



Channel modelling for DNA-based data storage

DA Walker

 **orcid.org/ 0000-0003-4842-1632**

Dissertation accepted in fulfilment of the requirements for the degree *Master of Engineering in Computer and Electronic Engineering* at the North-West University

Supervisor: Prof ASJ Helberg

Graduation: August 2024

DECLARATION

I, Daniël Andreas Walker, hereby declare that the dissertation titled “Channel modelling for DNA-based storage” is my own original work and has not already been submitted to any other university or institution for examination.

ABSTRACT

DNA-based storage offers a high-density data storage solution, but the synthesis and sequencing processes cause high error rates. Accurate models of the errors and error distributions are required due to the high synthesis cost. Currently, available channel models fail to represent all aspects of the errors introduced during the DNA data storage process. This study proposes new models capable of representing DNA data storage channel error bursts. A Markov model is used alongside substring characteristics derived from a MinION dataset. The methods for creating the proposed models are described. The proposed models are compared to currently available models. We show that the proposed models represent string-specific characteristics, error runs, bursts, and error-free runs better than similar existing models.

Key terms - Channel Modelling, DNA-Based Storage, Markov Models

TABLE OF CONTENTS

Declaration.....	i
Abstract	ii
List of tables	xi
List of Figures	xiv
CHAPTER 1 INTRODUCTION	1
1.1 Background	1
1.1.1 Introduction.....	1
1.1.2 Errors	1
1.1.3 Error correction coding for DNA data storage	2
1.1.4 Channel models.....	3
1.1.5 Statistical models.....	4
1.1.6 Machine learning in channel modelling	5
1.1.7 Datasets	6
1.1.8 Conclusion.....	6
1.2 Research problem and objectives.....	7
1.2.1 Research problem	7
1.2.2 Objectives:.....	7
1.2.2.1 Create a model for the DNA data storage channel.....	7
1.2.2.2 Compare the created models to each other and to existing solutions	7
1.3 Research Methodology	7
1.3.1 Collection and preparation of data	7
1.3.2 Statistical models.....	8
1.3.3 Comparison of models.....	8
1.4 Chapter division	8
1.5 Publication	9
CHAPTER 2 LITERATURE STUDY	10
2.1 DNA	10
2.2 DNA data storage	10
2.2.1 Comparison of DNA storage to traditional storage	10
2.2.2 Comparison of <i>in vivo</i> and <i>in vitro</i> approaches.....	11
2.2.3 DNA data storage process.....	13
2.2.4 Synthesis methods	14
2.2.4.1 Column-based oligo synthesis	14

2.2.4.2	Array-based oligo synthesis.....	15
2.2.5	Polymerase chain reaction.....	15
2.2.6	Sequencing technologies.....	16
2.2.6.1	Sanger sequencing.....	16
2.2.6.2	Next Generation Sequencing.....	17
2.2.6.3	Nanopore sequencing.....	17
2.2.6.4	FASTQ	18
2.2.7	Base callers.....	18
2.2.8	Trace reconstruction.....	19
2.2.9	Errors in DNA data storage.....	19
2.2.10	Dependency of errors on sequence characteristics	19
2.2.10.1	Homopolymer runs	19
2.2.10.2	Position within DNA string	20
2.2.10.3	GC content	20
2.2.10.4	Problematic substrings	21
2.2.10.5	Error bursts and runs.....	21
2.2.11	Encoding	21
2.3	Types of channel models.....	22
2.3.1	IID models	22
2.3.2	Markov models	22
2.3.2.1	Fritchman models.....	24
2.3.2.2	Gilbert-Elliot models.....	24
2.4	Estimating Markov model parameters	25
2.4.1	Baum-Welch algorithm	25
2.4.2	Viterbi estimation	27
2.4.2.1	Viterbi algorithm.....	27
2.4.2.2	Viterbi training.....	29
2.4.3	Comparison of Baum-Welch and Viterbi estimation	29
2.5	Existing DNA data storage models	29
2.5.1	DNA storage error simulation pipeline	29
2.5.2	Skewed DNASimulator model.....	30
2.5.3	DeepSimulator.....	31
2.5.4	Badread.....	32
2.5.5	Channel model with memory	33
2.6	Available datasets	35
2.6.1	Clustered Nanopore Reads dataset.....	36
2.6.2	DNA Fountain Code dataset.....	36

2.6.3	Practical high-capacity low-maintenance storage dataset.....	36
2.6.4	Performance assessment of DNA sequencing platforms	36
2.6.5	Nanopore Technologies Benchmark Datasets	37
2.7	String comparison algorithms	37
2.7.1	Hamming distance	37
2.7.2	Levenshtein distance	37
2.7.3	Damerau-Levenshtein Distance.....	38
2.7.4	Edit script using Levenshtein distance	39
2.7.5	Myers difference algorithm.....	39
2.8	Summary	39
CHAPTER 3	EXPERIMENTAL DESIGN	40
3.1	Objective	40
3.2	Selecting a dataset	40
3.3	Processing the dataset	41
3.4	Type of model	41
3.5	Selecting existing models for comparison	42
3.6	Experimental process	43
3.6.1	Markov model.....	43
3.6.2	Gilbert-Elliott model	43
3.6.3	Sequences of outcomes	43
3.6.4	Improved model.....	44
3.6.5	IID model	44
3.6.6	Comparison of models.....	44
3.7	Comparison areas	44
3.7.1	Error distribution	45
3.7.2	Error types	45
3.7.3	Homopolymer runs and problematic substrings	46
3.7.4	GC content	46
3.8	Statistical metrics	46
3.8.1	Correlation coefficient	46
3.8.2	Skewness	47
3.8.3	Root mean square error.....	47
3.9	Technologies used	48
3.10	Summary	49
CHAPTER 4	IMPLEMENTATION	50
4.1	Levenshtein Distance Error Identification	51

4.1.1	Verify error identification using Levenshtein distance.....	52
4.2	Data statistics	53
4.3	Detecting errors in a dataset	54
4.4	Strings of outcomes	55
4.5	Strings of outcomes to train the independent Gilbert-Elliot models	55
4.6	Deriving model parameters.....	56
4.6.1	Markov model.....	57
4.6.2	Independent Gilbert-Elliot models	57
4.7	Generating model outputs	58
4.8	Generating model outputs – independent Gilbert-Elliot models.....	59
4.9	Populating strings of outcomes with bases	61
4.10	Creating an IID model.....	61
4.11	Training the Memory model on the CNR dataset	62
4.12	Comparison of models.....	64
4.12.1	Error runs	64
4.12.2	Error bursts.....	64
4.12.3	Error-free runs	65
4.13	Homopolymer analysis	65
4.14	GC content analysis	66
4.15	Running digital sum analysis	66
4.16	Problematic substring analysis.....	67
4.17	Swap error analysis.....	68
4.18	Statistical metrics.....	69
4.19	Summary	69
CHAPTER 5	VERIFICATION	70
5.1	Levensthein Distance Error Identification	70
5.1.1	Individual examples	70
5.1.2	Verify error identification using Levenshtein distance.....	74
5.2	Detecting errors in a dataset	75
5.3	Strings of outcomes	75
5.4	Strings of outcomes to train the independent Gilbert-Elliot models	75
5.5	Deriving model parameters.....	75
5.6	Data statistics	76
5.7	Generating model outputs	77
5.8	Creating an IID model.....	77
5.9	Training the Memory model on the CNR dataset	77
5.10	Comparison of models.....	78

5.10.1	Importing data.....	78
5.10.2	Error-free strings.....	78
5.10.3	Average string length.....	78
5.10.4	General statistics	78
5.10.5	Error rate at each position	78
5.10.6	Error runs	79
5.10.7	Error bursts.....	81
5.10.8	Error-free runs	83
5.11	Homopolymer analysis	84
5.12	GC content analysis	85
5.13	Running digital sum analysis	86
5.14	Problematic substring analysis	87
5.14.1	Cartesian product	87
5.14.2	Problematic substring analysis	87
5.15	Swap error analysis.....	90
5.16	Summary	91
CHAPTER 6	ANALYSIS OF MARKOV MODELS.....	92
6.1	Model parameters.....	92
6.1.1	Independent Gilbert-Elliot models.....	92
6.1.2	Markov models	92
6.1.3	Summary of model parameters.....	93
6.2	Comparison of different Markov models	94
6.2.1	General statistics	94
6.2.2	Error runs	95
6.2.3	Error bursts.....	96
6.2.4	Error-free runs	96
6.3	Summary	97
CHAPTER 7	ANALYSIS OF THE DATASET.....	98
7.1	Error rates at each position	98
7.2	Homopolymer analysis	98
7.3	Problematic substring analysis	99
7.3.1	Substrings of length 2.....	99
7.3.2	Substrings of lengths 4 and 6	99
7.3.3	Summary	100
7.4	GC content analysis	100
7.5	Running digital sum	101

7.6	Swap errors.....	102
7.7	Summary	102
CHAPTER 8	IMPROVED MODEL.....	104
8.1	Error rates at the start and the end of the string	104
8.2	Substring characteristics.....	104
8.3	Substring model	105
8.4	Models that combine a Markov model with the substring characteristics	105
8.4.1	Model components	107
8.4.1.1	Markov model as a base.....	107
8.4.1.2	Substring data with reduced error rate	107
8.4.2	Methods for implementing substring characteristics.....	108
8.4.2.1	Consecutive windows	108
8.4.2.2	Sliding forward windows	108
8.4.2.3	Sliding backward windows.....	108
8.4.3	Implementing negative values in the substring data.....	109
8.4.4	Combining Markov model outputs with outputs generated from substring characteristics.....	109
8.4.5	Resultant models	110
8.5	Summary	110
CHAPTER 9	RESULTS.....	111
9.1	Dataset and models used for comparison.....	111
9.1.1	Dataset.....	111
9.1.2	Newly created models	111
9.1.3	7MM	112
9.1.4	IID model and Memory model.....	113
9.2	General statistics.....	113
9.3	Error rate at each position	114
9.4	Error runs	117
9.5	Error bursts.....	119
9.6	Error-free runs	121
9.7	Homopolymer analysis	123
9.8	GC content	125
9.9	Running digital sum	127
9.10	Problematic substrings.....	129
9.10.1	Substrings of length 2	129
9.10.2	Substrings of length 4	131

9.10.3	Substrings of length 6	134
9.11	Swap errors.....	139
9.12	Discussion	140
9.12.1	7MM	140
9.12.2	Substring model.....	140
9.12.3	Combined model	141
9.12.4	Sliding forwards model	141
9.12.5	Sliding backwards model	141
9.13	Summary	142
CHAPTER 10 CONCLUSION		144
10.1	Research problem	144
10.2	Method.....	144
10.2.1	Dataset.....	144
10.2.2	Existing models	144
10.2.3	Proposed models.....	144
10.2.4	Areas for comparison.....	145
10.3	Results	145
10.4	Summary	146
10.5	Future work.....	147
References.....		149
APPENDIX A LEVENSHTAIN DISTANCE ERROR IDENTIFICATION ALGORITHM		167
APPENDIX B ALGORITHM FOR POPULATING STRINGS OF OUTCOMES WITH BASES		169
APPENDIX C PYTHON IMPLEMENTATION OF THE MEMORY MODEL		172
APPENDIX D ALGORITHM FOR DETECTING ERROR RUNS		175
APPENDIX E ALGORITHM FOR DETECTING ERROR BURSTS.....		176
APPENDIX F ALGORITHM FOR DETECTING ERROR-FREE RUNS.....		178
APPENDIX G ALGORITHM FOR DETECTING ERRORS IN HOMOPOLYMER RUNS		179
APPENDIX H EXAMPLE STRING FROM THE DATASET USED TO VERIFY THE LEVENSHTAIN DISTANCE ERROR IDENTIFICATION ALGORITHM		182
APPENDIX I EXAMPLE STRING TO VERIFY THE GENERATION OF STRINGS OF OUTCOMES.....		183
APPENDIX J EXAMPLE STRING TO VERIFY THE GENERATION OF STRINGS OF OUTCOMES FOR INDEPENDENT GILBERT-ELLIOT MODELS.....		184

APPENDIX K ERRORS IN EXAMPLE STRINGS USED TO VERIFY THE ALGORITHM TO CALCULATE STATISTICS.....	185
APPENDIX L EXAMPLES FOR VERIFYING THE CALCULATION OF ERROR RATES AT EACH STRING POSITION	186
APPENDIX M SUMMARY OF GC CONTENT ANALYSIS FOR A SAMPLE STRING	189
APPENDIX N SUMMARY OF RDS ANALYSIS FOR A SAMPLE STRING	191
APPENDIX O EXAMPLES FOR PROBLEMATIC SUBSTRING ANALYSIS	193
APPENDIX P TRANSITION AND EMISSION MATRICES OF DIFFERENT MARKOV MODELS	199
APPENDIX Q COMPARISON OF MARKOV MODELS	206
APPENDIX R ANALYSIS OF THE DATASET	212
APPENDIX S GENERAL STATISTICS OF DIFFERENT MODELS.....	231
APPENDIX T ERROR RATES PRODUCED AT DIFFERENT POSITIONS.....	232
APPENDIX U ANALYSIS OF ERROR RUNS.....	235
APPENDIX V ANALYSIS OF ERROR BURSTS	238
APPENDIX W ANALYSIS OF HOMOPOLYMERS.....	240
APPENDIX X ANALYSIS OF GC CONTENT	254
APPENDIX Y RUNNING DIGITAL SUM ANALYSIS	255
APPENDIX Z PROBLEMATIC SUBSTRING ANALYSIS.....	263
APPENDIX AA MODELLING ERROR RUNS IN DNA DATA STORAGE USING A MARKOV MODEL.....	272

LIST OF TABLES

Table 2-1: Percentage of characters and strings accurately reconstructed by trace reconstruction algorithms when comparing the skewed DNASimulator model to the Clustered Nanopore Reads dataset [108]	31
Table 3-1: Summary of characteristics of different datasets	40
Table 3-2: Meaning of different skewness values	47
Table 3-3: Technologies used	49
Table 4-1: Numerical representation of each type of observation	55
Table 6-1: Summary of model parameters.....	93
Table 6-2: Comparison of different Markov models	94
Table 7-1: Statistics for error rates of different length substrings in the dataset	100
Table 7-2: Correlation coefficient between RDS value and error rate	102
Table 9-1: Probability of starting in each state for the 7MM model.....	113
Table 9-2: Summary of model statistics.....	114
Table 9-3: RMSLE for error runs produced by each model.....	119
Table 9-4: RMSLE for error bursts produced by each model	121
Table 9-5: RMSLE for error-free run lengths produced by each model.....	123
Table 9-6: NRMSE for error rates in homopolymers produced by each model normalised for mean error rate produced by the model	125
Table 9-7: NRMSE for error rates of length 6 windows with different GC contents produced by each model normalised for mean error rate	127
Table 9-8: Pearson correlation coefficient between RDS value and error rate	128
Table 9-9: NRMSE for error rate produced by each model for different RDS values.....	128
Table 9-10: Comparison of error rates produced by different models for different substrings of length 2	131
Table 9-11: Statistics for error rates produced by each model within length 4 substrings	132
Table 9-12: Comparison of error rates produced by different models for different substrings of length 4	133
Table 9-13: Statistics for error rates produced by each model within length 6 substrings	134
Table 9-14: Comparison of error rates produced by different models for different substrings of length 6	138
Table 9-15: Percentage of errors that make up swap errors produced by each model	139
Table 9-16: Comparison of different models.....	142
Supplementary Table L-1: Number of errors at different positions within the example data set.....	187
Supplementary Table L-2: Error rate at different positions within the example data set	188
Supplementary Table M-1: Summary of the error rate of substrings with different GC contents in the example string.....	190

Supplementary Table N-1: Summary of RDS analysis for a sample string.....	192
Supplementary Table O-1: Analysis of substrings of length 2 for example strings	193
Supplementary Table O-2: Summary of errors at the first position of substrings of length 2 for example strings.....	194
Supplementary Table O-3: Summary of error rates at the first position of substrings of length 2 for example strings	195
Supplementary Table O-4: Summary of errors at the second position of substrings of length 2 for example strings	196
Supplementary Table O-5: Summary of error rates at the second position of substrings of length 2 for example strings	197
Supplementary Table O-6: Summary of problematic substring analysis of substrings of length 6 for an example string	198
Supplementary Table P-1: Transition matrix of GE model pertaining insertion errors	199
Supplementary Table P-2: Transition matrix of GE model pertaining deletion errors	199
Supplementary Table P-3: Transition matrix of GE model pertaining substitution errors.....	199
Supplementary Table P-4: Emission matrix of the three GE models.....	199
Supplementary Table P-5: Transition matrix of 7MMLS model	200
Supplementary Table P-6: Emission matrix of 7MMLS model	200
Supplementary Table P-7: Transition matrix of 7MMIT model	201
Supplementary Table P-8: Emission matrix of 7MMIT model.....	201
Supplementary Table P-9: Transition matrix of 7MM model	202
Supplementary Table P-10: Emission matrix of 7MM model.....	202
Supplementary Table P-11: Transition matrix of 4MM model.....	202
Supplementary Table P-12: Emission matrix of 4MM model.....	203
Supplementary Table P-13: Transition matrix of 6MM model.....	203
Supplementary Table P-14: Emission matrix of 6MM model.....	203
Supplementary Table P-15: Transition matrix of 9MM model.....	204
Supplementary Table P-16: Emission matrix of 9MM model.....	204
Supplementary Table P-17: Transition matrix of 11MM model.....	205
Supplementary Table P-18: Emission matrix of 11MM model.....	205
Supplementary Table R-1: Length 4 substrings in the dataset with the highest error rates ...	219
Supplementary Table R-2: Length 4 substrings in the dataset with the lowest error rates	219
Supplementary Table R-3: Length 6 substrings in the dataset with the highest error rates ...	222
Supplementary Table R-4: Length 6 substrings in the dataset with the lowest error rates	222
Supplementary Table S-1: Distribution of error types produced by each model	231
Supplementary Table Z-1: Error rate of each length 2 substring produced by each model ...	263
Supplementary Table Z-2: Statistics for error rates produced by each model within length 2 substrings.....	264

Supplementary Table Z-3: Length 4 substrings with the highest error rate produced by each model	268
Supplementary Table Z-4: Length 4 substrings with the lowest error rate produced by each model	269
Supplementary Table Z-5: Length 6 substrings with the highest error rate produced by each model	270
Supplementary Table Z-6: Length 6 substrings with the lowest error rate produced by each model	271

LIST OF FIGURES

Figure 2-1: Simplified representation of the DNA data storage process.....	13
Figure 3-1: Areas in which models are compared.....	45
Figure 4-1: Workflow for creating a Markov model.....	50
Figure 4-2: Combining outputs of three independent Gilbert-Elliott models.....	60
Figure 5-1: Evaluating the error identification algorithm on different example strings.....	72
Figure 5-2: Examination of error runs identified by the model.....	80
Figure 5-3: Examination of error bursts identified by the model	82
Figure 5-4: Detection of error-free runs on an example string.....	83
Figure 5-5: Example strings for homopolymer analysis.....	84
Figure 5-6: Example strings for the analysis of problematic substrings of length two.....	88
Figure 5-7: Example string for the analysis of problematic substrings of length six.....	90
Figure 6-1: Comparison of error run lengths produced by different Markov models	95
Figure 8-1: Workflow for creating models combining a Markov model and substring characteristics, and generating output strings with these models	106
Figure 9-1: Comparison of error rates produced at each position by each model	115
Figure 9-2: Scatterplots representing error positions in the first 300 strings of the dataset and the output of each model.....	116
Figure 9-3: Comparison of error runs produced by each model.....	118
Figure 9-4: Comparison of error burst lengths produced by each model.....	120
Figure 9-5: Comparison of error-free run lengths produced by each model	122
Figure 9-6: Comparison of error rates produced within homopolymers by each model.....	124
Figure 9-7: Comparison of GC content of windows and error rate produced by each model for windows of length 6	126
Figure 9-8: Comparison of error rates produced by each model for each substring of length 2	130
Figure 9-9: Comparison of distribution of error rates between different substrings of length 6 produced by different models	135
Figure 9-10: Comparison of error rates produced by each model for each substring of length 6 ordered according to error rate produced by the model	136
Figure 9-11: Comparison of error rates produced by each model for each substring of length 6 ordered according to error rate in the dataset.....	137
Supplementary Figure H-1: Evaluating the error identification algorithm on an example from the dataset	182
Supplementary Figure I-1: Generated string of outcomes compared to original and noisy DNA strings.....	183
Supplementary Figure J-1: Strings of outcomes for each error type compared to original string of outcomes.....	184

Supplementary Figure M-1: GC content analysis for an example string.....	189
Supplementary Figure N-1: RDS analysis for a sample string	191
Supplementary Figure Q-1: Comparison of run lengths for runs of substitution errors produced by different Markov models.....	206
Supplementary Figure Q-2: Comparison of run lengths for runs of deletion errors produced by different Markov models.....	207
Supplementary Figure Q-3: Comparison of run lengths for runs of insertion errors produced by different Markov models.....	208
Supplementary Figure Q-4: Comparison of error burst lengths produced by different Markov models.....	209
Supplementary Figure Q-5: Comparison of errors per burst produced by different Markov models	210
Supplementary Figure Q-6: Comparison error-free run lengths produced by different Markov models.....	211
Supplementary Figure R-1: Error rates at different positions of the dataset.....	212
Supplementary Figure R-2: Error rate within homopolymers of different lengths in the CNR dataset.....	213
Supplementary Figure R-3: Error rate at the position right after homopolymers of different lengths in the CNR dataset	214
Supplementary Figure R-4: Error rate at the position right before homopolymers of different lengths in the CNR dataset.....	215
Supplementary Figure R-5: Error rate of each substring of length 2 in the dataset	216
Supplementary Figure R-6: Distribution of number of occurrences of different substrings of length 4 in the dataset.....	217
Supplementary Figure R-7: Distribution of error rates between different substrings of length 4 in the dataset.....	218
Supplementary Figure R-8: Distribution of number of occurrences of different substrings of length 6 in the dataset.....	220
Supplementary Figure R-9: Distribution of error rates between different substrings of length 6 in the dataset.....	221
Supplementary Figure R-10: Distribution of GC content of different strings in the dataset	223
Supplementary Figure R-11: Comparison of error rates in strings of different GC contents in the dataset	224
Supplementary Figure R-12: Comparison of GC content of windows and error rate for windows of length 6 in the dataset	225
Supplementary Figure R-13: Distribution of different RDS values in the dataset	226
Supplementary Figure R-14: Comparison of RDS values and error rates for the dataset	227
Supplementary Figure R-15: Comparison of RDS values and error rates of different types for the dataset.....	228
Supplementary Figure R-16: Comparison of RDS values calculated from the back of the string and error rates for the dataset	229
Supplementary Figure R-17: Comparison of RDS values calculated from the back of the string and error rates of different types for the dataset	230

Supplementary Figure T-1: Comparison of insertion rates produced at each position by each model	232
Supplementary Figure T-2: Comparison of deletion rates produced at each position by each model	233
Supplementary Figure T-3: Comparison of substitution rates produced at each position by each model	234
Supplementary Figure U-1: Comparison of insertion error runs produced by each model	235
Supplementary Figure U-2: Comparison of deletion error runs produced by each model	236
Supplementary Figure U-3: Comparison of substitution error runs produced by each model	237
Supplementary Figure V-1: Comparison of errors per burst produced by each model	238
Supplementary Figure V-2: Start positions of bursts produced by each model	239
Supplementary Figure W-1: Comparison of error rates produced before homopolymers by each model	240
Supplementary Figure W-2: Comparison of error rates produced after homopolymers by each model	241
Supplementary Figure W-3: Comparison of error rates produced within homopolymers of A by each model	242
Supplementary Figure W-4: Comparison of error rates produced before homopolymers of A by each model	243
Supplementary Figure W-5: Comparison of error rates produced after homopolymers of A by each model	244
Supplementary Figure W-6: Comparison of error rates produced within homopolymers of C by each model	245
Supplementary Figure W-7: Comparison of error rates produced before homopolymers of C by each model	246
Supplementary Figure W-8: Comparison of error rates produced after homopolymers of C by each model	247
Supplementary Figure W-9: Comparison of error rates produced within homopolymers of G by each model	248
Supplementary Figure W-10: Comparison of error rates produced before homopolymers of G by each model	249
Supplementary Figure W-11: Comparison of error rates produced after homopolymers of G by each model	250
Supplementary Figure W-12: Comparison of error rates produced within homopolymers of T by each model	251
Supplementary Figure W-13: Comparison of error rates produced before homopolymers of T by each model	252
Supplementary Figure W-14: Comparison of error rates produced after homopolymers of T by each model	253
Supplementary Figure X-1: Comparison of error rates produced by each model for strings of different GC contents	254

Supplementary Figure Y-1: Comparison of error rates produced by each model for different RDS values calculated from the front of the string	255
Supplementary Figure Y-2: Comparison of insertion rates produced by each model for different RDS values calculated from the front of the string	256
Supplementary Figure Y-3: Comparison of deletion rates produced by each model for different RDS values calculated from the front of the string	257
Supplementary Figure Y-4: Comparison of substitution rates produced by each model for different RDS values calculated from the front of the string	258
Supplementary Figure Y-5: Comparison of error rates produced by each model for different RDS values calculated from the back of the string.....	259
Supplementary Figure Y-6: Comparison of insertion rates produced by each model for different RDS values calculated from the back of the string.....	260
Supplementary Figure Y-7: Comparison of deletion rates produced by each model for different RDS values calculated from the back of the string.....	261
Supplementary Figure Y-8: Comparison of substitution rates produced by each model for different RDS values calculated from the back of the string.....	262
Supplementary Figure Z-1: Comparison of distribution of error rates between different substrings of length 4 produced by different models.....	265
Supplementary Figure Z-2: Comparison of error rates produced by each model for each substring of length 4 ordered according to error rate in the dataset.....	266
Supplementary Figure Z-3: Comparison of error rates produced by each model for each substring of length 4 ordered according to error rate produced by the model.....	267

CHAPTER 1 INTRODUCTION

1.1 Background

1.1.1 Introduction

Due to the large amounts of data that are constantly produced in the current technological era, traditional storage media are hard-pressed to keep up with growing demands. It is predicted that the volume of globally stored data will increase to 175 ZB by 2025 [1]. Many corporations are investigating new storage methods, including DNA data storage systems. The advantages of DNA data storage systems include the high data density provided by DNA, with a theoretical limit of $\sim 4.5 \times 10^7$ GB/g [2]. Furthermore, DNA molecules remain stable for thousands of years, which is a very desirable attribute for archival data storage. This is demonstrated by the fact that scientists have successfully sequenced mitochondrial DNA from a bear that is 300 000 years old [3]. Lastly, DNA storage systems require much less power than conventional archival methods, thus maintenance costs are much lower [4].

DNA itself is a molecule consisting of four bases or nucleotides, namely Adenine, Cytosine, Guanine and Thymine (A, C, G and T). Data can then be stored via a four-letter alphabet, represented by these four bases. The DNA data storage process consists of encoding data to this alphabet and thereafter synthesising the corresponding DNA strings. Polymerase chain reaction (PCR) is used to amplify DNA strands, i.e. creating more copies of each strand before the data can be read. In order to read the data, the DNA strings should then be sequenced, whereafter the data can be decoded. Synthesis and PCR occur via various chemical reactions and are usually done in laboratories. Sequencing is done via specialised machinery, and common sequencing technologies include Illumina and Oxford Nanopore sequencing [5].

1.1.2 Errors

During the various stages of the DNA data storage process, different types of errors can occur. Deletion errors may occur when one or more nucleotides are deleted from the string, i.e., not written during synthesis or not read during sequencing. Insertion errors may occur when nucleotides are added to the string in erroneous locations during the synthesis process or read in an erroneous location during sequencing. Lastly, substitution errors may occur when a nucleotide is substituted for a different nucleotide, for example, a C can be read as a T [5].

Additionally, entire DNA sequences may be lost. Sequences may break up in storage. PCR bias may also lead to a loss of sequences. PCR bias refers to when certain strands are amplified more than others. As PCR is implemented in many cycles, this may result in large amounts of certain

strands but negligible amounts of others. This could reduce the likelihood that certain strands are sequenced and may lead to a loss of strands [5].

Lastly, secondary structures may form within DNA strands due to hybridisation. Hybridisation refers to the process by which two single-stranded DNA molecules that complement each other bond together to form a single double-stranded molecule [6]. This may occur within a single strand when different parts of that strand bond together, yielding loops or other secondary structures within that DNA strand. Strands with secondary structures cannot be sequenced, leading to a loss of data.

Burst errors (multiple errors in a row) are also common [7]. Depending on the synthesis and sequencing technology used, as well as the storage conditions, error rates may vary. Nanopore sequencing is a modern type of sequencing which is popularly used due to its low cost, but has the disadvantage of high error rates. Nanopore sequencing may result in an error rate of more than 10% [5]. Due to these high error rates, redundancy is required to ensure data integrity. To maintain the main advantage of DNA data storage, namely data density, efficient error correction codes are used to maximise data integrity while minimising redundancy.

Contrary to the assumption made by many models, research indicates that errors do not occur uniformly, but certain areas of a DNA string may see more errors [8]–[10]. Areas with a higher GC content may see an increase in errors [9]. Furthermore, errors may be more likely to occur at the beginning or end of a DNA string than at the centre [8]. Many current models also assign PCR bias randomly, while certain molecules may have characteristics that influence PCR bias. Church [10] also states in some of his earliest work that errors may not be uniformly distributed.

1.1.3 Error correction coding for DNA data storage

In order to reduce error rates, various constraints are introduced when designing error correction codes for DNA data storage.

Firstly, the maximum G and C content of a DNA string is constrained. This is due to the fact that a GC content much higher or lower than 50% can lead to increased error rates [7].

Secondly, the number of homopolymers, that is repetitions of the same base, is limited via run-length constraints. Long runs of homopolymers can lead to increased errors [7]. Many codes exclude any homopolymers. Some codes allow homopolymers up to a certain length, normally three, as this does not seem to significantly impact error rates [11].

The total length of each codeword is also constrained as current synthesis technology can only reliably synthesise DNA strings of 100-200 nucleotides [12].

Furthermore, codewords should be designed in order to prevent hybridisation. Certain algorithms that can predict the formation of secondary structures, such as NUPACK, may aid in this design [13].

Many codes implement additional constraints, such as mutual Hamming distance constraints for error correction purposes [14].

Due to the above-mentioned constraints as well as the high error rates present in DNA data storage channels, a large amount of research is being done on specialised error correction encoding for this novel storage medium.

Many experiments have been done to develop error correction codes for DNA data storage. Many experiments relied on redundancy in order to ensure data integrity. Goldman et al. [15], for example, used ternary digits to encode the data. Thereby, each nucleotide could be selected to differ from the previous base to prevent homopolymers. Furthermore, different DNA strings were created to overlap, thereby creating four-fold physical redundancy. A total of 739 kB was encoded.

Grass et al. [16] used Reed-Solomon codes to correct burst errors. Overlap was also implemented in order to create physical redundancy. A total of 83 kB was encoded into 4991 DNA segments of 158 nucleotides each.

Due to current sequencing technology and the use of PCR, multiple copies of the same DNA string are usually read [17]. This gives DNA data storage an inherent physical redundancy. Certain error correction methods take advantage of this and compare different reads of the same molecule. As errors are unlikely to occur at the same position in multiple strings, different reads may be compared to reduce the number of errors. Immink et al. [18] used such a method. Due to the reduction in errors, this encoding scheme used codewords capable of correcting a single error in each DNA string.

1.1.4 Channel models

Current technologies make the DNA synthesis and sequencing processes very expensive. In 2021, the cost of DNA storage was estimated at \$1 300 000/TB, while tape storage is \$7/TB [19]. Although the costs are predicted to decrease in the coming years, it is still a barrier to entry for prospective researchers. Furthermore, highly specialised equipment is needed, including sequencing machines. This makes the process inaccessible to many researchers. Therefore, simulation environments that allow researchers to test their encoding schemes are vital for the development of improved error correction methods. Channel models may be used to simulate the occurrence of errors in a DNA storage system. Error correction codes can then be tested on the model to determine their effectiveness.

Many current channel models for the DNA data storage system use statistics from experiments to model the occurrences of errors. One such model designed specifically for Nanopore sequencing distributes errors uniformly throughout the sequence. The number of errors introduced is based on the error rate of the technology. Due to the large amount of resulting molecules after running many PCR cycles (e.g. 40 cycles), a constant output size is generated by taking a uniform sub-sample after each cycle. Synthesis error rates are assumed to be negligible for short DNA strings [20].

Another model [21] used assumes that the synthesiser creates an equal number of copies of each DNA strand. Furthermore, PCR bias is disregarded. Error rates for synthesis and sequencing are combined and the total number of errors are injected into the DNA string.

DeSP (DNA storage error simulation pipeline) is another popular channel modelling tool [22]. This model accounts for both lost sequences, as well as within-sequence errors, namely errors occurring within a specific DNA string, such as insertions, deletions and substitutions. Statistical error rates are introduced independently at each stage of the DNA storage process. Errors are once again uniformly distributed throughout each DNA string. Furthermore, molecules breaking up into smaller pieces that cannot be sequenced successfully are statistically simulated in the storage process. PCR bias is also accounted for.

1.1.5 Statistical models

A statistical approach capable of modelling communications channels is the Markov chain. It uses a fixed probability that the channel will transition between different states at a specified time instant, based on the state at the previous time instant [23]. States may include an erroneous state or an errorless state. A study has attempted to model the DNA storage channel using a Markov chain [24]. Instead of time instants, the characterisation is based on the current position within the DNA string. A shortcoming of this model is that it does not properly represent long-run errors. On the other hand, Markov models have been implemented in wireless communication with great success [25]. This indicates that additional variations of Markov models may be explored.

A Gilbert-Elliot channel refers to a special case of the Markov chain, consisting of only two states [26]. Gilbert-Elliot models assume that a channel can only switch between a good and a bad state. These models have been used in communication channels to estimate the channel state where constantly probing the channel is too costly [27].

An additional model considered in channel coding for wireless communication channels is the Fritchman model [25]. In existing literature, a Fritchman model is shown to contain memory and

is therefore able to model bursts of errors. Memory, in this instance, indicates that the current state of the channel is dependent on prior channel states. Furthermore, if correctly configured, Fritchman models are also capable of representing insertion and deletion errors. This model may be expanded to the DNA storage channel.

While these models have been shown to perform well for communications channels, a few key factors should be considered when addressing the DNA storage system. Many previous implementations of these models were designed for binary channels. Furthermore, insertion and deletion errors were not always accounted for. Therefore, the only type of error considered is when a bit is flipped (a 0 is turned into a 1 or vice versa). This requires expansion to consider substitution errors over a four-letter alphabet, as well as insertion and deletion errors, if the models are to be implemented in the DNA data storage environment. Furthermore, these models rely on statistical analysis of a specific dataset to deduce model parameters. Therefore, generalisation may prove difficult.

1.1.6 Machine learning in channel modelling

While statistical methods have regularly been used to model communications channels, recent years have seen an increase in the use of machine learning to characterise these channels. With the large quantities of data available, machine learning may be used to create models based on the behaviour of communication channels.

The Long Short-Term Memory (LSTM) deep learning model has been studied for wireless channel estimation in 5G communications [28]. This model was found to predict the channel state with an accuracy of 85%. Convolutional neural networks (CNNs) as well as recurrent neural networks (RNNs) have also been implemented in previous studies detailing communication channel modelling [29].

While there are many examples of machine learning models for wireless communication channels, there are also some cases where such models are used in the realm of DNA data storage. DeepSimulator, for example, uses AI to simulate the electrical current produced by Nanopore devices [30]. This current can then be decoded to simulate the sequencing process. A shortcoming of DeepSimulator is that the same kind of error regularly appears in the same position in different sequences, which is not representative of a true DNA storage channel [24].

Another study proposed a machine learning model that is capable of predicting the formation of secondary structures in a DNA string [31]. This is very useful when designing codes, and the model performs as accurately and much faster than traditional algorithms. It should be noted that the model only predicts the formation of secondary structures, but not any other types of errors. Due to the data-hungry nature of machine learning and the high cost of acquiring datasets

experimentally, this model was trained on synthetic data. Existing algorithms were used to predict hybridisation of given strings in order to create the dataset.

1.1.7 Datasets

In order to implement machine channel models, large amounts of experimental data regarding the DNA storage channel are required. The following freely available datasets have been identified thus far:

1. Clustered Nanopore Reads dataset

A set of 10 000 DNA strings that were each read multiple times. The 270 000 resultant strings after sequencing are also available [32].

2. DNA Fountain Code dataset

The fountain code is a specific type of error correction code. The authors of this study applied this code to DNA storage and made the experimental data available [33].

3. Practical high-capacity low-maintenance storage dataset

The synthesis and sequencing dataset produced by the experiment by Goldman et al. as described above is available [15].

4. Performance assessment of DNA sequencing technologies

The experimental data for a study done on Next Generation Sequencing technology is available [34].

5. Nanopore Technologies Benchmark Datasets

This collection of datasets includes sequencing data provided by Oxford Nanopore Technologies and Amazon for nanopore sequencing machines [35].

1.1.8 Conclusion

DNA data storage is a novel storage method with the advantage of a high data density. Furthermore, once a DNA archive is established, maintenance costs are very low. DNA can also remain stable for long periods of time. Disadvantages of DNA data storage include high error rates. Due to the unique nature of errors caused by the DNA data storage process, specialised error correction codes are required. Testing such codes experimentally is expensive, therefore accurate error models are required to make code development more accessible. Current channel models have various shortcomings, consequently, improved channel models are required. Various unexplored methods for modelling DNA storage channels have been described.

1.2 Research problem and objectives

1.2.1 Research problem

Current channel models for DNA storage do not encompass all aspects of the channel, including different error types and bursts of errors. A model that addresses the shortcomings of current solutions is required to simulate the DNA data storage channel.

1.2.2 Objectives:

Two objectives are stated for this study.

1.2.2.1 Create a model for the DNA data storage channel

Statistical models are frequently used to model communication channels. Therefore, many established modelling methods exist that have been proven to provide accurate results. These models may also be able to accurately model the DNA data storage channel. At least one suitable model shall be identified and applied to the DNA data storage environment. As many such models were created for a binary environment, the modelling method shall be modified to ensure that it works on a four-letter alphabet. Thereby, a method for modelling the DNA data storage channel shall be proposed.

1.2.2.2 Compare the created models to each other and to existing solutions

In order to evaluate the performance of the created models, the models are compared. The performance of the created models and existing models is compared on an existing data set. Thereby, the efficacy of the proposed modelling method is ascertained.

1.3 Research Methodology

1.3.1 Collection and preparation of data

For statistical modelling, data is required. Existing published datasets are used in this study.

The datasets are prepared to ensure that the data is represented in the correct format. It is confirmed that all data is adequately labelled. This includes the detection of errors within a sequenced DNA string when compared to the desired string. These are used to create the models.

1.3.2 Statistical models

Statistical models as described above are created. This includes Gilbert-Elliott and Markov models. Model parameters are derived by analysing the available datasets.

1.3.3 Comparison of models

The created models are compared to each other and to currently available DNA data storage models found in literature. The overall error rate of each model is determined and compared to the error rate of experimental data. The ability of each model to represent bursts of errors is also assessed. Furthermore, the ability of the models to represent errors due to specific causes is assessed. This includes high GC content within a specific DNA string, the position within a DNA string and homopolymers of various lengths in different positions.

1.4 Chapter division

The following chapter titles appear in the dissertation:

1. Introduction

The introduction provides an overview, some background, as well as the research aim and objectives.

2. Literature study

The literature study elaborates on available research and solutions. Furthermore, literature on techniques to be used is also examined.

3. Experimental design

The experimental design is described in this chapter, including the choice of dataset, algorithms and comparison metrics.

4. Implementation

The implementation of the experimental method is discussed in this chapter. The algorithms used are described, as well as the technologies used to implement these algorithms.

5. Verification

In this chapter, the algorithms used are examined for accuracy through various examples.

6. Analysis of Markov models

The implementation of Markov models is described in this chapter. The ability of the models to represent error runs and bursts is also analysed.

7. Analysis of the dataset

The dataset is analysed in different areas to determine the effect of different characteristics on errors within the DNA strings.

8. Improved model

The implementation of an improved model with the aim of representing additional characteristics of the dataset is described in this chapter.

9. Results

The improved model is compared to the dataset and existing models from literature in this section. The results are discussed in this section to evaluate the performance of each model.

10. Conclusion

An overview of the study is given along with a conclusion. Future studies and improvements that may be made to the proposed models are discussed in this section.

1.5 Publication

The research described in this dissertation led to the following publication:

- “Modelling Error Runs in DNA Data Storage Using a Markov Model” [36] was published at the Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2023. This paper can be found in Appendix AA.

Some of the published work is included in this dissertation. Subsections of Section 1.1 have been published in the SATNAC paper. Additionally, the SATNAC paper describes the 7MM model further described in Section 9.1.3 and analysed in Chapter 9.

CHAPTER 2 LITERATURE STUDY

2.1 DNA

Deoxyribonucleic acid, commonly referred to as DNA, is a molecule in living cells that carries the genetic information required for the development and function of an organism [37]. DNA consists of monomeric units called bases. There are four types of bases, namely Adenine, Cytosine, Guanine and Thymine. The bases are referred to by the letters A, C, G and T respectively. These bases are connected into a strand with deoxyribose, a sugar, between each base. A nucleoside refers to a base that is linked to a sugar [38]. A nucleotide refers to a nucleoside that is bonded to a phosphoric acid [38]. In literature, these terms are often used interchangeably. Various molecule groups can react chemically with either the sugar or the phosphate. The sugar is referred to as the 3' end, while the phosphate is referred to as the 5' end of the nucleotide [39]. In naturally occurring DNA, two such strands are bound together via hydrogen bonds in order to create a double helix structure. Base pairing rules dictate that A binds with T and C with G, therefore the two DNA strands in a single DNA molecule are complementary. A single-stranded DNA string is also referred to as an oligo [40].

2.2 DNA data storage

In order to store data in DNA, data is encoded onto a four-letter alphabet that corresponds to the four types of bases. The following example shows how binary data can be encoded onto a quaternary alphabet: $00 \rightarrow A, 01 \rightarrow C, 10 \rightarrow G, 11 \rightarrow T$. After conversion to this quaternary alphabet, DNA strings containing the data are synthesised. The DNA can be stored *in vivo* or *in vitro*. *In vivo* DNA refers to DNA within living organisms, while *in vitro* refers to non-living DNA [41].

2.2.1 Comparison of DNA storage to traditional storage

Due to its high data density, DNA-based storage is being investigated as a replacement for traditional storage methods. In 2016, Oxford Nanopore reported sequencing speeds of 400 to 450 bases per second [42]. When a base is taken to represent four bits, this corresponds to a read speed of around 1800 bits per second, as shown in equation (2-1):

$$\begin{aligned} \text{bits per second} &= \text{bases per second} \times \text{bits per base} = 450 \times 4 \\ &= 1800 \text{ bits per second} \end{aligned} \quad (2-1)$$

This equates to 225 bytes per second. While the read speed is projected to increase in the future, the current technological limitations make DNA data storage more suitable for archival storage, where read speeds do not need to be as fast as active storage.

Current archival data storage technologies include tape drives and hard disk drives (HDDs). In 2019, HDDs possessed an areal data density around two orders of magnitude higher than tape drives. Tape, on the other hand, has a much larger recording surface than HDDs, therefore this advantage is negated and tape may achieve a data density of three times that of HDDs of a similar form factor [43]. Tapes are therefore regularly viewed as the state-of-the-art technology for archival data storage. Modern tapes can achieve data transfer speeds of 400 MB/s, which is orders of magnitude faster than current DNA sequencing technologies [44]. DNA has a theoretical data density of approximately $\sim 4.5 \times 10^7$ GB/g, which is millions of times more than that of traditional data storage [45].

Tape currently remains the cheapest data storage method, with a price of \$7/TB as reported in 2021 [19]. The cost of DNA storage was estimated at \$1 300 000/TB in the same year [19]. This cost only includes the purchase costs and excludes maintenance and energy costs. While the costs for DNA data storage remain high, the costs are rapidly decreasing. DNA sequencing costs have decreased by an order of 5 from 2001 to 2022 [46]. Synthesis costs are also decreasing, but more slowly. From 1994 to 2008, sequencing costs decreased more than 10-fold [47]. Twist Bioscience can currently synthesise DNA at \$0.07 per base pair, indicating another 10-fold decrease in synthesis costs since 2008 [48].

In an experiment, DNA was stored under conditions that simulated 10°C storage for a period of 2000 years [16]. The data stored on the DNA was recovered successfully, indicating the longevity of DNA as storage method. It has been shown that DNA can be stored at room temperature for extended time periods if it is isolated from water and oxygen [49]. While cold storage is able to preserve DNA for exceptionally long periods of time, dry storage at room temperature may lead to reduced costs [50]. The 2021 European Inventor Award was awarded to engineers for devising a method to store DNA strands within silica, where they can be protected from decay for millennia [16], [51]. It has also been suggested that DNA be stored in an environment like the Global Crop Diversity Trust's Svalbard Global Seed Vault, which requires no permanent on-site staff [52], [53]. Tape storage, however, has an expected lifetime of 30 years, after which the data should be transferred to new media [54].

2.2.2 Comparison of *in vivo* and *in vitro* approaches

In vivo DNA refers to DNA within living organisms, while *in vitro* refers to non-living DNA, such as synthetic DNA stored in test tubes. Both methods have been investigated for use in DNA data

storage and each one has its own advantages and disadvantages. *In vivo* DNA storage has been tested on bacteria, such as *E. Coli*, although other microorganisms may also be considered. Experiments on *in vivo* storage using *E. Coli* date as far back as 1988, when a picture was stored on the bacteria in the form of a 5×7 matrix [55]. *In vivo* storage has the advantage of replication through biological methods. These methods are orders of magnitude faster than the methods used to copy artificial DNA strands [56]. Spontaneous mutation may occur within living cells. The rate of spontaneous mutations in *E. coli* was found to be 1×10^{-3} per genome per generation [57]. While this rate is relatively small, the generation time of the bacteria is only 20 to 30 minutes. Biosafety should also be considered, as there is not enough evidence to indicate if inserted DNA can affect the gene expression of host cells [58]. Therefore, care should be taken to prevent any potential biological impacts that may be caused by inserting artificial DNA fragments into biological cells. Consequently, after several years of storage, such mutations may result in significant error rates in the stored data. Furthermore, *in vivo* storage has a limited data density, as each living cell can only store a limited amount of data [59]. These two factors negate the main advantages of DNA data storage, namely longevity and data density. Therefore, *in vitro* DNA storage is considered as a more suitable alternative for high-density data storage than *in vivo* DNA.

In vitro storage has been explored to a much greater extent than *in vivo* methods. Artificial DNA strings are created and stored in pools of oligos. Current technology can only reliably synthesise oligos of 100 to 200 nucleotides [12]. The data is split among the oligos, and each oligo has to be identified in order to reconstruct the data. Therefore, an address is encoded on each oligo in order to identify it [7]. Longer addresses are required for larger oligo pools, leading to a reduced data density. A possible solution is to use multiple smaller oligo pools, although this may make certain operations, such as amplification, more cumbersome. Furthermore, improvements in technology may yield longer oligos and reduce the number of oligos required to store the same amount of data, thereby reducing the required address length. Polymerase chain reaction (PCR) is used to amplify the DNA strands, i.e. creating copies of each strand [60]. This process is much slower than natural replication.

The rate of decay of the DNA is characterised by its half-life. At room temperature, the half-life of DNA is relatively short. Research indicates that strings of 100 base pairs have a half-life of 150 years when preserved in bone at 25°C [61]. When stored under protected conditions, such as very low temperatures, the DNA decays much slower. Scientists have successfully recovered the genome of a horse from bone that was in permafrost for 560 to 780 thousand years [62]. Grass et al. [16] stored 83 kB of data encapsulated in silica and successfully retrieved the data after simulating storage at room temperature for 2000 years. This indicates the longevity of DNA when protected from air and water. Since *in vitro* techniques do not introduce artificial DNA to living

organisms, biosafety and ethical implications are reduced. *In vitro* DNA storage therefore addresses the disadvantages of *in vivo* storage, including longevity, data density and biosafety if the correct measures are taken. This study focusses on *in vitro* DNA storage techniques.

2.2.3 DNA data storage process

The *in vitro* DNA data storage process consists of various steps [7], as shown in Figure 2-1. Firstly, data is encoded onto a four-letter alphabet in order to match the four DNA bases. During this stage, codewords should be selected to implement error correction with the aid of logical redundancy [58]. Furthermore, coding should also be implemented in such a manner that codewords fulfil any constraints set by channel characteristics. Error correction coding and constrained coding can be implemented in successive steps or combined into a single step with codes that have combined properties [63]. Many coding methods are based on principles similar to traditional channel coding methods. Error correction coding and channel constraints are explored in more detail in Sections 2.2.9 - 2.2.11.

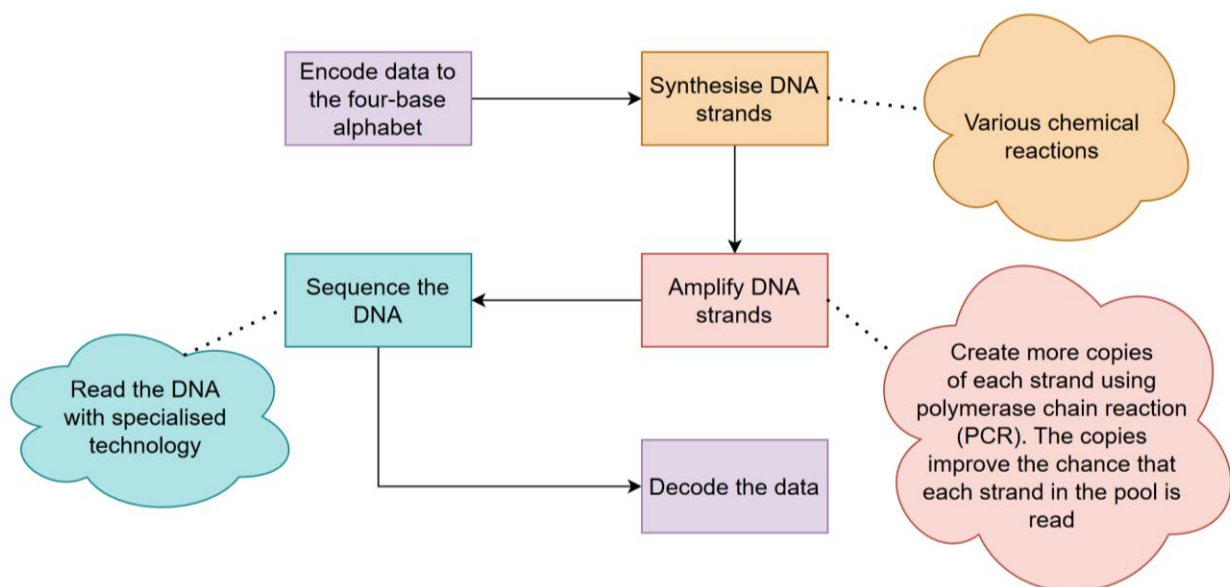


Figure 2-1: Simplified representation of the DNA data storage process

The maximum length of the codewords should match the length of the oligos to be generated. Addresses should also be added to the codewords in order to identify individual oligos. Thereafter, the DNA strings are synthesised in a lab using chemical reactions or other methods. The DNA strings are then transferred to a storage medium such as silica. When the DNA is to be read, PCR is applied to amplify the oligos in the pool. A sample of the oligo pool is taken and sequenced with specialised machinery in order to read the data. The data can then be decoded. Each of these steps is described in greater detail in the following sections.

2.2.4 Synthesis methods

More research has been conducted on DNA sequencing than synthesis, therefore synthesis methods are one of the main limitations regarding the DNA data storage process. Nonetheless, various *in vitro* DNA synthesis methods exist with different advantages.

2.2.4.1 Column-based oligo synthesis

Column-based oligo synthesis is a long-standing DNA synthesis technique that can be implemented via a variety of approaches, each using different chemicals. These approaches include phosphodiester, H-phosphonate, phosphotriester and phosphoramidite, which is the most modern [64]–[67]. Phosphoramidite synthesis makes use of a solid support to which the oligo chains are attached. A nucleoside bound with dimethoxytrityl (DMT) is attached to this solid support. The DMT prevents other nucleotides from attaching to the nucleoside. DMT can be removed by using trichloroacetic acid. After removing the DMT, a new DMT-protected nucleoside is coupled to the 5' group of the oligo, thereby growing the oligo chain. A capping step binds acetyl to any unreacted 5' ends, rendering them inert in order to reduce the occurrence of deletion errors. A phosphite triester is formed during this step, to which an iodine oxidation is added in order to create a structural support in the form of a phosphate backbone for the oligo. These steps are known as detritylation, coupling, capping and oxidation. They are repeated to form a cycle, allowing the entire DNA string to be synthesised by adding each nucleotide individually. After the desired string has been synthesised, the oligos can be removed from the solid support. In 2016, this method was capable of producing oligos of a length of 100 nucleotides at a cost of less than \$0.15 per nucleotide [12]. The length of oligos that this method can synthesise is limited by the yield of each turn of the cycle, also known as the coupling efficiency. If oligos of length 101 are desired, a 99% yield at each cycle will only result in ~36% of the molecules synthesised completely. It should also be noted that various errors may occur within each of these completed molecules, rendering the number of usable molecules even fewer. The formula for the final yield is given by [68]:

$$\text{Final yield} = \text{coupling efficiency}^{\text{desired string length}-1} = 99\%^{101-1} = 36.6\% \quad (2-2)$$

Column-based oligo synthesis using phosphoramidite contains mostly single-base deletion errors, i.e. a base is not added to the output string during synthesis. This is mainly due to lower efficiency in the coupling and capping processes, or failure to remove the DMT at each step in the cycle. In 2016, this method was stated to achieve an error rate of 0.5% or better [12].

2.2.4.2 Array-based oligo synthesis

Array-based oligo synthesis is another collection of synthesis methods. Array-based synthesis uses similar steps to column-based synthesis, but key mechanisms differ in order to make the processes much more efficient. Photolabile phosphoramidites are deprotected selectively using photolithography in microarray synthesis. Modern techniques use programmable micromirror devices to direct light onto specific points of the array [69], [70]. Another modern technique uses ink-jet printing to print individual nucleotides on the array surface, removing the requirement for photolabile molecules [71]. Furthermore, semiconductor-based electrochemical acid production can be used to selectively deprotect nucleosides [72]. In 2016, oligos could be produced by microarrays for less than \$0.001 per nucleotide [12]

Array-based synthesis can produce hundreds of thousands of oligos, but only small amounts of each oligo are produced [7]. Furthermore, when the oligos are removed from the array, all of the oligos are mixed into a single pool. This should be taken into account when synthesising DNA for data storage purposes using this method.

Multiple short oligos can be added together in order to construct a longer oligo. Ligase chain reaction is a method commonly used to this end. A ligase is an enzyme that catalyses the process of chemically bonding two molecules together. This method does not place a constraint on the length of DNA strings that can be synthesised [73]. PCR is required to amplify the DNA after ligation, therefore the length of the synthesised strings is usually kept below 5000 bases to increase reliability [12]. This method is expensive and is not regularly used in DNA data storage experiments.

While being a cheaper alternative than column-based synthesis, array-based synthesis suffers from higher error rates than column-based methods [7], [12].

2.2.5 Polymerase chain reaction

PCR uses DNA polymerase, an enzyme that catalyses the synthesis of a new DNA strand alongside a template strand, to amplify DNA strands [74]. This is an important step in the DNA data storage process, as large oligo pools may only contain a few copies of each type of oligo. Additional copies are required to sequence reliably and to store for future use.

The PCR process involves three main steps [74]. PCR works on single-stranded DNA molecules, therefore, when double-stranded molecules are considered, the DNA is first heated to separate the strands. This step is not required when working with an oligo library. Thereafter, short molecules called primers are added to each end of the template DNA strands. The primers should match the ends of the template strands, therefore codewords for DNA data storage should be designed with this in mind. Lastly, the DNA polymerase is used to extend the 3' end of each

primer. Keeping base pairing rules in mind, this results in a complementary copy of the original DNA strand. Since each single-stranded DNA molecule is converted into two strands, the number of DNA molecules is doubled. This process can be repeated multiple times to create millions of copies of each original molecule.

There are various DNA polymerases that can be used in PCR. Modern polymerases render PCR a high-fidelity process. Polymerases with high specificity allow the process to target specific oligos by only extending molecules with the correct primers. Polymerase with a high thermostability are also desirable, as heat is used to separate the DNA strands. Lastly, modern DNA polymerases possess high processivity, allowing the PCR process to occur at a higher rate [75].

Thermocyclers are regularly used to automate the PCR process [74]. These devices automate temperature cycling in order to separate strands and form new strands.

PCR has been shown to be a high fidelity process, with error rates in the order of 10^{-6} [76].

2.2.6 Sequencing technologies

In order to read DNA strings, various sequencing techniques with different advantages and disadvantages have been developed. These techniques are examined in the following subsections.

2.2.6.1 Sanger sequencing

Sanger sequencing is a first generation sequencing technique that employs the sequencing by synthesis method [77]. Although it is not used regularly anymore, it has formed the basis for many modern sequencing techniques. Sanger sequencing works on single-stranded DNA. The strand is connected to deoxynucleotide triphosphates (dNTPs), which provide the required nucleotides to create a double-stranded structure. Dideoxynucleotide triphosphates (ddNTPs) are also added. The sequence will extend as dNTPs attach until a ddNTP is added. The ddNTP terminates the sequence, as it has no 3' end and is therefore unable to extend further [77]. ddNTPs for each type of nucleotide are added, namely ddATP, ddGTP, ddCTP and ddTTP. Base pairing rules ensure that the correct type of ddNTP attaches at the correct position. Each ddNTP includes a fluorescent marker that indicates the type of base. A laser may then be used to read the sequence by detecting the fluorescent intensity [78].

Sanger sequencing remains in use in scenarios where low numbers of target strings are to be sequenced [79]. Therefore, Sanger sequencing is not a viable option for DNA data storage channels, as large numbers of strings are required to store large amounts of data.

2.2.6.2 Next Generation Sequencing

Next Generation Sequencing (NGS) technologies often employ sequencing by synthesis methods similar to those used in Sanger sequencing. NGS methods usually rely on shorter reads than Sanger sequencing [80]. These methods generally have higher error rates than Sanger sequencing, but rely on high coverage to minimise errors. Coverage refers to reading multiple copies of the same DNA sequence. An average of the reads can then be used to determine the most probable string and reduce errors that only occur in single reads. The main advantage of NGS methods when compared to Sanger sequencing is their ability to sequence large numbers of strings simultaneously [79].

Illumina is one of the best-known companies that provide NGS platforms. Illumina has introduced multiple different platforms, such as HiSeq X Ten and MiniSeq. It has been found that error rates vary between platforms [81]. HiSeq X Ten has shown a median error rate of 0.087%, while MiniSeq has a median error rate of 0.613 %. However, it should be noted that HiSeq X Ten is a very expensive platform, and in 2014 the cost of the system was stated to be \$10 million [82]. Additionally, the consumables required to sequence a human genome have a cost of \$1000. The MiniSeq is a much cheaper alternative, and the platform can be leased from Illumina for \$1600 per month as of 2023 [83].

2.2.6.3 Nanopore sequencing

Nanopore sequencing is a modern sequencing technique often referred to as third generation sequencing [84]. Oxford Nanopore Technologies (ONT) has created the MinION device, which is one of the most well-known nanopore sequencers [85]. A nanopore refers to a nanoscale protein hole that is set in an electrically resistant polymer membrane [86]. Nanopores are housed in a flow cell. An example of a naturally occurring nanopore is α -hemolysin, which has an inner diameter of 1 nm. This diameter is on the same scale as DNA and can be manufactured in laboratories, therefore it is suitable for use in nanopore sequencing technology [87]. Other proteins may also be used, and studies [88], [89] have explored solid state nanopores, created from synthetic materials, for use in future technology as well.

By applying a constant voltage, an ionic current is produced in the nanopore, which in turn drives negatively charged single-stranded DNA molecules through the pore. As the DNA strand moves through the nanopore, it induces changes in the ionic current. These changes correspond to the different nucleotides within the DNA strand. Computational algorithms known as base callers are used to decode these changes and identify which bases are moving through the nanopore, thereby sequencing the strand in real time [90]. Base callers are described in Section 2.2.7.

An advantage of nanopore technology is its compact size, as the MinION device is a handheld package. Furthermore, the device can read long DNA strands in excess of 4 million bases. The device is also faster than other sequencing methods, capable of reaching speeds of up to 420 bases per second. The flow cells of the MinION device can sequence up to 50 gigabases before requiring replacement [91]. As of 2023, a MinION starter kit, containing a MinION device and 6 flow cells, costs \$4900 [92]. This renders nanopore sequencing technology highly accessible when compared to other sequencing methods. ONT also supplies various desktop solutions, capable of larger throughputs than the MinION device. These desktop nanopore sequencers may prove favourable for DNA data storage systems due to their cost-effectiveness.

ONT continuously makes new advancements and brings improvements to its technology. As of 2023, the error rate of ONT devices is stated to be between 3.2% and 7.9%, depending on the platform used [93].

2.2.6.4 FASTQ

The output data of the sequencing process is generally stored in FASTQ files [94]. This file format allows the storage of a sequence of bases, along with the quality score of each base. The quality score Q_{PHRED} gives an indication of the certainty of each base and is defined in terms of the error probability P_e :

$$Q_{PHRED} = -10 \times \log_{10} P_e \quad (2-3)$$

2.2.7 Base callers

Nanopore sequencing platforms emit currents that represent the different DNA bases read by the sequencer. Algorithms known as base callers decode these current values to show which bases are sequenced. These bases can then be stored in FASTQ files.

Guppy is the production base caller used by MinION devices [95]. Guppy makes use of a bi-directional recurrent neural network. This is a form of machine learning described in [96]. Bonito is an open source base caller for ONT platforms that was released for research purposes [97]. MinION is continuously improving its base caller algorithms and multiple versions of Guppy and Bonito have been released.

Different base callers may read bases with different amounts of accuracy. As of 2020, Guppy 4.0 boasted a median read accuracy of around 96%, while Bonito 0.3 CRF achieved a median accuracy of close to 98% [95].

Albacore is another popular base caller developed by ONT. As of 2019, Albacore boasts a similar accuracy to Guppy [98]. The main difference is that Albacore is CPU (central processing unit)

based, while Guppy can utilise a GPU (graphical processing unit) as well, thereby providing increased computational speed. Albacore has since been discontinued by ONT.

2.2.8 Trace reconstruction

As stated in Section 2.2.6, each DNA string within a pool of oligos may be sequenced multiple times. This results in multiple copies of the same string, each with different errors.

Algorithms are required to cluster multiple reads of the same sequence together. Clustering algorithms are regularly based on edit distance [99]. When comparing two strings, edit distance refers to the number of insertions, deletions and substitutions required to change the first string to the second string. String comparison algorithms, including algorithms for computing the edit distance, are examined in Section 2.7.

Additional algorithms are required to reach a consensus on what is the correct string by combining multiple noisy reads. This is referred to as trace reconstruction or consensus. For this purpose, multiple different algorithms have been proposed [32], [100]–[103].

2.2.9 Errors in DNA data storage

The three basic error types within DNA data storage systems are substitutions, deletions and insertions [5]. Substitution errors may occur when an erroneous base is read instead of the correct base, for example, a C can be read as a T. Deletion errors may occur when one or more nucleotides are deleted from the string, either when they are not written during synthesis or not read during sequencing. Lastly, insertion errors may occur when nucleotides are added to the string in erroneous locations during the synthesis process, or read in an erroneous location during sequencing.

Swap errors are common in some types of molecular communication channels [104]. Swap errors occur when two consecutive symbols are read in the incorrect order, for example ‘AGCT’ is read as ‘ACGT’, i.e. the G and the C are swapped. These errors are, however, not often reported in literature regarding DNA data storage.

2.2.10 Dependency of errors on sequence characteristics

Errors in DNA sequences are often assumed to be independent and identically distributed, regardless of the characteristics of the string itself [105]. In this section, various aspects that may affect the distribution of errors are examined.

2.2.10.1 Homopolymer runs

When examining both Nanopore and NGS data, a strong correlation has been shown between homopolymer run lengths and an increase in error rates at the homopolymer run [106], [107]. A

homopolymer run is multiple consecutive repetitions of the same base in the DNA string. It was seen that homopolymers of C or G induce a larger increase in error rate than homopolymers of A or T in Nanopore sequencing data. A study has determined that homopolymer runs mainly lead to an increase in insertion and deletion errors in NGS sequencing, while the substitution rate remains similar to the rest of the string [9].

2.2.10.2 Position within DNA string

A study on NGS data indicated that error rates may be dependent on the position within the read, with most of the errors occurring in the second half of the read [106]. Another study also confirmed a rise in error rates towards the end of the sequenced string [81].

A major increase in error rate has been seen at the start and end of the sequenced string when examining Nanopore sequencing data [108]. It was found that 90% of errors originate at the first three and final positions of the DNA string. A similar study on NGS data also found major increases in error rates at the start and the end of the string [109].

2.2.10.3 GC content

Multiple studies have shown that strings with very high or very low GC contents in DNA strings induce reduced accuracy in NGS platforms [110], [111]. This is due to PCR bias, where strings with high or low GC contents are amplified less, leading to reduced coverage and worse results after trace reconstruction.

A study that divided sequenced strings into substrings of length 100 found that substrings with extremely high or low GC contents showed increased error rates [9]. Insertion and deletion rates rose by up to two orders of magnitude, while substitution rates were affected to a lesser degree.

A similar study on Nanopore sequencing data also showed an increase in error rate as GC content increases when inspecting substrings of length 100 [107].

In binary channel mediums, running digital sum (RDS) constraints are regularly employed [112]. The RDS is used to measure the total number of 1's or 0's up to the current point in the transmission. At the start of the transmission, the RDS is 0. As bits are transmitted, each 1 received increases the RDS by 1, and each 0 decreases the RDS by 1. The transmission may be broken up into segments, and the RDS of each individual segment may be calculated. A similar method has been proposed for DNA data storage [14]. As the total GC content may influence the error rate, G's and C's increase the RDS, similar to 1's in a binary channel. A's and T's play a similar role to 0's in a binary channel.

2.2.10.4 Problematic substrings

The term problematic substrings, or sequence-specific-errors, denote any substrings that lead to an increase in error rates when compared to the average error rate of the complete DNA string. This may be substrings with a high GC content, or substrings containing homopolymers.

Additionally, short repeated regions have been found to show increased error rates in a study done on Nanopore sequencing data [107]. Short repeated regions refer to any short substring, consisting only of a few bases, that repeats itself. For example, CCACCA. The substring CCA is repeated twice in this short repeated region.

2.2.10.5 Error bursts and runs

Burst errors have been shown to be common in DNA data storage systems [7]. High throughput synthesis methods have been shown to produce bursts of substitution or deletion errors. An error burst is defined as an area where errors occur at a high density [113].

Additionally, error runs – multiple consecutive errors – have been shown to be common. A study has shown that runs of deletions introduced during synthesis may account for over 10% of all deletion errors [114].

2.2.11 Encoding

Due to the errors that occur within the DNA data storage process, error correction codes are regularly applied. Various different error correction encoding methods have been proposed for DNA data storage systems. An encoding scheme making use of a fountain code has been proposed [33]. Reed-Solomon codes have also been employed in multiple studies [7], [11], [16].

Additionally, to reduce the occurrence of errors due to variations in GC content or the occurrence of homopolymer runs, constrained codes are employed. Constrained codes aim to create codewords that enforce certain constraints, such as a limited number of consecutive repetitions of the same base within the code word. Various studies have proposed constrained coding methods that enforce a balanced GC content [7], [18]. Both of these codes also prevent any homopolymer runs of more than 2 bases.

Another proposed encoding scheme prevented homopolymers of any kind by using a ternary alphabet instead of a quaternary alphabet [15]. By excluding the previous base at each point in the string, the next base is selected by converting the ternary value to one of the three remaining bases.

Multiple studies have proposed encoding schemes that employ both constrained coding and error correction coding [7], [14], [15], [18]. Certain studies were able to test their encoding schemes on

experimental data [14], [15]. Due to the high costs of such experiments, other studies were limited to the use of channel models to test their encoding schemes [21], [101]. Lastly, there are studies that do not validate their encoding schemes through testing, but propose theoretical proofs and examinations [7], [18].

2.3 Types of channel models

There are various methods for creating channel models. This section shall focus on the IID model due to its common use for modelling DNA data storage channels. Furthermore, the Markov model shall be examined due to its efficiency in modelling bursts, as bursts of errors are a common occurrence in DNA data storage channels, as discussed in Section 2.2.10.5. State-of-the-art models designed and implemented specifically for DNA data storage channels shall be discussed in Section 2.5.

2.3.1 IID models

The independent and identically distributed model (IID) assumes that a set of random variables is mutually independent. Furthermore, the variables are drawn from the same probability distribution [115].

An example of an IID model for the DNA data storage channel requires three probability distributions. Firstly, the probability for each type of error – insertion, deletion and substitution – is taken. Thereafter, if an insertion is generated, the probability for each type of base to be inserted is taken. Otherwise, if a substitution is generated, the probability for each other type of base to replace the original base is taken. The three required probabilities can be calculated from a training set. Errors can then be inserted into DNA strings according to these probability distributions in an IID manner.

IID models have been used in literature describing error correction codes as a means to test the codes for DNA data storage channels without the requirement of physical experiments [21], [101]. Additionally, several proposed models for the DNA data storage channel in literature assume that errors are IID [22], [116], [117]. Another model simulates the output current produced by Nanopore devices by adding noise in an IID manner [118].

2.3.2 Markov models

The Markov model is a method for creating a discrete channel model [119]. The model requires a training sequence $X = [x_1, x_2, \dots]$. The model is able to produce a sequence of output symbols

$Y = [y_1, y_2, \dots]$. The Markov model is an abstract model that can be represented by a small set of parameters. The derivation of these parameters is discussed in Section 2.4.

The Markov model consists of a number of states and a number of possible outcomes. The number of states is selected when deriving the model, while the number of possible outcomes is determined by the phenomena that are modelled. The model is represented by two matrices, the transition matrix and the emission matrix. The notation described in [119] is adopted in this dissertation and is summarised below.

The transition matrix consists of the probability of transitioning to each other state, given the current state [119]. The entry in the i^{th} row and j^{th} column of the transition matrix A is represented by a_{ij} as given in equation (2-4). a_{ij} is the conditional probability to transition from state i to state j . S_{t+1} and S_t represent the next state and the current state respectively.

$$a_{ij} = P(S_{t+1} = j | S_t = i) \quad (2-4)$$

In each state, there is a probability distribution that determines the probability for each possible outcome. Examples of outcomes are a successful transition or the occurrence of an error. In this example, there are two possible outcomes, but models may be created with more different outcomes. The emission matrix B represents the probability of each outcome, given the current state [119]. The entry in the i^{th} row and k^{th} column of the emission matrix is given by b_{ik} . e_k represents the k^{th} outcome, while S_t represents the current state:

$$b_{ik} = P(e_k | S_t = i) \quad (2-5)$$

The Markov model is defined by the transition and emission matrices. At each instance, given the current state, an outcome is selected according to the probability distribution from the emission matrix. The next state is selected according to the probability distribution from the transition matrix. The model is commonly called a hidden Markov model (HMM), as the states are hidden from the user while the outputs are generated.

Markov models are normally assumed to be stationary, indicating that the values of the transition and emission matrices do not vary with t . However, the probability to start in each state may be calculated.

A Markov model has memory, i.e. the occurrence of errors at a previous point in time may affect the occurrence of errors at the current time. This allows the model to effectively represent error bursts and runs. The memory of the Markov model is contained in the state of the model, as the current state of the model depends on the previous states. For a first order Markov model, the probability of the current state only depends on the previous state. For an m^{th} order Markov model, the probability of the current state depends on the previous m states.

In DNA data storage, Markov models have been considered for modelling the GC content of DNA strings [120]. In addition, models described as Markov models have been used to model the occurrence of errors within DNA strings [24].

2.3.2.1 Fritchman models

A Fritchman model is a special case of the Markov model, where each state has an error probability of either 1 or 0 [121]. Once again, the notation described in [119] is adopted in this dissertation and summarised below. A model consisting of N states is divided into two groups. Group A contains states $S_1, S_2, S_3, \dots, S_k$. Group B contains states $S_{k+1}, S_{k+2}, S_{k+3}, \dots, S_N$. States from group A are denoted as good states. The error probabilities for these states are all 0. States from group B are denoted as bad states. The error probabilities for these states are all 1. As an example, a Fritchman model with four bad states and two good states may have the following emission matrix:

$$B = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \quad (2-6)$$

In a Fritchman model, there are no transitions between different good states. Furthermore, there are no transitions between different bad states. Transitions only occur between a good state and a bad state, or between a state and itself. An example transmission matrix for the model with four bad states and two good states is given:

$$A = \begin{bmatrix} a_{11} & 0 & 0 & 0 & a_{15} & a_{16} \\ 0 & a_{22} & 0 & 0 & a_{25} & a_{26} \\ 0 & 0 & a_{33} & 0 & a_{35} & a_{36} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & 0 \\ a_{61} & a_{62} & a_{63} & a_{64} & 0 & a_{66} \end{bmatrix} \quad (2-7)$$

When computing the values for the transition and emission matrix, namely the transition probabilities and error probabilities within each state, the values for the emission matrix are set as initial values and remain constant through the computation process. The transition matrix is determined in the same manner as a Markov model, and this process is described in Section 2.4.

2.3.2.2 Gilbert-Elliot models

A Gilbert-Elliot model is a special case of the Markov model and Fritchman model that contains two states [26], [119]. The first state is an error-free state where the probability of errors occurring

is 0. The second state has a probability p for errors to occur. This allows the model to represent error bursts, as bursts of errors may contain accurate transmissions scattered between the errors. The transition and emission matrices are determined in the same manner as a Markov model, and this process is described in Section 2.4.

2.4 Estimating Markov model parameters

To define a Markov model, as described in Section 2.3.2, the transmission and emission matrices of the model are required. The Baum-Welch algorithm [122] and the Viterbi training method [123] are two popular methods for estimating the model parameters. These methods are implemented in the MATLAB Statistics and Machine Learning Toolbox [124], making them readily available to MATLAB users. Therefore, these two algorithms are discussed in this section.

2.4.1 Baum-Welch algorithm

The Baum-Welch algorithm uses expectation-maximisation to find the parameters for a Markov model. The algorithm was first described in the late 1960's and early 1970's [122]. The notation from [119] is used and the algorithm is summarised as follows:

Let the transition matrix A of the hidden Markov model be defined in the same manner as in Section 2.3.2:

$$a_{ij} = P(S_{t+1} = j | S_t = i) \quad (2-8)$$

Furthermore, let the emission matrix B be defined, according to Section 2.3.2, as:

$$b_{ik} = P(e_k | S_t = i) \quad (2-9)$$

With O_i referring to the i^{th} outcome, let the sequence of outcomes used for training be defined as:

$$O = \{O_1, O_2, \dots, O_T\} \quad (2-10)$$

Furthermore, let the initial state distribution be given by:

$$\Pi = [\pi_1 \quad \pi_2 \quad \dots \quad \pi_N] \quad (2-11)$$

with π_i referring to the probability of starting in the i^{th} state.

The algorithm requires assumed initial values for the transition and emission matrices, represented as $\Gamma(A, B)$.

The forward and backward variables are computed according to the forwards-backwards algorithm, which is described below [125]–[127]. Once again, notation from [119] is used, and the algorithm is summarised below.

The forward variables α are defined as:

$$\alpha_t(i) = P(O_1, O_2, \dots, O_T, s_t = i | \Gamma) \quad (2-12)$$

for $t = 1, 2, \dots, T$ and $i = 1, 2, \dots, N$.

The forward variables are initialised and then computed recursively:

$$\alpha_1(i) = \pi_i b_i(O_1), \quad i = 1, 2, \dots, N \quad (2-13)$$

$$\alpha_{t+1}(j) = [\sum_{i=1}^N \alpha_t(i) a_{ij}] b_j(O_{t+1}), \quad i = 1 \dots N, 1 \leq t \leq T-1, 1 \leq j \leq N \quad (2-14)$$

The backward variables β are defined as:

$$\beta_t = P(O_{t+1}, O_{t+2}, \dots, O_T | s_t = i, \Gamma) \quad (2-15)$$

for $t = 1, 2, \dots, T$.

The backward variables are initialised and then computed recursively:

$$\beta_T(i) = 1, \quad i = 1, 2, \dots, N \quad (2-16)$$

$$\beta_t(i) = \sum_{j=1}^N \beta_{t+1}(j) a_{ji} b_j(O_{t+1}), \quad i = 1 \dots N, 1 \leq t \leq T-1, 1 \leq j \leq N \quad (2-17)$$

Two temporary variables are defined. $\gamma_t(i)$ denotes the probability of being in state i at time t given the observed sequence O . $\xi_t(i, j)$ denotes the probability of being in state i at time t and state j at time $t+1$ given the observed sequence O . These variables are calculated according to the Bayes method, which was originally described in [128]. Using the notation from [119] yields the following equations:

$$\xi_t(i, j) = P(s_t = i, s_{t+1} = j | O, \Gamma) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O | \Gamma)} \quad (2-18)$$

$$\gamma_t(i) = P(s_t = i | O, \Gamma) = \frac{\alpha_t(i) \beta_t(i)}{P(O | \Gamma)}, \quad i = 1, 2, \dots, N \quad (2-19)$$

The updated state transition probabilities can then be found:

$$a_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \quad (2-20)$$

The updated emission probabilities can also be found using:

$$b_{jk} = \frac{\sum_{t=1}^T \gamma_t(j) b_j(O_t)}{\sum_{t=1}^T \gamma_t(j)} \quad (2-21)$$

As the forward and backward variables may tend to zero for large datasets, a scaling factor C_t may be used:

$$C_t = \sum_{i=1}^N \alpha_t(i) \quad (2-22)$$

This results in the following scaled forward and backward variables:

$$\overline{\alpha}_t(j) = \alpha_t(j)/C_t \quad (2-23)$$

$$\overline{\beta}_T(i) = \beta_T(i)/C_t \quad (2-24)$$

The above process is an iterative process and requires some sort of stopping criteria. Convergence is usually determined when the values of A and B no longer change within a given number of significant figures between iterations. Otherwise, the change in the value of $P(O|\Gamma)$ may be examined. If the value differs very little between iterations, convergence has been reached. The value can be expressed as the log-likelihood ratio [119]:

$$\log_{10} P(O|\Gamma) = \sum_{t=1}^T \log_{10} C_t \quad (2-25)$$

Values found for A and B are not unique and may differ if different initial values for A and B are used [119]. The Baum-Welch algorithm always converges to a maximum likelihood solution [113]. Maximum likelihood indicates that, assuming the model, the observed data is the most probable outcome [129].

2.4.2 Viterbi estimation

Another method to estimate the emission and transition matrices of a Markov model makes use of the Viterbi algorithm [130], which is described in Section 2.4.2.1. This estimation method [123] shall be referred to as the Viterbi training process, and is explained in Section 2.4.2.2.

2.4.2.1 Viterbi algorithm

The Viterbi algorithm is a recursive algorithm used for finding the most likely sequence of states produced by a Markov model [130], [131]. The notation from [132] is adapted to match the notation for Markov models described in Section 2.3.2, and is summarised below.

The Viterbi algorithm requires the transition and emission matrices, A and B that define the Markov model as input. These matrices are defined according to Section 2.3.2 as:

$$a_{ij} = P(S_{t+1} = j | S_t = i) \quad (2-26)$$

$$b_{ik} = P(e_k | S_t = i) \quad (2-27)$$

In order to describe the Viterbi algorithm, multiple variables are defined according to [132]. For a Markov model with K number of states, X represents a sequence of states where x_i is an element of the different states of the Markov model $S = \{s_1, s_2, \dots, s_K\}$:

$$X = (x_1, x_2, \dots, x_T) \quad (2-28)$$

Y represents the sequence of observations generated by X . Each element y_n of Y is an element of $O = \{o_1, o_2, \dots, o_N\}$ which represents the different possible observations.

$$Y = (y_1, y_2, \dots, y_T) \quad (2-29)$$

Y is a required input for the algorithm, and X is returned by the algorithm. The algorithm attempts to find the values of X which are most likely to produce Y . $\hat{X} = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_T)$ represents the most likely path to generate Y found thus far.

Tr_1 and Tr_2 are $K \times T$ matrices. Each element of Tr_1 , namely $Tr_1[i, j]$, stores the probability of the most likely path so far, i.e. $\hat{X}$, with $\hat{x}_j = s_i$. Tr_2 represents the previous values of $\hat{X}$ for the most likely path so far. $Tr_2[i, j]$ takes on the value of $\hat{x}_{j-1}$ for the most likely path found thus far for $\hat{X} = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_j)$ with $\hat{x}_j = s_i$. Values for Tr_1 and Tr_2 are calculated recursively [132].

As described in [132] and using the notation from above, the matrices Tr_1 and Tr_2 can be calculated with the following algorithm:

Viterbi algorithm

// Initialization

for $i = 1, 2, 3, \dots, K$

$Tr_1[i, 1] = 1$

$Tr_2[i, 1] = 0$

// Recursion

for $j = 2, 3, \dots, T$

for $i = 1, 2, 3, \dots, K$

$Tr_1[i, j] = b_{iy_j} \cdot \max_k (Tr_1[k, j-1] \cdot a_{ki})$

$Tr_2[i, j] = \arg\max_k (Tr_1[k, j-1] \cdot a_{ki})$

// Termination

$x_T = \arg\max_k (Tr_1[k, T] \cdot a_{k0})$

// Traceback

for $j = T, T-1, T-2, \dots, 2$

$$x_{j-1} = Tr_2[x_j, j]$$

The different values of x make up the sequence X which is returned by the algorithm.

2.4.2.2 Viterbi training

Viterbi training makes use of the Viterbi algorithm to estimate values for the transition matrix A and emission matrix B [123].

Initial values for the emission and transition matrices are defined as A^0 and B^0 by the user. Additionally, a set of observations Y is used for training. The Viterbi algorithm is then used to find a sequence of states X that maximises the likelihood of the observations in Y . For values in X where the sequence is in state l , the probability distribution for emissions in that state is defined as $P(B_l)$. B_l represents the l^{th} row of B , which can then be estimated using maximum likelihood estimation. This is done for each row of B and the resultant emission matrix B^1 replaces B^0 . Similarly, A^1 is estimated using maximum likelihood estimation and replaces A^0 .

This process is repeated by replacing A^m and B^m with A^{m+1} and B^{m+1} until a convergence is reached. It has been shown that convergence is reached in a finite number of steps [133]. Convergence may be determined when consecutive values of A and B no longer differ within a certain number of significant figures.

2.4.3 Comparison of Baum-Welch and Viterbi estimation

Studies have indicated that, while Viterbi training may converge faster than the Baum-Welch algorithm, models trained using the Viterbi algorithm may yield suboptimal results [134], [135]. However, Viterbi training has a much lower computational cost than the Baum-Welch algorithm.

2.5 Existing DNA data storage models

A variety of models have been created with the goal of representing the DNA data storage channel. In Section 2.3.1, it was stated that IID models are commonly used for this purpose. In this section, various state-of-the-art models are described that use different methods to represent the DNA data storage channel.

2.5.1 DNA storage error simulation pipeline

The DNA storage error simulation pipeline (DeSP) [22] is a model for the DNA data storage channel that simulates errors generated at each stage of the process. This model is able to

simulate entire sequences that are lost, for example, due to PCR bias and decay during storage, as well as errors within a specific sequence.

Firstly, the synthesis process is simulated. The number of strings to be generated is specified by the user. Furthermore, a coupling efficiency value is specified, allowing the software to compute the number of strings yielded according to equation (2-2). Thereafter, errors are injected into the generated strings using an IID method.

Decay is then simulated. The ratio of lost sequences is computed according to equations derived from literature [136]. A number of sequences are removed according to this ratio. Additionally, substitutions of G with T can be added to the sequences according to a probability distribution.

PCR is then simulated. Due to the low error rates in PCR, no within-sequence errors are added at this stage. Since the effect of PCR bias is costly to compute, the values are selected according to a normal distribution, which introduces negligible PCR bias.

As only a sample of the entire pool of DNA strings may be sequenced, sampling is simulated next. The percentage of the pool that is sampled is defined by the user as p_s . Each string generated in the previous step then has a p_s chance of being sampled. This allows DeSP to simulate lost sequences and a change in the distribution of the sequences within the sample set when compared to the entire pool.

Lastly, sequencing is simulated. Errors are added to strings in the sampled set using an IID method.

For each stage of the system, error rates may be user-defined if the user wants to simulate a specific technology, and has knowledge of the error rates produced by that technology. DeSP also allows users to use the default values set in the simulation software.

2.5.2 Skewed DNASimulator model

A model was created in order to simulate the occurrence of errors within the DNA data storage channel. The model considers error probabilities in the same manner as an IID model, but also adds certain skews [108]. In order to create the model, the Clustered Nanopore Reads dataset was examined [32]. The probability for each type of error, given the current base, is found and incorporated into the model. Furthermore, the probabilities for deletion runs of different lengths are also incorporated. Errors are distributed independently and identically according to these probability distributions. Additionally, two skews are added.

It was found that there was a large increase in error rate at the first three positions of the DNA string, as well as at the final position. Therefore, a skew was added to the IID model to represent these increased error rates. Furthermore, it was found that certain types of errors dominate certain

positions. For example, it was found that insertions of G are more probable at the end of the string than throughout the rest of the string, while deletions of C are more probable at the start of the string than throughout the rest of the string. Therefore, a second skew was added to represent the 10 most probable types of errors.

This model was tested in a low coverage system, where each strand was read 5 times. A trace reconstruction algorithm was applied to the generated strings, and the percentage accuracy achieved for single bases as well as complete strings were computed. These values were compared to the accuracies achieved by the trace reconstruction algorithm on the dataset. The analysis was done using two trace reconstruction algorithms, namely BMA (Bitwise Majority Alignment) look-ahead as presented in [102] and iterative reconstruction, which is presented in [103]. The values are given in Table 2-1, taken from [108].

Table 2-1: Percentage of characters and strings accurately reconstructed by trace reconstruction algorithms when comparing the skewed DNASimulator model to the Clustered Nanopore Reads dataset [108]

Data	BMA		Iterative	
	Base	String	Base	String
Clustered Nanopore Reads dataset	87.74%	29.04%	90.32%	66.70%
Skewed DNASimulator model	88.67%	44.78%	77.39%	33.87%

In the data, it is seen that the percentage accuracy achieved by the BMA algorithm on the dataset as well as the model outputs closely correspond when comparing single bases. When using the BMA algorithm on entire strings, as well as single bases and strings when using the iterative algorithm, the accuracy achieved on the dataset and model outputs do not correspond as closely. This indicates that an improved model may be required.

2.5.3 DeepSimulator

DeepSimulator is a simulator for Nanopore sequencing that makes use of deep learning [30]. DeepSimulator 1.5 is an updated and improved version of the original DeepSimulator [137]. Since DeepSimulator 1.5 shows various improvements over the original version, the focus of this section shall be DeepSimulator 1.5.

DeepSimulator makes use of a bi-directional long short-term memory (Bi-LSTM) [138] model which is extended via deep canonical time warping (DCTW) [139]. This leads to a context-dependent model. Due to the high computational time of this method, a context-independent

model was also implemented to reduce the computational complexity, albeit with a compromise in accuracy. The simulator allows the user to choose between the two implementations.

The dataset used to train the model is a Nanopore sequencing dataset that contains the sequencing data of a strain of *Pandoraea pnomenusa*. Since this is the sequencing data of the genome of a specific species, DeepSimulator was not trained to encompass the entire DNA data storage channel, but rather only the sequencing step.

The workflow of DeepSimulator is now described. Input strings are given by the user, as well as the coverage (number of times each input string is read). DeepSimulator acts by simulating the output current generated by Nanopore devices during the sequencing process. A base caller is then applied to this output current to determine the output string of bases. DeepSimulator supports the use of Guppy and Albacore. DeepSimulator, therefore, does not simulate errors by directly introducing insertions, deletions or substitutions. The simulator rather simulates erroneous currents generated by the Nanopore device, which are then converted to erroneous bases by the base caller.

Research has shown that the original DeepSimulator regularly produces the same type of error in the same position in different DNA strings, which is not characteristic of experimental Nanopore data [24]. However, DeepSimulator 1.5 implements a low-pass filter to more closely mimic real Nanopore data, as without the filter the model generates square wave signals that are not representative of real Nanopore sequencing signals. By filtering out the high-frequency components, the model much more closely represents real-world Nanopore signals.

2.5.4 Badread

Badread is a model designed specifically to simulate long DNA reads. Badread was designed to prioritise user control over model accuracy, as the model allows users to set the desired error rate [140].

A k mer is described as a group of k successive bases. The Badread model makes use of k mers of length 7. For each k mer, the probability of that k mer to be correctly sequenced is found. Furthermore, for each k mer, the probability of each possible erroneous read of that k mer to occur is found.

For example, the k mer ACAGTTG has a 67.0% probability of being sequence error free. That same sequence has a 12.7% probability to be sequenced as ACGGTTG, and a 1.4% probability to be sequenced as ACCAGTTG. The probability is calculated for each possible read of that same sequence. The probability of an accurate read and the 25 erroneous sequences with the highest likelihood are stored and used by the model. This process is repeated for each possible k mer of length 7.

To generate a read when presented with an input sequence, the Badread model chooses a random k mer of length 7 from the input sequence. The probability distribution calculated for that k mer is used to select a k mer to replace the original k mer. As there is a chance for the k mer to be sequenced error free, the k mer may be replaced by the same bases. Alternatively, an erroneous sequence may be selected to replace the k mer.

This process is repeated until enough errors are produced within the sequence that the percentage of correct bases that remain in the string corresponds to the percentage of correct bases as defined by the user. The percentage of correct bases is defined according to three values: the mean, maximum and standard deviation. The appropriate percentage of correct bases for the current string is selected according to a beta distribution. Alternatively, the percentage of correct bases can be defined according to the maximum and mean Q_{PHRED} values. By defining the percentage of correct bases, the user is able to choose a desired error rate to simulate.

Three versions of the model were trained on Nanopore reads datasets from 2018, 2020 and 2023 respectively. Additionally, a model was trained on the PacBio dataset. A random model, with an equal chance for insertions, deletions and substitutions is also available. The user may select a model to use. Users may also specify a file path to a different model if the user has trained their own model. This allows the workflow to use a user-defined model if the user has trained a model for a specific dataset.

2.5.5 Channel model with memory

A channel model for the DNA data storage channel using Nanopore sequencing was implemented [24], [105]. The model was created to contain a memory of previous error events, and shall therefore be referred to as the Memory model.

A $kmer_t$ is described as a group of k successive bases, where x_t represents the base at position t of the considered DNA string:

$$kmer_t = (x_{t-k+1}, x_{t-k+2}, \dots, x_t) \quad (2-30)$$

$$kmer_{t+1} = (x_{t-k+2}, x_{t-k+3}, \dots, x_{t+1}) \quad (2-31)$$

As $kmer_t$ moves through a nanopore, the device produces an electrical current c_t . As $kmer_{t+1}$ moves through, an electrical current c_{t+1} is produced. This sequence of current values is then read as bases by the base caller.

The Memory model considers events that can occur at each position of the DNA string. The event at position t is represented by the value e_t . A transmission event indicates that the base at position t of the input DNA string has been transmitted successfully. A deletion event indicates that the base at position t of the input DNA string has been deleted. A substitution event indicates that

the base at position t of the input DNA string has been replaced with a different base. An insertion event indicates that one or more bases have been inserted at position t of the input DNA string. When an insertion occurs, an additional variable L_t is used to indicate the number of bases that were inserted.

The Memory model proposes that event e_t depends on $kmer_t$. The value of k can be adjusted by the user. The model is described as a k^{th} order Markov model [24]. The model also considered the previous event, as event e_{t-1} may influence event e_t . This allows the model to have memory and thereby attempt to represent error bursts.

The definition of the model is now summarised using the description and notation from [105]. The following probabilities define the model:

$$P(e_t | kmer_t, e_{t-1}) \quad (2-32)$$

$$P(L_t | kmer_t, e_t = \text{Insertion}) \quad (2-33)$$

In order to determine the new base in the case of a substitution, the following probability is used:

$$P(B_t | kmer_t, e_t = \text{Substitution}) \quad (2-34)$$

In order to simulate channels with different error rates, for example channels that use different sequencing techniques, a scaling variable α is introduced. By using Monte Carlo simulations, the value of α is determined for channels with different error rates. This results in the following probabilities:

$$\tilde{P}^\alpha(e_t = D | kmer_t, e_{t-1}) = \alpha P(e_t = D | kmer_t, e_{t-1}) \quad (2-35)$$

$$\text{for } D \in \{\text{Insertion}, \text{Deletion}, \text{Substitution}\}$$

$$\tilde{P}^\alpha(e_t = D | kmer_t, e_{t-1}) = (1 - \alpha) P(e_t = D | kmer_t, e_{t-1}) \quad (2-36)$$

$$\text{for } D = \text{Transition}$$

For $t \in [2, 3, \dots, k]$, no complete $kmer$ is observed, therefore additional probability distributions are used for these positions:

$$P(e_t | x_t) \text{ for } t \in [2, 3, \dots, k] \quad (2-37)$$

$$P(L_t | x_t, e_t = \text{Insertion}) \text{ for } t \in [2, 3, \dots, k] \quad (2-38)$$

$$P(B_t | x_t, e_t = \text{Substitution}) \text{ for } t \in [2, 3, \dots, k] \quad (2-39)$$

Additionally, increases in errors are observed at the start and end of the DNA strings. The occurrence of insertions and deletions are observed to be independent of the corresponding base of the input sequence [24], thus this base is not considered. However, the base has to be

considered when choosing a replacement base if a substitution occurs. Therefore, the following probability distributions are considered for the first and last positions of the string of length N :

$$P(e_t) \text{ for } t \in \{1, N\} \quad (2-40)$$

$$P(L_t | e_t = \text{Insertion}) \text{ for } t \in \{1, N\} \quad (2-41)$$

$$P(B_t | x_t, e_t = \text{Substitution}) \text{ for } t \in \{1, N\} \quad (2-42)$$

The entire model is therefore described with probability distributions (2-33) to (2-42).

The dataset used to train the model consists of DNA strings synthesised by Thermofisher. The strings were sequenced with a MinION sequencer using the Guppy base caller. Any probability distributions that were not observed in the dataset were determined by averaging over the other, known, probability distributions.

To use the model, an input sequence is supplied by the user. Additionally, the number of reads is supplied, as well as the length of the $kmer$ to consider. As MinION sequencers use $kmers$ of length 6, the default value of k is 6 [105]. The value of α cannot be set by the user.

In [105], this model is compared to DeepSimulator [137], the Badread model [140] and an IID model created from the same dataset. The models are compared in terms of Kulback-Leibler (KL) divergence [105]. KL divergence is a similarity metric that evaluates the similarity between two probability distributions. In [141], the KL divergence D_{KL} is defined as follows:

$$D_{KL}(P||Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx \quad (2-43)$$

P and Q represent the two probability distributions that are compared. x represents the different outcomes in set X , and the integral is evaluated over all of the outcomes in X .

The KL divergence was averaged between the model outputs and a training set for each of the tested models. The memory model was stated to perform better than DeepSimulator 1.5, Badread and the IID model. It was, however, found that the Memory model does not accurately represent long error runs [24].

2.6 Available datasets

To create a channel model for the DNA data storage process, a large amount of experimental data regarding the DNA storage channel is required. Multiple freely available datasets have been identified.

2.6.1 Clustered Nanopore Reads dataset

The Clustered Nanopore Reads (CNR) dataset is a dataset for the DNA data storage process released by Microsoft [32]. The dataset is released with the hope of furthering research on DNA data storage and is available for use given proper citation. The data set consists of a *Centers.txt* file and a *Clusters.txt* file.

The centers file contains 10000 DNA strings that were generated uniformly. Each string contains 110 bases. Each string, or center as it is termed in the dataset, starts on a new line. These strings were synthesised by Twist Bioscience and thereafter amplified using PCR. The strings were then sequenced using ONT MinION, which is a form of Nanopore technology. The resultant 269709 noisy reads were stored in the clusters file. The reads are arranged into clusters according to the different strings of the centers file, and each cluster is separated by a separator line of '=' signs. Each string in the clusters file also starts on a new line. The total size of the dataset is 30.07 MB.

2.6.2 DNA Fountain Code dataset

This dataset was created to test the DNA Fountain error correction code [33]. The dataset consists of DNA strings of length 152. The strings are designed specifically with error correction in mind and therefore conform to certain parameters. These parameters dictate that the strings do not contain any homopolymers that are longer than three bases. Furthermore, all of the strings have a GC content between 45% and 55%. In total, 72000 DNA strings were generated. The strings were amplified using PCR and sequenced using an Illumina MiSeq flow cell, resulting in 32 million erroneous output sequences.

2.6.3 Practical high-capacity low-maintenance storage dataset

This dataset contains synthesis and sequencing data for an experiment done by Goldman et al. [15]. The dataset contains 153335 DNA strings that were synthesised and amplified using PCR. The strings were then sequenced using Illumina HiSeq 2000 technology. The size of the dataset is approximately 36 GB. In order to reduce the error rate, the 153335 input strings were designed to contain no homopolymers of any length. Each of the input strings has a length of 117 nucleotides.

2.6.4 Performance assessment of DNA sequencing platforms

This comprehensive dataset aims to act as a benchmark for the comparison of different sequencing technologies [34]. The dataset includes sets of sequences derived using different sequencing technologies such as Illumina HiSeq and ONT MinION. The dataset was created by sequencing samples of human and bacterial DNA.

2.6.5 Nanopore Technologies Benchmark Datasets

Various datasets are made available by Amazon through the Registry of Open Data on AWS [35], [142]. This release contains a number of datasets, including the following:

- Genome in a bottle samples
- Tumour normal pair sequencing
- HG002 Genome in a Bottle Sample GM24385
- Drosophilidae sequencing data

The datasets are all created by sequencing natural-occurring DNA strings, such as genomes, with Oxford Nanopore technology.

2.7 String comparison algorithms

There are various algorithms for comparing two strings with each other. These algorithms are valuable for comparing original DNA strings to sequenced strings. The algorithms can be used to find the differences between the two strings, which is indicative of the errors that are introduced in the DNA data storage process.

2.7.1 Hamming distance

The Hamming distance is a string comparison metric used to compare two strings of equal length. The Hamming distance is defined as the number of positions at which the corresponding characters differ between the two strings [143]. The Hamming distance has been used in the field of bioinformatics to determine the genetic distance between members of a population [144].

2.7.2 Levenshtein distance

The Levenshtein distance is a method to determine the edit distance between two strings [145]. The Levenshtein distance calculates the number of single-character edits – namely insertions, deletions and substitutions – between the two strings, thereby finding the minimum number of edits required to transform the first string to the second. The Levenshtein distance originally did not consider substitutions as a type of edit, but rather represented a substitution with an insertion and deletion at the same position. However, in literature, the Levenshtein distance is often taken to include substitutions [146]–[148]. Therefore, in this study, the Levenshtein distance shall be defined to take substitutions into account alongside insertions and deletions.

The Levenshtein distance can be defined as the result of a recursive algorithm [146]. Let $head(z)$ denote the first character in a string z and $tail(z)$ denote the rest of the string. Additionally, let

$len(z)$ denote the length of a string z . The Levenshtein distance between strings x and y can then be defined according to:

$$lev(x, y) = \begin{cases} len(x) & \text{if } len(y) = 0, \\ len(y) & \text{if } len(x) = 0, \\ lev(tail(x), tail(y)) & \text{if } head(x) = head(y), \\ 1 + \min \begin{cases} lev(tail(x), y) \\ lev(x, tail(y)) \\ lev(tail(x), tail(y)) \end{cases} & \text{otherwise} \end{cases} \quad (2-44)$$

In the above equation, the term $lev(tail(x), y)$ indicates the occurrence of a deletion. Therefore, the corresponding character is present in string x but not in string y . The term $lev(x, tail(y))$ indicates an insertion. Therefore, the corresponding character is present in string y but not in string x . Lastly, the term $lev(tail(x), tail(y))$ indicates a substitution, where the character from string x is replaced by a different character in string y . These terms may be used to derive an edit script algorithm, which is examined in Section 2.7.4.

The above algorithm is a simple definition but an inefficient method. More efficient implementations that deliver the same result have been devised, such as the iterative method [148]. Additionally, the Levenshtein distance algorithm may be modified to award different weights to different types of errors.

As the distance was initially considered for binary sequences, the edit operations were mentioned as insertion, deletion and reversals. The term reversal indicates that a specified binary digit is substituted with the other binary digit, for example, a 0 is substituted with a 1. As DNA consists of a quaternary alphabet, substitutions are considered instead of replacements. Therefore, in this context, a substitution refers to a specified DNA base that is replaced with one of the three other DNA bases. The single-character edits considered by the Levenshtein distance therefore correspond to the basic error types found within DNA.

The algorithm has also been used previously for comparing DNA strings in the field of bioinformatics. The Levenshtein distance paved the way for algorithms employed in the commonly used BLAST programme [147], [149].

2.7.3 Damerau-Levenshtein Distance

The Damerau-Levenshtein distance is an edit distance metric similar to the Levenshtein distance. The main difference is that while the Levenshtein distance only considers insertions, deletions and substitutions, the Damerau-Levenshtein distance also considers swaps [150]. The original paper only considers strings with one or fewer errors, but the algorithm is expanded to consider strings with multiple errors [151]. The Damerau-Levenshtein distance has been used for the purpose of clustering similar enzymes together in a study within the field of bioinformatics [152].

2.7.4 Edit script using Levenshtein distance

The edit script between two strings refers to a set of instructions that can change the source string to the destination string by applying the least amount of insertions, deletions and substitutions. Furthermore, the positions at which these edits occur and the affected characters are also recorded within the edit script. An edit script algorithm based on the iterative implementation of the Levenshtein distance algorithm has been developed and is described in detail in [153].

2.7.5 Myers difference algorithm

The Myers difference algorithm is a greedy algorithm that determines the edit script between two strings [154]. The Myers difference algorithm equates finding the edit script between two strings via the longest common substring to a shortest path problem. The algorithm is shown to perform much faster than traditional methods if the difference between the two strings is small, but may perform slower than traditional methods for strings that are very dissimilar or almost completely different. The original algorithm does not take substitutions into account, but instead treats a substitution as an insertion and deletion at the same position. This algorithm has been employed in the field of bioinformatics with the purpose of aligning DNA sequences [155].

2.8 Summary

In this chapter, existing literature on the DNA data storage process was examined. The DNA data storage process was described, detailing the mechanisms that introduce errors and the types of errors that are introduced. Additionally, modelling methods such as IID models and Markov models were described. Various existing models for the DNA data storage process were described and the shortcomings of these models were identified. Different publicly available datasets that may be used to create models for the DNA data storage process were also identified. Lastly, various string comparison methods were examined. These methods may be used to compare input and output strings of the DNA data storage process in order to identify any errors that were introduced during the process.

A model of the DNA data storage process can now be created to take the different types of errors introduced in the system into account. The model shall be created using one of the identified datasets, while the dataset shall be processed using the string comparison algorithms examined in Section 2.7. The proposed model shall aim to improve on the shortcomings of existing models.

CHAPTER 3 EXPERIMENTAL DESIGN

3.1 Objective

The objective of this study is to devise a method for the DNA data storage channel. The method shall be able to accurately model error runs and bursts. Furthermore, the method shall represent other areas of the channel as well, such as the effect of homopolymer runs and variations in GC content. The model shall then be compared to the dataset and to existing models in order to ascertain the efficacy of the proposed modelling method.

3.2 Selecting a dataset

Multiple datasets were examined in Section 2.6. The characteristics of each dataset are summarised in Table 3-1.

Table 3-1: Summary of characteristics of different datasets

Dataset	Clustered Nanopore Reads	DNA Fountain Code	Practical high-capacity low-maintenance storage	Performance assessment of DNA sequencing platforms	Nanopore Technologies Benchmark
Reference	[32]	[33]	[15]	[34]	[35], [142]
Is the DNA synthesised?	Yes	Yes	Yes	No	No
Do the strings contain homopolymers?	Yes	Limited to shorter than 4	No	Yes	Yes
Is GC content varied?	Yes	Limited between 45 and 55%	Yes	Yes	Yes

The objective is to model the occurrence of errors within the DNA data storage process. Therefore, a dataset that encompasses the entire DNA data storage process from sequencing to synthesis is required. This ensures that all aspects of the process that can influence the error rate

are considered when creating the model. The datasets discussed in Sections 2.6.4 and 2.6.5 make use of naturally occurring DNA sequences that have not been synthesised. As these datasets do not encompass the entire DNA data storage process, they are not considered further.

Additionally, a model that can accurately represent the influence of different phenomena on the errors within the DNA strings is desired. Thus, a dataset that contains a large variety of phenomena within the input strings, such as homopolymers and varied GC content, is desirable. This excludes the datasets discussed in Sections 2.6.2 and 2.6.3, as they have restrictions on homopolymers and GC content.

The remaining dataset is the CNR dataset as discussed in Section 2.6.1. The dataset encompasses the entire DNA data storage process, as the strings are taken from syntheses to sequencing. Furthermore, since the strings are generated uniformly, homopolymers and a varied GC content are present. The CNR dataset therefore shows desirable characteristics for devising a model for the DNA data storage process and shall therefore be used.

3.3 Processing the dataset

As the objective is to model the occurrence of errors within the DNA data storage process, it is required to identify errors within the DNA strings of the dataset. As insertion and deletion errors may alter the length of a DNA string, the Hamming distance cannot be used. In order to find errors within each string, the edit script algorithm based on the Levenshtein distance shall be used [153].

This algorithm is described in Section 2.7.4 and shall be referred to as the Levenshtein Distance Error Identification algorithm. Pseudocode for the algorithm is given in Appendix A. This algorithm is chosen as the Levenshtein distance is a well-known string comparison metric. Therefore, functions to compute the Levenshtein distance have been thoroughly developed and tested in different programming languages. Furthermore, as the Levenshtein distance encompasses the three basic types of errors that occur within DNA strings, it has previously been applied in the field of bioinformatics. Since research does not describe swap errors in DNA data storage systems, the Damerau-Levenshtein distance is not used.

3.4 Type of model

In Section 2.5.5, it was seen that a shortcoming in one of the existing state-of-the-art models for DNA storage channels is its inability to accurately represent long error runs [24]. Due to its effectiveness in modelling error runs, a Markov model shall be used to represent the DNA data storage channel in an attempt to create an improved model.

Since Markov models derived using the Baum-Welch algorithm are seen to perform more optimally than models derived with the Viterbi algorithm, the Baum-Welch algorithm shall be used to derive the transition and emission matrices for the Markov model [134], [135].

As a Gilbert-Elliot model can be trained using the same training method as a Markov model, a model based on the Gilbert-Elliot model shall also be created. The model shall consist of an independent Gilbert-Elliot model for each type of error, namely insertions, deletions and substitutions. The errors generated by the three Gilbert-Elliot models shall then be combined in order to create a complete model.

In Section 2.2.10, it was stated that various characteristics of the DNA string can influence the error rate. As a Markov model may not be able to accommodate all the characteristics of the dataset, a combined approach shall be taken. The combined approach shall combine the Markov model with statistics derived from the dataset. After analysing the dataset, the best performing Markov model shall be selected and altered to be more representative of the characteristics found in the dataset.

3.5 Selecting existing models for comparison

It was found that DeepSimulator 1.5 does not accurately represent certain characteristics of experimental data [24]. The Memory model described in Section 2.5.4 attempts to improve on the results produced by DeepSimulator [24]. Additionally, when compared to the BadReads model [140], it was shown that the Memory model performs better [105]. Furthermore, the Memory model has been trained on a dataset created from nanopore sequencing data, similar to the CNR dataset chosen to derive our model.

The Skewed DNASimulator model [108] and DeSP [22] described in Sections 2.5.2 and 2.5.1 both make use of the IID method to model errors. Although the Skewed DNASimulator adds some skews, the model parameters were not made publicly available. Therefore, the Memory model shall be used for comparison. As the Memory model [24] was originally trained using a different dataset than the CNR dataset [32] selected for this study, the Memory model shall be retrained using the CNR dataset. This is further discussed in Section 4.11.

Additionally, an IID model shall be created for the CNR dataset. In Section 2.3.1, it is shown that IID models are regularly used for testing purposes. An IID model is simple to create, requiring few parameters to implement. Therefore, an IID model shall also be used for comparison.

3.6 Experimental process

The experimental process consists of multiple steps. Firstly, the dataset is processed by detecting the errors within each noisy read in the dataset. This is done using the Levenshtein Distance Error Identification algorithm, as described in Section 2.7.4 [153]. The dataset is then split into a training and a testing set.

Additionally, statistics are calculated from the training set. This includes the overall error rate and the rate of each type of error. The distribution of bases for insertion errors is also calculated, i.e. the rate of insertion for each of the four bases. Furthermore, the distribution of bases for substitution errors is calculated, i.e. given the current base, the rate of substitution for each of the three remaining bases.

Thereafter, the Markov and Gilbert-Elliot models can be created. An IID model can also be created, and the models can be compared to each other, and with the dataset and Memory model.

3.6.1 Markov model

A string of outcomes is created for each string in the training set. Four outcomes are considered, namely a successful transition or one of the three types of errors – insertion, deletion or substitution.

This sequence of outcomes is then used to train a Markov model. Models with varying parameters are trained in order to fine-tune the parameters and create the most effective model. The Baum-Welch algorithm is used to train the models. These models are capable of generating strings of outcomes given the length of an input DNA string.

3.6.2 Gilbert-Elliot model

Three additional sequences of outcomes are created for each string in the training set – one for each type of error. Therefore, each sequence contains only two types of outcomes, namely a successful transition and one of the three types of errors. A Gilbert-Elliot model is trained on each of the three sets of sequences of outcomes. Therefore, each model considers a different type of error. These independent Gilbert-Elliot models are also trained using the Baum-Welch algorithm. By combining the three models, a model can be created that considers all three types of errors. The model is capable of generating a string of outcomes given the length of an input DNA string.

3.6.3 Sequences of outcomes

Each of the created models produces a sequence of outcomes, but not specific bases. These strings of outcomes can then be used to create strings of bases. An input string is used as template. Wherever the string of outcomes contains a successful transmission, the corresponding

base from the input string is added to the output string. If the sequence of outcomes contains a deletion, the corresponding base is omitted. When the sequence contains a substitution, the corresponding base from the input string is replaced, and the new base is selected according to the probability distributions calculated from the training set. When the sequence of outcomes contains an insertion, an additional base, which is selected according to the probability distributions calculated from the training set, is added.

3.6.4 Improved model

After creating Markov models with different parameters, the models shall be compared to the dataset. The ability of the models to represent error bursts and runs shall be assessed. The best performing Markov model shall be selected. Thereafter, the dataset shall be analysed further to determine the different characteristics that the Markov model is unable to take into account. An improved model shall then be created. This model shall use the chosen Markov model as a base and attempt to represent the other characteristics of the dataset. After creating the improved model, strings of bases shall be generated in order to analyse the efficacy of the model.

3.6.5 IID model

An IID model is also created. This model shall serve as reference for comparison. The model uses the probability distribution of each type of error as calculated above to generate output strings with independently distributed errors.

3.6.6 Comparison of models

Strings of bases are generated using the Memory model as well for an additional comparison. After strings of bases have been generated by each model, errors are identified within these strings using the same edit script algorithm as used on the dataset. Thereafter, the models can be compared to each other and to the dataset in various areas.

3.7 Comparison areas

In order to determine the efficacy of the developed models, they shall be examined in various different areas. The models shall be compared to each other and to the testing set in each of these areas. The areas in which the models are compared to the dataset are summarised in Figure 3-1.

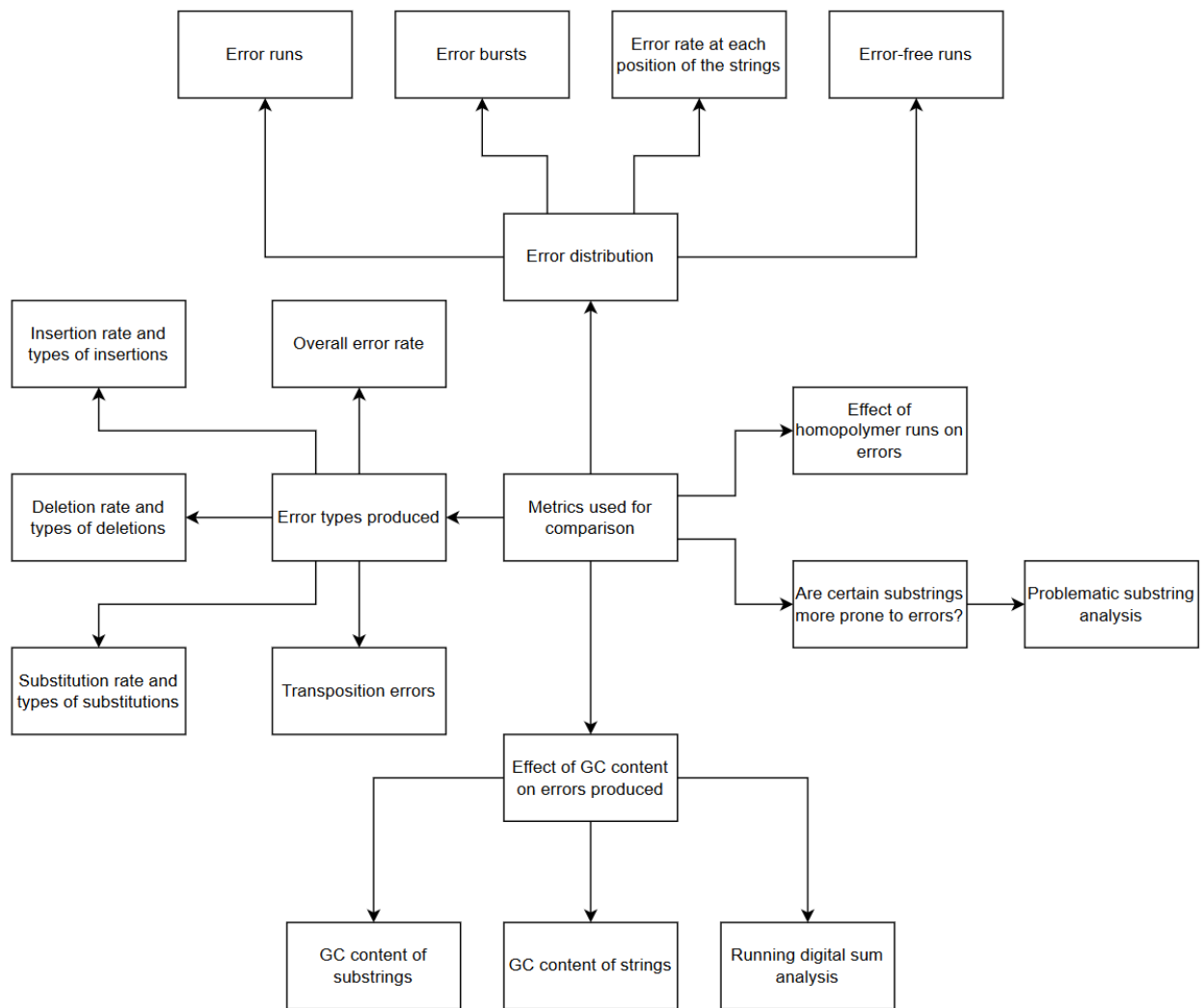


Figure 3-1: Areas in which models are compared

3.7.1 Error distribution

Firstly, it shall be verified if the errors are distributed independently or dependently throughout a DNA string. This shall be done by inspecting the error runs and error bursts found within the testing set and the model outputs. Furthermore, the error rate produced by each model at each position of the output strings shall be examined. Error-free runs shall also be examined, as they may give an indication of the error distribution.

3.7.2 Error types

The types of errors produced by each model shall also be examined. This includes the overall error rate and the rate of each type of error. Swap errors shall also be examined. Additionally, the type of deletions shall be examined, i.e. the probability for each type of base to be deleted. The type of insertions shall be examined in the same manner. Lastly, the type of substitution shall be

examined. This includes the probability for each type of base to be substituted, as well as the probability for each type of base to replace the current base given the current base.

3.7.3 Homopolymer runs and problematic substrings

The effect of homopolymer runs shall be examined. The error rate within homopolymer runs shall be calculated and compared, as well as the error rate right before and right after the run. This shall give an indication of whether homopolymer runs lead to increased error rates within the test set, as well as the ability of each model to represent this aspect. Problematic substrings shall also be examined. The error rate shall be computed for each possible substring of a certain length. This shall indicate if certain substrings are more prone to errors, and if the models are able to accurately reproduce this aspect.

3.7.4 GC content

The effect of GC content on the error rate shall also be examined. The GC content of substrings shall be compared to the error rate of the substrings to see if the local GC content has an effect on the error rate. The overall GC content of a string shall be compared to the error rate of the string to determine if the overall GC content has an effect. Additionally, a running digital sum value shall be computed. G and C bases shall increase the sum, while A and T bases shall decrease the sum. The error rate can then be compared to the RDS to determine if there is a correlation.

3.8 Statistical metrics

3.8.1 Correlation coefficient

The Pearson correlation coefficient may be used to measure the strength of the relationship between two variables. The coefficient is given by r in [156]:

$$r = \frac{\sum_{n=1}^N (a[n] - \bar{a}) \sum_{n=1}^N (b[n] - \bar{b})}{\sqrt{\sum_{n=1}^N (a[n] - \bar{a})^2 \sum_{n=1}^N (b[n] - \bar{b})^2}} \quad (3-1)$$

$a[n]$ and $b[n]$ represent the n^{th} values of the two respective variables, while $\bar{a}$ and $\bar{b}$ represent the means of the two respective variables.

The values of r range from -1 to 1 . Values close to zero indicate that there is no correlation between the two variables. Values close to -1 represent a strong negative correlation, while values close to 1 represent a strong positive correlation.

3.8.2 Skewness

The skewness of a distribution may be calculated as a measure of the symmetry of the distribution [157]. The meanings of different skewness values are summarised in Table 3-2. The skewness of a distribution is calculated according to [158]:

$$m_i = \frac{1}{N} \sum_{n=1}^N (x[n] - \bar{x})^i \quad (3-2)$$

$$g_1 = \frac{m_3}{m_2^{3/2}} \quad (3-3)$$

where g_1 represents the skewness of the distribution. N is the number of samples in the set, while $x[n]$ refers to the n^{th} sample. $\bar{x}$ is the mean of the set.

Table 3-2: Meaning of different skewness values

g_1 value	Direction of skewness	Meaning
≈ 0	Symmetrically distributed	Both tails are of similar length
< 0	Skewed left	The left tail of the distribution is longer than the right tail
> 0	Skewed right	The right tail of the distribution is longer than the left tail

3.8.3 Root mean square error

The root mean square error (RMSE) is used to compare the difference between measured and estimated values given by a model [159]. The RMSE is calculated with:

$$RMSE = \sqrt{\frac{1}{N} \sum_{n=1}^N (x[n] - y[n])^2} \quad (3-4)$$

where N indicates the number of samples that are considered. $x[n]$ represents the n^{th} measured sample, while $y[n]$ represents the n^{th} value estimated by the model.

The value of the RMSE is directly proportional to the size of the errors encountered. Therefore, in instances where certain errors may be very large, the root mean squared logarithmic error (RMSLE) may be considered. The RMSLE is calculated using:

$$RMSLE = \sqrt{\frac{1}{N} \sum_{n=1}^N (\log(1 + x[n]) - \log(1 + y[n]))^2} \quad (3-5)$$

Lastly, to compare models with different scales, the normalised root mean square error (NRMSE) may be considered. This is useful for comparing the distribution of values for models that produce error rates within the DNA strings. The NRMSE is defined by:

$$NRMSE = \sqrt{\frac{1}{N} \sum_{n=1}^N \left(\frac{x[n]}{\bar{x}} - \frac{y[n]}{\bar{y}} \right)^2} \quad (3-6)$$

where $\bar{x}$ represents the mean of the observed values and $\bar{y}$ represents the mean of the predicted values:

3.9 Technologies used

In order to implement and test the model, multiple technologies shall be used. Due to its open source nature and ease of use, most of the data handling shall occur in Python. Furthermore, various additional modules within Python shall be used. In certain instances, the functions provided by the MATLAB toolboxes are invaluable, therefore MATLAB shall be used for these cases. The different technologies, as well as the versions used, are listed in Table 3-3.

MATLAB toolboxes are thoroughly tested and well documented [160]. Therefore, it is assumed that the accuracy of any functions provided by these toolboxes is acceptable. The Python Standard Library is distributed with the Python programming language [161], [162]. It is therefore assumed that the functions provided within this library are of adequate accuracy in order to meet the standards of the Python Software Foundation.

The *matplotlib* library is a third-party module often used to visualise data within Python. Its use is considered standard and the visualisation functions therein are viewed as accurate [163]. *SciPy* is a Python third-party module that contains various algorithms aimed at scientific use. It is widely used in the scientific community and the functions therein are considered accurate [164].

Numpy is another third-party module that is often used to handle data in the form of matrices. Data can be stored in the form of matrices, and the module can be used to perform various calculations on these matrices. The module is often used to represent matrices in scientific literature. Therefore, its use is seen as standard and the functions therein are viewed as accurate [165].

The *Levenshtein* module is a third-party Python module capable of computing the Levenshtein distance between two strings. Its accuracy shall be verified through testing.

Table 3-3: Technologies used

Technology	Version	Documentation
MATLAB	R2022b	[166]
MATLAB Statistics and Machine Learning Toolbox	R2022b	[124]
MATLAB Text Analytics Toolbox	R2022b	[167]
Python	3.9.12	[168]
Python Standard Library	3.9.12	[162]
Python matplotlib	3.5.1	[169]
Python SciPy	1.7.3	[170]
Python NumPy	1.21.5	[171], [172]
Python Levenshtein	0.12.2	[173]

3.10 Summary

In this chapter, the experimental design that shall be used for this study is described. The CNR dataset [32] was selected to train the proposed model. Additionally, the workflow for training Markov and Gilbert-Elliott models was described. It was decided to compare the proposed models to existing models, namely the IID and Memory models [24]. Furthermore, multiple areas for comparison were identified. Lastly, the technologies to be used were selected as Python and MATLAB, along with various toolboxes and libraries. The implementation described in Chapter 4 shall follow the experimental design described in this chapter.

CHAPTER 4 IMPLEMENTATION

This chapter describes the implementation of the experiment. Firstly, the implementation of the error identification algorithm is discussed, followed by its application to the dataset. The calculation of statistics from the dataset is explained. Thereafter, the algorithm to derive a string of outcomes for each string in the dataset is discussed. The discussion is extended to strings of outcomes specifically for training the independent Gilbert-Elliot models in Section 4.5.

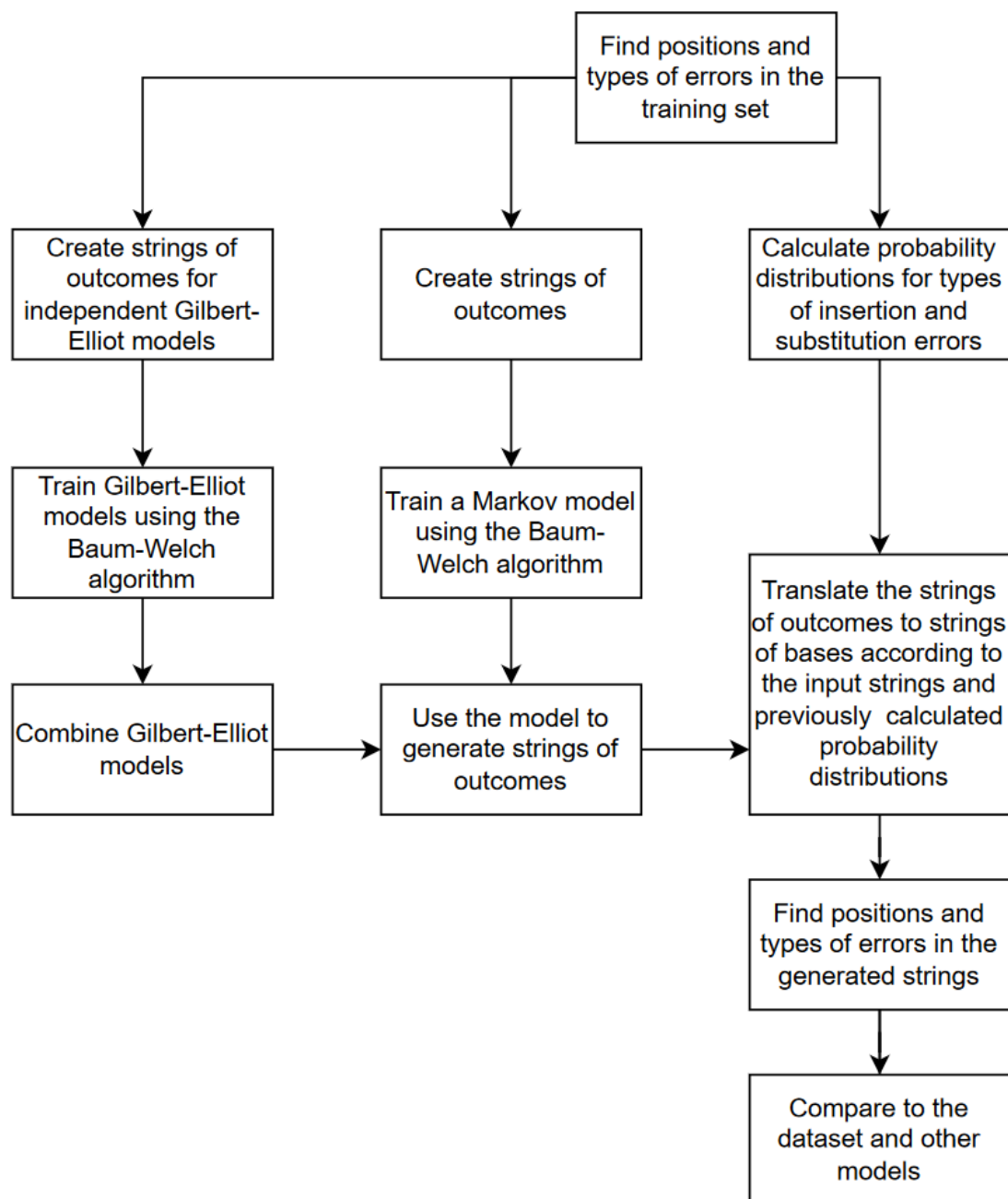


Figure 4-1: Workflow for creating a Markov model

The creation of the Markov and Gilbert-Elliot models is also described in this chapter. Figure 4-1 summarises the process for training Markov and Gilbert-Elliot models. The method for generating noisy DNA strings with these models is discussed. The creation of an IID model and modifications to the Memory model follow. Finally, the methods used to compare the different models are discussed.

The implementation of an improved model shall be described in Chapter 8. The code files for the implementation are stored on GitHub and may be made available per request.

4.1 Levenshtein Distance Error Identification

In order to identify errors, an algorithm based on the Levenshtein distance [153] is used to compare a noisy string to an error-free string. This is a Python implementation of the algorithm as described in Section 2.7.4. Pseudocode is given in Appendix A. As this algorithm can be applied to the dataset and to strings generated by models, this algorithm corresponds to the blocks titled “Find positions and types of errors in the training set” and “Find positions and types of errors in the generated strings” in Figure 4-1.

The algorithm takes two strings, representing the original string and a noisy string, as input. The algorithm iterates through both strings simultaneously in order to identify any discrepancies. The discrepancies are recorded as errors and the type of errors, bases involved and positions of the errors are returned in the form of a list. Errors include insertions, deletions and substitutions. Each type of error is recorded in the following format:

- Substitution:
 - ‘s’, position in the noisy string, position in the original string, base in the original string, base in the noisy string
- Deletion:
 - ‘d’, position in the noisy string, position in the original string, base deleted from the original string,
- Insertion:
 - ‘i’, position in the noisy string, position in the original string, base inserted in the noisy string,

Both the position in the original string and the position in the noisy string are recorded, as the position may differ due to insertion and deletion errors affecting the length of the strings. The position of an insertion error is taken as before the current position in the string, therefore an insertion at the start of the string takes place at position 0. Multiple insertions in a row share the same position in the original string, but consecutive positions in the noisy string.

4.1.1 Verify error identification using Levenshtein distance

This file uses the *Levenshtein* module in Python to compare the Levenshtein distance between the input and output DNA strings to the number of errors identified by the Levenshtein Distance Error Identification algorithm. In this manner, the accuracy of the error identification algorithm can be verified, as the algorithm should detect the same number of errors as the value of the Levenshtein distance.

The *distance* function in the *Levenshtein* Python module is used [174]. The *distance* function requires two input arguments, namely the two strings to be compared. The file further requires multiple text files as input. The required files include the centers and clusters files containing the dataset. Additionally, a file containing the errors identified by the error identification algorithm is required.

L is defined as the number of lines in the cluster file. $numErrors[i]$ is defined as the number of errors identified by the Levenshtein Distance Error Identification algorithm in the i^{th} string from the cluster file. The algorithm is given below.

Verify error identification

```
// Initialization
j = 0
currentCenter = centers[j]
difference = []
largerVal = []
// Iteration
for i = 1,2,3, ..., L
    currentCluster == clusters[i]
    if currentCluster == seperator
        j = j + 1
        currentCenter = centers[j]
    distance = lev(currentCenter, currentCluster)
    append to difference: |numErrors[i] - distance|
    if numErrors[i] > distance:
        append to largerVal: 'numErrors'
    else if numErrors[i] < distance:
        append to largerVal: 'distance'
    else:
        append to largerVal: 'equal'
```

After completion of the algorithm, the minimum and maximum differences are identified from the list using the *min* and *max* functions built into Python. The *statistics* Python module, which is part of the Python Standard Library, is used to find the mean, median and standard deviation values for the list [175].

The number of times that the Levenshtein distance is larger than the number of errors identified is counted. The number of times that the number of errors identified is larger than the Levenshtein distance is also counted. This is done using the *count* method built into Python. If the Levenshtein Distance Error Identification algorithm performs inaccurately, this may give an indication of where the problem lies.

The same test is done using the *editDistance* function from the MATLAB Text Analytics Toolbox [176]. The function also requires two input arguments, namely the two strings to be compared. While other edit distance algorithms may be specified, the function uses the Levenshtein distance by default.

Any discrepancies between the values derived by the MATLAB *editDistance* and Python *distance* functions, and the numbers of errors identified by the Levenshtein Distance Error Identification algorithm, indicate errors in the Python code.

4.2 Data statistics

In order to decide which specific bases to substitute or insert, additional statistics are required from the dataset. The function to calculate these statistics corresponds to the block titled “Calculate probability distributions for types of insertion and substitution errors” in Figure 4-1. Additionally, the results produced by this function may be used to compare the different models, thereby the function can also correspond to the block titled “Compare to the dataset and other models”.

A function was created that iterates through all the errors in a given list and counts the number of errors of each type that occurs. The probability of each different type of error is then calculated. The probability of each different type of base to be affected is also calculated. The values are stored in dictionaries, with the dictionary keys representing the type of error and the affected base. The dictionary values represent the calculated values.

The following values are calculated and returned by the function:

- Total error rate
- Probability of each type of error to occur (insertion, deletion or substitution), given that an error occurs

- Probability of substitution of each different base given that a substitution error occurs and the current base
- Probability of insertion of each different base given that an insertion error occurs
- Probability of deletion of each different base given that a deletion error occurs

4.3 Detecting errors in a dataset

In order to process the dataset, a Python algorithm iterates through the clusters.txt and centers.txt files in the dataset simultaneously. The clusters.txt file contains the noisy reads, while the centers.txt file contains the corresponding original string. Therefore, the current original string is referred to as the current center. The current line in each file is represented by the current cluster value and current center value respectively. When a separator line is detected in the clusters file, the algorithm moves to the next value in the centers file. At each iteration, the algorithm compares the current cluster value to the current center value using the Levenshtein Distance Error Identification algorithm. The results of the algorithm are appended to a list, along with the index of the current cluster and center.

After the algorithm has iterated through the entire list of clusters, the errors found can be stored in a text file. The format is as follows:

- Underscores (“_”) separate the center index, cluster index and first error from each other
- Semi-colons (“;”) separate different errors in the same line from each other
- Hashtags (“#”) separate the different components of each error from each other
- The list of errors of each different string starts in a new line

The following errors found within the first cluster DNA string are used as an example:

[‘s’, 17, 17, ‘C’, ‘T’]

[‘i’, 31, 31, ‘G’]

[‘i’, 32, 31, ‘T’]

[‘d’, 80, 78, ‘G’]

Since this is the first string, the index of the cluster and center are both 0. This yields the following list of errors as stored in the text file:

0_0_s#17#17#C#T;i#31#31#G;i#32#31#T;d#80#78#G

The above algorithm can be applied to any file containing a list of sorted clusters and the accompanying *centers* file, given that the centers and clusters are in the same order. This

algorithm corresponds to the block titled “Find positions and types of errors in the training set” in Figure 4-1.

4.4 Strings of outcomes

Since a Markov model requires a string of observations to train, the errors found in the noisy reads should be converted to a sequence of observations. This is done by iterating through each original string. At each position, the list of errors is inspected to see if an error has occurred at that position. If no error is detected, a ‘t’ is added to the string of observations, indicating a successful transmission. A ‘d’, ‘s’, or ‘i’ is added to represent deletion, substitution or insertion errors respectively.

Representing the observations using alphabet letters improves interpretability, but many algorithms require numerical values to function. Therefore, this alphabet representing the four types of observations is converted to numbers for future use according to the following table. The strings of observations are stored in a text file for future use. This algorithm corresponds to the block titled “Create strings of outcomes” in Figure 4-1.

Strings of observations are useful for detecting error runs and bursts. Therefore, this algorithm can be applied to the outputs of different models, as well as the errors identified within the noisy strings of the dataset. As the alphabetical representation is more interpretable than the numerical representation, the alphabetical representation is returned for cases where numerical values are not required. The alphabetical and numerical representation of each observation is given in Table 4-1.

Table 4-1: Numerical representation of each type of observation

Observation	Alphabetical representation	Numerical representation
Transmission	‘t’	1
Insertion	‘i’	2
Deletion	‘d’	3
Substitution	‘s’	4

4.5 Strings of outcomes to train the independent Gilbert-Elliot models

In order to train independent Gilbert-Elliot models that represent each type of error, different strings of observations are required. To achieve this, the function iterates through each string of

observations previously found for the dataset. For each string of observations, three new strings are derived – one for each type of error.

For insertion errors, if the observation is an insertion, represented by a '2', a '2' is added to the string of observations representing insertion errors. For all other observations, a '1' is added. Although using the values of '0' and '1' may be more intuitive, the mathematical software utilised to train the models does not support the use of '0' in this instance. Therefore the values of '1' and '2' are used to represent the different cases.

For deletion errors, if the observation is a deletion, represented by a '3' in the original string of observations, a '2' is added to the string representing deletion errors. For all other observations, a '1' is added.

Likewise, for substitution errors, if the observation is a substitution, represented by a '4' in the original string of observations, a '2' is added to the string representing substitution errors. For all other observations, a '1' is added.

These three sets of strings of observations are then stored in three individual text files for future use. This algorithm corresponds to the block titled “Create strings of outcomes for independent Gilbert-Elliot models” in Figure 4-1.

4.6 Deriving model parameters

MATLAB was used to derive the parameters for the Markov and Gilbert-Elliot models. The MATLAB code file requires a text file containing the training data as input. The format of the input file is as follows:

- Each new line represents a different DNA string
- Each line contains a string representing the string of outcomes of that DNA string

The text file is read by MATLAB and the strings of observations are converted into arrays of integers. Initial transition and emission matrices are defined by the user.

The true values of the matrices are then estimated using the *hmmtrain* function from the MATLAB Statistics and Machine Learning Toolbox. The function requires an array containing the training data as input. Furthermore, the initial matrices as defined above are given as input values for the function. A tolerance for convergence as well as the maximum iterations are also specified. Lastly, the function allows the user to choose between the Baum-Welch and Viterbi algorithms to estimate the values. The Baum-Welch algorithm is the default option and is also used in this implementation.

Finally, the estimated transition and emission matrices are stored in separate text files. The different rows of the matrices are stored in the different lines of the file, while semi-colons (;) separate the columns within each row. The names of the files are user-defined.

4.6.1 Markov model

Below is an extract of the input file used to train the Markov model, representing two different strings.

[illegible]

The values for the transition and emission matrices are randomly defined by the user. Any non-zero and non-unity probability values are acceptable. The dimensions of these matrices are $m \times m$ and $m \times n$ respectively. m represents the desired number of states for the model. n represents the number of different observations, which is four in this case.

In order to create Gilbert-Elliott models for each type of error, the other types of errors are taken as transitions, while the error focused on is represented by a '2'. For the example string of observations 41131112, if only deletions are considered, the string of observations shall be 11121111. This process is explained in further detail in Section 4.5.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (4-1)$$

A value of 1 indicates a 100% chance for the corresponding observation to occur in the specified state, while a value of 0 indicates a 0% chance. Setting the values as such prevents the training algorithm from changing these values, and they will therefore remain constant throughout training.

This process is repeated three times, once for insertion errors, once for deletion errors and once for substitution errors to train three different models. The three models are then combined into a single model, as explained in Section 4.8.

According to the definitions described in Sections 2.3.2.1 and 2.3.2.2, by setting the emission matrix to the value of equation (4-1), the model can also be described as both a Gilbert-Elliot and a Fritchman model. For clarity, the model shall be referred to as a Gilbert-Elliot model throughout the rest of this study.

4.7 Generating model outputs

In order to generate outputs from the Markov model trained within MATLAB, the emission and transition matrices are imported into Python.

A random number is generated using the *random* Python module, which is part of the Python Standard Library [177]. This number is used to determine the starting state according to the uniform probability distribution between all the states of the model.

A string of outputs is generated by iterating through the length of the corresponding input string. At each position, a random number is generated to determine the state according to the transition matrix. Another random number is generated to determine the occurrence of errors according to the emission matrix. If a successful transmission is generated, a 't' is added to the string of outputs. Otherwise, a 'd', 's' or 'i' is appended to represent the different types of errors. As insertions add additional bases to the string, if an insertion is generated, another generation is done at the same position of the string.

Insertion errors are handled as occurring before the current position in the input string. Therefore, when the last position of the string is reached, an additional emission is predicted to accommodate insertions happening after the final base of the string. This prediction only takes insertion errors into account, as substitutions or deletions do not occur before or after a base, but at the same position as the affected base. If an insertion is generated, an additional emission is generated. In order to allow for strings of insertions to occur, this process repeats until the generated emission is not an insertion.

This algorithm corresponds to the block titled “Use the model to generate strings of outcomes” in Figure 4-1. The string of outcomes can then be populated with specific bases according to the algorithm that is described in Section 4.9.

4.8 Generating model outputs – independent Gilbert-Elliot models

A model is devised that uses three independent Gilbert-Elliot models to make predictions. Three Gilbert-Elliot models are trained in MATLAB, for insertion errors, deletion errors and substitution errors respectively. This process was described in Section 4.6.2. The process to create an output string by combining the three models is summarised in Figure 4-2. In order to generate outputs from the independent Gilbert-Elliot models, the three transition matrices are imported into Python. As the emission matrices are all set to the value from equation (4-1), they do not have to be imported.

A random number is generated using the *random* Python module. This number is used to determine the starting state for each of the three independent Gilbert-Elliot models according to a uniform distribution.

A string of observations is generated by iterating through the length of the corresponding input string. Since the model consists of three independent Gilbert-Elliot models, three distinct state variables are required to store the current state of each model. At each position of the input string, three random numbers are generated in order to determine the state according to the three respective transition matrices.

Firstly, the model pertaining to insertion errors is considered. Since the emission matrix has a 0% error rate in the first state and a 100% error rate in the second state, the state of the model is used to determine if an insertion error is generated. If no insertion error is generated, the system considers the model pertaining to deletion errors. If no deletion error is generated, the system considers the model pertaining to substitution errors.

If a successful transmission is generated, a ‘t’ is added to the string of outputs. Otherwise, a ‘d’, ‘s’ or ‘i’ is appended to represent the different types of errors. As insertions add additional bases to the string, if an insertion is generated, another generation is done at the same position of the string.

Similarly to the Markov model, insertion errors are handled as occurring before the current position in the input string. Therefore, when the last position of the string is reached, an additional emission is predicted to accommodate insertions happening after the final base of the string. This prediction only takes insertion errors into account, as substitutions or deletions do not occur

before or after a base, but at the same position as the affected base. If an insertion is generated, an additional emission is generated. In order to allow for strings of insertions to occur, this process repeats until the generated emission is a transmission.

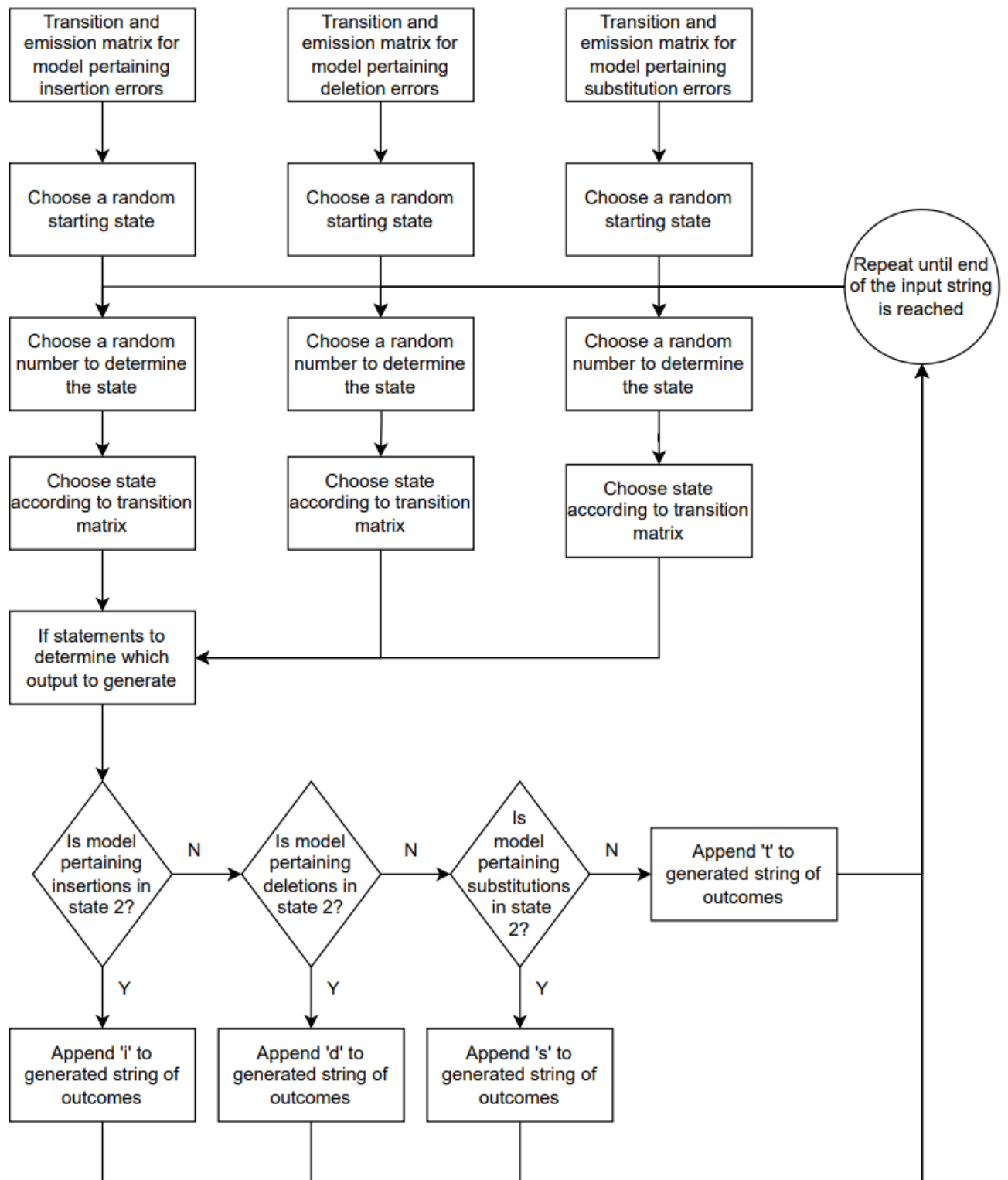


Figure 4-2: Combining outputs of three independent Gilbert-Elliot models

This algorithm corresponds to the blocks titled “Combine Gilbert-Elliot models” and “Use the model to generate strings of outcomes” in Figure 4-1. The string of outcomes can then be populated with specific bases according to the algorithm that is described in Section 4.9.

4.9 Populating strings of outcomes with bases

The models described in Sections 4.7 and 4.8 produce strings of outcomes. These strings represent the positions and type of errors that occur within an output DNA string, but the specific bases are not yet added. The following algorithm is used to populate a string of outcomes with specific bases. Pseudocode for the algorithm is given in Appendix B. This algorithm corresponds to the block titled “Translate the strings of outcomes to strings of bases according to the input strings and calculated probability distributions” in Figure 4-1.

Firstly, the function to calculate data statistics from Section 4.2 is used to calculate the rates for different types of insertion and substitution errors from the same training set used to train the model.

The function then proceeds to create a string containing the actual bases by iterating through the string of observations and comparing it to the input DNA string. If the observation is a transition, the corresponding base of the input string is appended to the string of bases. If the observation is a deletion, the corresponding base is omitted. If the observation is an insertion, the base that is inserted is selected according to the probability distribution calculated for different types of insertions. This base is then appended to the string of bases. If the observation is a substitution, the corresponding base of the input string is identified and the base that replaces it is selected according to the probability distribution calculated for different types of substitutions. This base is then appended to the string of bases.

The noisy strings of bases generated by the model are stored in a text file. Errors can then be detected using the previously defined Levenshtein Distance Error Identification algorithm. This ensures consistency when the errors generated by the model are compared to errors generated by different models, as well as the errors that occur in the original dataset. This error data is stored in the same format as described in Section 4.2.

4.10 Creating an IID model

As a comparison to the other models, an IID model is created based on the dataset. The model is created using the same training set as the other models to ensure consistency.

The function from Section 4.2 is used to calculate the overall error rate, rate of each type of error and rates for different types of insertion and substitution errors from the same training set used to create the model.

The *random* Python module is used to generate random numbers used for probabilities throughout this model.

To generate the output bases, the model iterates through the input string. At each position, the model determines whether an error occurs according to the error rate of the training set. If no error occurs, the corresponding base of the input string is added to the output string. If an error occurs, the model determines if it is an insertion, deletion or substitution error according to the corresponding probability distribution. If the error is a deletion error, the corresponding base of the input string is omitted and the model moves to the next base. If a substitution is predicted, the model uses the probability distributions for different types of substitution errors to select a base to replace the corresponding base of the input string. If an insertion error is predicted, the base to be inserted is selected according to the probability distribution for different types of insertions. The model makes an additional prediction at the same position of the input string as insertion errors add bases to the string.

Insertion errors are once again handled as if occurring before the corresponding base of the input string. Therefore, when the last position of the string is reached, an additional prediction is made to accommodate insertions happening after the final base of the string. This prediction only takes insertion errors into account. In order to allow for strings of insertions to occur, this process repeats until the generated prediction is error-free.

As a means to ensure consistency, the strings of bases generated by the model are stored in a text file, whereafter errors can be detected using the Levenshtein Distance Error Identification algorithm.

4.11 Training the Memory model on the CNR dataset

The original implementation for the Memory model [24] is available on GitHub and was done in the Julia programming language [178]. The model takes an input sequence and generates a corresponding output sequence. The model was not originally trained on the CNR dataset [32] used for this study, and shall therefore be retrained using the CNR dataset. To retrain the model, the model is implemented in the Python programming language. The implementation is done according to the method described in [105], which is summarised in Section 2.5.5.

The following probabilities are defined according to the description provided in [24], [105] and summarised in Section 2.5.5:

$$P(e_t | kmer_t, e_{t-1}) \quad (4-2)$$

$$P(L_t | kmer_t, e_t = \text{Insertion}) \quad (4-3)$$

$$P(B_t | kmer_t, e_t = \text{Substitution}) \quad (4-4)$$

$$P(e_t | x_t) \text{ for } t \in [2, 3, \dots, k] \quad (4-5)$$

$$P(L_t | x_t, e_t = \text{Insertion}) \text{ for } t \in [2, 3, \dots, k] \quad (4-6)$$

$$P(B_t | x_t, e_t = \text{Substitution}) \text{ for } t \in [2, 3, \dots, k] \quad (4-7)$$

$$P(e_t) \text{ for } t \in \{1, N\} \quad (4-8)$$

$$P(L_t | e_t = \text{Insertion}) \text{ for } t \in \{1, N\} \quad (4-9)$$

$$P(B_t | x_t, e_t = \text{Substitution}) \text{ for } t \in \{1, N\} \quad (4-10)$$

Insertion errors are taken as occurring before the current base of the input string. In order to account for insertion errors after the last base of the input string, the following probabilities are added:

$$P(e_t = \text{Insertion}) \text{ for } t = N + 1 \quad (4-11)$$

$$P(L_t | e_t = \text{Insertion}) \text{ for } t = N + 1 \quad (4-12)$$

Additionally, the following probability is added to determine which bases are inserted:

$$P(B | e_t = \text{Insertion}) \text{ for } t \in [1, 2, \dots, N, N + 1] \quad (4-13)$$

These probabilities are calculated from a training set. The model is implemented using *kmers* of length 6 as this is the default *kmer* of the original model [105]. The Python implementation is summarised as pseudocode in Appendix C.

A for-loop was added to the main file of the implementation in order to generate multiple output sequences and write the sequences to a text file. A large number of outputs are generated via the for-loop in order to evaluate the model accurately. Therefore, multiple output sequences are generated for each input sequence. The output strings are clustered together according to the corresponding input strings, and different clusters are separated by a line of symbols. This file can then be read and used as input for the Levenshtein Distance Error Identification algorithm. This error data is stored in the same format as described in Section 4.1, allowing the data to be compared to that of the other models, as well as the dataset.

4.12 Comparison of models

In this section, the different models are compared in various areas. Firstly, the lists of errors generated by each model, as well as the errors in the dataset, are imported. It is ensured that all of the lists contain the same number of strings for an accurate comparison. The following calculations are made on the model outputs and the dataset:

- Due to the formatting of the lists of errors, by counting the empty entries, the number of error-free strings generated by each model is found.
- The average lengths of the output strings are calculated from the length of each string.
- The length of each entry in the lists of errors is counted to find the total number of errors, which is then used to calculate the error rate generated by each model.

By using the function to calculate various statistics defined in Section 4.2, the total error rate of each model is calculated once again and can be compared to the previous result for verification purposes. Furthermore, the rate of each type of error is calculated, as well as the rates of insertions for different bases. Additionally, the rates of different types of substitutions are calculated. Lastly, the number of deletions of each type of base is calculated.

The number of errors generated at each position of the input string is counted by iterating through all the errors identified in the dataset and the model outputs. The error rate at each position is then calculated. Furthermore, this is done for each different type of error.

4.12.1 Error runs

Runs of errors generated by each model can be identified by using lists of outcomes. These lists of outcomes are generated by the function described in Section 4.4. Multiple consecutive symbols in a string of outcomes that indicate errors constitute an error run. Furthermore, multiple identical symbols in a string of outcomes constitute a run of errors of the same type. By iterating through all the strings of outcomes for a specific model, the total number of error runs, as well as the number of runs of each type of error can be counted. This is done for different types of runs in order to give an indication of the run lengths produced by each model, as well as the runs occurring within the dataset. Pseudocode for this algorithm is presented in Appendix D.

4.12.2 Error bursts

An error burst is defined as multiple errors within close proximity of each other. By setting a threshold that defines the proximity, the bursts generated by each model can be detected from the strings of outcomes. This threshold can be defined by the user as an input value to this algorithm. The threshold value is the maximum number of error-free transmissions between two errors that is considered a burst.

This algorithm iterates through a string of outcomes and counts the number of consecutive symbols representing a successful transmission. If this number is less than the defined threshold, a burst is identified. This process repeats until the number of consecutive successful transmissions is greater than the defined threshold, indicating the end of the burst. In the process, the total length of the burst, the total number of errors in the burst and the starting position of the burst are recorded. The number of bursts starting at each position in the string can thus be counted. Furthermore, the number of bursts of different lengths can be counted, as well as the number of bursts with different amounts of errors. Pseudocode for this algorithm is presented in Appendix E.

4.12.3 Error-free runs

An error-free run is defined as multiple consecutive successful transmissions. By iterating through a string of outcomes and counting consecutive symbols that represent successful transmissions, error-free runs can be identified. As soon as a symbol representing an error is detected, the run is terminated. The length of the run is recorded, and the algorithm counts the number of error-free runs of different lengths. Pseudocode for this algorithm is presented in Appendix F.

4.13 Homopolymer analysis

A homopolymer is defined as multiple consecutive bases of the same type. In Section 2.2.10.1, it is indicated that homopolymers may lead to an increase in errors. This function analyses this attribute and can be applied to the dataset, as well as the strings generated by different models.

A homopolymer is identified by iterating through a string of bases. Each base is compared to the previous base. If the two bases are the same, a homopolymer is identified. The number of consecutive identical bases is counted. As soon as a different base is detected, the homopolymer is terminated and its length is recorded. As each base is compared to its predecessor, the algorithm starts at the second base in order to avoid any errors.

By iterating through all the strings in a set of data, homopolymers can be identified within each string. The position at which each homopolymer starts is recorded. The length of the homopolymer is used to find the position at which the homopolymer ends. By simultaneously iterating through the list of errors identified for that dataset, the position of each error can be compared to the position of the homopolymer. Thereby, the number of errors within the homopolymer can be found. Furthermore, the number of errors right before and right after the homopolymer can also be recorded respectively. These values can then be compared to the length of the homopolymer in order to investigate the correlation between homopolymer length

occurrences of errors. The type of bases comprising the homopolymer is also recorded, allowing the error rates induced by homopolymers of different bases to also be compared. Pseudocode for this algorithm is presented in Appendix G.

4.14 GC content analysis

In Section 2.2.10.3, it is suggested that the GC content of a DNA string may influence the number of errors. Therefore, this function analyses this attribute and can be applied to the dataset, as well as the strings generated by different models.

The GC content can be found by iterating over a given DNA string and counting the number of G's and C's within that string. The percentage GC content can be found by dividing this value by the length of the string. This algorithm can also be applied to a specific section of a string, for example, a window within the string.

The function iterates through all the strings in a given dataset. A window length is defined by the user, and the current string is divided into multiple windows. The following window starts where the previous window ends, thereby removing any overlap between different windows. At each window position, the algorithm iterates through the list of errors identified within the dataset. The position of each error is compared to the position of the window. Thereby the error rate within the window can be found. The error rate as well as the GC content of the window is recorded, allowing the correlation between error rate and GC content to be explored.

Furthermore, a similar algorithm can be applied but with sliding windows instead. The influence of overlapping windows can then be analysed. The same algorithm is also applied to the entire string, thereby exploring the effect of the overall GC content on the error rate.

4.15 Running digital sum analysis

The RDS presents an additional way of investigating the influence of GC content on error rate. The RDS can be calculated by iterating through a DNA string. The sum starts at a value of 0. If a G or a C is encountered, the sum is increased by 1. If another type of base is encountered, namely A or T, the sum is decreased by 1. This allows the analysis to be normalised at each position throughout the entire length of an input string, thereby delivering different results than the GC content analysis of Section 4.14.

In order to compare the effect of the sum on the error rate, the algorithm iterates through all the strings in a given dataset as well as the list of errors detected for that dataset. For each error, the

RDS at the position of the error is recorded. Thereafter, the error rate for different RDS values can be calculated.

This algorithm can be applied by calculating the RDS from the start of the DNA string or from the end of the DNA string. The start refers to the leftmost base when reading the DNA string as stored in the dataset, while the end refers to the rightmost base. The results can then be compared.

4.16 Problematic substring analysis

Certain combinations of bases may have increased error rates. To find all combinations of bases and the error rate over all the occurrences of each combination, a function is created. The function requires the alphabet of bases and the number of subsequent bases to examine. Furthermore, a file containing all the noisy strings in the dataset as well as a file containing all the original strings in the dataset is required, namely the clusters and centers files respectively. The file containing the errors detected in the dataset is also required. For DNA data storage, where the alphabet is always the same four bases, the alphabet may remain constant. By providing different files, the performance of different models regarding problematic substrings may be compared to that of the dataset.

The *product* function from the *itertools* Python module is used to find the Cartesian product of the bases – that is, all combinations of the specified length of all sequences of the input symbols. The function requires two input variables, namely the list of symbols and the desired length of the Cartesian product. The four DNA bases are given as input symbols, while the desired length may be varied to analyse different combinations of runs. The *itertools* module is found in the Python Standard Library [179].

The algorithm simultaneously iterates through the clusters and centers files simultaneously. For each cluster, the corresponding center is examined. The algorithm iterates through each base in the center. The bases after the center are sampled to identify the current substring. The sample size is defined by the number of subsequent bases to examine, as given by the user.

The algorithm then examines the list of errors for that particular string. All the errors with locations within the substring are identified. The number of errors are recorded, as well as the type of errors. Thereby the error rate within the substring and the rate of each type of error can be calculated.

After the algorithm iterates through all the strings, the total number of errors of each type within each different substring is recorded, as well as the number of occurrences of that substring. Thereby, error rates can be calculated over all the substrings of each type. This can be used to identify substrings with higher or lower error rates than the average error rate of the dataset.

Additionally, when ordering the substrings according to error rate, the number of substrings that are in the same order for each model as the substrings in the dataset is counted. This is done by ordering the substrings according to ascending error rate in the dataset and counting the number of substrings that have a larger error rate than the previous substring. This is done for each model.

4.17 Swap error analysis

When classifying errors as deletions, substitutions or insertions, a swap can be defined in three ways. An example string is used as explanation: AACGTG is transformed into AAGCTG. For this example, the string indices will start at position 0.

1. Two substitutions

Two substitutions directly after one another, where each character is substituted for the other character. For the example string above:

- The C in position 2 is substituted with a G
- The G in position 3 is substituted with a C

2. An insertion and a deletion

An insertion of a character, followed by a correct transmission, followed by a deletion of the same character that was inserted. For the example string above:

- A G is inserted before position 2
- The C in position 2 is transmitted successfully
- The G in position 3 is deleted

3. A deletion and insertion

A deletion of a character, followed by a correct transmission, followed by an insertion of the same character that was deleted. For the example string above:

- The C in position 2 is deleted
- The G in position 3 is transmitted successfully
- A C is inserted before position 4

To ensure that all swaps are detected, an algorithm that checks for all three classifications is used. The algorithm iterates through the list of errors produced by the Levenshtein Distance Error Identification algorithm for each cluster in the dataset. For the error list of each cluster, the three checks are performed by comparing the positions of the errors, as well as which bases were affected by each error. The two symbols that were swapped, as well as the manner in which the swap was recorded (e.g. two substitutions), are appended to a list. The number of swaps that

occurred in the dataset is now known. The percentage of errors that account for swaps can also be calculated by the following equation:

$$\% \text{ swaps} = \frac{2 \times \text{number of swaps}}{\text{total number of errors}} \times 100\% \quad (4-14)$$

Two times the number of swaps is taken as two single base errors make up one swap. The swaps in the output data of a model can be calculated in a similar manner.

4.18 Statistical metrics

The *pearsonr* function of the *SciPy* Python module is used to determine the correlation between two variables. The function uses equation (3-1) as described in Section 3.8.1 [180]. The *skew* function of the *SciPy* Python module is used to determine the skewness of a probability distribution. The function uses equation (3-2) as described in Section 3.8.2 [181].

The RMSE, RMSLE and NRMSE are calculated according to equations (3-4), (3-5) and (3-6) respectively. The *log* and *sqrt* functions of the *math* Python module are used to implement the equations. The *math* module is a part of the Python Standard Library [182].

4.19 Summary

In this chapter, the implementation of various algorithms was described. This includes using the Levenshtein Distance Error Identification algorithm to process the dataset. The implementation of Markov and Gilbert-Elliott models was also discussed. Furthermore, the creation of an IID model on the CNR dataset was discussed, as well as the training of the Memory model on the CNR dataset. Lastly, algorithms used to analyse the different models were described, including algorithms to detect error runs, bursts and error-free runs.

In Chapter 5, these algorithms are verified for accuracy. Thereafter, the results generated by the algorithms presented in this chapter are discussed.

CHAPTER 5 VERIFICATION

This section covers the verification of each of the algorithms described in Chapter 4. Each algorithm is verified via specific test cases. The test cases are designed to cover a variety of occurrences as well as outliers that might induce software errors. Additionally, the test cases serve as examples of the use of each algorithm. Where possible, results are verified by comparing the outputs of multiple different algorithms.

5.1 Levenshtein Distance Error Identification

5.1.1 Individual examples

In order to evaluate the accuracy of the Levenshtein Distance Error Identification algorithm, various example strings are examined. The results are depicted in Figure 5-1. In each case, an original string is compared to a noisy string with certain characteristics. The errors within each string are indicated, as well as the positions of the errors. This is compared to the output given by the algorithm below.

In Figure 5-1 a), the algorithm is evaluated on a string with multiple insertion errors at the start of the string. The algorithm output matches the format given in Section 4.1. The first value indicates the type of error. The second and third values indicate the position of the error in the noisy string and original string respectively. Thereafter, the affected bases are given. As seen in the algorithm output below, the position of the insertion error in the original string is the position before which the insertion occurs. Furthermore, it is seen that the algorithm output matches the perceived errors by indicating three consecutive insertion errors at the start of the string.

['i', 0, 0, 'C']

['i', 1, 0, 'C']

['i', 2, 0, 'G']

In Figure 5-1 b), the algorithm is evaluated on a string with multiple deletion errors at the start of the string. It is seen that the algorithm output matches the perceived errors by indicating three consecutive deletion errors at the start of the string. As seen in the algorithm output below, the position of the deletion error in the noisy string is the position at which the corresponding base would have been if the deletion had not occurred.

['d', 0, 0, 'A']

['d', 0, 1, 'A']

['d', 0, 2, 'A']

In Figure 5-1 c), the algorithm is evaluated on a string with multiple substitution errors at the start of the string. It is observed that the algorithm output matches the perceived errors by indicating two consecutive substitution errors at the start of the string. The base in the original string is given first, followed by the base in the erroneous string. The algorithm output is given below.

['s', 0, 0, 'A', 'G']

['s', 1, 1, 'A', 'C']

In Figure 5-1 d), the algorithm is evaluated on a string with multiple insertion errors at the end of the string. It is seen that the algorithm output matches the perceived errors by indicating three consecutive insertion errors at the end of the string. The algorithm output is given below.

['i', 6, 6, 'A']

['i', 7, 6, 'A']

['i', 8, 6, 'A']

In Figure 5-1 e), the algorithm is evaluated on a string with multiple deletion errors at the end of the string. It is seen that the algorithm output matches the perceived errors by indicating two consecutive deletion errors at the end of the string. The output of the algorithm is given below.

['d', 4, 4, 'G']

['d', 4, 5, 'T']

In Figure 5-1 f), the algorithm is evaluated on a string with multiple substitution errors at the end of the string. It is seen that the algorithm output matches the perceived errors by indicating three consecutive substitution errors at the final positions of the string. The algorithm output is given below.

['s', 4, 4, 'G', 'A']

['s', 5, 5, 'T', 'A']

['s', 6, 6, 'T', 'A']

positions 1, 2 and 3 of the string. Furthermore, two deletion errors are indicated at positions 5 and 7 of the original string. When comparing this to the errors indicated in Figure 5-1, a discrepancy is noted as the errors in the figure are given to be at positions 6 and 7. Upon further examination, it is seen that the base 'T' occurs at both positions 5 and 6 of the original string, and deleting either of these bases will result in the same noisy output string. In the dataset, the positions of the errors are not given. Therefore, in such instances where multiple different errors may result in the same output string, it is not known which of the errors occurred during the sequencing process. Therefore, the algorithm output is accurate, as the errors detected by the algorithm result in the same noisy string. The algorithm output is given below.

['s', 1, 1, 'A', 'G']

['s', 2, 2, 'A', 'G']

['s', 3, 3, 'C', 'G']

['d', 5, 5, 'T']

['d', 6, 7, 'A']

In Figure 5-1 h), the algorithm is evaluated on a string with multiple consecutive errors of different types, simulating an error run. It is seen that the algorithm output matches the perceived errors by indicating insertion errors at positions 0 and 2 of the noisy string. Furthermore, the algorithm output matches the substitution errors perceived at positions 7 and 8 of the noisy string. Lastly, the three consecutive deletion errors at the end of the string, right after the substitution errors, are also correctly indicated. The output of the algorithm is given below.

['i', 0, 0, 'A']

['i', 2, 1, 'T']

['s', 7, 5, 'T', 'G']

['s', 8, 6, 'T', 'G']

['d', 9, 7, 'A']

['d', 9, 8, 'A']

['d', 9, 9, 'C']

Figure 5-1 i) depicts a string with a swap error – two consecutive bases that are swapped with each other. In Figure 5-1 j), the algorithm is evaluated on the string. Since the algorithm only detects insertion, deletion and substitution errors, the algorithm cannot detect a swap error. However, the algorithm is still capable of perceiving swap errors as a combination of two errors, namely an insertion and a deletion. In this case, the algorithm detects the swap as an insertion of

the base 'G' at position 3 of the noisy string and a deletion of the base 'G' at position 4 of the original string. This results in the same noisy output string as if a swap error was introduced. The algorithm output is given below.

['i', 3, 3, 'G']

['d', 5, 4, 'G']

The algorithm was also tested on an example from the dataset. A noisy string was selected from the clusters.txt file and compared to the corresponding string from the centers.txt file. It is seen that the algorithm gives the correct output, indicating a substitution at position 17, insertions at positions 31 and 32 of the noisy string, and a deletion at position 78 of the original string. The output of the algorithm is given below.

['s', 17, 17, 'C', 'T']

['i', 31, 31, 'G']

['i', 32, 31, 'T']

['d', 80, 78, 'G']

The strings and the errors that occurred within the strings are depicted in Appendix H.

5.1.2 Verify error identification using Levenshtein distance

While certain examples for the Levenshtein Distance Error Identification algorithm were examined individually, it is useful to evaluate the algorithm across a wider range of examples. Therefore, the algorithm is evaluated on the entire dataset. In each instance, the number of errors identified for that combination of strings is compared to the Levenshtein distance calculated by the *Levenshtein* Python module. Additionally, this value is compared to the value provided by the *editDistance* function in MATLAB. Since MATLAB toolboxes are verified to be accurate, by comparing the results of the Python code to the results obtained via the MATLAB function, the accuracy of the Python code may be verified. The results showed no discrepancies, therefore, for each example in the dataset, the Levenshtein Distance Error Identification algorithm identified the correct number of errors.

It is verified that the entire dataset is examined by counting the number of strings. This equates to 279709 strings, which corresponds to the number of strings in the dataset. This indicates that the Levenshtein Distance Error Identification algorithm identified the correct number of errors for the entire dataset. Furthermore, when also taking the individual examples that were examined into account, the algorithm is shown to perform accurately.

Additionally, the accuracy of the *Levenshtein* Python module was verified. The module may thus be used in conjunction with other Python code throughout the rest of the implementation.

5.2 Detecting errors in a dataset

To ensure that every entry in the dataset is examined, the total number of original strings and the total number of noisy strings handled are counted. The total number of original strings that are examined is found to be 10000, which corresponds to the number of strings in the *centers.txt* file. The total number of noisy strings that are examined are found to be 279709, which corresponds to the number of strings in the *clusters.txt* file. This indicates that the Levenshtein Distance Error Identification algorithm is applied to every string in the dataset.

5.3 Strings of outcomes

In order to determine if the algorithm derives the correct string of outcomes, an example string is selected from the dataset. The list of errors for the example string is found using the Levenshtein Distance Error Identification algorithm, whereafter the string of outcomes is derived. The selected example string features each type of error, as well as consecutive errors. The string of outcomes matches the positions of the erroneous and error-free bases in the example string. This indicates that the algorithm is accurate. The results are summarised in Appendix I.

5.4 Strings of outcomes to train the independent Gilbert-Elliott models

To determine if the correct strings of outcomes are derived for each type of error, the same example string as in Section 5.3 is used as it features each type of error as well as consecutive errors. The string of outcomes as found in Section 5.3 is used to derive the strings for insertion, deletion and substitution errors. The results are examined to ensure that each string shows the correct errors. The strings of outcomes match the positions of the erroneous and error-free bases in the example string, indicating that the algorithm is accurate. The results are summarised in Appendix J.

5.5 Deriving model parameters

In order to verify that the model parameters are correctly derived, the format of the output is examined. In Section 4.6, the format of the matrices containing the model parameters was

described. The matrix dimensions must match the desired number of states and outputs. It is verified that the algorithm provides the correct output format.

The accuracy of the derived model parameters is verified by analysing the model itself. This is done by comparing the model output to the dataset. Comparison is done on statistics such as error rate, but also includes the analysis of how the model handles homopolymers, GC-content, RDSs, swap errors and problematic substrings.

5.6 Data statistics

The calculation of the overall error rate is verified using the dataset as reference. This is done by calculating the Levenshtein distance between each noisy string and the corresponding original string. As the Levenshtein distance is equal to the number of errors, the total number of errors can be found. This value is divided by the total number of bases in the dataset to find the error rate.

The *distance* function used to calculate the Levenshtein distance allows the weights of each type of error to be set. The weights of insertion and deletion errors are set to 0 and the weight of substitution errors is set to 1. Thus, by again finding the distance between each noisy string and the corresponding original string, the total number of substitution errors can be found. Thereby, the substitution error rate can also be found. Likewise, by setting the weight of insertion errors to 1 and that of deletions and substitutions to 0, the insertion error rate can be found. Lastly, by setting the weight of deletion errors to 1 and the weights of the other types of errors to 0, the deletion error rate can be found.

Example strings are used to verify that the probability distributions for different types of insertion, deletion and substitution errors are correctly calculated. After running the algorithm on the example strings, it is determined that the algorithm calculates the rate of each type of insertion, deletion or substitution error correctly. The errors contained within the example strings are depicted in Appendix K.

The rate of each type of substitution should add up to the total rate of substitution errors. Additionally, the rate of each type of insertion should add up to the total rate of insertion errors. Likewise, the rate of each type of deletion error should add up to the total rate of deletion errors. Lastly, the total error rate should equal the sum of the rate of deletion errors, substitution errors and insertion errors.

The algorithm successfully fulfils all of the abovementioned conditions, thereby confirming its accuracy for the test cases.

5.7 Generating model outputs

The algorithm that provides the emission and transition matrices is verified in Section 5.5. Therefore, these matrices are assumed to be accurate and it is only confirmed that they are correctly imported into the Python file.

Additionally, the length of the string of outcomes is confirmed for an example. For an error-free string, or a string that only includes substitution and deletion errors, the string of outcomes should have the same length as the input string provided to the model. For each insertion error, the length of the string of outcomes increases by one.

Lastly, the file in which the strings generated by the model are stored, is checked. The file should contain the correct number of strings, and the strings should be stored in the correct format.

These checks are performed for both the outputs of the Markov model and the independent Gilbert-Elliott models.

5.8 Creating an IID model

The same function that was verified in Section 5.6 was used to calculate the statistics required to create the IID model. This includes the error rate, probability distribution for the different types of errors as well as the probabilities for different types of insertions and substitutions. Therefore, the computation of these statistics is verified. The algorithm is run once again on the model outputs to confirm that the probabilities were correctly translated to the model outputs. Additionally, the output file is checked to contain the correct number of generated strings and that the strings are stored in the correct format.

5.9 Training the Memory model on the CNR dataset

The file in which the strings generated by the model are stored, is checked in order to ensure that the model produced the desired output strings. The file should contain the correct number of strings and the strings should be stored in the correct format. The accuracy of the implemented model is verified by comparing the model outputs to the dataset, which shall be done in Chapter 9.

5.10 Comparison of models

5.10.1 Importing data

In order to ensure that the correct data is imported, the number of strings imported from the dataset, as well as each model, is counted. In each case, the same number of strings are imported, thereby indicating that the correct data is used for the comparison of the different models.

5.10.2 Error-free strings

Two additional checks are performed to confirm the correct number of error-free strings provided by each model. Firstly, the original and noisy strings are checked and any cases where the strings are the same are counted as error-free strings. Secondly, the Levensthein distance for each original string and corresponding noisy string is found. If the Levensthein distance is 0, it indicates an error-free string. Both these results match the results originally found by counting the number of errors in each string. This indicates that the number of error-free strings was found accurately.

5.10.3 Average string length

In order to verify the algorithm that finds the average string length, the average string length of a list of strings of which the average length is known is found. Each string in the centers.txt file is of length 110, therefore the average length is known as 110. The algorithm is applied to the strings in this file. The algorithm gives a matching answer of 110, confirming the accuracy of the algorithm.

5.10.4 General statistics

The function to calculate statistics from the dataset that was verified in Section 5.6 was used to calculate the general statistics. As this function is already verified, the statistics of the data set and of each model are assumed to be calculated accurately.

5.10.5 Error rate at each position

This algorithm only counts the number of errors at each position and divides those values by the total number of strings to find the error rate at each position, as the position of each error is already identified via the Levenshtein Distance Error Identification algorithm. By using a small example data set, the validity of this algorithm can be examined.

The example data set as well as the results for this subsection are shown in Appendix L. It is found that the algorithm correctly identifies the error rate at each position, matching the values calculated manually. This indicates that the algorithm performs accurately for this small test set.

The accuracy of the algorithm for the entire dataset can be verified by averaging the error rate, as calculated by the algorithm, over all the positions and comparing that to the error rate found by the algorithm of Section 4.2. The values match, indicating that the algorithm performs accurately over a larger dataset.

5.10.6 Error runs

In order to determine the efficacy of the algorithm to detect error runs, multiple examples are analysed. The example strings, which are depicted in Figure 5-2, are designed to contain various edge cases. The locations of errors are indicated, as well as the length of each error run. In each case, the algorithm is able to detect a run of consecutive errors and also indicates runs of specific types of errors.

Figure 5-2 a) depicts a string with a run of three insertion errors at the start of the string. The algorithm detects a run of insertion errors with a run length of three, as well as a run of errors of length three.

In Figure 5-2 b), a run of three deletions occurs at the start of the string. The algorithm successfully detects a run of three deletions and the corresponding run of three errors.

In Figure 5-2 c), a run of two substitutions occurs at the start of the string. Once again, the algorithm successfully detects a run of two substitutions and the corresponding run of two errors.

A string with a run of three insertions at the end is depicted in Figure 5-2 d). The algorithm successfully detects the run of three insertions as well as a run of three errors.

Figure 5-2 e) depicts a run of two deletions at the end of a string. The algorithm accurately detects a run of two deletions and a run of two errors.

Figure 5-2 f) depicts a string with multiple consecutive insertions at the end. The algorithm successfully detects a run of three substitutions, as well as a run of three errors.

Figure 5-2 g) depicts a string with multiple error runs. The algorithm successfully detects a run of three substitutions as well as a run of two deletions. Corresponding error runs of lengths three and two are recorded.

In Figure 5-2 h) a run containing different types of errors is depicted. The algorithm successfully detects a run of two insertion errors, a run of two substitution errors and a run of three deletion errors. Furthermore, the algorithm is capable of detecting that the substitutions and deletions combine into a longer error run of five errors. Lastly, a run of two errors is also detected, thereby accounting for the insertion errors.

Legend		
Deletion	Insertion	Substitution

a) Multiple insertions at the start of the string

Position in original string		0	1	2	3	4	5		
Base in original string		A	A	A	C	G	T		
Base in noisy string	C	C	G	A	A	A	C	G	T
Position in noisy string	0	1	2	3	4	5	6	7	8
Run Length	3								

b) Multiple deletions at the start of the string

0	1	2	3	4	5
A	A	A	C	G	T
			C	G	T
			0	1	2
3					

c) Multiple substitutions at the start of the string

0	1	2	3	4	5	5
A	A	A	C	G	T	T
G	C	A	C	G	T	A
0	1	2	3	4	5	5
2						

d) Multiple insertions at the end of the string

0	1	2	3	4	5			
A	A	A	C	G	T			
A	A	A	C	G	T	A	A	A
0	1	2	3	4	5	6	7	8
						3		

e) Multiple deletions at the end of the string

0	1	2	3	4	5
A	A	A	C	G	T
A	A	A	C		
0	1	2	3		
				2	

f) Multiple substitutions at the end of the string

0	1	2	3	4	5	6
A	A	A	C	G	T	T
A	A	A	C	A	A	A
0	1	2	3	4	5	6
				3		

g) Multiple error runs in the same string

0	1	2	3	4	5	6	7	8	9
A	A	A	C	G	T	T	A	A	C
A	G	G	G	G	T			A	C
0	1	2	3	4	5			6	7
			3			2			

h) A run containing different types of errors

	0	1	2	3	4	5	6	7	8
	A	A	C	G	T	T	A	A	C
G	T	A	A	C	G	G	G		
0	1	2	3	4	5	6	7		
2						5			

i) An example of a swap error

0	1	2	3	4	5	6
A	A	A	C	G	T	T
A	A	A	G	C	T	T
0	1	2	3	4	5	6
	1			1		

Figure 5-2: Examination of error runs identified by the model

A swap error is depicted in Figure 5-2 i). Since a swap is classified by the algorithm as an insertion followed by a successful transmission and then a deletion, it is detected as two individual errors and not an error run.

In each example that was analysed, the algorithm correctly detected error runs. Runs of individual error types, as well as runs of errors in general, were detected. It was also shown that the algorithm can detect multiple runs in the same string.

5.10.7 Error bursts

In order to determine the efficacy of the algorithm to detect error bursts, multiple examples which are designed to contain edge cases are analysed. The example strings are depicted in Figure 5-3 and in each case the locations of errors are indicated. When a burst occurs, the burst length, number of errors within that burst and the starting position of the burst are indicated.

Figure 5-3 a), b) and c) depict strings with consecutive errors at the start of each string. These examples each contain a different type of error. In each case, the algorithm successfully detects the starting position of the run as position 0 of the string. The correct burst length and number of errors within the burst are also detected.

In Figure 5-3 d), a run of three insertion errors occurs at the end of the string. The algorithm once again successfully determines the starting position of the burst as well as the number of errors and burst length. Similarly, a run of two deletions is depicted at the end of the string in Figure 5-3 g). This burst is also correctly detected.

A swap error is depicted in Figure 5-3 e). Since a swap is classified by the algorithm as an insertion followed by a successful transmission and then a deletion, it is detected as a burst of two errors. The starting position of the burst is successfully detected, as well as the length of the burst which is three. The burst length is three, as three bases are affected, including the inserted and deleted bases and the successfully transmitted base.

Figure 5-3 f) depicts a string with multiple error runs in close proximity. The algorithm successfully classifies this as an error burst. The correct starting position of the burst is identified. Furthermore, the burst length is correctly identified as 7, while the number of errors within the burst is found to be 5.

In Figure 5-3 h) a string that contains multiple bursts is depicted. The starting positions of both bursts are successfully detected, as well as the length of each burst. The number of errors in each burst is also successfully determined.

Legend

Deletion

Insertion

Substitution

a) Multiple insertions at the start of the string

Position in original string		0	1	2	3	4	5		
Base in original string		A	A	A	C	G	T		
Base in noisy string	C	C	G	A	A	A	C	G	T
Position in noisy string	0	1	2	3	4	5	6	7	8
Burst Length	3								
Errors per burst	3								
Starting position of burst	0								

b) Multiple deletions at the start of the string

0	1	2	3	4	5
A	A	A	C	G	T
			C	G	T
			0	1	2
3					
3					
0					

c) Multiple substitutions at the start of the string

0	1	2	3	4	5
A	A	A	C	G	T
G	C	A	C	G	T
0	1	2	3	4	5
2					
2					
0					

d) Multiple insertions at the end of the string

0	1	2	3	4	5
A	A	A	C	G	T
A	A	A	C	G	T
0	1	2	3	4	5
			6	7	8
			3		
			3		
			6		

e) An example of a swap error

0	1	2	3	4	5	6
A	A	A	C	G	T	T
A	A	A	G	C	T	T
0	1	2	3	4	5	6
				3		
				2		
				3		

f) Multiple groups of errors in close proximity

0	1	2	3	4	5	6	7	8	9
A	A	A	C	G	T	T	A	A	C
A	G	G	G	G	T			A	C
0	1	2	3	4	5			6	7

g) Multiple deletions at the end of the string

0	1	2	3	4	5
A	A	A	C	G	T
A	A	A	C		
0	1	2	3		
				2	
				2	
				4	

h) Multiple burst errors in the same string

0	1	2	3	4	5	6	7	8	9
A	A	T	A	A	C	G	G	G	G
0	1	2	3	4	5	6	7	8	9

Figure 5-3: Examination of error bursts identified by the model

In each example that was analysed, the algorithm correctly detected error bursts. The starting position and length of each burst were successfully found, as well as the number of errors within each burst. It was also shown that the algorithm can detect multiple bursts in the same string.

5.10.8 Error-free runs

In order to determine whether the algorithm identifies error-free runs correctly, an example string from the dataset is used. The list of outcomes for the example string, along with the length of the error-free substrings, is depicted in Figure 5-4. Each error-free substring is indicated in a different colour. Between positions 31 and 32 of the string of outcomes, an error-free string of length 0 is found. This is due to the fact that there are two consecutive errors, meaning that there are 0 consecutive error-free bases between them.

Position	0	1	2	3	4	5	6	7	8	9	10	...
String of outcomes	t	t	t	t	t	t	t	t	t	t	t	...
Length of error-free substring	17											

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	...
t	t	t	t	t	t	s	t	t	t	t	t	t	t	t	t	t	...
17							13										

28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	...
t	t	t	i	i	t	t	t	t	t	t	t	t	t	t	t	t	...
13			0		47												

45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	...
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...

62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	...
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...
47																	

79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	...
t	d	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...
47		31															

96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	
31																

Figure 5-4: Detection of error-free runs on an example string

Furthermore, an example string that is error-free is used. The algorithm correctly identifies an error-free run that is the same length as the input string. When applying the algorithm to the entire dataset, the number of error-free runs that are the same length as the input strings of the dataset should equal the number of error-free strings in the dataset, as found in Section 4.12.

In all three cases, the algorithm results correspond to the correct values, thereby confirming that the algorithm is accurate.

5.11 Homopolymer analysis

To verify the capability of the algorithm to detect homopolymers within DNA strings, multiple examples are used. The examples are depicted in Figure 5-5. In each case, the string is given. Homopolymers are indicated on the string, while the length of each homopolymer is given below the string.

In Figure 5-5 a), a string with multiple homopolymers of the same base is given. The algorithm successfully detects three distinct homopolymers of base 'A' and gives the lengths as three, two and seven respectively.

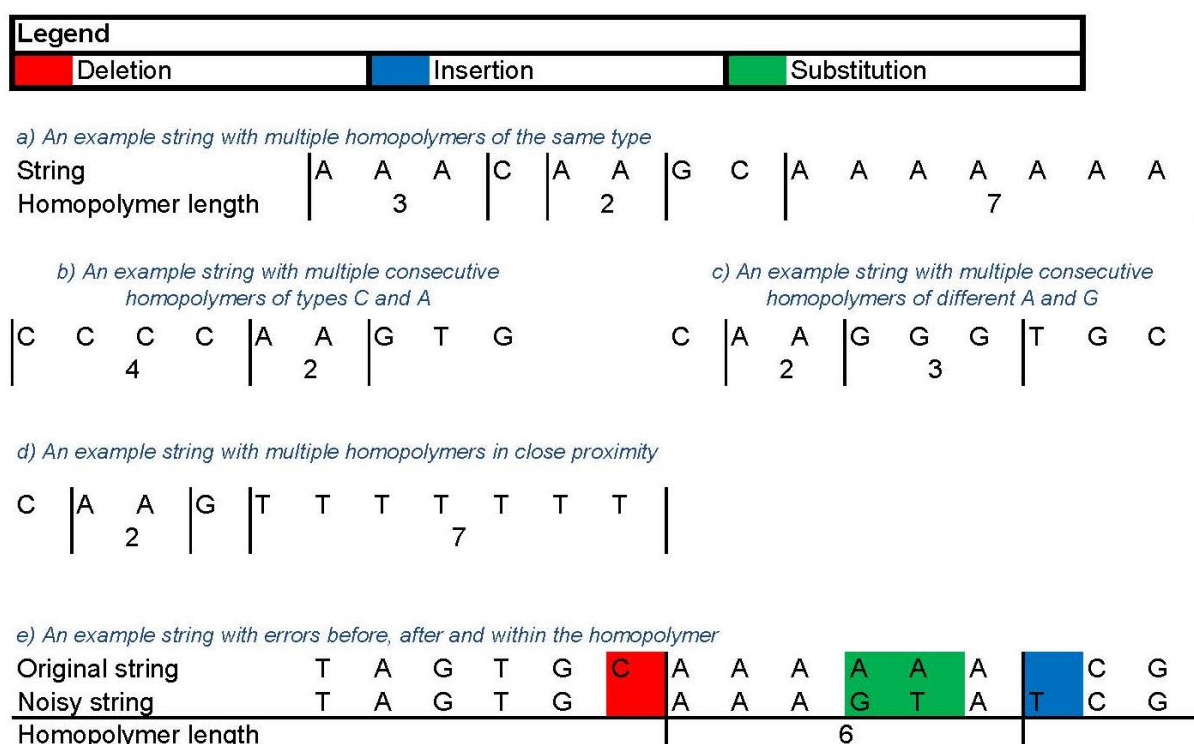


Figure 5-5: Example strings for homopolymer analysis

Figure 5-5 b) depicts a string with multiple consecutive homopolymers that consist of different bases. The algorithm successfully detects a homopolymer of base 'C' and length four. Furthermore, a homopolymer of base 'A' and length two is also detected accurately.

Figure 5-5 c) similarly depicts a string with multiple consecutive homopolymers that consist of different bases. The algorithm successfully detects a homopolymer of base 'A' and length two and another homopolymer of base 'G' and length three.

Figure 5-5 d) depicts another string with multiple homopolymers. The algorithm successfully detects a homopolymer of base 'A' and length two and another homopolymer of base 'T' and length seven.

Figure 5-5 e) is used to verify the ability of the algorithm to detect errors within, before and after a homopolymer. Original and noisy strings are given. The positions of the errors are also indicated. It is seen that the original string contains a homopolymer of base 'A' and length six. The homopolymer contains two errors. Furthermore, an error precedes the homopolymer and another error occurs right after the homopolymer. The algorithm successfully detects the homopolymer, as well as the errors within, before and after the homopolymer.

In Figure 5-5 a) through d), homopolymers of each base are represented. Furthermore, consecutive homopolymers are also present. Homopolymers of the shortest possible length, namely two, are present. A homopolymer of a comparatively large length, namely seven, is also present. Homopolymers at the start and end of strings are also incorporated. In this manner, the algorithm to detect homopolymers is verified over a variety of test cases. Lastly, via the example in Figure 5-5 e), it is verified that the algorithm can detect errors within, before and after a homopolymer.

5.12 GC content analysis

In order to test the GC content analysis algorithm, an example string is selected from the dataset. The example string is depicted in Appendix M in Supplementary Figure M-1, along with the locations of errors within the string. The string is divided into substrings of length 6 and the GC content of each substring is found.

Supplementary Table M-1 in Appendix M summarises the results. The number of occurrences of each different possible GC content value of a substring is counted. Since the length of each substring is six, GC content values can range between 0 and 6. The number of bases within substrings of a certain GC content is found with:

$$N_{bases} = N_{occurrences} \times L_{substring} \quad (5-1)$$

$L_{substring}$ indicates the length of the substring, which is taken as 6 in this algorithm. $N_{occurrences}$ represents the number of occurrences of substrings with a specific GC content value.

The number of errors within substrings with a specific GC content value is also counted. Finally, the error rate for a substring with a specific GC content value can be calculated by dividing the number of errors by the number of bases. This can be done for each GC content value from 0 to 6.

Additionally, the overall GC content for the string is found to be 60. Including the final two bases of the string results in a string length of 110, therefore the percentage GC content is found as 54.5%. The string contains 4 errors, equating to a 0.036 error rate within this string.

The algorithm calculates the correct values for this example string. This algorithm is then applied to the entire testing set, for which each base cannot be individually verified. To validate the results, the average error rate found by this algorithm is calculated over all the strings and is compared to the error rate found with the algorithm from Section 4.2. Both algorithms provide equal values, giving an indication that this algorithm performs correctly when applied to a large number of strings.

5.13 Running digital sum analysis

In order to evaluate the algorithm that examines the effect of the RDS on the occurrence of errors, a string from the dataset is used as an example. The algorithm performs accurately, and the results are summarised in Appendix N. Supplementary Figure N-1 depicts the example string, as well as the location of any errors. At each position of the string, the current RDS is noted.

In Section 5.12, the same example string was used. In that section, the GC content of the string was found as 60 out of 110 bases. This indicates that there are 50 non-GC bases in the string. Therefore, there are 10 more GC bases than non-GC bases. This corresponds to the RDS value at the last position of the string, as shown in Supplementary Figure N-1. By comparing the results of this algorithm with the results of the algorithm described in Section 4.14, it can be verified that the algorithm correctly determines the RDS.

The analysis is summarised in Appendix N. In Supplementary Table N-1, the number of occurrences of each different RDS value is counted. The number of errors that occur when the RDS is a specific value is also noted. This can be used to calculate the error rate when the RDS is a specific value by dividing the number of errors by the number of occurrences of that RDS value. The calculated error rates are also depicted in the table.

In all instances of the examined string, the algorithm correctly calculates the RDS. The algorithm also correctly identifies if an error occurs at a specific position, and records the RDS at that position. Thereby, the algorithm can determine the error rate at different RDS values.

5.14 Problematic substring analysis

5.14.1 Cartesian product

In order to verify that the Cartesian product of all the bases is correctly calculated, the length of the Cartesian product can be calculated and compared to the length of the output. The length of the Cartesian product can be calculated as follows:

$$Len_{cartesian} = k^n \quad (5-2)$$

k represents the alphabet size, while n represents the length of the string. In all cases, $k = 4$ as there are four possible bases in the DNA alphabet.

When using a small string length, the Cartesian product can be calculated manually. The results can be compared to that provided by the algorithm. For a string length of 2, the Cartesian product of the four-letter DNA alphabet is as follows:

[('A', 'A'), ('A', 'C'), ('A', 'G'), ('A', 'T'),
 ('C', 'A'), ('C', 'C'), ('C', 'G'), ('C', 'T'),
 ('G', 'A'), ('G', 'C'), ('G', 'G'), ('G', 'T'),
 ('T', 'A'), ('T', 'C'), ('T', 'G'), ('T', 'T')]

In this case, the algorithm once again performs correctly. Therefore, when considering that the algorithm provides Cartesian products of the correct length, it is assumed that the algorithm is also accurate for longer string lengths.

5.14.2 Problematic substring analysis

Tables summarising the results of this subsection are available in Appendix O. In order to examine the validity of the algorithm that identifies problematic substrings, a set of example strings is analysed. Substrings of length 2 are used as only 16 possible substrings exist, therefore the error rate of each substring can be calculated by hand and depicted. The example strings are given in Figure 5-6. The errors within each string are indicated, and the substrings are given below each string. The substrings are shown in two rows. The top row contains all the substrings that start at even positions of the string, while the bottom row contains all the substrings that start at the odd positions of the string.

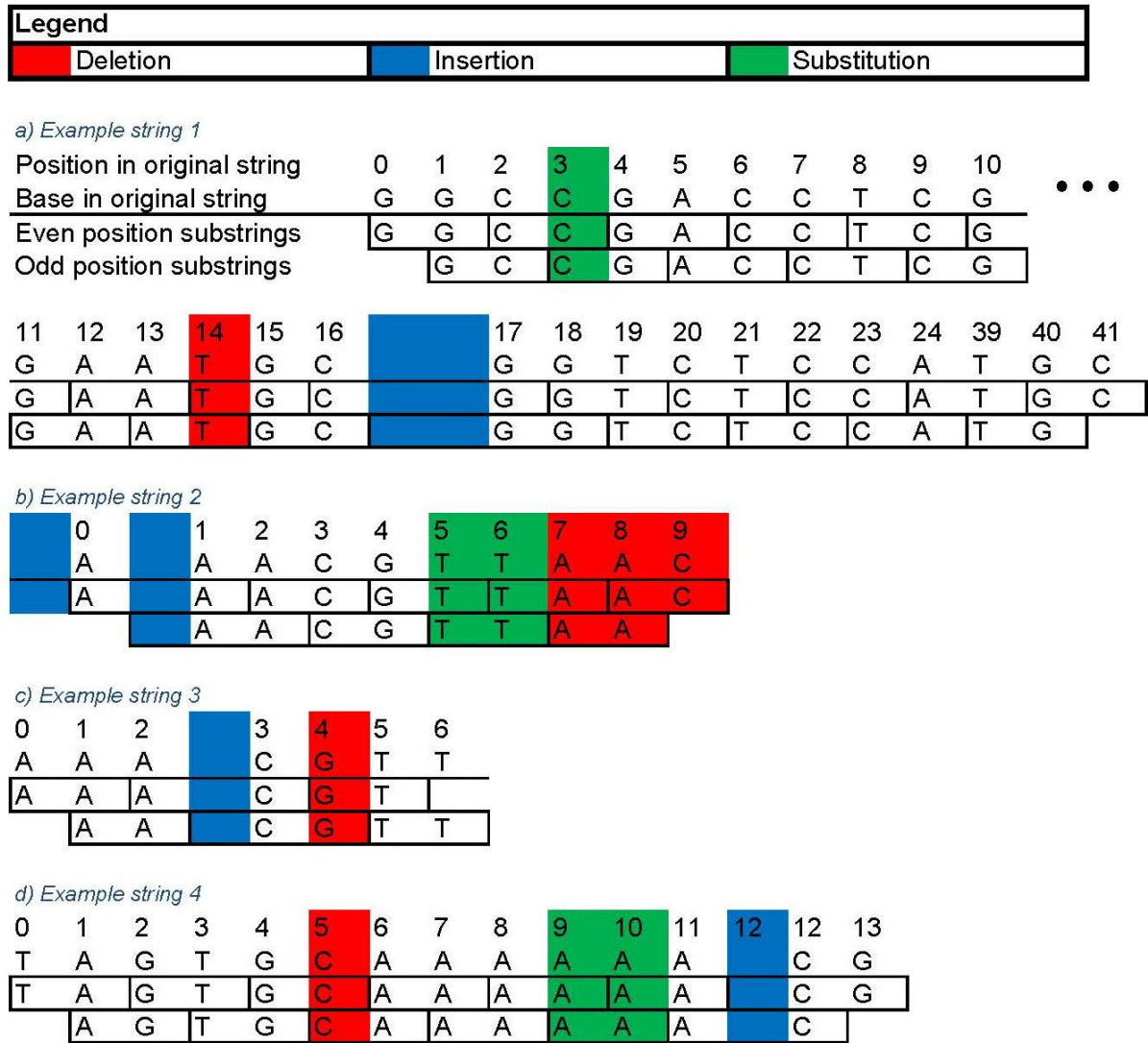


Figure 5-6: Example strings for the analysis of problematic substrings of length two

The number of occurrences of each substring is counted over all of the example strings. Letting n represent the number of bases in a substring and m the number of occurrences of that substring, the total number of bases within that substring can be found as:

$$num_{bases} = n \times m \quad (5-3)$$

Furthermore, the total number of errors that occur within each substring over all of the example strings can be counted. By dividing this value by the number of bases contained by each substring, the error rate of each substring can be found. The results for the set of example strings are depicted in Supplementary Table O-1.

Additionally, the errors that occur at the first position of each substring can be counted over all the example strings. The number of errors of each type – namely insertions, deletions and substitutions – can also be counted. The results are shown in Supplementary Table O-2.

As the number of errors that occur at the first position of each substring is known, the error rates can be calculated for the first position of each substring. The number of errors counted for each substring is divided by the number of occurrences of each substring. The rates for insertion, deletion and substitution errors can also be calculated in the same manner. The results are shown in Supplementary Table O-3.

Similarly to the above calculations, the number of errors that occur can be counted for each position of each substring. Since the substrings used in this example are of length 2, each substring only has two positions where errors can occur. The number of errors that occur at the second position of each substring is summarised in Supplementary Table O-4. The errors can also be calculated and the results are summarised in Supplementary Table O-5.

For the example strings above, the algorithm is able to correctly count the number of occurrences of each possible substring of length 2. The total number of errors within each substring can also be counted successfully, allowing the algorithm to calculate the error rate of each substring. Furthermore, the number of errors that occur at each position of each substring can be counted, allowing the algorithm to find the error rate at each position of each substring. This can also be done for each individual type of error. The algorithm is capable of successfully calculating these values for a set of multiple example strings.

In the examples above, the algorithm is shown to perform accurately for substrings of length 2. An additional example string is used to evaluate the performance of the algorithm for longer substrings. An example string is depicted in Figure 5-7. The locations of errors are given. The string is divided into substrings of length 6, which are also depicted.

The number of occurrences of each substring can once again be counted, similar to the previous example. Furthermore, the number of errors that occur within each substring over all the occurrences of that substring can be found. The error rate can then be calculated for each substring. These values are summarised in Supplementary Table O-6. Due to the large number of possible substrings of length 6, only substrings that contain errors according to the example string from Figure 5-7 are depicted in Supplementary Table O-6.

The algorithm is capable of correctly identifying substrings of length 6. Furthermore, the algorithm can accurately compute the error rate of each substring found in the example. The examples shown indicate that the algorithm performs accurately for substrings of lengths 2 and 6.

Furthermore, the algorithm performs accurately when presented with an example set containing multiple strings.

Legend	
 Deletion	 Insertion
	 Substitution
Position in original string	0 1 2 3 4 5 6 7
Base in original string	G G C C G A C C
Substrings	G G C C G A C C T
	G C C G A C C C C
	G C C G A C C T C
	C C C A C C T C G
	C C C A C C T C G G
	C C C A C C T C G G A
	C C C A C C T C G G A
	C C C A C C T C G G A
	C C C A C C T C G G A
	C C C A C C T C G G A

Figure 5-7: Example string for the analysis of problematic substrings of length six

5.15 Swap error analysis

To determine if this algorithm is capable of detecting swaps, the example string from Figure 5-1 i) is used. The algorithm should detect a swap at positions 3 and 4 of the string, with the affected bases being a 'C' and a 'G'. The algorithm successfully detects the swap.

5.16 Summary

In this chapter, the implemented code was verified for its accuracy. This includes the Levensthein Distance Error Identification algorithm that is used to process the dataset. Functions to calculate various statistics and find error runs and bursts were also validated. Additionally, functions that can analyse different aspects of the dataset, such as the errors produced by homopolymers and variations in GC content, were also verified.

Additionally, the different models that were trained were verified to produce output strings of the correct format. This includes the proposed Markov and Gilbert-Elliot models, as well as the versions of the IID and Memory models that have been trained on the CNR dataset. The efficacy of the models can now be tested.

CHAPTER 6 ANALYSIS OF MARKOV MODELS

Different Markov models were created and analysed in order to determine which model performs most accurately in terms of generating error bursts, error runs and error-free runs. The emission and transition matrices of all the models mentioned in this chapter are given in Appendix P.

6.1 Model parameters

The *hmmtrain* function used to train the Markov models allows various parameters to be configured. Different Markov models were trained using different combinations of parameters. All the models were trained using the Baum-Welch algorithm. The following parameters were varied in order to create different Markov models:

- Number of states
- Training set size
- Tolerance
- Maximum iterations

When the values of the transmission and emission matrices differ less than the specified tolerance value, convergence is achieved. The maximum iterations value limits the number of iterations – if this number of iterations has been completed but convergence has not been determined through tolerance, the training algorithm stops.

A training set *X* containing 13486 strings was uniformly selected from the dataset. The remaining strings in the dataset were added to the test set. A smaller training set *Y* was selected uniformly from training set *X*. Set *Y* contains 1349 strings.

6.1.1 Independent Gilbert-Elliott models

As described in Section 4.6.2, three independent Gilbert-Elliott models were trained. Each of these models was trained to represent a different type of error. All three of these models were trained using a tolerance of 10^{-3} and 1000 maximum iterations. These models were trained using training set *Y*. The combined model consisting of these three models shall be referred to as the GE model. The transition matrices of the three models are given in Appendix P. The three models share the same emission matrix, which is given in Supplementary Table P-4.

6.1.2 Markov models

The first Markov model trained is referred to as model 7MMLS. This model has 7 states and makes use of training set *X*. The tolerance is set to 10^{-3} and the maximum iterations to 1000.

A second Markov model is trained using the same training set and number of states, but with a decreased tolerance and increased maximum iterations of 10^{-4} and 5000 respectively. This model is referred to as model 7MMIT. By comparing this model to 7MMLS, the impact of adjustments in the tolerance and maximum iterations values may be assessed.

In order to assess the impact of the training set size, the 7MM model was trained using training set Y . The model has 7 states and tolerance and maximum iteration values of 10^{-3} and 1000 respectively, allowing for a comparison with the 7MMLS model.

To compare the impact of the number of states, models with 4, 6, 9 and 11 states were trained. These models are known as the 4MM, 6MM, 9MM and 11MM models respectively. These models were all trained using training set Y and with the same tolerance and maximum iteration values of 10^{-3} and 1000 respectively.

6.1.3 Summary of model parameters

Table 6-1 summarises the parameters used to train each of the models. The three independent Gilbert-Elliot models each used the same training parameters.

Table 6-1: Summary of model parameters

Identifier	Description	Number of states	Training set size	Tolerance	Maximum iterations
4MM	4 state Markov model	4	1349	10^{-3}	1000
6MM	6 state Markov model	6	1349	10^{-3}	1000
7MM	7 state Markov model	7	1349	10^{-3}	1000
7MMLS	7 state Markov model – large training set	7	13486	10^{-3}	1000
7MMIT	7 state Markov model – improved tolerance	7	13486	10^{-4}	5000
9MM	9 state Markov model	9	1349	10^{-3}	1000
11MM	11 state Markov model	11	1349	10^{-3}	1000
GE	Independent Gilbert-Elliot models	2	1349	10^{-3}	1000

6.2 Comparison of different Markov models

The different Markov models are now compared to each other and to the dataset. The testing set consists of the dataset excluding training set X . Therefore, the testing set contains 256223 DNA strings. Each model was used to generate the same number of strings. The strings were analysed and the results are described in the following subsections. Supplementary material for this section is found in Appendix Q.

6.2.1 General statistics

The error rate produced by each model is given in Table 6-2. Additionally, the percentage of error-free strings produced by each model, as well as the average string length, are given.

Table 6-2: Comparison of different Markov models

Model	Error rate	Percentage of error-free strings	Average string length
CNR Dataset	0.058	1.86 %	109.69
4MM	0.062	0.56 %	109.52
6MM	0.062	0.81 %	109.41
7MM	0.064	0.55 %	109.39
7MMLS	0.061	0.78 %	109.55
7MMIT	0.061	0.76 %	109.65
9MM	0.062	0.68 %	109.31
11MM	0.063	0.63 %	109.27
GE	0.061	0.20 %	110.14

It is seen that all of the models produce similar error rates which are slightly higher than that of the dataset. Additionally, all of the models produce fewer error-free strings than the dataset, with the GE model performing the worst in this aspect. This increased error rate and reduction in error-free strings may be due to the fact that the starting state of each model is chosen randomly from a uniform distribution. This is amplified by the fact that, for each string generated by the models, a new start state is selected. Therefore, the models may on average start in states with higher error rates than is required, leading to an increase in errors. Lastly, it is seen that all of the models produce similar average string lengths.

6.2.2 Error runs

Next, the error runs produced by each model are evaluated. Figure 6-1 depicts the number of error runs of different lengths produced by each model that contain any type of error, including multiple different types of errors in the same run. In Figure 6-1, it is seen that all of the models produce slightly fewer error runs of any type. The GE model performs the poorest, indicating that runs of errors of different types are not independent of each other, and should therefore be modelled together such as with the other models. 4MM performs the best of the Markov models, while 9MM performs the worst.

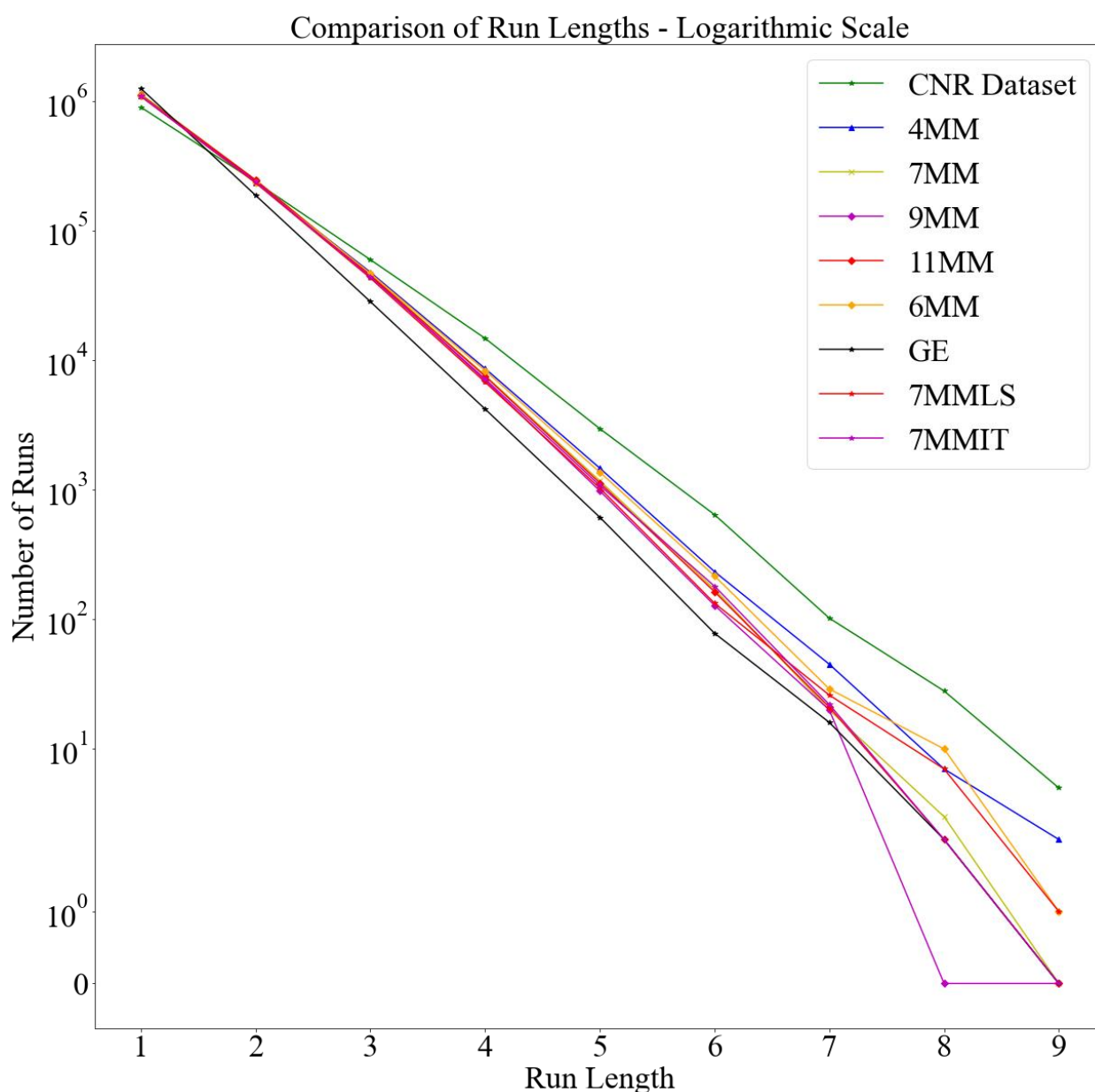


Figure 6-1: Comparison of error run lengths produced by different Markov models

Additionally, the models are compared in terms of the number of error runs of different lengths produced for each type of error. The results are summarised in Supplementary Figure Q-1 to Supplementary Figure Q-3

It is seen that all of the models produce slightly fewer runs of substitution errors than the dataset, especially at long lengths. Regarding substitution errors, 6MM performs the best, while the GE model performs the worst.

It is seen that all of the models produce slightly fewer runs of insertion errors than the dataset. 4MM performs the best, while 9MM performs the worst in regards to insertion errors.

It is seen that all of the models produce accurate amounts of runs of deletion errors. 7MM performs the best. At different run lengths, different models perform the worst. For example, for deletion runs of length 5, 7MMLS produces much fewer runs than the dataset, but for runs of length 7, 11MM and 9MM produces the least amount of runs.

6.2.3 Error bursts

Error bursts generated by each model are examined next. A burst is defined as multiple errors with three or fewer correct transmissions between them. A burst can consist of any type of error, including multiple types of errors within the same burst.

It is seen that most of the models accurately generate bursts of different lengths with different amounts of errors. The Markov models all show similar performance. The GE model has the worst performance, thereby indicating that bursts of errors of different types may not be independent of each other. The results are summarised in Supplementary Figure Q-4 and Supplementary Figure Q-5.

6.2.4 Error-free runs

Lastly, error-free runs, i.e. runs of consecutive successful transitions, of different lengths produced by each model, are examined. Once again, all of the models perform similarly except for the GE model. The GE model has the worst performance. This may be due to the fact that each of the three independent models that make up the GE model may interrupt the error-free runs generated by the other models. As an example, if the model that represents insertions produces an error-free run, this run may be broken up by errors generated by the models that represent substitutions and deletions.

9MM and 6MM perform the worst of the Markov models, especially at longer error-free run lengths. The results are summarised in Supplementary Figure Q-6.

6.3 Summary

Multiple Markov models were created and compared to each other. The models were compared in terms of the error rate, average string length and number of error-free strings generated by each model. In each case, all of the models performed very similarly and produced results close to the dataset.

Additionally, the models were compared in their ability to produce error runs and bursts, as well as error-free runs. While slight differences were detected, most of the models produced results that closely match the dataset. The GE model, however, underperformed in multiple instances when compared to the other models.

The Markov models that performed the worst in various tests pertaining to error runs, bursts and error-free runs were 4MM, 6MM, 9MM, 11MM and 7MMLS. These models shall therefore not be examined further.

The remaining models to be compared are the 7MM and 7MMIT models. The 7MM model showed a clear improvement over the other models at modelling runs of deletion errors, and is therefore chosen for further use and examination. While the model showed an increased error rate when compared to the dataset, this may be improved by selecting a starting state for the model in accordance with the error rate of the dataset. An improved implementation of the model is presented in Section 9.1.3 in order to address this shortcoming.

The model performs adequately in each examined area. The overall performance of the model is therefore better than the models that were considered to be the worst in certain tests. An additional advantage of the 7MM is that it is not data intensive or computationally expensive to train. The 7MM model uses training set Y and tolerance and maximum iteration values of 10^{-3} and 1000 respectively.

CHAPTER 7 ANALYSIS OF THE DATASET

The dataset is analysed in various aspects in order to determine the effect of different characteristics on the errors within the DNA strings. These aspects and characteristics are described in Section 3.7. Supplementary material for this chapter is presented in Appendix R.

7.1 Error rates at each position

Firstly, the error rate at each position of the DNA strings is examined in order to determine if certain areas, such as the beginning of the string, show increased error rates. The rate of each type of error at each position is also examined.

A marked increase in errors at the start of the string is found. Additionally, a smaller increase in errors is seen at the end of the string. It is also seen that, while each type of error shows an increase at these positions, insertion errors dominate the start and end of the string. The results are summarised in Supplementary Figure R-1.

7.2 Homopolymer analysis

The effects of homopolymer runs are also analysed. Error rates within homopolymer runs are examined. Error rates at the position right before the homopolymer runs and at the position right after the homopolymer runs are also examined. The analysis is done for homopolymer runs of each different type of base, as well as homopolymer runs of all types of bases combined. The results are summarised in Supplementary Figure R-2 to Supplementary Figure R-4.

It is found that the error rate within homopolymers of length two is approximately the same as the overall error rate of the dataset, which was found to be 0.058. An increase in error rate is seen as the length of the homopolymer increases. Although homopolymers of A and T show a lower error rate than homopolymers of G and C, all types of homopolymers show an increase in error rate as homopolymer length increases.

It is seen that there is no strong connection between homopolymer length and the error rate right before or right after the homopolymer. The error rates right before and right after homopolymer runs of different lengths are seen to be around 0.058, which is the overall error rate of the dataset.

7.3 Problematic substring analysis

Different substrings in the dataset were analysed according to the algorithm described in Section 4.16. Substrings of lengths 2, 4 and 6 were examined.

7.3.1 Substrings of length 2

In order to analyse the occurrence of problematic substrings, substrings of length 2 were analysed. It is seen that homopolymers of G and C are the substrings with the highest error rates. The results are summarised in Supplementary Figure R-5.

7.3.2 Substrings of lengths 4 and 6

In order to confirm that an adequate number of each substring is in the dataset to allow for a statistical analysis of high certainty, the distribution of number of occurrences of each substring is examined. It is seen that the number of occurrences follow a normal distribution, and that each substring of length 4 occurs at least 9500 times. Each substring of length 6 occurs at least 2000 times. This allows for a high degree of certainty when calculating the error rates of different substrings. The results for substrings of lengths four and six are depicted in Supplementary Figure R-6 and Supplementary Figure R-8 respectively.

The distribution of error rates of different substrings is examined. It is observed that the error rate varies greatly. Certain substrings of length 4 have an error rate of up to 0.125, which is much higher than the average error rate of the dataset. Certain substrings of length 6 have an error rate of up to 0.225. Furthermore, it can be seen that the error rates of different substrings of length 4 and length 6 do not follow a normal distribution. This is an indication that certain substrings are problematic and can lead to increased error rates, while other substrings might lead to decreased error rates. The results for substrings of lengths 4 and 6 are depicted in Supplementary Figure R-7 and Supplementary Figure R-9 respectively.

The ten substrings of lengths 4 and 6 with the highest error rates are examined. It is found that all of these substrings have error rates well above the average error rate of the dataset. Furthermore, it is seen that most of these substrings have high GC contents and all of these substrings contain homopolymers. This indicates a correlation between GC content and error rate, as well as a correlation between homopolymer runs and an increased error rate. The results for substrings of lengths 4 and 6 are summarised in Supplementary Table R-1 and Supplementary Table R-3 respectively.

The ten substrings of lengths 4 and 6 with the lowest error rates are examined. It is found that all of these substrings have error rates well below the average error rate of the dataset. Furthermore, it is seen that most of these substrings have lower GC contents. In each case, the GC content is

either equal to less than half of the substring. Only three of these substrings contain homopolymer runs, and none of the homopolymer runs are of the bases C or G. All of the homopolymer runs in these substrings are of length three or shorter. The results for substrings of lengths four and six are summarised in Supplementary Table R-2 and Supplementary Table R-4 respectively.

7.3.3 Summary

Table 7-1 summarises various statistics for the different length substrings that were analysed above. It is seen that the error rate between different substrings varies to a greater degree when longer substrings are analysed than when shorter substrings are analysed. The substrings of length 6 show the highest maximum and lowest minimum error rates, as well as the largest standard deviation. The substrings of length 2 show the lowest maximum and highest minimum error rates, as well as the smallest standard deviation. This may be attributed to the fact that substrings of length 6 contain longer homopolymers and a more varied GC content, thereby leading to more types of substrings that can be prone to errors.

Table 7-1: Statistics for error rates of different length substrings in the dataset

Substring length	2	4	6
Maximum error rate	0.0740	0.1251	0.2259
Minimum error rate	0.0395	0.0305	0.0238
Mean error rate	0.0562	0.0558	0.0557
Median error rate	0.0535	0.0548	0.0539
Standard deviation	0.0101	0.0137	0.0148
Number of different substrings	16	256	4096

7.4 GC content analysis

To confirm that an adequate number of strings with each different GC content value are in the dataset to allow for statistical analysis of high certainty, the distribution of the number of occurrences of strings with different GC content values is examined. The results are summarised in Supplementary Figure R-10.

It is seen that the number of occurrences follows a normal distribution. It is also seen that not all GC content values are available in the dataset. Strings with very high or very low GC contents do

not exist in the dataset. Therefore, the GC content analysis shall only be done with the available strings.

The average error rate within strings with different GC contents is examined. The results are summarised in Supplementary Figure R-11. When examining GC content values that have a high degree of certainty, i.e. GC content values around 55 out of 110, it is seen that, as the GC content increases, the error rate also increases. However, the error rate stays between the values of 0.055 and 0.065, which is relatively close to the average error rate of the dataset of 0.058.

The error rate of substrings with different GC contents is examined next. Substrings of length 6 were analysed. A marked increase in error rate is found as the GC content increases. Substrings with low GC contents may have error rates as low as 0.050, which is much lower than the 0.058 error rate of the dataset. Furthermore, substrings with very high GC contents have error rates as high as 0.080. Hereby, it is seen that the local GC content has a large influence on the error rate in that part of the string. The results are summarised in Supplementary Figure R-12.

7.5 Running digital sum

In order to confirm that an adequate number of bases with different RDS values are in the dataset to allow for a high-certainty statistical analysis, the distribution of the number of occurrences of bases with different RDS values is examined. The results are shown in Supplementary Figure R-13 and summarised below.

It is seen that the number of occurrences follows a normal distribution. It is also seen that not all RDS values are available in the dataset. Bases with very high or very low RDS values do not exist in the dataset. Therefore, the RDS analysis shall only be done with the available values. Furthermore, it is seen that there are not many occurrences of RDS values above 20 and below -20 , therefore outliers may skew the results when analysing these values. Consequently, only RDS values between -20 and 20 shall be analysed.

Additionally, as seen in Section 7.1, error rates increase at the start and the end of a string. Since the RDS at the start of the string is always 0, and then becomes either -1 or 1 , this may lead to an increased average error rate for RDS values between -1 and 1 . Therefore, the first two and final positions of the DNA strings are left out when analysing the effect of the RDS on the error rate.

A comparison between RDS values and error rates for bases with that value is made. Firstly, the RDS as calculated from the start of the string is considered. The error rate is found to increase as the RDS increases above 0 for each error type, as well as the overall error rate. Furthermore,

it is seen that the error rate decreases as the RDS decreases below 0. The error rate reaches a minimum at a RDS of -4 , and as the RDS decreases further, the error rate increases again. The results are summarised in Supplementary Figure R-14 and Supplementary Figure R-15.

The RDS as calculated from the back of the string is now considered. A strong increase in error rate is seen as the RDS increases above 0. Furthermore, it is seen that the error rate decreases as the RDS decreases, reaching a minimum error rate at a RDS value of -15 . The results are summarised in Supplementary Figure R-16 and Supplementary Figure R-17.

Additionally, to confirm if there is a correlation between the RDS value and the error rate, the Pearson correlation coefficient was calculated. RDS values between -10 and 10 were considered. It is seen that for all types of errors, there is a strong correlation between the RDS value and the error rate. For the overall error rate, deletion rate and insertion rate, the correlation is stronger when calculating the RDS from the end of the string. For the insertion rate, the correlation is stronger when calculating the RDS from the start of the string. The results are summarised in Table 7-2.

Table 7-2: Correlation coefficient between RDS value and error rate

Type of error	RDS calculated from the front of the string	RDS calculated from the back of the string
Overall error rate	0.847	0.905
Insertion rate	0.833	0.620
Deletion rate	0.805	0.847
Substitution rate	0.811	0.908

7.6 Swap errors

It is found that 2.23% of the errors within the dataset contribute to swap errors. Using the error rate of the dataset as 5.8% as shown in Table 6-2, the probability of an error being part of a swap error is found to be 0.13%.

7.7 Summary

The dataset was analysed to determine which characteristics may influence the error rate within the channel. It was seen that homopolymers lead to an increase in error rate. Furthermore, it is seen that certain substrings are problematic and have an increased error rate when compared to

other substrings. The effect of GC content was also analysed and indicated that an increase in local GC content can lead to an increased error rate. Lastly, it was found that swap errors only make up a small portion of the total number of errors.

This gives an indication of which characteristics should be incorporated in order to create an improved model.

CHAPTER 8 IMPROVED MODEL

In Chapter 6 it is seen that a Markov model with 7 states is able to accurately model error runs and bursts within the DNA data storage process. This model, however, does not represent the influence of the bases in the input string on the error rate. Aspects such as the influence of homopolymers and the GC content, which are examined in Chapter 7, are not accommodated. Therefore, additional models are devised in an attempt to represent these aspects of the dataset.

In order to create these proposed models, a training set is created by uniformly sampling 50% of the dataset. This training set includes the 1349 substrings used to train the 7MM model. Thereby, the other 50% of the dataset may be used as a test set in order to verify the efficacy of the proposed models. This training set is used to create the models described in this chapter, while the testing set is used to determine the efficacy of these models in Chapter 9. Although both the testing and the training sets are expected to show similar statistical properties, the use of a testing set acts as a validation method for the performance of the models on unseen data. This is similar to current best practices in machine learning, where the performance of a model is validated on a different set of data than the data used to train the model [183].

8.1 Error rates at the start and the end of the string

Insertion, deletion and substitution rates for the first and second positions of the strings in the training set are calculated (positions 0 and 1). Errors can then be generated for these positions according to the appropriate probability distributions. Similarly, insertion rates are calculated for insertions that occur after the final position of the string. Insertions are then generated according to this probability distribution. The errors generated at this stage can be added to strings of outcomes generated by the models, overwriting the existing outcomes in their respective positions.

8.2 Substring characteristics

In order to create models that take different substrings into account, different substrings are characterized.

As MinION sequencers use *kmers* of length 6 [105], substrings of length 6 are used. Each string in the training set is analysed using the algorithm from Section 4.16 and the data is recorded for the use of creating improved models.

For each substring, the error rate at each position of the substring is found. Furthermore, the insertion, deletion and substitution rates at each position of the substring is found. Using this data, the proposed models can attempt to represent problematic substrings, as well as the influence of the GC content of different substrings. Furthermore, homopolymers up to length 6, which is the length of the substrings, may be taken into account via this method.

8.3 Substring model

The first proposed model shall be referred to as the Substring model. The model shall make use of the substring data described in Section 8.2 to generate errors within a DNA string. This may allow the model to capture the behaviour of the channel with regard to homopolymers and GC content, although error runs and bursts are not explicitly incorporated within this modelling method. The efficacy of the model in representing these aspects shall be examined in Chapter 9.

When presented with an input string, the Substring model creates a string of outcomes. For the Substring model to produce a string of outcomes, the input string is divided into non-overlapping substrings of length 6. As the length of the input strings in the dataset is 110, which is not divisible into equal substrings of length 6, substrings start at position 2 of the input string – thus excluding positions 1 and 0. There are 108 remaining bases, equating to 18 substrings of length 6.

Each substring is compared to the analysis on the dataset. Outcomes are generated for each position of each substring according to the probability distributions for that specific type of substring.

Errors at the start and end positions of the strings of outcomes are added according to the method described in Section 8.1. Finally, the strings of outcomes are populated by specific bases using the algorithm described in Section 4.9.

8.4 Models that combine a Markov model with the substring characteristics

Models are now created that combine the errors generated by a Markov model and the data derived from substrings in an attempt to accurately represent error runs and bursts, as well as characteristics from different substrings.

Figure 8-1 depicts the workflow for creating these models and generating outputs with the models. The workflow is able to accommodate different methods for incorporating the data derived from substrings, namely through consecutive windows, windows that slide forward and windows that slide backward over the DNA string. These methods and the different aspects of the workflow are discussed throughout the rest of this subsection.

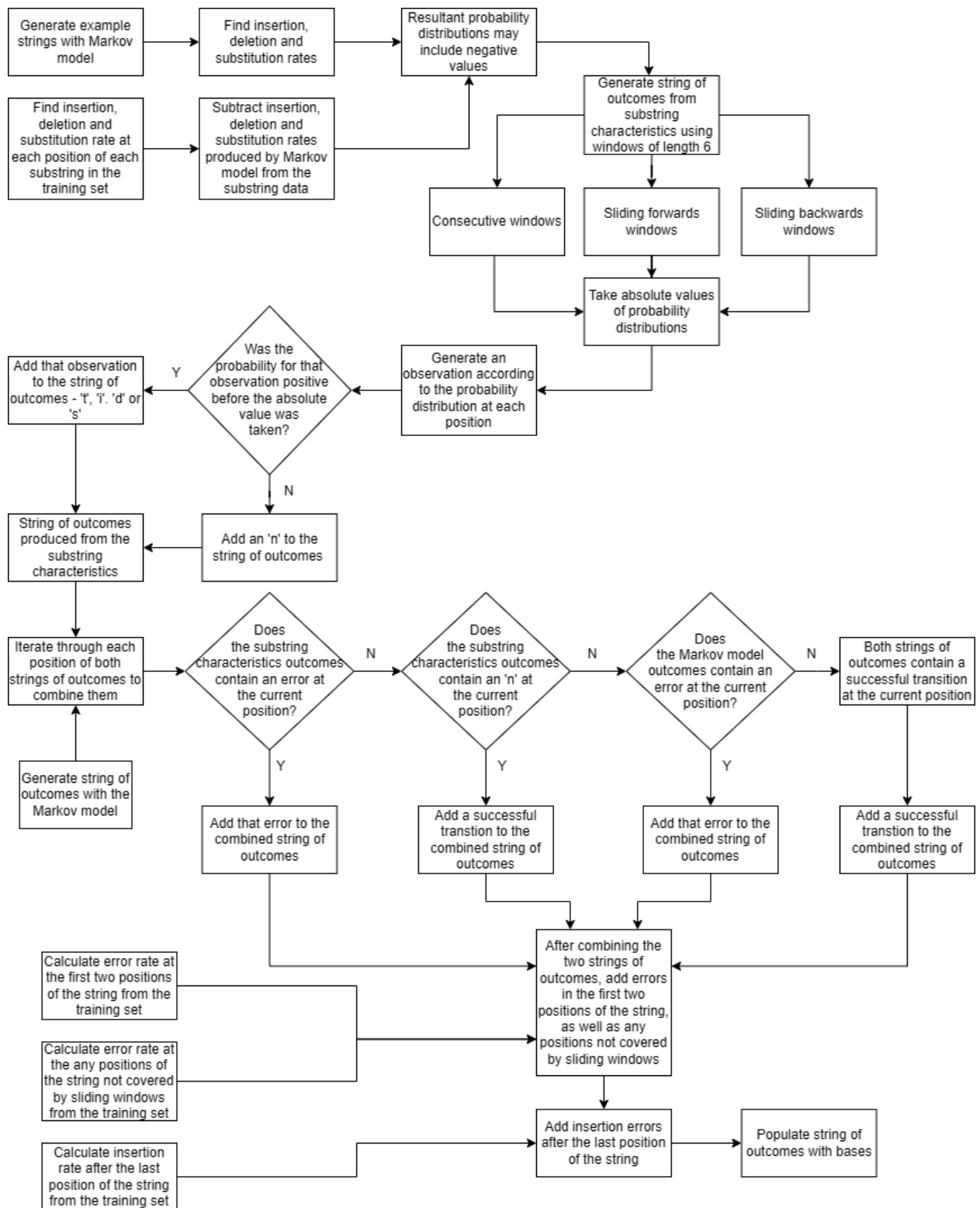


Figure 8-1: Workflow for creating models combining a Markov model and substring characteristics, and generating output strings with these models

8.4.1 Model components

Three models are proposed that shall each consist of a Markov model as base and a secondary model using data on the characteristics of different substrings, as presented in Section 8.2. The use of the Markov model shall attempt to accurately model error runs and bursts. The use of the substring characteristics shall attempt to represent the effect of problematic substrings, homopolymers and GC content on the error rate within the DNA string.

Additionally, error rates for the start and the end of the DNA string shall be calculated and added to the model, as a marked increase in errors was found at these positions. Swap errors are not explicitly modelled, as Section 7.6 showed that they make up a very small percentage of the total errors that occur within the DNA string.

8.4.1.1 Markov model as a base

The improved model will use the 7MM Markov model with 7 states as a base. The 7MM model is chosen, as Chapter 6 indicated that the 7MM model outperforms the other examined Markov models when tasked with representing runs of deletion errors, while performing adequately in each other examined area. The Markov model was trained with the Baum-Welch algorithm on a training set with 1349 strings for 1000 iterations on each string. Using a Markov model as base shall allow the proposed models to represent error runs and bursts. Errors at the start of the string shall be added separately using the method described in Section 8.1. Therefore, a modification is made to the Markov model regarding the starting state. The original 7MM model chose a starting state randomly. The Markov model used to create new proposed models throughout this section is modified to always start in the first state, as that is a low error rate state.

8.4.1.2 Substring data with reduced error rate

When errors generated by the Markov model and the substring characteristics are added to the same string, the total error rate generated may become too high. Therefore, the error rate indicated by the substring data is reduced by the error rate produced by the Markov model.

The Markov model is used to create a number of example strings. The average insertion, deletion and substitution rate produced by the Markov model is found. These values are subtracted from the insertion, deletion and substitution rates contained by the substring data in order to ensure that the total error rate does not become too large. The resultant values are stored for use within the proposed models.

It should be noted that subtracting the insertion, deletion and substitution rates produced by the Markov model from the rates contained in the substring data may result in negative values if the

rates produced by the Markov model are larger than the rates for a specific substring. These negative values are also stored for future use.

8.4.2 Methods for implementing substring characteristics

The three proposed models that combine the Markov model with the substring characteristics shall each implement substring characteristics in a different manner.

8.4.2.1 Consecutive windows

In this model, the substring data is used in a similar manner as the Substring model proposed in Section 8.3. In order to produce a string of outcomes, the input string is divided into non-overlapping substrings of length 6. At each position of each substring, the model references the data corresponding to that substring. Errors are produced at each according to the probability distribution for different types of errors for that position of that specific substring. Errors at the starting positions of the strings of outcomes are added according to Section 8.1, as well as insertions at the ends of the strings.

8.4.2.2 Sliding forward windows

For this model to produce a string of outcomes, a sliding window of length 6 and step size 1 is used. The window starts at the first position of the DNA string and compares the 6 bases within the window to the substring data. The probability distribution for that specific type of substring is used to predict errors that occur at the last position of the substring. The window thereafter moves forward one base and the process continues until the sliding window reaches the end of the DNA string.

As the sliding window is of length 6 and starts at the first position of the DNA string, this model cannot predict error rates at the first 5 positions of the DNA strings. Therefore, error rates for these positions are independently calculated and errors are added to the string of outcomes accordingly. Additionally, insertions are added to the end of the strings as described in Section 8.1.

8.4.2.3 Sliding backward windows

This model uses a sliding window of length 6 and step size 1. The window starts at the last position of the DNA string and compares the 6 bases within the window to the substring data. The probability distribution for that specific type of substring is used to predict errors occurring at the first position of the substring. The window thereafter moves backwards one base, and the process continues until the sliding window reaches the first position of the DNA string.

The sliding window has a length of 6 and starts at the last position of the DNA string, therefore the window cannot be used for observations at the last 5 positions of the DNA strings. Consequently, error rates for these positions are independently calculated, and errors are added to the string of outcomes accordingly.

Errors at the starting positions of the strings of outcomes are added according to the method described in Section 8.1 to simulate the increased error rate at the start of the DNA string that was shown in Section 7.1. Additionally, insertions after the last position of the DNA string are added according to the method described in Section 8.1.

8.4.3 Implementing negative values in the substring data

After subtracting the insertion, deletion and substitution rates produced by the Markov model from the rates contained in the substring data, some of the new probability distributions may contain negative values. When choosing an outcome based on these probability distributions, the absolute values of the probabilities are used.

If an outcome is selected, it is checked if the value of that probability, before the absolute value was taken, was positive. If it was positive, the selected outcome is added to the string of outcomes as is. If the value of the probability was negative, an 'n' is added to the string of outcomes instead of the selected outcome.

8.4.4 Combining Markov model outputs with outputs generated from substring characteristics

The Markov model as described in Section 8.4.1.1 is used as base for the three proposed models. Each of the three proposed models shall use a different form of windows, as described in Sections 8.4.2.1 to 8.4.2.3.

When presented with an input string, the Markov and model based on substring data each creates a string of outcomes. As before, the Markov model does not regard the specific bases of the input string, but only produces a string of outcomes of the correct length. The Substring model, however, does consider the bases of the input string.

There are now two separate strings of outcomes, one created by the Markov model and one created using the substring data. The two strings of outcomes shall be combined to create a combined string of outcomes. If both strings of outcomes contain a successful transition, a successful transition will be added to the combined string. If either of the strings of outcomes contains an error, that error will be added to the combined string. If both strings of outcomes contain errors, but the errors are of different types, the error generated from the substring data takes precedence. This is due to the fact that different substrings may contain errors of different

types, for example certain substrings may contain more deletions while other substrings may contain more insertions. If the outcomes generated from the substring data produce an 'n' value, a successful transition is added to the string of outcomes regardless of the observation produced by the Markov model. This allows the substring data to reduce the error rate in low error rate substrings by removing errors generated by the Markov model.

For each of these three models, the strings of outcomes are populated by specific bases using the algorithm described in Section 4.9.

8.4.5 Resultant models

The model that uses the consecutive windows of Section 8.4.2.1, shall be referred to as the Combined model. The model that uses the sliding windows of Section 8.4.2.2, shall be referred to as the Sliding forwards model. Lastly, the model that uses the windows that slide backwards as discussed in Section 8.4.2.3, shall be referred to as the Sliding backwards model.

8.5 Summary

In this section, four models were proposed. The proposed models were trained on the same training set consisting of 50% of the dataset, which corresponds to 134855 strings. The four proposed models make use of data derived from different substrings in the training set in order to better represent certain aspects of the dataset. These aspects include the effect of GC content and homopolymers on the errors induced in the DNA data storage process, as identified in Chapter 7. The proposed models incorporate the substring characteristics by making use of sliding windows when examining the input DNA strings.

Additionally, three of the proposed models combine the characteristics of the substrings with a Markov model. By combining the Markov model with the substring characteristics, these models aim to represent error runs and bursts, as well as aspects such as the effect of GC content on the errors within the DNA strings. The models are evaluated on their ability to represent different characteristics of the dataset in Chapter 9.

CHAPTER 9 RESULTS

In this chapter, the proposed Substring, Combined, Sliding forwards and Sliding backwards models are compared to an updated version of the 7MM model as well as the existing IID and Memory models. The ability of the models to represent error runs, error bursts and error-free runs is compared against a testing set derived from the CNR dataset. The models are also assessed according to their ability to represent errors induced by homopolymer runs and different GC contents within an input DNA string. Lastly, the error rates that the models produce within different substrings are assessed.

9.1 Dataset and models used for comparison

A testing set, which excludes the data used to train the models, is sampled from the dataset. The newly proposed models, as well as the 7MM model, are used to generate a number of strings for comparison against this test set. The 7MM model is updated to represent the dataset more accurately by calculating the probability of starting in each state instead of choosing a starting state from a uniform probability distribution. The updated 7MM model used in this chapter is described in Section 9.1.3. Furthermore, the IID model and Memory model are used to generate strings in order to compare the proposed models to existing models.

9.1.1 Dataset

In order to train the proposed models, a training set was sampled from the CNR dataset. The remainder of the dataset, thus excluding the training set, consists of 134854 DNA strings. These remaining strings are used as a test set for comparison against the proposed models. When referring to the dataset throughout this chapter, this testing subset of the CNR dataset is referred to.

9.1.2 Newly created models

The Substring model, Combined model, Sliding forwards model and Sliding backwards models are each used to generate a set of 134854 strings, using randomly selected strings from the Centers file as input strings. The Levenshtein Distance Error Identification algorithm is then applied to the generated strings to find the errors within these strings. The performance of the models can then be compared to the other models and to the dataset.

9.1.3 7MM

As a comparison, the 7MM model as described in Chapter 6 is used. This may give an indication of the impact that combining the substring data with the Markov model has on the ability of the Markov model to accurately model error runs and bursts.

The 7MM model is updated to correctly model the increased error rate at the start of the DNA strings. This updated 7MM model has been published in the paper “Modelling Error Runs in DNA Data Storage Using a Markov Model” [36] that was produced during this study.

Starting probabilities were calculated for the starting state of the 7MM model. In order to calculate the starting probabilities for the model, only the states with the highest probability of each error type are considered. This simplifies the calculation of the starting probabilities of each state. The following states are selected according to the transition matrix of the model, as given in Supplementary Table P-9:

- State 1 for successful transitions
- State 4 for substitution errors
- State 5 for insertion errors
- State 7 for deletion errors

By calculating the insertion rate at the first position of the training set, this value can be divided by the probability of emitting an insertion error in state 5, thereby finding the probability of the model to start in state 5 and achieve the correct insertion rate at the start of the string. Similarly, the probability to start in state 7 can be calculated by considering the deletion rate at the start of the string. Lastly, the probability to start in state 4 can be calculated in the same manner by considering the substitution rate at the start of the string.

Lastly, the probability to start in state 1, which is an error-free state, is found by calculating the probability of not starting in any of the other states. The calculations are given as:

$$P_4 = \frac{\text{substitution rate at position 0}}{\text{probability of a substitution in state 4}} \quad (9-1)$$

$$P_5 = \frac{\text{insertion rate at position 0}}{\text{probability of an insertion in state 5}} \quad (9-2)$$

$$P_7 = \frac{\text{deletion rate at position 0}}{\text{probability of a deletion in state 7}} \quad (9-3)$$

$$P_1 = 1 - P_4 - P_5 - P_7 \quad (9-4)$$

The resultant probabilities are given in Table 9-1.

Table 9-1: Probability of starting in each state for the 7MM model

State	Probability
1	0.754
2	0.000
3	0.000
4	0.027
5	0.191
6	0.000
7	0.028

9.1.4 IID model and Memory model

As described in Section 3.5, outputs are generated using the IID model and the Memory model. These models serve as benchmarks for comparison of the newly created models. Both models were trained on the same training set as the proposed models. The implementation and training of these models are described in Sections 4.10 and 4.11.

The Memory model allows the user to change the length of *kmers* that are considered. The default *kmer* length considered by the Memory model is 6 as MinION sequencers use *kmers* of length 6 [105]. Therefore, this parameter is left at the default value.

9.2 General statistics

In this subsection, general statistics of the different models are compared. This includes the error rate, number of error-free strings and average string length produced by each model. Additionally, the distribution of error types produced by each model is examined. By examining these statistics, it can be verified if the models are able to represent the basic characteristics of the dataset.

The error rate produced by each model is summarised in Table 9-2. Furthermore, the percentage of strings that are error-free is given, as well as the average length of the strings produced by the model.

It is seen that the error rate produced by the 7MM model closely matches that of the dataset. Furthermore, the 7MM model produces the closest number of error-free strings to the dataset of any of the models. This is a large improvement when compared to the results achieved by this model in Section 6.2.1. This improvement can be attributed to the fact that probabilities of starting in different states were calculated and added to the model in Section 9.1.3.

The Substring model and IID model also produce similar error rates to the dataset, but both of these models produce much fewer error-free strings. The other proposed models all produce error rates that are 0.011 to 0.013 above that of the dataset. The IID model and proposed models all produce similar average string lengths to the dataset. The Memory model produces a slightly lower error rate than the dataset, but still produces a similar average string length.

Table 9-2: Summary of model statistics

Model	Error rate	Percentage of error-free strings	Average string length
CNR Dataset	0.058	1.82	109.68
7MM	0.057	1.27	109.73
Substring model	0.058	0.14	109.31
Combined model	0.071	0.22	109.16
Sliding forwards model	0.069	0.32	109.30
Sliding backwards model	0.071	0.22	109.19
IID model	0.058	0.14	109.65
Memory model	0.057	0.74	109.58

The distribution of error types produced by each model is also examined. It is found that all of the models produce similar distributions of error types, namely insertions, deletions and substitutions, than the dataset. The results are depicted in Appendix S, Supplementary Table S-1.

9.3 Error rate at each position

This section examines the error rates produced by each model at different positions within the DNA strings. This examines whether the models are able to simulate the increases in errors at the start and end of the DNA strings, as shown in Section 7.1. Supplementary material for this subsection is available in Appendix T.

Figure 9-1 depicts the average error rate produced by each model at each position of the DNA string. The proposed models all exhibit an increase in error rate at the start of the DNA string, similar to the dataset. It is also noted that the proposed models show an increase in error rate at the end of the DNA string, but this increase is much larger than shown by the dataset. The Memory

model shows an increase in error rate at the end of the strings similar in size to that shown by the dataset.

Similar analyses for insertion, deletion and substitution rates produced by each model at each position of the DNA string, show similar characteristics as for the overall error rate. The results are depicted in Supplementary Figure T-1, Supplementary Figure T-2 and Supplementary Figure T-3.

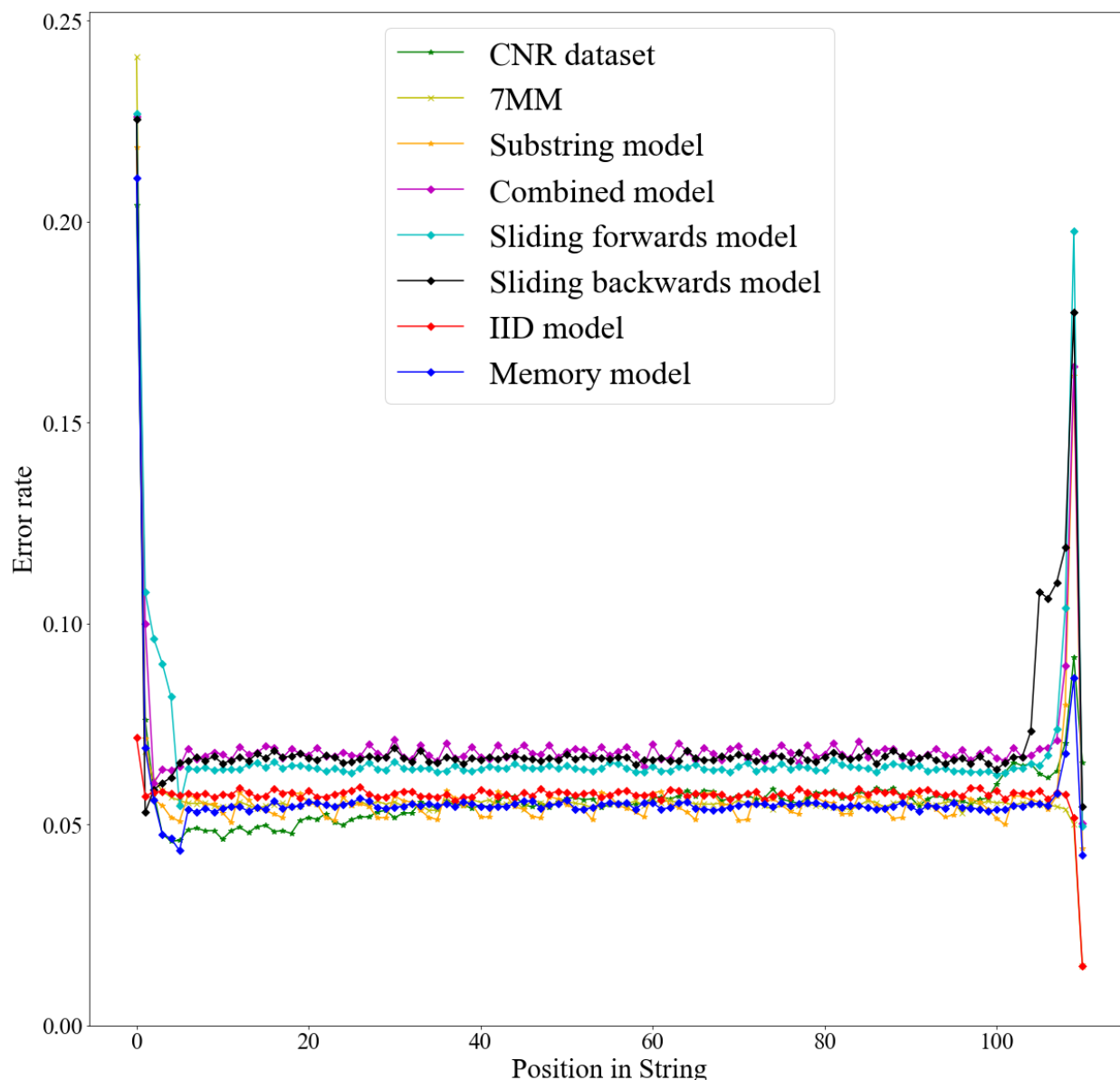


Figure 9-1: Comparison of error rates produced at each position by each model

Figure 9-2 a) shows a scatterplot of the errors produced in the first 300 strings in the dataset. At the right of the scatterplot, it is seen that only insertions occur at the last position of each string, indicating insertions that occur after the last base of the input string.



Figure 9-2: Scatterplots representing error positions in the first 300 strings of the dataset and the output of each model

Since the first 300 strings produced by each model are plotted in Figure 9-2 b) to h), and each input string may be sequenced multiple times, some successive output strings correspond to the same input string. Multiple errors in the same position in successive strings (represented on the y-axis) represent errors in the model outputs in the same position for the same input string. Such errors can indicate aspects such as homopolymer runs or other attributes of the input string, which lead to an increase in errors at that position.

As the 7MM does not inspect individual bases of the input string when producing errors at different positions, it does not reflect these dependencies accurately. The other proposed models do indicate these dependencies, as does the Memory model.

9.4 Error runs

This subsection examines the number of error runs produced by each model and compares that to the runs in the dataset. Special attention shall be given to long-run errors, as in Section 2.5.5 it was found that existing models do not accurately model long-run errors. Figure 9-3 represents the number of error runs of different lengths produced by each model with a logarithmic scale.

The IID model and the Substring models produced much fewer runs than the dataset, especially as the run length increases. This indicates that increased error rates within specific substrings as implemented in the Substring model are not capable of accurately simulating error run events.

All of the other proposed models produce slightly fewer error runs than the dataset. This indicates that, by combining substring characteristics with the Markov model in the Combined, Sliding forwards and Sliding backwards models, the models' ability to produce error runs is not negatively affected.

The 7MM and Sliding backwards models performed the best for run lengths of length 8 and 9 respectively. This indicates that these two models are the most accurate at representing long-run errors. Although the Memory model performs similarly to the proposed models, it produced slightly fewer error runs at long run lengths, i.e. runs of lengths 8 and 9.

When analysing runs of insertion, deletion and substitution errors respectively, it is found that the 7MM, Combined, Sliding forwards and Sliding backwards models are all capable of representing deletion runs very accurately. These models all produce slightly fewer insertion runs of long lengths than the dataset. However, a larger discrepancy is seen in the ability of these four models to accurately model long runs of substitutions, as all four models produce fewer runs than the dataset. Supplementary material for this subsection is available in Appendix U.

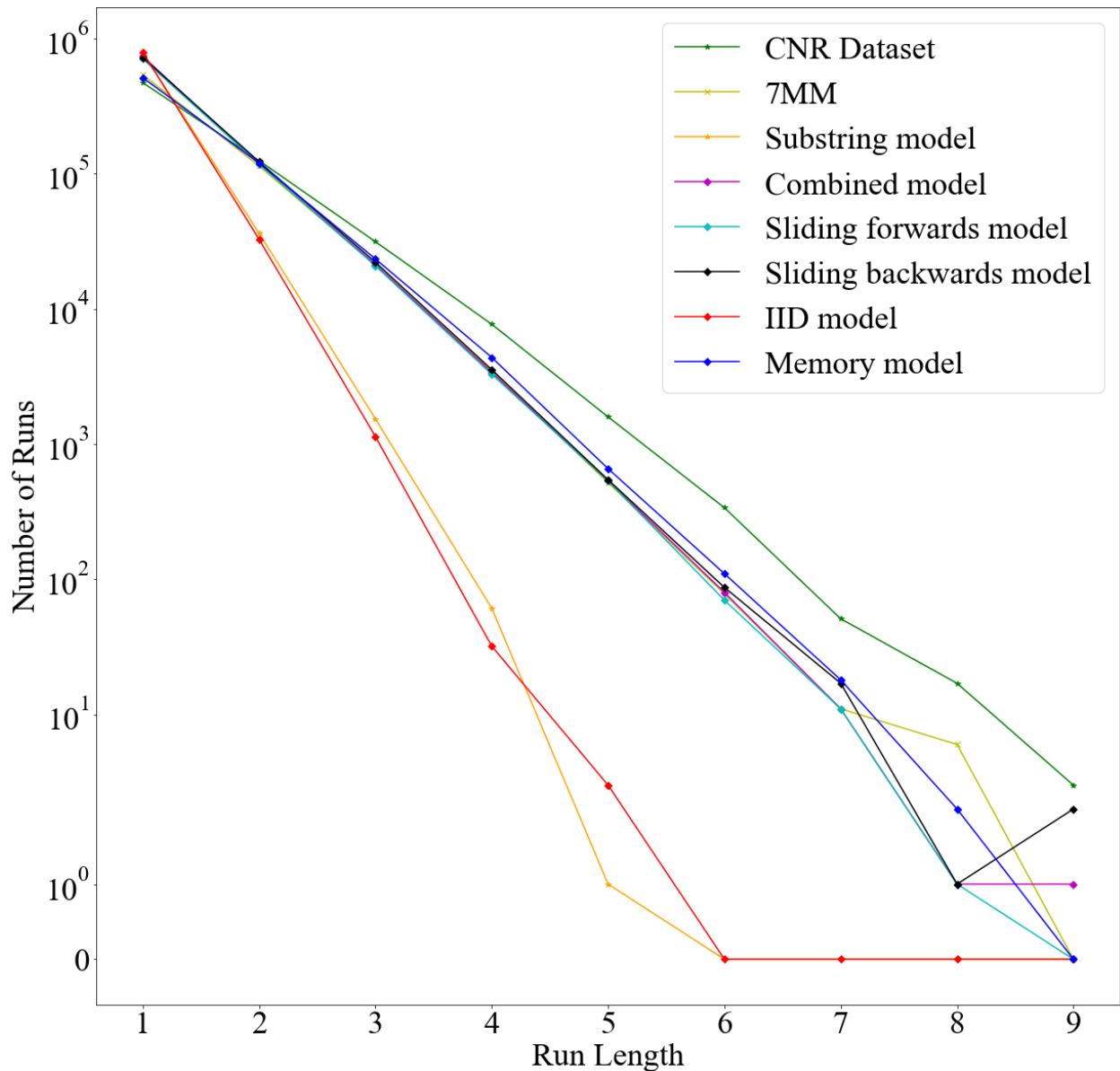


Figure 9-3: Comparison of error runs produced by each model

Furthermore, the RMSLE between runs produced by each model and runs found in the dataset was calculated. These values are depicted in Table 9-3. It is found that the 7MM, Combined, Sliding forwards, Sliding backwards and Memory models all perform much better than the Substring and IID models. The Memory and 7MM models have the smallest and second smallest RMSLE values when considering error, insertion and substitution runs. This indicates that these models perform well regarding error runs in general, but they may still be outperformed by the Sliding backwards model when only considering long run lengths, as seen in Figure 9-3.

Table 9-3: RMSLE for error runs produced by each model

Model	Error runs	Insertion runs	Deletion runs	Substitution runs
7MM	0.438	0.394	0.196	0.569
Substring model	1.704	1.448	1.214	1.401
Combined model	0.496	0.463	0.180	0.585
Sliding forwards model	0.534	0.413	0.176	0.601
Sliding backwards model	0.456	0.463	0.198	0.645
IID model	1.703	1.655	1.271	1.431
Memory model	0.424	0.224	0.230	0.484

9.5 Error bursts

In this section, the error bursts produced by each model are compared to the bursts found in the dataset. An error burst is defined as multiple errors with three or fewer successful transitions between them. Therefore, each burst has a length of at least two and contains at least two errors. By this definition, every error run is also an error burst, or part of an error burst. Bursts are examined to see if the models can represent errors in close proximity that may depend on each other, but also contain correct transitions between them. Supplementary material for this subsection is found in Appendix V.

Figure 9-4 compares the number of bursts of each length produced by each model to the number of bursts in the dataset. The length of the burst is the number of bases that the burst spans, including the number of erroneous bases and the number of successful transitions encompassed by the burst. It is seen that the IID model, Memory model and Substring model produce much fewer bursts than the dataset, especially at long lengths. The other proposed models all perform similarly to the dataset, with the 7MM being the most accurate at shorter burst lengths and the Combined and Sliding backwards models the most accurate at burst lengths of more than 25 bases.

Additionally, the number of bursts containing different numbers of errors produced by the models was compared to the bursts in the dataset. This does not include successful transitions between the errors within a burst, but only the number of errors within the bursts. Once again, it is found that the IID and Substring models produce much fewer bursts than the dataset, especially at long burst lengths. The Memory model also produces fewer bursts than the dataset. All of the other proposed models produce bursts that closely resemble the dataset. The Sliding backwards model

performs the best, especially at bursts with large amounts of errors. The results are depicted in Supplementary Figure V-1.

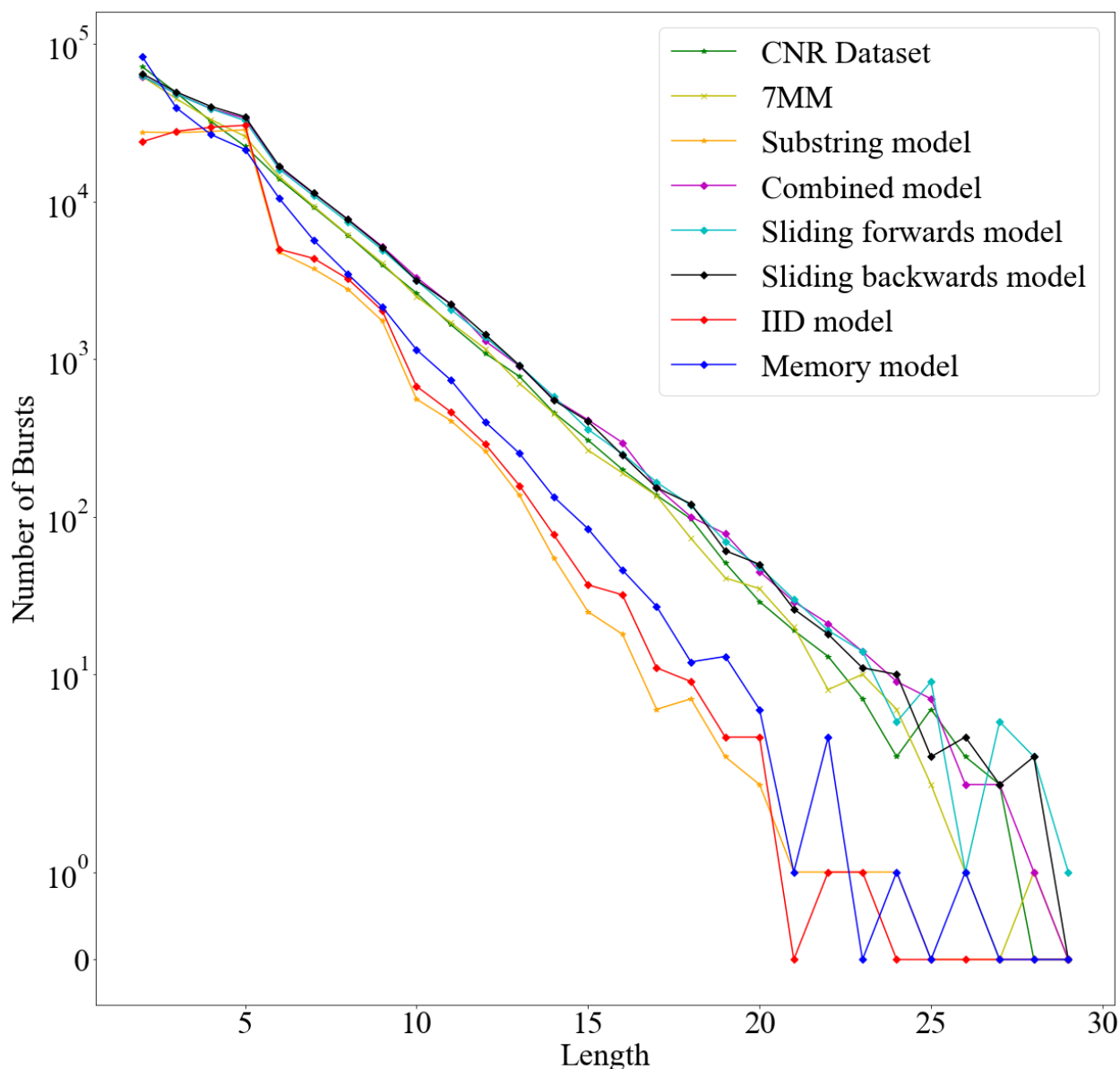


Figure 9-4: Comparison of error burst lengths produced by each model

Furthermore, the RMSLE between bursts produced by each model and bursts found in the dataset was calculated. It is found that the 7MM performs the best when examining burst length, while the Combined model performs the best when examining the number of errors per burst. The Sliding backwards model performs very similarly to the Combined model, and the Sliding forwards model also performs very well. The results are depicted in Table 9-4.

Table 9-4: RMSLE for error bursts produced by each model

Model	Burst length	Errors per burst
7MM	0.033	0.153
Substring model	0.301	1.525
Combined model	0.094	0.102
Sliding forwards model	0.083	0.120
Sliding backwards model	0.096	0.109
IID model	0.293	1.472
Memory model	0.153	0.462

The number of bursts produced by each model that starts at each position of a DNA string was also analysed. The 7MM, Combined, Sliding forwards and Sliding backwards models all closely mimic the dataset. The Sliding backwards model does, however, produce an increase in bursts at the end of the DNA string, which is not representative of the dataset. The results are depicted in Supplementary Figure V-2.

9.6 Error-free runs

In this subsection, the number of error-free runs of different lengths produced by each model is examined. As it was shown in Section 6.2 that errors are not independently distributed in the dataset, it is now examined whether error-free areas are also not independently distributed. Error-free runs are defined as multiple errorless bases directly after one another. A run of length one indicates a single error-free base between two errors.

Figure 9-5 depicts the number of error-free runs of different lengths produced by each model. A sharp increase is seen on the right-hand side of the graph for the dataset and all of the proposed models. This value corresponds to error-free runs of length 110. A string with an error-free run length of 110 is not necessarily an error-free string. Insertion errors before the first base or after the last base may occur within the string, thereby still leaving a string of 110 consecutive successful transmissions.

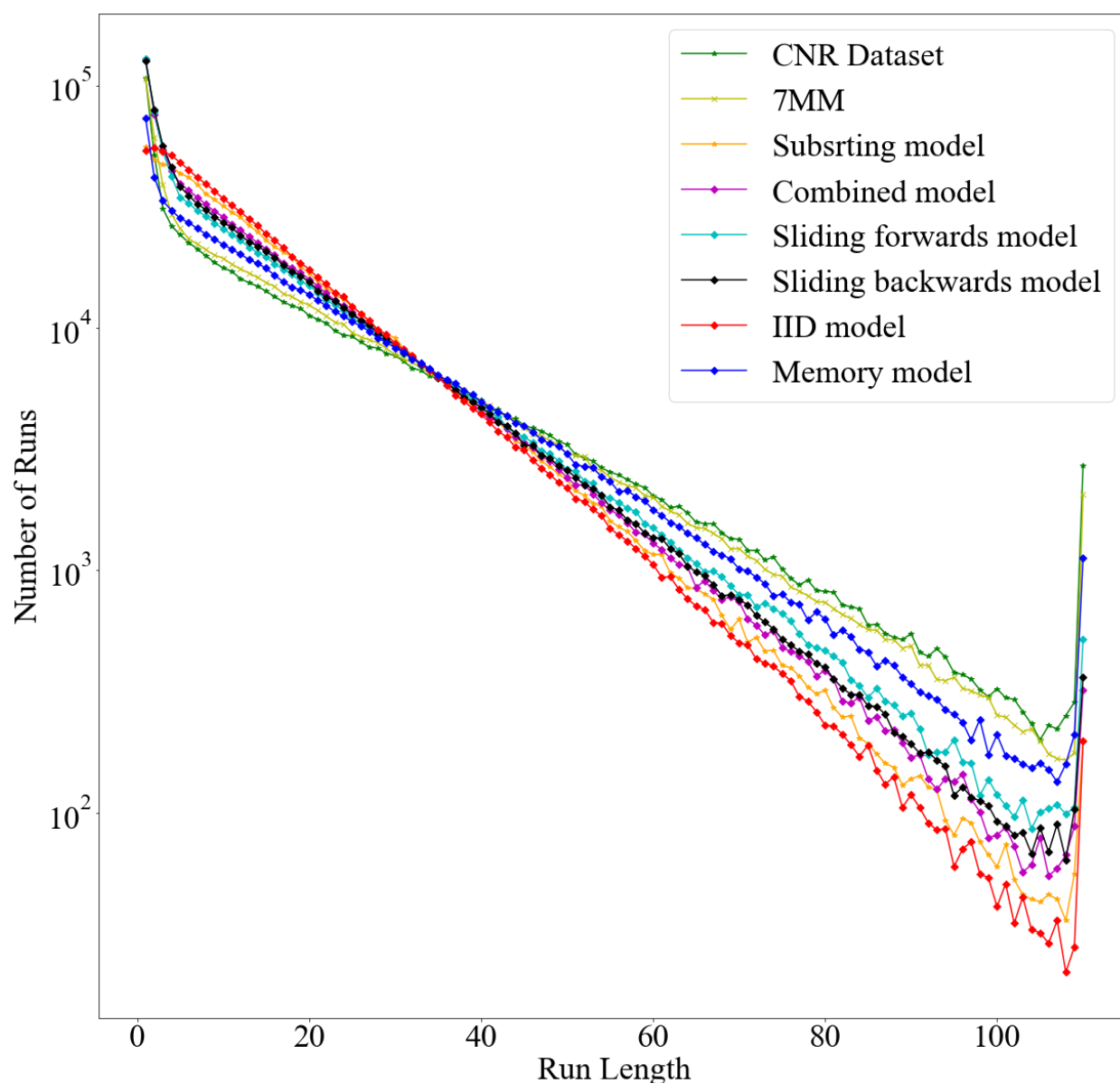


Figure 9-5: Comparison of error-free run lengths produced by each model

7MM model is seen to be the only model that closely resembles the dataset throughout the entire range of error-free run lengths. The other models all follow similar trends to each other. The IID model performs the worst of the compared models. Calculating the RMSLE between the error-free run lengths produced by each model and the error-free run lengths found in the dataset reflects this. The 7MM model shows the lowest RMSLE, while the IID model shows the highest RMSLE. These results are summarised in Table 9-5.

Table 9-5: RMSLE for error-free run lengths produced by each model

Model	Error-free run length
7MM	0.047
Substring model	0.239
Combined model	0.203
Sliding forwards model	0.171
Sliding backwards model	0.196
IID model	0.273
Memory model	0.092

9.7 Homopolymer analysis

In Section 2.2.10.1, it was found that literature indicates that homopolymer runs have an effect on the errors produced within DNA strings. The analysis of the dataset in Section 7.2 confirmed this. Therefore, the ability of each model to represent the effect of homopolymer runs on the errors produced within DNA strings is analysed in this subsection. Supplementary material for this subsection is available in Appendix W.

Figure 9-6 compares the error rate within homopolymers of different lengths produced by each model. This includes homopolymer runs of any type of base. It is seen that the CNR dataset shows a clear increase in error rate as the length of the homopolymer run increases. The Sliding backwards model most closely mimics this increase in error rate. The Combined and Substring models also show an increase in error rate as the length of the homopolymer runs increases, but the increases produced by these models are much less severe than the increase seen in the dataset.

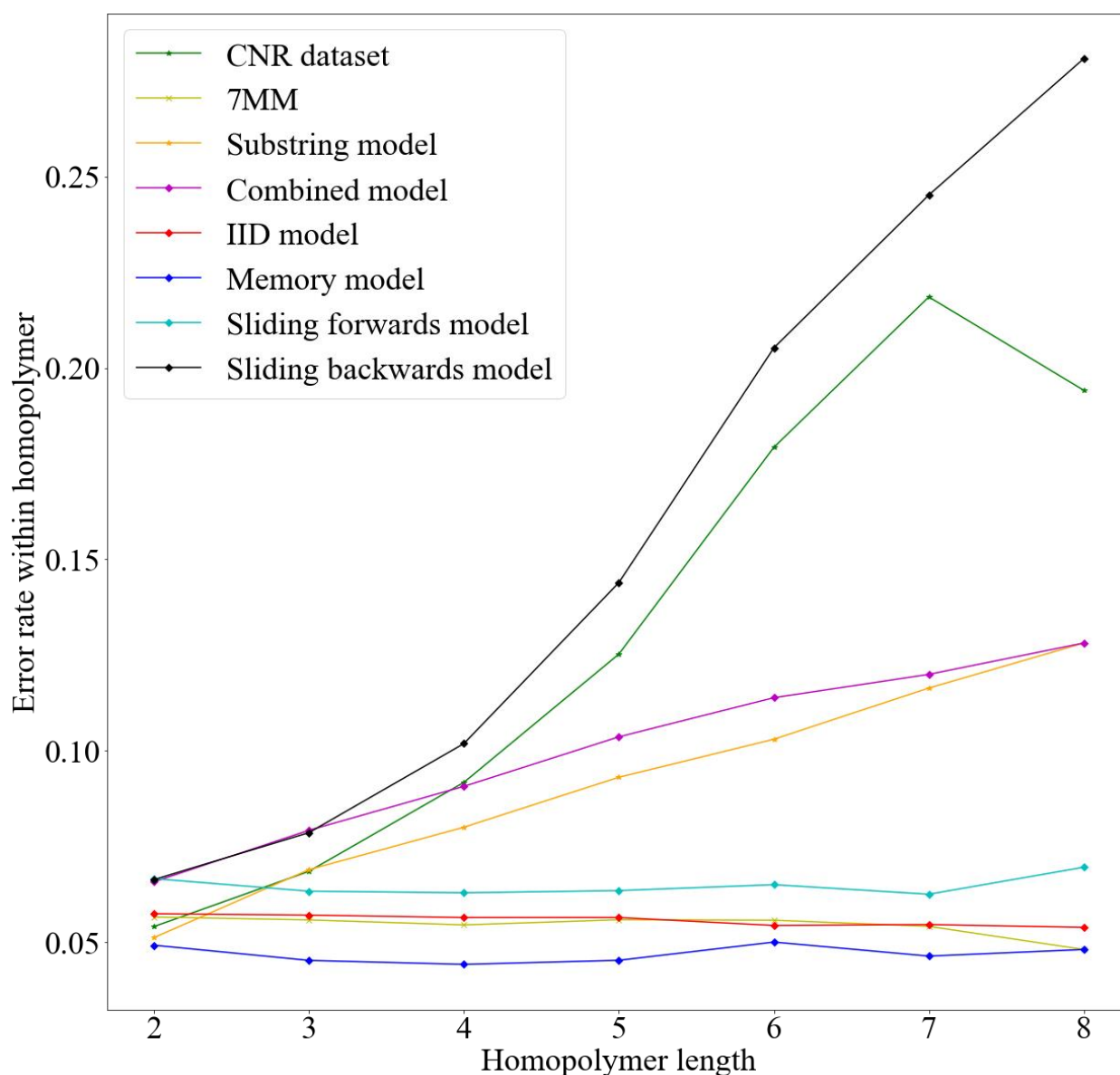


Figure 9-6: Comparison of error rates produced within homopolymers by each model

Section 7.2 indicated that the length of the homopolymer run does not influence the error rate right before or right after the homopolymer, as the error rate does not significantly change for homopolymers of different lengths. The error rate produced by each model at the base right before and the base right after homopolymers of different lengths was also examined. All of the proposed models show similar tendencies to the dataset. In both cases, it is seen that the Substring model produces the closest error rate to the error rate seen in the dataset. The results are depicted in Supplementary Figure W-1 and Supplementary Figure W-2.

Similar analyses were done for error rates within, before and after homopolymer runs of each type of base, i.e. homopolymer runs of A, C, G and T. In each case, the results are similar to those for homopolymers of any type of base. In each case, the Sliding backwards model

represents the error rate within the homopolymer runs the best, while the Substring model represents the error rate right before and right after the homopolymer runs the best. The results are depicted in Supplementary Figure W-3 to Supplementary Figure W-14.

Lastly, the NRMSE is calculated for the error rate produced by each model for each type of homopolymer compared to the error rate found in homopolymers in the dataset. It is seen that in each case the Sliding backwards model performs the best by a large margin. The results are summarised in Table 9-6.

Table 9-6: NRMSE for error rates in homopolymers produced by each model normalised for mean error rate produced by the model

Model	Type of homopolymer				
	Any base	A	C	G	T
7MM	1.523	1.314	2.030	2.065	1.091
Substring model	0.858	0.809	1.214	1.066	0.714
Combined model	1.039	0.914	1.469	1.538	0.801
Sliding forwards model	1.513	1.369	1.867	2.056	1.151
Sliding backwards model	0.249	0.101	0.460	0.260	0.140
IID model	1.513	1.345	1.962	2.041	1.107
Memory model	1.594	1.562	1.767	2.154	1.275

9.8 GC content

In Section 2.2.10.3, literature examining the influence of GC content on the error rate of DNA strings was examined. GC content was shown to have a negative influence on the error rate in DNA strings. In Section 7.4, the dataset was examined and the negative influence of GC content on the error rate was confirmed. It was seen that the GC content of the entire string influences

the error rate of that string. Additionally, it was found that the error rate within a specific substring is influenced by the GC content of that substring. Therefore, this subsection examines the ability of each model to represent the influence of the GC content on the error rate for both complete strings and substrings. Supplementary material for this subsection can be found in Appendix X.

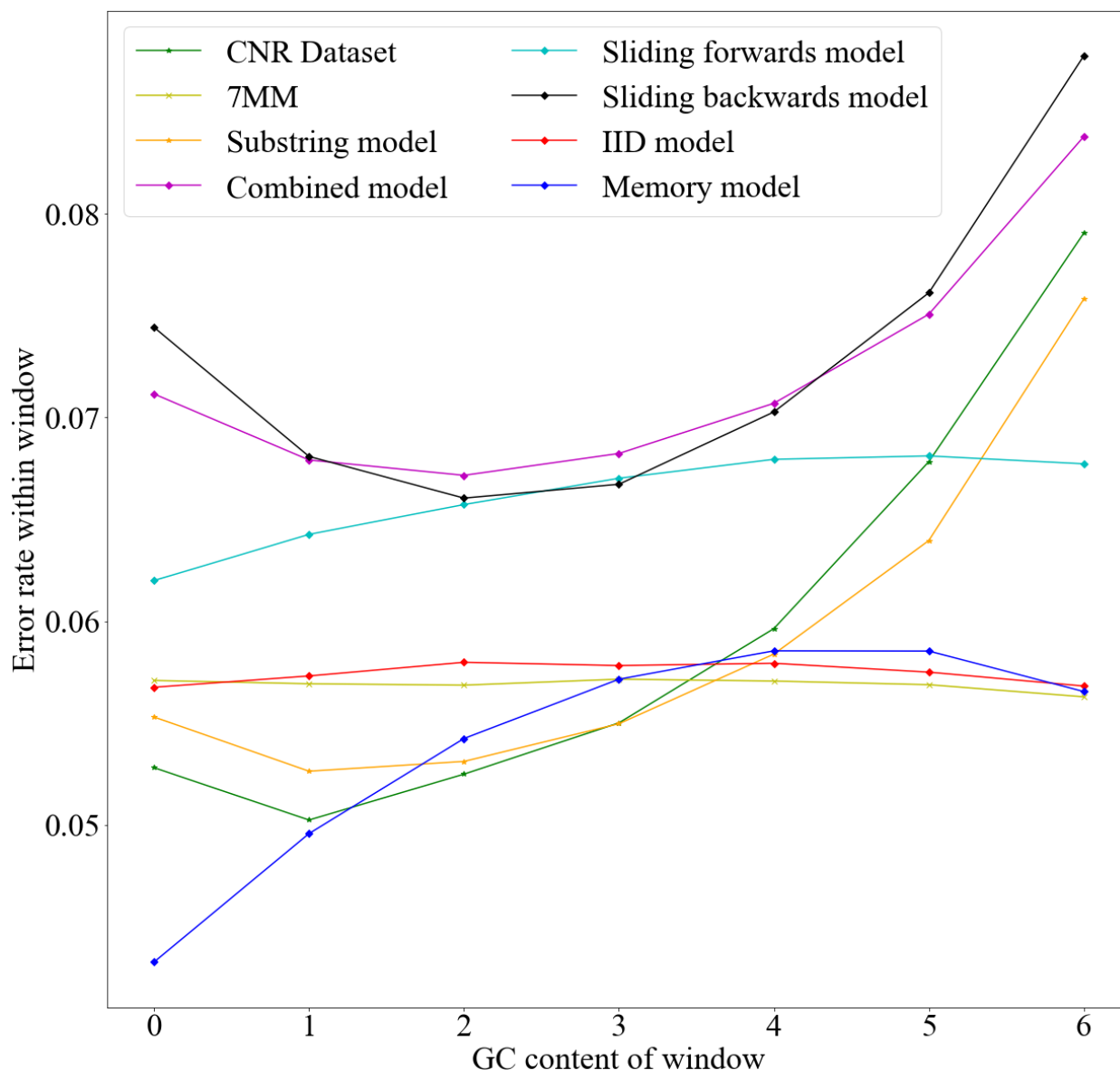


Figure 9-7: Comparison of GC content of windows and error rate produced by each model for windows of length 6

MinION sequencers use *kmers* of length 6 [105], therefore substrings of length 6 are analysed in this section. Figure 9-7 depicts the average error rate for substrings with different GC contents as produced by each model.

Furthermore, the NRMSE between the error rate produced by each model for windows with different GC content values and the dataset was found. It is seen that the Substring model

performs the best. The Sliding backwards and Combined models also perform well. The results are summarised in Table 9-7.

Table 9-7: NRMSE for error rates of length 6 windows with different GC contents produced by each model normalised for mean error rate

Model	NRMSE of error rates of windows of different GC contents
7MM	0.175
Substring model	0.042
Combined model	0.098
Sliding forwards model	0.165
Sliding backwards model	0.090
IID model	0.175
Memory model	0.165

The average error rate produced by each model for strings with different GC content values was also examined. It should be noted that few strings with extremely high or low GC contents exist in the dataset, resulting in outlier values. Therefore, this analysis shall focus on the values with a GC content close to 50%. It is found that the Substring model most closely mimics the dataset, showing a rising error rate as the GC content increases. The results are depicted in Supplementary Figure X-1.

9.9 Running digital sum

In Section 2.2.10.3, it is shown that the RDS may be used to determine the influence of the GC content of a DNA string on the error rate of that string. In Section 7.5, the dataset was analysed using an RDS that is calculated from the start or the back of the string and increases for each base that is a C or G and decreases for each A or T. The error rate of the dataset was shown to increase as the RDS increases above 0 and decreases as the RDS decreases below 0. This subsection examines the strings produced by the models to ascertain whether this trend is reflected by the models. This is done by finding the correlation between RDS and error rate for each model, as well as the NRMSE between error rates for different RDS values when comparing each model to the dataset. RDS values between -10 and 10 are considered in order to reduce the number of outliers. Supplementary material for this subsection is found in Appendix Y.

The Pearson correlation coefficient between the RDS value and the error rate at that RDS value was found for the dataset and for the strings produced by each model. When examining the RDS from the start of the string, it is found that the Substring model has the value closest to that of the dataset. The Combined model and Memory model also perform well. When examining the RDS from the back of the string, the Memory model shows the closest correlation, while the Sliding backwards model also performs well. These results are depicted in Table 9-8.

Table 9-8: Pearson correlation coefficient between RDS value and error rate

Model	RDS calculated from the front of the string	RDS calculated from the back of the string
CNR dataset	0.833	0.908
7MM	0.105	-0.323
Substring model	0.885	0.464
Combined model	0.770	0.419
Sliding forwards model	0.413	0.117
Sliding backwards model	0.394	0.706
IID model	-0.298	-0.010
Memory model	0.902	0.809

Table 9-9: NRMSE for error rate produced by each model for different RDS values

Model	RDS calculated from the front of the string	RDS calculated from the back of the string
7MM	0.037	0.045
Substring model	0.021	0.037
Combined model	0.025	0.036
Sliding forwards model	0.034	0.042
Sliding backwards model	0.028	0.031
IID model	0.056	0.057
Memory model	0.021	0.033

In Table 9-9, the NRMSE values are depicted, comparing the error rate for different RDS values produced by each model to that of the dataset. It is seen that the Substring and Memory models perform the best for RDS values calculated from the front of the DNA strings, as these models share the lowest NRMSE value. The Sliding backwards model has the lowest NRMSE value for RDS values computed from the back of the DNA strings. These findings correspond to the results indicated by the Pearson correlation coefficients summarised in Table 9-8, as the Substring, Sliding backwards and Memory models performed the best in that regard as well.

9.10 Problematic substrings

Section 2.2.10.4 discussed literature indicating that certain substrings may have increased error rates. Additionally, Section 7.3 verified this aspect by examining substrings within the dataset. The effect of GC content on the error rate of the substrings may be reflected in which substrings show the highest and lowest error rates. Furthermore, the effect of homopolymers may be reflected in the same manner. Therefore, the error rates produced by each model in different substrings are investigated in this subsection. Substrings of length 2, 4 and 6 are examined. Supplementary material for this subsection can be found in Appendix Z.

9.10.1 Substrings of length 2

Figure 9-8 compares the error rate produced by each model for each substring of length 2. Substrings are ordered according to ascending error rate in the dataset. The values plotted in Figure 9-8 are given in Supplementary Table Z-1. Substrings are ordered alphabetically in the table.

It is seen that the two substrings with the highest error rate in the dataset are homopolymer runs of G and C. The Sliding backwards model is the model that best mimics the dataset. It is seen that the Sliding backwards model has the same top three substrings with the highest error rates, but does not mimic the error rates of the other substrings in the dataset well.

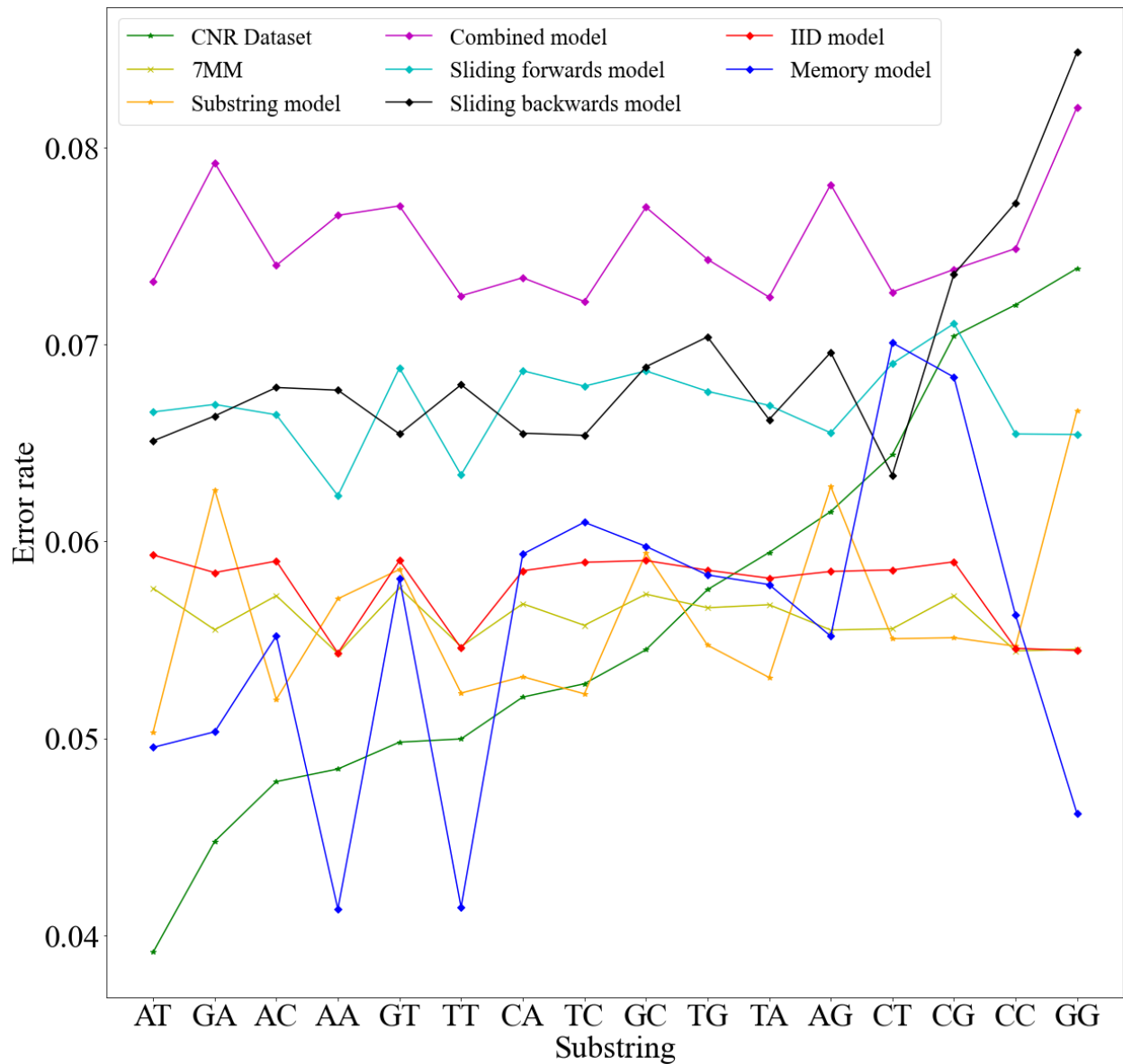


Figure 9-8: Comparison of error rates produced by each model for each substring of length 2

In order to confirm the results derived from Figure 9-8, the NRMSE is found for each model. Firstly, the substrings are ordered according to the error rate produced by each model and compared to the substrings in the dataset, which are ordered according to the error rate in the dataset. Thereafter, the substrings are ordered according to the error rate in the dataset for both the models and the dataset, and the NRMSE is found once again. This indicates if the models produce similar distributions of error rates for different substrings and if the models give similar error rates to the same substrings as found in the dataset. The results are summarised in Table 9-10, along with the number of substrings that are in the same order as the dataset for each model when ordered according to the error rate.

It is seen that the Combined and Sliding backwards models both have the highest number of substrings in the correct order. The Memory model has the lowest NRMSE when substrings are ordered according to error rate produced by the model, whereas the Sliding backwards model has the lowest error rate when substrings are ordered according to error rate in the dataset.

Table 9-10: Comparison of error rates produced by different models for different substrings of length 2

Model	Number of substrings in the correct order	NRMSE of error rates of different substrings when substrings for each model are in increasing order	NRMSE of error rates of different substrings when substrings for each model are ordered according to substrings in the dataset
7MM	8	0.151	0.178
Substring model	8	0.096	0.160
Combined model	9	0.158	0.187
Sliding forwards model	6	0.141	0.167
Sliding backwards model	9	0.108	0.127
IID model	8	0.149	0.184
Memory model	6	0.064	0.175

The standard deviation in error rate produced by each model for substrings of length 2 is also examined. It is seen that the CNR dataset has a large standard deviation, while all of the models have smaller standard deviations. The Memory model has the closest standard deviation to the dataset. The dataset has a standard deviation of 0.010, while the Memory model has a standard deviation of 0.008. The Substring and Sliding backwards models both have a standard deviation of 0.005, and are the closest of the proposed models. The values are summarised in Supplementary Table Z-2, along with the minimum, maximum, median and mean error rate and the skew of the distribution.

9.10.2 Substrings of length 4

When examining the distribution of error rates produced by each model for different substrings of length 4 in Section 7.3.2, it was found that the dataset does not follow a normal distribution, and certain substrings have a much higher error rate than others. Various statistics are now examined

in order to compare the error rate produced by different models for different substrings to the dataset.

The Combined and Sliding backwards models have large maximum error rates and most closely mimic the dataset in this regard, although these two models are unable to represent the substrings with the lowest error rates. The Substring and Memory models have the lowest minimum error rates, and both closely mimic the dataset in this regard. It is seen that the Sliding backwards model has the closest standard deviation to the dataset. The Substring and Memory models perform the next best and both have standard deviation values of 0.008. Additionally, the Substring model has the closest skewness to the dataset, thereby showing a similar distribution. Table 9-11 summarises these values, as well as the minimum, maximum, median and mean error rates.

Table 9-11: Statistics for error rates produced by each model within length 4 substrings

Model	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
Maximum error rate	0.124	0.057	0.098	0.107	0.075	0.142	0.059	0.078
Minimum error rate	0.030	0.052	0.039	0.064	0.057	0.060	0.052	0.033
Mean error rate	0.056	0.056	0.055	0.073	0.066	0.068	0.058	0.055
Median error rate	0.055	0.056	0.055	0.071	0.065	0.066	0.058	0.055
Standard deviation	0.013	0.001	0.008	0.006	0.003	0.009	0.001	0.008
Skew	1.382	-1.118	1.489	2.356	0.287	3.920	-1.891	-0.102

The error rate of each different substring of length 4 is also examined. It is seen that the Substring model mimics the dataset the best, but is unable to represent the substrings with the highest error rates. The Sliding backwards model also mimics the dataset, but is once again unable to represent the substrings with the lowest error rates. Supplementary Figure Z-2 shows the error rate produced by each model for each substring of length 4 according to ascending error rate in

the dataset. Supplementary Figure Z-3 shows the error rate produced by each model for each substring of length 4 ordered according to ascending error rate produced by the model.

Additionally, the NRMSE for the error rate of different substrings of length 4 and the number of substrings that are in the correct order are found in the same manner as was done for substrings of length 2. It is seen that all of the models produce a similar number of substrings that are in the correct order, with values ranging between 120 and 132. The Substring model shows the best NRMSE in both tested cases and, when taking both cases into account, is seen to perform better than the other models overall by a large margin.

Table 9-12: Comparison of error rates produced by different models for different substrings of length 4

Model	Number of substrings in the correct order	NRMSE of error rates of different substrings when substrings for each model are in increasing order	NRMSE of error rates of different substrings when substrings for each model are ordered according to substrings in the dataset
7MM	130	0.223	0.244
Substring model	129	0.102	0.146
Combined model	128	0.170	0.194
Sliding forwards model	132	0.188	0.227
Sliding backwards model	120	0.126	0.168
IID model	128	0.223	0.248
Memory model	132	0.110	0.243

Due to the large number of different substrings of length 4, the error rate of each individual substring cannot be examined as was done with substrings of length 2. Therefore, only the substrings with the ten highest and ten lowest error rates are examined. These substrings are examined to evaluate whether the models allocate the highest and lowest error rates to the same substrings as the dataset.

It is found that all of the ten substrings with the highest error rates in the dataset contain homopolymer runs of C or G. The Substring model mimics this the best, and all but one of the substrings with the highest error rate produced by this model contain homopolymers of C or G. The Sliding backwards and Combined models also mimic this. For each of these two models, all

but two of the substrings with the highest error rates produced by the model contain homopolymers of C or G.

It is found that the Substring and Combined models both share six of the strings with the highest ten error rates with the dataset. The Sliding backwards model performs the next best, sharing five of the ten strings. The Substring model shares three of the top ten strings with the lowest error rates with the dataset.

The ten substrings of length 4 with the highest and lowest error rates produced by each model are given in Supplementary Table Z-3 and Supplementary Table Z-4. For each model, the substrings that also appear in the tables for the dataset are highlighted in each table.

9.10.3 Substrings of length 6

In this section, substrings of length 6 are examined. Figure 9-9 depicts the distribution of error rates produced by each model for different substrings. It is seen that the dataset does not follow a normal distribution and has certain substrings with very high error rates. The Sliding backwards model represents this the best. None of the models are capable of representing the substrings in the dataset with the lowest error rates successfully.

Table 9-13: Statistics for error rates produced by each model within length 6 substrings

Model	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
Maximum error rate	0.226	0.061	0.127	0.128	0.081	0.260	0.062	0.082
Minimum error rate	0.024	0.050	0.038	0.057	0.054	0.054	0.048	0.031
Mean error rate	0.056	0.055	0.055	0.071	0.065	0.068	0.058	0.055
Median error rate	0.054	0.055	0.054	0.070	0.065	0.065	0.058	0.055
Standard deviation	0.015	0.001	0.008	0.006	0.004	0.011	0.001	0.007
Skew	2.112	0.003	1.465	2.036	0.377	5.565	-0.433	0.034

Firstly, various statistics are now examined to confirm the results derived above. The dataset shows a large skewness, which is most closely matched by the Combined model, followed by the

Substring model. The Sliding backwards model has a much larger positive skewness, confirming that the model does not represent substrings with lower error rates accurately. The large standard deviation seen in the error rates of different substrings within the dataset is only represented by the Sliding backwards model. All of the models have lower standard deviations than the dataset. Table 9-13 summarises these statistics derived from the error rates of different substrings of length 6, including the minimum, maximum, median and mean error rates produced by each model.

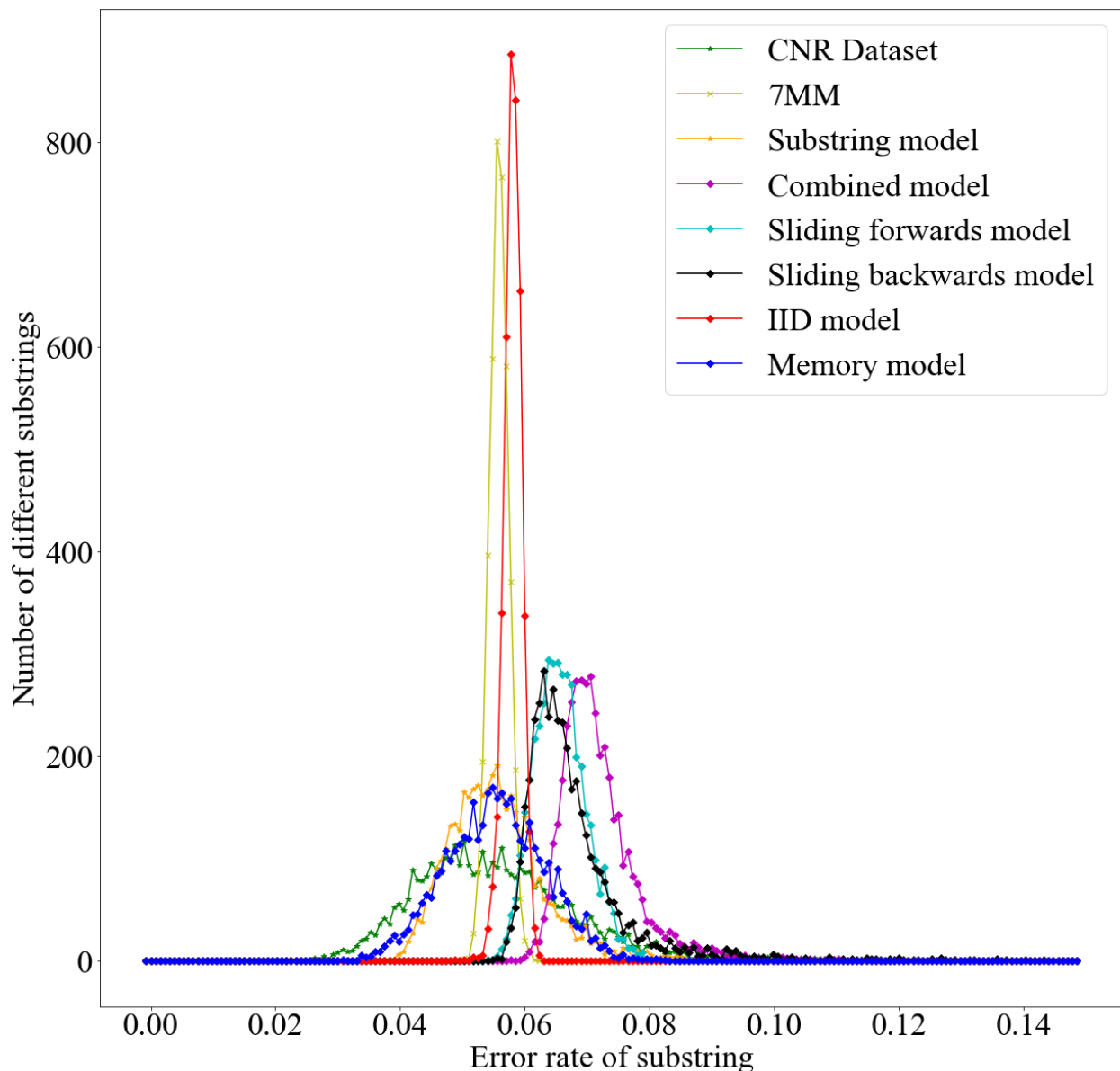


Figure 9-9: Comparison of distribution of error rates between different substrings of length 6 produced by different models

Figure 9-10 depicts the error rate produced by each model for each substring, ordered according to the error rates produced by that model. It is seen that the Memory model has substrings with

the most similar error rates to the dataset on the low end of the spectrum. On the high end of the spectrum, the Sliding backwards and Combined models have substrings with the most similar error rates to the dataset.

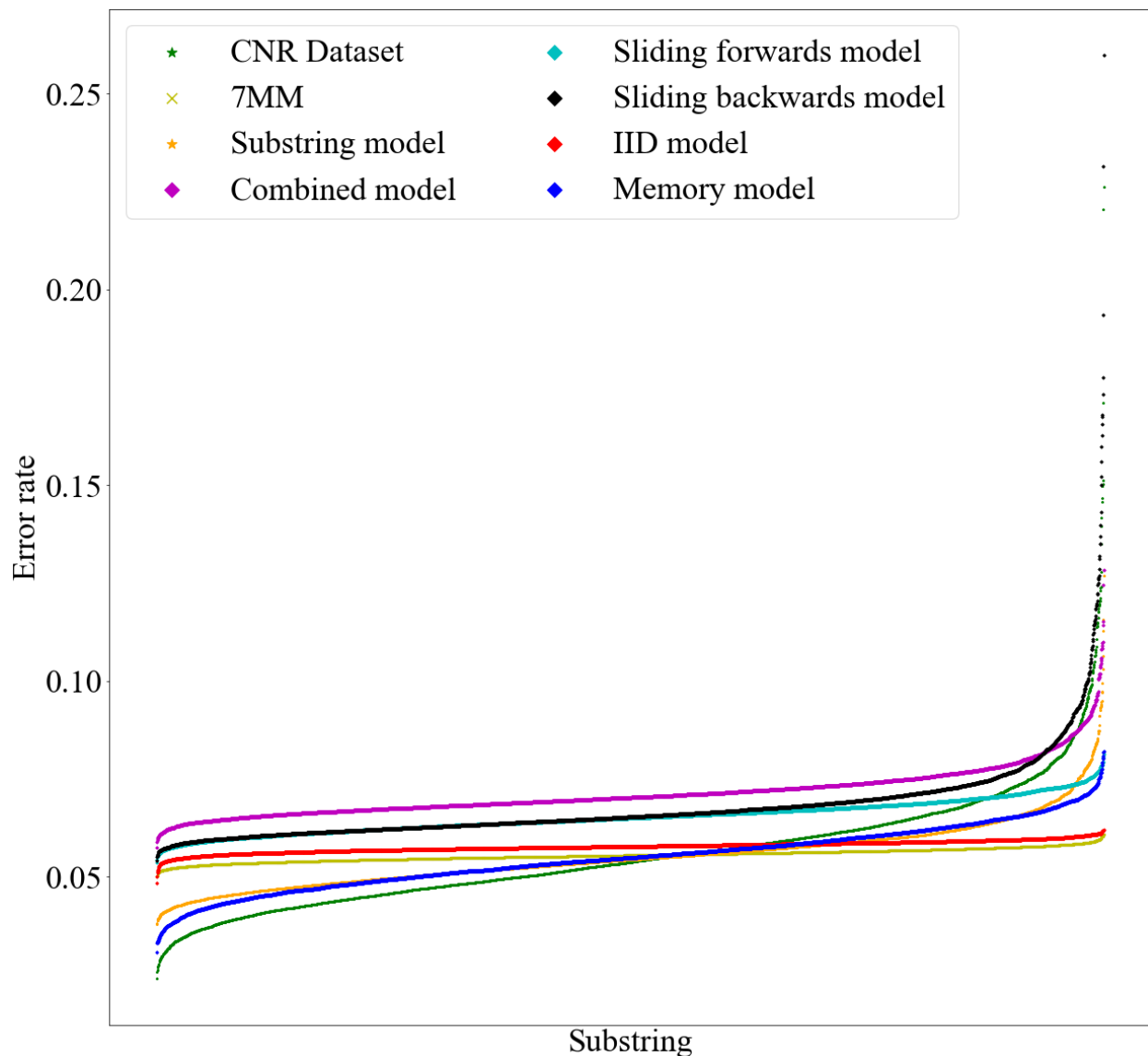


Figure 9-10: Comparison of error rates produced by each model for each substring of length 6 ordered according to error rate produced by the model

To determine whether the models produced similar error rates for each substring than the error rate for that substring in the dataset, substrings are now examined according to the error rate in the dataset. Figure 9-11 shows the error rate produced by each model for each substring of length 6, with substrings ordered according to ascending error rate in the dataset. The Sliding backwards model mimics this order in the dataset the best, as a marked increase in error rate produced by this model is seen at the right of the figure, along with an increase shown by the dataset. However, the Sliding backwards model is unable to represent the substrings with the lowest error rates. The

Substring model also mimics the increase in error rate from left to right shown by the dataset, but is unable to represent the substrings with the highest error rates accurately. All of the proposed models show an oscillation in error rates between consecutive substrings when substrings are ordered according to ascending error rate in the dataset. This indicates that the models do not produce the same error rate as the dataset for each substring.

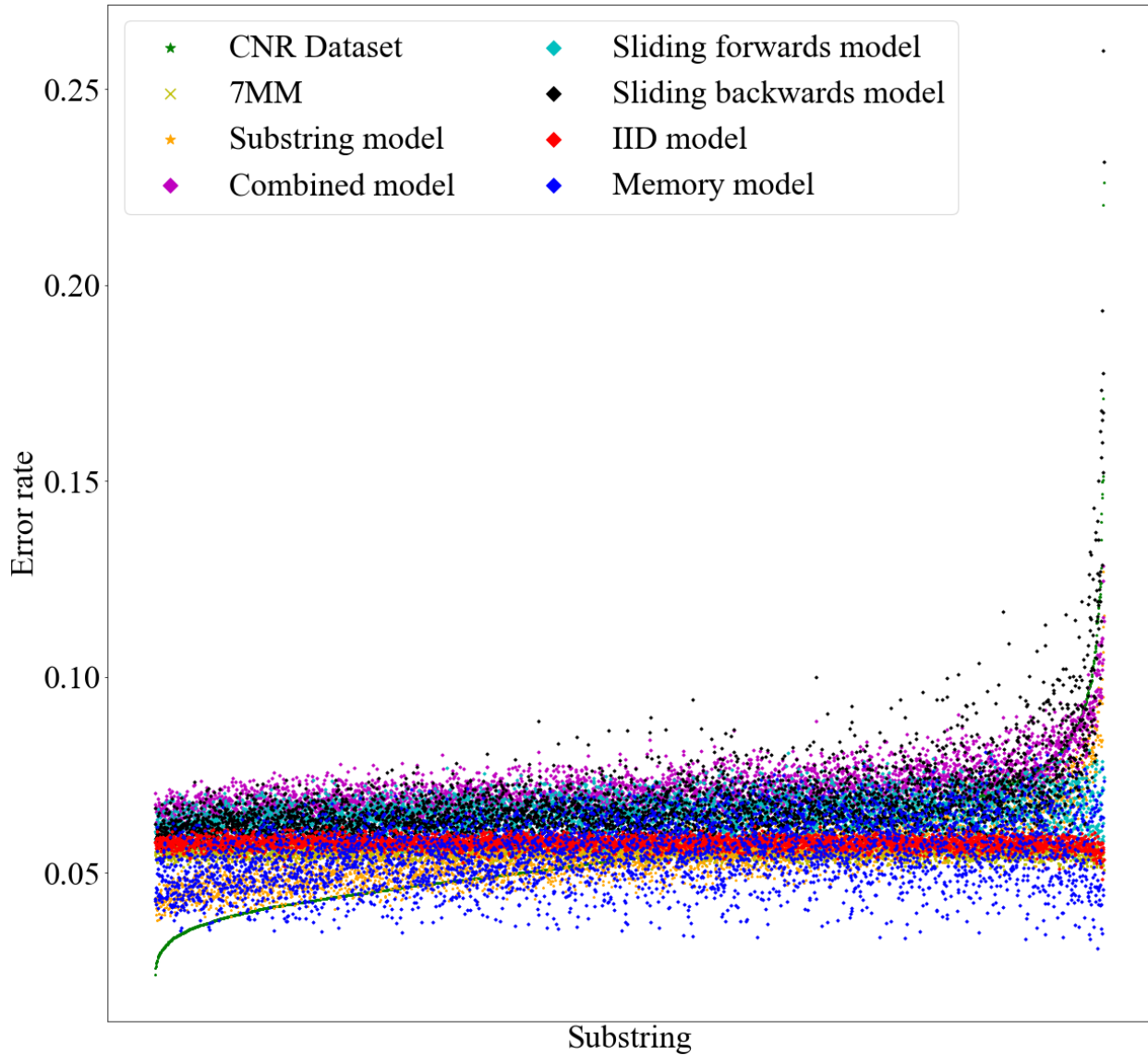


Figure 9-11: Comparison of error rates produced by each model for each substring of length 6 ordered according to error rate in the dataset

To confirm the results derived from the figures above, the NRMSE for the error rate of different substrings of length 6 and the number of substrings that are in the correct order are found in the same manner as was done for substrings of lengths 2 and 4. All of the models produce between 2026 and 2049 substrings that are in the correct order. The Substring model shows the best NRMSE in both the tested cases. This is similar to the results found for substrings of length 4.

Table 9-14: Comparison of error rates produced by different models for different substrings of length 6

Model	Number of substrings in the correct order	NRMSE of error rates of different substrings when substrings for each model are in increasing order	NRMSE of error rates of different substrings when substrings for each model are ordered according to substrings in the dataset
7MM	2049	0.232	0.262
Substring model	2026	0.117	0.148
Combined model	2032	0.174	0.198
Sliding forwards model	2052	0.200	0.246
Sliding backwards model	2032	0.122	0.179
IID model	2044	0.237	0.265
Memory model	2044	0.135	0.262

Once again, the ten substrings with the highest and lowest error rates are examined. It is found that the Substring model performs the best for substrings with the highest error rates, as the model shares eight substrings with the dataset. The Combined and Sliding backwards models also perform well, sharing seven and six strings with the dataset respectively.

None of the models share many of the substrings with the lowest error rates with the dataset. Therefore, the characteristics of the substrings with the lowest error rates are examined. None of the substrings with the lowest error rates in the dataset contain homopolymer run lengths of longer than 2. Furthermore, all of the substrings have a 50% or less GC content, corresponding to the findings of Section 7.4 that substrings with a lower GC content have reduced error rates.

The Combined model mimics the characteristics of the substrings with the lowest error rates in the dataset the best. The substrings with the lowest error rates produced by the Combined model contain no homopolymer run lengths of longer than 2, and all have a GC content of 50% or less.

The Substring and Sliding backwards models also mimic the dataset. Exceptions include one of the substrings with the lowest error rate produced by the Substring model that has a homopolymer of length 3, while two of the substrings produced by the Sliding backwards model have homopolymer runs of length 3. Supplementary Table Z-5 depicts the ten substrings with the highest error rate produced by each model, while Supplementary Table Z-6 depicts the ten

substrings with the lowest error rate produced by each model. For each model, the substrings that also appear in the table for the dataset are highlighted in each table.

9.11 Swap errors

This subsection examines the number of swap errors produced by each model. Section 2.2.9 indicated that swap errors are prevalent in other types of communication channels. As it was shown that swap errors are not often mentioned in literature regarding DNA-based storage, and Section 7.6 indicated that swap errors comprise a very small percentage of total errors within the dataset, none of the proposed models explicitly consider swap errors.

The percentage of errors produced by each model that make up swaps are summarised in Table 9-15. Furthermore, the probability for a swap to occur is calculated from the total error rate and percentage of errors that make up swap errors.

Table 9-15: Percentage of errors that make up swap errors produced by each model

Model	Percentage of errors making up swap errors	Total error rate	Probability for swap error to occur
CNR Dataset	2.24 %	0.058	0.0013
7MM	1.04 %	0.057	0.0006
Substring model	1.00 %	0.058	0.0006
Combined model	1.00 %	0.071	0.0007
Sliding forwards model	1.02 %	0.069	0.0007
Sliding backwards model	1.07 %	0.071	0.0008
IID model	0.85 %	0.058	0.0005
Memory model	0.74 %	0.057	0.0004

It is seen that the dataset contains a much larger chance for a swap error to occur than produced by any of the models. The percentage of errors that make up swaps in the dataset is more than double that of any of the models. The five proposed models all produced similar numbers of swap errors.

The IID and Memory models perform the worst in this aspect. These results indicate that, while swaps only make up a small percentage of the total errors in the dataset, models that do not

explicitly include a mechanism to represent swap errors cannot accurately simulate the occurrence of swap errors.

9.12 Discussion

Five models that each make use of a different modelling method were proposed in order to simulate the DNA data storage channel. The models were compared to each other, existing models and a testing dataset.

The existing models that were considered are the IID model and the Memory model. While representing the overall error rate accurately, the IID model performed poorly in the other aspects of the dataset that were examined. The Memory model also produced an overall error rate similar to that of the dataset.

The Memory model represents error runs well, but produces too few long run errors. Furthermore, the Memory model produces much fewer error bursts than exists in the dataset. It was found that the Memory model does not represent error-free runs and errors generated by homopolymers accurately. Some of the proposed models are able to improve on these areas as discussed in the following subsections.

9.12.1 7MM

The 7MM model consists of a seven-state Markov model. The model produces a string of outcomes, i.e. correct transmissions or insertion, deletion or substitution errors. The model does not take the bases of the input string into account when generating observations. The string of observations is then populated by specific bases by considering the input string.

It was found that the 7MM model performed the best in various aspects, including the modelling of error runs and bursts, as well as error-free runs. Furthermore, the 7MM model produced the number of error-free strings closest to the dataset. The model also produced an average string length and error rate similar to the dataset. However, the model was not able to represent string-specific characteristics accurately. These characteristics include the effect of GC content and homopolymer runs on the error rate, as well as the effect of problematic substrings.

9.12.2 Substring model

The Substring model uses characteristics of different substrings. The model uses consecutive windows of length 6 to generate a string of observations while considering the substrings of the input string. The string of observations is then populated by considering the bases of the input strings.

The Substring model performed the best at representing the effect of GC content on errors. The model also represented the effect of homopolymer runs on errors, but is not as accurate as some of the other models in this regard. The model was able to accurately represent the distribution of error rates of different substrings of length 6, except for substrings with very high error rates. The model produced an average string length and error rate close to that of the dataset. However, the model was unable to accurately represent error runs and bursts or error-free runs and strings.

9.12.3 Combined model

The Combined model combines the methods of the 7MM with characteristics of different substrings. The Markov model creates a string of observations without consulting the bases of the input string. Another string of observations is created using consecutive windows of length 6 while considering the substrings of the input string. The two strings of outcomes are then combined and populated with bases.

The Combined model was able to represent the effect of GC content on errors. The model also represented the effect of homopolymer runs on errors, but is not as accurate as some of the other models in this regard. The model was able to represent the distribution of error rates of different substrings of length 6 but to a lesser degree than the Substring model. The model produced an average string length close to that of the dataset. The model was also able to accurately represent error runs and bursts. However, the model produced a much higher error rate than the dataset, and was unable to accurately represent error-free runs and strings.

9.12.4 Sliding forwards model

The Sliding forwards model uses a similar method to the Combined model, but instead of using consecutive windows this model uses a sliding window. This model was also able to accurately represent error runs and bursts. However, the model produced a much higher error rate than the dataset. Furthermore, the model could not represent error-free runs and strings accurately, nor the effect of GC content or homopolymer runs on the error rate.

9.12.5 Sliding backwards model

The Sliding backwards model uses a similar method to the Combined model, but instead of using consecutive windows this, model uses a sliding window that starts at the back of the string and moves towards the start of the string.

The Sliding backwards model produces an average string length similar to the dataset, but produces much fewer error-free strings. Furthermore, the model produces a much higher error rate than the dataset, and is unable to represent error-free runs accurately.

The model was able to produce error runs and bursts accurately, and had a similar accuracy in representing long error bursts to the 7MM model. The model was the best at representing the effect of homopolymer runs on the error rate. The model was able to accurately represent the distribution of error rates of different substrings of length 6, except for substrings with very low error rates. The model was able to represent the effect of GC content on errors.

9.13 Summary

In this chapter, five proposed models were examined. This includes the 7MM, Substring, Combined, Sliding forwards and Sliding backwards models. The models were compared to each other and to a testing set. Furthermore, the models were compared to two models from literature – the regularly used IID model and the state-of-the-art Memory model [105]. The advantages and shortcomings of each model were examined.

The comparison of the different models in various areas are summarised in Table 9-16. Symbols are used for brevity, with an = indicating that the model performs adequately in that area. An – indicates that the model performs poorly in that area, while an + indicates that the model performs well in that area.

Table 9-16: Comparison of different models

Model	General statistics	Error runs	Error bursts	Error-free runs	Homopolymers	GC content	Problematic substrings
IID	=	–	–	–	–	–	–
Memory	=	=	=	=	–	–	=
7MM	+	+	+	+	–	–	–
Substring	+	–	–	–	=	+	+
Combined	=	+	+	–	=	+	+
Sliding forwards	=	+	+	–	–	–	=
Sliding backwards	=	+	+	–	+	+	+

When examining general statistics, the 7MM model was found to perform the closest to the dataset, producing a similar overall error rate, string length and number of error-free runs to the

dataset. The 7MM, Combined, Sliding forwards and Sliding backwards models are all able to accurately model error runs, but the 7MM and Sliding backwards models performed better than the rest. The 7MM, Combined, Sliding forwards and Sliding backwards models are all able to accurately represent bursts of errors. The 7MM model is the only model that accurately represents error-free runs of different lengths.

The Sliding backwards model represents the errors induced by homopolymer runs the best. The Substring model represents the effect of GC content on the error rate the best, but the Combined and Sliding backwards models are also capable of representing this effect. The Substring model performs the best when compared to the dataset in terms of the effect of the RDS on the error rate.

The Substring and Sliding backwards models both represented the distribution of error rates between different substrings to a greater extent than the other models when compared to the dataset. The substrings with the highest and lowest error rates generated by these two models were similar to the substrings with the highest and lowest error rates in the dataset. The Combined model also performed well in this regard.

All of the models produced a smaller percentage of swap errors than the dataset.

CHAPTER 10 CONCLUSION

10.1 Research problem

This study attempted to create a model for the DNA data storage process that addresses the shortcomings of existing models. Various existing models were examined in Section 2.5 and it was concluded that none of the existing models were able to represent all of the characteristics of the DNA data storage channel accurately. It was especially evident that even state-of-the-art channel models, such as the Memory model [24] could not represent long runs of errors accurately. As literature [105] has found that the Memory model outperforms current deep learning models such as DeepSimulator [30], deep learning models were not assessed further.

10.2 Method

10.2.1 Dataset

In order to model the DNA data storage channel, the CNR dataset [32] was selected to train the model. The dataset consists of input DNA strings that were sequenced using Nanopore technology, as well as the corresponding reads that contain erroneous bases. An algorithm based on the Levenshtein distance [153] was used to identify the positions and types of errors within the strings. The dataset was divided into a training set and testing set.

10.2.2 Existing models

Existing models were identified to use for comparison. The models identified were the IID model and the Memory model [105]. Both of these models were implemented in Python in order to train them on the same training set used for the proposed models.

10.2.3 Proposed models

A Markov model was selected as a starting point. Markov models with different parameters were trained and evaluated. The best performing model was selected, and various areas in which the model performed inadequately were identified. Various models were then created in an attempt to improve on the Markov model, resulting in five proposed models.

1. 7MM

The 7MM model consists of a seven-state Markov model which is used to produce a string of outcomes – transmissions, deletions, insertions or substitutions. The string of outcomes is then populated with specific bases from the input string. Additional probability

distributions are included to determine which bases are substituted or inserted and to represent increased error rates at the beginning and end of the DNA strings.

2. Substring model

This model uses consecutive windows of length 6 to analyse different substrings within the input DNA string. Probability distributions representing the occurrence of errors within different substrings are used to generate a string of outcomes, which is then populated with specific bases.

3. Combined model

The Combined model incorporates a Markov model and consecutive windows of length 6 to analyse different substrings. A string of outcomes is created by the Markov model, similar to that of the 7MM model. A second string of outcomes is created according to probability distributions that represent the occurrence of errors within different substrings, similar to the Substring model. The two strings of outcomes are combined to create a single string of outcomes that contains errors generated by both methods. This string of outcomes is then populated with specific bases.

4. Sliding forwards model

This model is implemented similarly to the Combined model, but instead of using consecutive windows of length 6, the model uses a sliding window with a step size of one.

5. Sliding backwards model

This model is implemented similarly to the Sliding forwards model, but the sliding window starts at the back of the DNA string and moves to the start.

10.2.4 Areas for comparison

Areas to evaluate the dataset and models were identified. These areas include general statistics such as the error rate produced by the model. Furthermore, error runs, bursts and error-free runs produced by each model were examined. The errors induced by homopolymers of different lengths and the GC content of DNA strings were analysed. The error rates of different substrings were examined. Lastly, the number of swap errors produced by each model was compared.

10.3 Results

Most of the tested models were able to match the error rate in the training set. The exceptions are the Sliding forwards, Sliding backwards and Combined models, which showed increased error

rates. The 7MM model produced the closest number of error-free strings to the dataset, followed by the Memory model. All of the models produced similar average string lengths.

The IID and Substring models produced much fewer error runs than the dataset. By calculating the RMSLE, it was shown that all of the other models performed similarly to the dataset. The 7MM and Sliding backwards models performed slightly better than the other models for long run lengths. When considering error bursts, the IID, Substring and Memory models performed the worst. The other models all delivered error bursts that closely mimic the occurrence of bursts in the dataset. The 7MM model was the only model that could closely mimic the dataset in the representation of error-free runs of different lengths.

When considering errors induced by homopolymers of different lengths, the Sliding backwards model most closely mimicked the dataset and had the lowest NMRSE value. The Substring and Combined models also performed favourably, with lower NMRSE than the other models. All of the other models were unable to represent this phenomenon, and performed similarly to each other.

The Substring model was the only model that could effectively represent the errors induced by different GC contents in substrings of a DNA sequence, and showed a lower NRMSE than any of the other models. The Combined and Sliding backwards models also performed favourable, but to a lesser degree than the Substring model. All of the other models were unable to represent this dependency and produced much larger NRMSE values.

When examining the error rate within different substrings, none of the models were able to adequately represent this aspect. When ordering substrings according to the error rate, the order produced by each model differed from that of the dataset. The Combined, Sliding backwards and Substring models performed the best, yielding results that are the closest to that observed within the dataset. These models had the lowest NMRSE values, and the Substring model performed the best of the three.

All of the models produced a smaller percentage of swap errors than the dataset. The proposed models all produced similar percentages of swap errors of between 1.00% and 1.07%, compared to the 2.24% of swap errors observed in the dataset. The IID and Memory models performed the worst in this regard, each producing a swap error rate of 0.85% and 0.74% respectively.

10.4 Summary

In order to simulate the DNA data storage process, several modelling methods were proposed. Models were created from a Nanopore sequencing dataset using these methods. This resulted in

different models, each with different strong and weak points. The models may therefore each be used in different cases. While multiple models were examined, the following three models performed the best in various areas, and are therefore proposed for future use.

The 7MM model is the most accurate in representing error runs and bursts. This is the only proposed model that can accurately represent error-free runs. The model also produces a similar error rate to the DNA data storage channel. The model does not consider string-specific characteristics such as GC content or homopolymer runs. Therefore, it is proposed that this model is used in cases where error runs and bursts are an important factor, but the string-specific characteristics are not required.

The Substring model can accurately represent the effect of GC content on the error rate of a DNA string. The model also produces a similar error rate to the DNA data storage channel. Homopolymer runs are also represented, but to a lesser degree of accuracy than the Sliding backwards model. However, the model does not accurately represent error runs, bursts and error-free runs. It is therefore proposed that this model is used in cases where string-specific characteristics are important, but not the occurrence of error runs and bursts.

The Sliding backwards model can accurately represent error runs and bursts. Furthermore, the model is able to accurately represent the effect of GC content and homopolymer runs on the error rate. The model, however, does not accurately represent error-free runs and also produces a much higher overall error rate than the dataset. It is therefore proposed that this model is used in cases where error runs and bursts, as well as string-specific characteristics, are important, but not the overall error rate.

Additionally, the complexity of the proposed models should be noted. The 7MM model was originally trained on a very small training set of only 1349 DNA strings, and performs competitively when compared to the other models. The other models require multiple examples of each different substring of length 6 in order to determine the probability distributions that define those models. Such a small dataset cannot include enough different samples to train the other models, giving the 7MM model an added advantage due to its reduced complexity. Furthermore, the proposed Substring and Sliding backwards models are defined by fewer probability distributions than the Memory model, therefore these two models also have lower complexities than the Memory model.

10.5 Future work

In this study, five models were proposed in order to represent the errors within the DNA data storage channel. The models were assessed in various aspects, and shortcomings were noted.

The models were assessed in their ability to represent various characteristics of the DNA data storage channel. These characteristics include the overall error rate, length of strings generated and number of error-free strings generated. Furthermore, error runs and bursts, as well as error-free runs, were analysed. The effects of GC content and homopolymer runs on errors were also analysed, alongside the occurrence of problematic substrings. Although all of these characteristics were represented in at least one of the models, none of the models were able to encompass all of these characteristics accurately.

Therefore, it is proposed that future research includes the development of a single model that is able to represent all of these characteristics of the DNA data storage channel accurately. As it is a complex task to combine all of these characteristics into a single model, it is proposed that deep learning methods should be explored further. As an example, a Generative Adversarial Network (GAN) may be used. Additionally, more effective ways of combining Markov models with characteristics derived from substring data may be explored, such as adjusting the probabilities of the Markov model.

The models were developed and assessed using a dataset that consists of Nanopore sequencing data. It is proposed that future work includes the application of the modelling methods proposed in this study to different datasets. It can then be determined if the methods are applicable to other sequencing technologies.

Although the proposed models are capable of generating errors within an input DNA sequence, the models do not include PCR bias or sampling of the pool of DNA strings. These stages of the DNA data storage process may also be modelled for a more complete representation of the channel.

Lastly, none of the proposed models reflects the occurrence of swap errors in the dataset. Future work may therefore include explicitly adding swap errors to the models.

REFERENCES

- [1] D. Reinsel, J. Gants, and J. Rydning, "The digitization of the world - from edge to core," <https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf>.
- [2] L. Organick, Y. J. Chen, S. Dumas Ang, R. Lopez, X. Liu, K. Strauss, and L. Ceze, "Probing the physical limits of reliable DNA data retrieval," *Nat. Commun.*, vol. 11, no. 1, Dec. 2020, doi: 10.1038/s41467-020-14319-8.
- [3] J. Dabney, M. Knapp, I. Glocke, M. T. Gansauge, A. Weihmann, B. Nickel, C. Valdiosera, N. García, S. Pääbo, J. L. Arsuaga, and M. Meyer, "Complete mitochondrial genome sequence of a Middle Pleistocene cave bear reconstructed from ultrashort DNA fragments," *Proc Natl Acad Sci U S A*, vol. 110, no. 39, pp. 15758–15763, Sep. 2013, doi: 10.1073/pnas.1314445110.
- [4] A. Extance, "How DNA could store all the world's data," *Nature*, vol. 537, pp. 22–25, 2016.
- [5] R. Heckel, G. Mikutis, and R. N. Grass, "A characterization of the DNA data storage channel," *Sci Rep*, vol. 9, no. 1, Dec. 2019, doi: 10.1038/s41598-019-45832-6.
- [6] E. Sidransky, "Hybridization." Accessed: Nov. 25, 2023. [Online]. Available: <https://www.genome.gov/genetics-glossary/hybridization>
- [7] S. M. H. T. Yazdi, H. M. Kiah, E. Garcia-Ruiz, J. Ma, H. Zhao, and O. Milenkovic, "DNA-Based Storage: Trends and Methods," *IEEE Trans. Mol., Biol., and Multi-Scale Commun.*, vol. 1, no. 3, pp. 230–248, Sep. 2015, doi: 10.1109/TMBMC.2016.2537305.
- [8] C. Delahaye and J. Nicolas, "Sequencing DNA with nanopores: Troubles and biases," *PLoS One*, vol. 16, no. 10 October, Oct. 2021, doi: 10.1371/journal.pone.0257521.
- [9] M. G. Ross, C. Russ, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe, "Characterizing and measuring bias in sequence data," *Genome Biol.*, vol. 14, no. 5, May 2013, doi: 10.1186/gb-2013-14-5-r51.
- [10] J. Shendure, S. Balasubramanian, G. M. Church, W. Gilbert, J. Rogers, J. A. Schloss, and R. H. Waterston, "DNA sequencing at 40: Past, present and future," *Nature*, vol. 550, no. 7676, Oct. 2017, doi: 10.1038/nature24286.
- [11] G. M. Church, Y. Gao, and S. Kosuri, "Next-Generation Digital Information Storage in DNA," *New Series*, vol. 337, no. 6102, p. 1628, 2012, doi: 10.1126/science.

- [12] S. Kosuri and G. M. Church, "Large-scale de novo DNA synthesis: Technologies and applications," *Nature Methods*, vol. 11, no. 5. Nature Publishing Group, pp. 499–507, 2014. doi: 10.1038/nmeth.2918.
- [13] J. N. Zadeh, C. D. Steenberg, J. S. Bois, B. R. Wolfe, M. B. Pierce, A. R. Khan, R. M. Dirks, and N. A. Pierce, "NUPACK: Analysis and design of nucleic acid systems," *J. Comput. Chem.*, vol. 32, no. 1, pp. 170–173, Jan. 2011, doi: 10.1002/jcc.21596.
- [14] S. M. H. T. Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic, "A rewritable, random-access DNA-based storage system," May 2015, [Online]. Available: <http://arxiv.org/abs/1505.02199>
- [15] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. Leproust, B. Sipos, and E. Birney, "Towards practical, high-capacity, low-maintenance information storage in synthesized DNA," *Nature*, vol. 494, no. 7435, pp. 77–80, Feb. 2013, doi: 10.1038/nature11875.
- [16] R. N. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. J. Stark, "Robust chemical preservation of digital information on DNA in silica with error-correcting codes," *Angewandte Chemie - International Edition*, vol. 54, no. 8, pp. 2552–2555, Feb. 2015, doi: 10.1002/anie.201411378.
- [17] L. Organick, S. Dumas Ang, Y.-J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M. Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, C. Takahashi, S. Newman, H.-Y. Parker, C. Rashtchian, K. Stewart, G. Gupta, R. Carlson, J. Mulligan, D. Carmean, G. Seelig, L. Ceze, and K. Strauss, "Scaling up DNA data storage and random access retrieval." Accessed: Nov. 23, 2023. [Online]. Available: <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/12/Scaling-up-DNA-data-storage-and-random-access-retrieval.pdf>
- [18] T. T. Nguyen, K. Cai, K. A. Schouhamer Immink, and H. M. Kiah, "Capacity-Approaching Constrained Codes With Error Correction for DNA-Based Data Storage," *IEEE Trans Inf Theory*, vol. 67, no. 8, pp. 5602–5613, Aug. 2021, doi: 10.1109/TIT.2021.3066430.
- [19] T. Coughlin, "DNA Storage Update," <https://www.forbes.com/sites/tomcoughlin/2021/10/28/dna-storage-update/?sh=4fa56c5b2fb7>.
- [20] E. G. S. Antonio, T. Heinis, L. Carteron, M. Dimopoulou, and M. Antonini, "Nanopore Sequencing Simulator for DNA Data Storage," in *2021 International Conference on Visual Communications and Image Processing, VCIP 2021 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/VCIP53242.2021.9675388.

- [21] M. Zhang, K. Cai, K. A. S. Immink, and P. Chen, "Soft-Decision Decoding for DNA-Based Data Storage," in *Proceedings of 2018 International Symposium on Information Theory and Its Applications, ISITA 2018*, Institute of Electrical and Electronics Engineers Inc., Mar. 2019, pp. 16–20. doi: 10.23919/ISITA.2018.8664305.
- [22] L. Yuan, Z. Xie, Y. Wang, and X. Wang, "DeSP: a systematic DNA storage error simulation pipeline," *BMC Bioinformatics*, vol. 23, no. 1, Dec. 2022, doi: 10.1186/s12859-022-04723-w.
- [23] *MATLAB - A Fundamental Tool for Scientific Computing and Engineering Applications - Volume 2*. InTech, 2012. doi: 10.5772/3338.
- [24] B. Hamoum, E. Dupraz, L. Conde-Canencia, and D. Lavenier, "Channel Model with Memory for DNA Data Storage with Nanopore Sequencing," in *2021 11th International Symposium on Topics in Coding, ISTC 2021*, Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/ISTC49272.2021.9594243.
- [25] S. Achari, D. G. Holmes, and L. Cheng, "Symbol-Level Synchronization Channel Modeling with Real-World Application: From Davey-MacKay, Fritchman to Markov," *IEEE Access*, vol. 9, pp. 79752–79765, 2021, doi: 10.1109/ACCESS.2021.3084115.
- [26] E. N. Gilbert, "Capacity of a Burst-Noise Channel," *Bell System Technical Journal*, vol. 39, no. 5, pp. 1253–1265, Sep. 1960, doi: 10.1002/j.1538-7305.1960.tb03959.x.
- [27] A. Laourine and L. Tong, "Betting on Gilbert-Elliott channels," *IEEE Trans Wirel Commun*, vol. 9, no. 2, pp. 723–733, Feb. 2010, doi: 10.1109/TWC.2010.02.090055.
- [28] R. Varshney, C. Gangal, M. Sharique, and M. S. Ansari, "Deep Learning based Wireless Channel Prediction: 5G Scenario," *Procedia Comput Sci*, vol. 218, pp. 2626–2635, 2023, doi: 10.1016/j.procs.2023.01.236.
- [29] G. Peng, Z. Zhu, and Y. Liang, "Application of Deep Learning in Wireless Networks for Channel Estimation: A Survey," in *Journal of Physics: Conference Series*, IOP Publishing Ltd, Feb. 2022. doi: 10.1088/1742-6596/2203/1/012044.
- [30] Y. Li, R. Han, C. Bi, M. Li, S. Wang, and X. Gao, "DeepSimulator: A deep simulator for Nanopore sequencing," *Bioinformatics*, vol. 34, no. 17, pp. 2899–2908, Sep. 2018, doi: 10.1093/bioinformatics/bty223.
- [31] D. Buterez, "Scaling up DNA digital data storage by efficiently predicting DNA hybridisation using deep learning," *Sci Rep*, vol. 11, no. 1, Dec. 2021, doi: 10.1038/s41598-021-97238-y.

- [32] S. R. Srinivasavaradhan, S. Gopi, H. D. Pfister, and S. Yekhanin, "Trellis BMA: Coded Trace Reconstruction on IDS Channels for DNA Storage," in *IEEE International Symposium on Information Theory - Proceedings*, Institute of Electrical and Electronics Engineers Inc., Jul. 2021, pp. 2453–2458. doi: 10.1109/ISIT45174.2021.9517821.
- [33] Y. Erlich and D. Zielinski, "DNA Fountain enables a robust and efficient storage architecture," *Science* (1979), pp. 950–954, Mar. 2017, doi: 10.1126/science.aaj2038.
- [34] J. Foox, S. W. Tighe, C. M. Nicolet, J. M. Zook, M. Byrska-Bishop, W. E. Clarke, M. M. Khayat, M. Mahmoud, P. K. Laaguiby, Z. T. Herbert, D. Warner, G. S. Grills, J. Jen, S. Levy, J. Xiang, A. Alonso, X. Zhao, W. Zhang, F. Teng, Y. Zhao, H. Lu, G. P. Schroth, G. Narzisi, W. Farmerie, F. J. Sedlazeck, D. A. Baldwin, and C. E. Mason, "Performance assessment of DNA sequencing platforms in the ABRF Next-Generation Sequencing Study," *Nat Biotechnol*, vol. 39, no. 9, pp. 1129–1140, Sep. 2021, doi: 10.1038/s41587-021-01049-5.
- [35] "Oxford Nanopore Technologies Benchmark Datasets," <https://aws.amazon.com/marketplace/pp/prodview-ihy6a6dqhs3ii#resources>.
- [36] D. Walker and A. Helberg, "Modelling Error Runs in DNA Data Storage Using a Markov Model," in *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2023*, M. Smuts and A. Taute, Eds., Champagne Sports Resort, Central Drakensberg, KwaZulu-Natal, South Africa, Sep. 2023, pp. 113–118.
- [37] S. A. Bates, "Deoxyribonucleic acid (DNA)," <https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid>. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid>
- [38] R. H. Garret and C. M. Grisham, *Garret & Grisham's Biochemistry*, 2nd ed. Orlando: Saunders College Publishing, 1999.
- [39] "Phosphate backbone," Nature Education. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.nature.com/scitable/definition/phosphate-backbone-273/>
- [40] Twist Bioscience, "Oligo Pools," <https://www.twistbioscience.com/products/oligopools>.
- [41] "In vitro methods," European Chemicals Agency. Accessed: Nov. 06, 2023. [Online]. Available: <https://echa.europa.eu/support/registration/how-to-avoid-unnecessary-testing-on-animals/in-vitro-methods>
- [42] Oxford Nanopore Technologies, "Continuous development and improvement," <https://nanoporetech.com/about-us/continuous-development-and-improvement>.

- [43] “International Magnetic Tape Storage Roadmap,” 2019. Accessed: Jun. 24, 2023. [Online]. Available: <https://insic.org/wp-content/uploads/2019/07/INSIC-Technology-Roadmap-2019.pdf>
- [44] Oracle Corporation, “Oracle StorageTek LTO Tape Drives Datasheet,” 2022. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.oracle.com/a/ocom/docs/storage/tape-storage/lto-tape-drives-ds-033631.pdf>
- [45] A. El-Shaikh, M. Welzel, D. Heider, and B. Seeger, “High-scale random access on DNA storage systems,” *NAR Genom Bioinform*, vol. 4, no. 1, Mar. 2022, doi: 10.1093/nargab/lqab126.
- [46] K. A. Wetterstrand, “The Cost of Sequencing a Human Genome,” <https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost>. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost>
- [47] S. Mueller, J. R. Coleman, and E. Wimmer, “Putting Synthesis into Biology: A Viral View of Genetic Engineering through De Novo Gene and Genome Synthesis,” *Chemistry and Biology*, vol. 16, no. 3, pp. 337–347, Mar. 27, 2009. doi: 10.1016/j.chembiol.2009.03.002.
- [48] Twist Bioscience, “Genes,” <https://www.twistbioscience.com/products/genes>. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.twistbioscience.com/products/genes>
- [49] J. Bonnet, M. Colotte, D. Coudy, V. Couallier, J. Portier, B. Morin, and S. Tuffet, “Chain and conformation stability of solid-state DNA: Implications for room temperature storage,” *Nucleic Acids Res*, vol. 38, no. 5, pp. 1531–1546, Dec. 2009, doi: 10.1093/nar/gkp1060.
- [50] T. J. Anchordoquy and M. C. Molina, “Preservation of DNA,” *Cell Preserv Technol*, vol. 5, no. 4, pp. 180–188, Dec. 2007, doi: 10.1089/cpt.2007.0511.
- [51] R. N. Grass and W. Stark, “DNA-based data storage,” European Patent Office. Accessed: Nov. 18, 2023. [Online]. Available: <https://www.epo.org/en/news-events/european-inventor-award/meet-the-finalists/robert-n-grass-and-wendelin-stark#>
- [52] J. Bonnet, M. Colotte, D. Coudy, V. Couallier, J. Portier, B. Morin, and S. Tuffet, “Chain and conformation stability of solid-state DNA: implications for room temperature storage,” *Nucleic Acids Res*, vol. 38, no. 5, pp. 1531–1546, Mar. 2010, doi: 10.1093/nar/gkp1060.
- [53] “Seed Vault Data Portal,” <http://www.nordgen.org/sgsv>. Accessed: Jun. 24, 2023. [Online]. Available: <http://www.nordgen.org/sgsv>
- [54] J. W. C. Van Bogart, “Magnetic Tape Storage and Handling Guide Magnetic Tape Storage and Handling A Guide for Libraries and Archives.” [Online]. Available: <http://www.nlm.org>

- [55] J. Davis, "Microvenus," *Art journal*, vol. 55, no. 1, pp. 70–74, 1996.
- [56] M. Hao, H. Qiao, Y. Gao, Z. Wang, X. Qiao, X. Chen, and H. Qi, "A mixed culture of bacterial cells enables an economic DNA storage on a large scale," *Commun Biol*, vol. 3, no. 1, Dec. 2020, doi: 10.1038/s42003-020-01141-7.
- [57] H. Lee, E. Popodi, H. Tang, and P. L. Foster, "Rate and molecular spectrum of spontaneous mutations in the bacterium *Escherichia coli* as determined by whole-genome sequencing," *Proceedings of the National Academy of Sciences*, vol. 109, no. 41, Oct. 2012, doi: 10.1073/pnas.1210309109.
- [58] Y. Dong, F. Sun, Z. Ping, Q. Ouyang, and L. Qian, "DNA storage: research landscape and future prospects.," *Natl Sci Rev*, vol. 7, no. 6, pp. 1092–1107, Jun. 2020, doi: 10.1093/nsr/nwaa007.
- [59] L. Haye, "DNA data storage," Paris, Dec. 2021. Accessed: Jun. 24, 2023. [Online]. Available: https://www2.assemblee-nationale.fr/content/download/481482/4687200/version/1/file/OPECST_2021_0063_Note_stockage_ADN_en.pdf
- [60] M. Smith, "Polymerase chain reaction (PCR) ," National Human Genome Research Insitute. Accessed: Jun. 24, 2023. [Online]. Available: <https://www.genome.gov/genetics-glossary/Polymerase-Chain-Reaction>
- [61] M. E. Allentoft, M. Collins, D. Harker, J. Haile, C. L. Oskam, M. L. Hale, P. F. Campos, J. A. Samaniego, M. T. P. Gilbert, E. Willerslev, G. Zhang, R. P. Scofield, R. N. Holdaway, and M. Bunce, "The half-life of DNA in bone: measuring decay kinetics in 158 dated fossils.," *Proc Biol Sci*, vol. 279, no. 1748, pp. 4724–33, Dec. 2012, doi: 10.1098/rspb.2012.1745.
- [62] L. Orlando, A. Ginolhac, G. Zhang, D. Froese, A. Albrechtsen, M. Stiller, M. Schubert, E. Cappellini, B. Petersen, I. Moltke, P. L. F. Johnson, M. Fumagalli, J. T. Vilstrup, M. Raghavan, T. Korneliussen, A.-S. Malaspinas, J. Vogt, D. Szklarczyk, C. D. Kelstrup, J. Vinther, A. Dolocan, J. Stenderup, A. M. V. Velazquez, J. Cahill, M. Rasmussen, X. Wang, J. Min, G. D. Zazula, A. Seguin-Orlando, C. Mortensen, K. Magnussen, J. F. Thompson, J. Weinstock, K. Gregersen, K. H. Røed, V. Eisenmann, C. J. Rubin, D. C. Miller, D. F. Antczak, M. F. Bertelsen, S. Brunak, K. A. S. Al-Rasheid, O. Ryder, L. Andersson, J. Mundy, A. Krogh, M. T. P. Gilbert, K. Kjær, T. Sicheritz-Ponten, L. J. Jensen, J. V. Olsen, M. Hofreiter, R. Nielsen, B. Shapiro, J. Wang, and E. Willerslev, "Recalibrating Equus evolution using the genome sequence of an early Middle Pleistocene horse," *Nature*, vol. 499, no. 7456, pp. 74–78, Jul. 2013, doi: 10.1038/nature12323.

- [63] A. S. J. Helberg, W. A. Clarke, H. C. Ferreira, and A. J. H. Vinck, "A class of DC free, synchronization error correcting codes," *IEEE Trans Magn*, vol. 29, no. 6, pp. 4048–4049, Nov. 1993, doi: 10.1109/20.281386.
- [64] A. M. Michelson and A. R. Todd, "Nucleotides part XXXII. Synthesis of a dithymidine dinucleotide containing a 3': 5'-internucleotidic linkage," *J. Chem. Soc.*, vol. 0, no. 0, pp. 2632–2638, 1955, doi: 10.1039/JR9550002632.
- [65] R. H. Hall, A. Todd, and R. F. Webb, "644. Nucleotides. Part XLI. Mixed anhydrides as intermediates in the synthesis of dinucleoside phosphates," *Journal of the Chemical Society (Resumed)*, p. 3291, 1957, doi: 10.1039/jr9570003291.
- [66] H. G. Khorana, W. E. Razzell, P. T. Gilham, G. M. Tener, and E. H. Pol, "Syntheses OF Dideoxyribonucleotides," *J Am Chem Soc*, vol. 79, no. 4, pp. 1002–1003, Feb. 1957, doi: 10.1021/ja01561a065.
- [67] S. L. Beaucage and M. H. Caruthers, "Deoxynucleoside phosphoramidites—A new class of key intermediates for deoxypolynucleotide synthesis," *Tetrahedron Lett*, vol. 22, no. 20, pp. 1859–1862, Jan. 1981, doi: 10.1016/S0040-4039(01)90461-7.
- [68] "Technical Resources for Oligonucleotides," Thermofisher Scientific. Accessed: Nov. 18, 2023. [Online]. Available: <https://www.thermofisher.com/za/en/home/life-science/oligonucleotides-primers-probes-genes/custom-dna-oligos/oligo-technical-resources.html>
- [69] S. Singh-Gasson, R. D. Green, Y. Yue, C. Nelson, F. Blattner, M. R. Sussman, and F. Cerrina, "Maskless fabrication of light-directed oligonucleotide microarrays using a digital micromirror array," *Nat Biotechnol*, vol. 17, no. 10, pp. 974–978, Oct. 1999, doi: 10.1038/13664.
- [70] X. Gao, "A flexible light-directed DNA chip synthesis gated by deprotection using solution photogenerated acids," *Nucleic Acids Res*, vol. 29, no. 22, pp. 4744–4750, Nov. 2001, doi: 10.1093/nar/29.22.4744.
- [71] T. R. Hughes, M. Mao, A. R. Jones, J. Burchard, M. J. Marton, K. W. Shannon, S. M. Lefkowitz, M. Ziman, J. M. Schelter, M. R. Meyer, S. Kobayashi, C. Davis, H. Dai, Y. D. He, S. B. Stephanians, G. Cavet, W. L. Walker, A. West, E. Coffey, D. D. Shoemaker, R. Stoughton, A. P. Blanchard, S. H. Friend, and P. S. Linsley, "Expression profiling using microarrays fabricated by an ink-jet oligonucleotide synthesizer," *Nat Biotechnol*, vol. 19, no. 4, pp. 342–347, Apr. 2001, doi: 10.1038/86730.

- [72] A. L. Ghindilis, M. W. Smith, K. R. Schwarzkopf, K. M. Roth, K. Peyvan, S. B. Munro, M. J. Lodes, A. G. Stöver, K. Bernards, K. Dill, and A. McShea, "CombiMatrix oligonucleotide arrays: Genotyping and gene expression assays employing electrochemical detection," *Biosens Bioelectron*, vol. 22, no. 9–10, pp. 1853–1860, Apr. 2007, doi: 10.1016/j.bios.2006.06.024.
- [73] L.-C. Au, F.-Y. Yang, W.-J. Yang, S.-H. Lo, and C.-F. Kao, "Gene Synthesis by a LCR-Based Approach: High-Level Production of Leptin-L54 Using Synthetic Gene in *Escherichia coli*," *Biochem Biophys Res Commun*, vol. 248, no. 1, pp. 200–203, Jul. 1998, doi: 10.1006/bbrc.1998.8929.
- [74] Thermo Fisher Scientific Inc., "PCR Basics," Thermo Fisher. Accessed: Jun. 25, 2023. [Online]. Available: <https://www.thermofisher.com/za/en/home/life-science/cloning/cloning-learning-center/invitrogen-school-of-molecular-biology/pcr-education/pcr-reagents-enzymes/pcr-basics.html>
- [75] Thermo Fisher Scientific Inc., "DNA Polymerase—Four Key Characteristics for PCR," Thermo Fisher. Accessed: Jun. 25, 2023. [Online]. Available: <https://www.thermofisher.com/za/en/home/life-science/cloning/cloning-learning-center/invitrogen-school-of-molecular-biology/pcr-education/pcr-reagents-enzymes/dna-polymerase-characteristics.html>
- [76] J. Cline, "PCR fidelity of pfu DNA polymerase and other thermostable DNA polymerases," *Nucleic Acids Res*, vol. 24, no. 18, pp. 3546–3551, Sep. 1996, doi: 10.1093/nar/24.18.3546.
- [77] F. Sanger, S. Nicklen, and A. R. Coulson, "DNA sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, Dec. 1977, doi: 10.1073/pnas.74.12.5463.
- [78] A. Gomes and B. Korf, "Genetic Testing Techniques," in *Pediatric Cancer Genetics*, Elsevier, 2018, pp. 47–64. doi: 10.1016/B978-0-323-48555-5.00005-3.
- [79] "Key differences between next-generation sequencing and Sanger sequencing," Illumina, Inc. Accessed: Jul. 13, 2023. [Online]. Available: <https://emea.illumina.com/science/technology/next-generation-sequencing/ngs-vs-sanger-sequencing.html>
- [80] B. E. Slatko, A. F. Gardner, and F. M. Ausubel, "Overview of Next-Generation Sequencing Technologies.," *Curr Protoc Mol Biol*, vol. 122, no. 1, p. e59, Apr. 2018, doi: 10.1002/cpmb.59.

- [81] N. Stoler and A. Nekrutenko, "Sequencing error profiles of Illumina sequencing instruments," *NAR Genom Bioinform*, vol. 3, no. 1, Jan. 2021, doi: 10.1093/nargab/lqab019.
- [82] E. Check Hayden, "Is the \$1,000 genome for real?," *Nature*, Jan. 2014, doi: 10.1038/nature.2014.14530.
- [83] "Access the latest technology today with flexible financing solutions." Accessed: Nov. 22, 2023. [Online]. Available: <https://sapac.illumina.com/company/doing-business-with-illumina/illumina-financial-solutions.html>
- [84] Y. Fukasawa, L. Ermini, H. Wang, K. Carty, and M.-S. Cheung, "LongQC: A Quality Control Tool for Third Generation Sequencing Long Read Data," *G3 Genes[Genomes]Genetics*, vol. 10, no. 4, pp. 1193–1196, Apr. 2020, doi: 10.1534/g3.119.400864.
- [85] M. Jain, H. E. Olsen, B. Paten, and M. Akeson, "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community," *Genome Biol*, vol. 17, no. 1, p. 239, Dec. 2016, doi: 10.1186/s13059-016-1103-0.
- [86] "Flow cells," Oxford Nanopore Technologies. Accessed: Jul. 05, 2023. [Online]. Available: <https://nanoporetech.com/how-it-works/flow-cells-and-nanopores>
- [87] "Types of nanopores," Oxford Nanopore Technologies. Accessed: Jul. 05, 2023. [Online]. Available: <https://nanoporetech.com/how-it-works/types-of-nanopores>
- [88] Y. He, M. Tsutsui, Y. Zhou, and X.-S. Miao, "Solid-state nanopore systems: from materials to applications," *NPG Asia Mater*, vol. 13, no. 1, p. 48, Dec. 2021, doi: 10.1038/s41427-021-00313-z.
- [89] L. Xue, H. Yamazaki, R. Ren, M. Wanunu, A. P. Ivanov, and J. B. Edel, "Solid-state nanopore sensors," *Nat Rev Mater*, vol. 5, no. 12, pp. 931–951, Sep. 2020, doi: 10.1038/s41578-020-0229-6.
- [90] Y. Wang, Y. Zhao, A. Bollas, Y. Wang, and K. F. Au, "Nanopore sequencing technology, bioinformatics and applications," *Nat Biotechnol*, vol. 39, no. 11, pp. 1348–1365, Nov. 2021, doi: 10.1038/s41587-021-01108-x.
- [91] "MinION," Oxford Nanopore Technologies. Accessed: Jul. 06, 2023. [Online]. Available: <https://nanoporetech.com/products/minion>
- [92] "MinION Mk1C," Oxford Nanopore Technologies. Accessed: Jul. 06, 2023. [Online]. Available: <https://store.nanoporetech.com/minion-mk1c.html>

- [93] T. Zhang, H. Li, S. Ma, J. Cao, H. Liao, Q. Huang, and W. Chen, "The newest Oxford Nanopore R10.4.1 full-length 16S rRNA sequencing enables the accurate resolution of species-level microbial community profiling," *Appl Environ Microbiol*, vol. 89, no. 10, Oct. 2023, doi: 10.1128/aem.00605-23.
- [94] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants.," *Nucleic Acids Res*, vol. 38, no. 6, pp. 1767–71, Apr. 2010, doi: 10.1093/nar/gkp1137.
- [95] "How basecalling works." Accessed: Nov. 23, 2023. [Online]. Available: <https://nanoporetech.com/how-it-works/basecalling>
- [96] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997, doi: 10.1109/78.650093.
- [97] "Bonito." Accessed: Nov. 23, 2023. [Online]. Available: <https://github.com/nanoporetech/bonito>
- [98] R. R. Wick, L. M. Judd, and K. E. Holt, "Performance of neural network basecalling tools for Oxford Nanopore sequencing," *Genome Biol*, vol. 20, no. 1, p. 129, Dec. 2019, doi: 10.1186/s13059-019-1727-y.
- [99] C. Rashtchian, K. Makarychev, M. Racz, S. Ang, D. Jevdjic, S. Yekhanin, L. Ceze, and K. Strauss, "Clustering Billions of Reads for DNA Data Storage," in *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, New York: Curran Associates Inc., Dec. 2017, pp. 3362–3373.
- [100] T. Rausch, S. Koren, G. Denisov, D. Weese, A.-K. Emde, A. Döring, and K. Reinert, "A consistency-based consensus algorithm for *de novo* and reference-guided sequence assembly of short reads," *Bioinformatics*, vol. 25, no. 9, pp. 1118–1124, May 2009, doi: 10.1093/bioinformatics/btp131.
- [101] A. Lenz, I. Maarouf, L. Welter, A. Wachter-Zeh, E. Rosnes, and A. Graell i Amat, "Concatenated Codes for Recovery From Multiple Reads of DNA Sequences," in *2020 IEEE Information Theory Workshop (ITW)*, IEEE, Apr. 2021, pp. 1–5. doi: 10.1109/ITW46852.2021.9457675.
- [102] O. Sabary, A. Yucovich, G. Shapira, and E. Yaakobi, "Reconstruction Algorithms for DNA-Storage Systems," *Sci. Rep.*, vol. 14, no. 1951, Sep. 2024, doi: <https://doi.org/10.1038/s41598-024-51730-3>.

- [103] T. Batu, S. Kannan, and A. McGregor, "Reconstructing strings from random traces," *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms*, vol. 15, Jan. 2004, doi: 10.1145/982792.982929.
- [104] P.-J. Shih, C.-H. Lee, P.-C. Yeh, and K.-C. Chen, "Channel Codes for Reliability Enhancement in Molecular Communication," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 12, pp. 857–867, Dec. 2013, doi: 10.1109/JSAC.2013.SUP2.12130018.
- [105] B. Hamoum and E. Dupraz, "Channel Model and Decoder With Memory for DNA Data Storage With Nanopore Sequencing," *IEEE Access*, vol. 11, pp. 52075–52087, 2023, doi: 10.1109/ACCESS.2023.3278975.
- [106] N. Niklas, J. Pröll, M. Danzer, S. Stabentheiner, K. Hofer, and C. Gabriel, "Routine performance and errors of 454 HLA exon sequencing in diagnostics," *BMC Bioinformatics*, vol. 14, no. 1, p. 176, Dec. 2013, doi: 10.1186/1471-2105-14-176.
- [107] C. Delahaye and J. Nicolas, "Sequencing DNA with nanopores: Troubles and biases," *PLoS One*, vol. 16, no. 10, p. e0257521, Oct. 2021, doi: 10.1371/journal.pone.0257521.
- [108] M. Keoliya, P. Sharma, and D. Jevdjic, "Simulating Noisy Channels in DNA Storage," in *Proceedings - 2022 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 150–152. doi: 10.1109/ISPASS55109.2022.00019.
- [109] L. Organick, S. D. Ang, Y.-J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M. Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, C. N. Takahashi, S. Newman, H.-Y. Parker, C. Rashtchian, K. Stewart, G. Gupta, R. Carlson, J. Mulligan, D. Carmean, G. Seelig, L. Ceze, and K. Strauss, "Random access in large-scale DNA data storage," *Nat Biotechnol*, vol. 36, no. 3, pp. 242–248, Mar. 2018, doi: 10.1038/nbt.4079.
- [110] S. O. Oyola, T. D. Otto, Y. Gu, G. Maslen, M. Manske, S. Campino, D. J. Turner, B. MacInnis, D. P. Kwiatkowski, H. P. Swerdlow, and M. A. Quail, "Optimizing illumina next-generation sequencing library preparation for extremely at-biased genomes," *BMC Genomics*, vol. 13, no. 1, p. 1, 2012, doi: 10.1186/1471-2164-13-1.
- [111] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer, "Substantial biases in ultra-short read data sets from high-throughput DNA sequencing," *Nucleic Acids Res*, vol. 36, no. 16, Sep. 2008, doi: 10.1093/nar/gkn425.

- [112] M. Blaum, S. Litsyn, V. Buskens, and H. C. A. van Tilborg, "Error-correcting codes with bounded running digital sum," *IEEE Trans Inf Theory*, vol. 39, no. 1, pp. 216–227, 1993, doi: 10.1109/18.179362.
- [113] W. Turin, *Performance Analysis and Modeling of Digital Transmission Systems*. Boston, MA: Springer US, 2004. doi: 10.1007/978-1-4419-9070-9.
- [114] A. L. Gimpel, W. J. Stark, R. Heckel, and R. N. Grass, "A digital twin for DNA data storage based on comprehensive quantification of errors and biases," *Nat Commun*, vol. 14, no. 1, p. 6026, Sep. 2023, doi: 10.1038/s41467-023-41729-1.
- [115] L. Cao, "Beyond i.i.d.: Non-IID Thinking, Informatics, and Learning," *IEEE Intell Syst*, vol. 37, no. 4, pp. 5–17, Jul. 2022, doi: 10.1109/MIS.2022.3194618.
- [116] E. G. S. Antonio, T. Heinis, L. Carteron, M. Dimopoulou, and M. Antonini, "Nanopore Sequencing Simulator for DNA Data Storage," in *2021 International Conference on Visual Communications and Image Processing (VCIP)*, IEEE, Dec. 2021, pp. 1–5. doi: 10.1109/VCIP53242.2021.9675388.
- [117] E. A. G. Baker, S. Goodwin, W. R. McCombie, and O. M. Ramos, "SiLiCO: A Simulator of Long Read Sequencing in PacBio and Oxford Nanopore," *BioRxiv*, Sep. 2016, doi: 10.1101/076901.
- [118] L. Conde-Canencia and L. Dolecek, "Nanopore DNA Sequencing Channel Modeling," in *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*, IEEE, Oct. 2018, pp. 258–262. doi: 10.1109/SiPS.2018.8598361.
- [119] W. H. Tranter, *Principles of communication systems simulation with wireless applications*. Prentice Hall, 2004.
- [120] W. Song, K. Cai, M. Zhang, and C. Yuen, "Codes With Run-Length and GC-Content Constraints for DNA-Based Data Storage," *IEEE Communications Letters*, vol. 22, no. 10, pp. 2004–2007, Oct. 2018, doi: 10.1109/LCOMM.2018.2866566.
- [121] B. Fritchman, "A binary channel characterization using partitioned Markov chains," *IEEE Trans Inf Theory*, vol. 13, no. 2, pp. 221–227, Apr. 1967, doi: 10.1109/TIT.1967.1053975.
- [122] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, Feb. 1970, doi: 10.1214/aoms/1177697196.
- [123] J. Lember and A. Koloydenko, "The adjusted Viterbi training for hidden Markov models," *Bernoulli*, vol. 14, no. 1, Feb. 2008, doi: 10.3150/07-BEJ105.

- [124] The MathWorks Inc., “Statistics and Machine Learning Toolbox.” Accessed: Nov. 03, 2023. [Online]. Available: <https://www.mathworks.com/help/stats/>
- [125] L. E. Baum and J. A. Eagon, “An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology,” *Bulletin of the American Mathematical Society*, vol. 73, no. 3, pp. 360–363, 1967, doi: 10.1090/S0002-9904-1967-11751-8.
- [126] L. E. Baum and G. Sell, “Growth transformations for functions on manifolds,” *Pac J Math*, vol. 27, no. 2, pp. 211–227, Nov. 1968, doi: 10.2140/pjm.1968.27.211.
- [127] L. R. Rabiner, “A tutorial on hidden Markov models and selected applications in speech recognition,” *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989, doi: 10.1109/5.18626.
- [128] “LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S,” *Philos Trans R Soc Lond*, vol. 53, pp. 370–418, Dec. 1763, doi: 10.1098/rstl.1763.0053.
- [129] R. J. Rossi, *Mathematical Statistics: An Introduction to Likelihood Based Inference*. Hoboken: Wiley, 2018.
- [130] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Trans Inf Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967, doi: 10.1109/TIT.1967.1054010.
- [131] G. D. Forney, “The viterbi algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973, doi: 10.1109/PROC.1973.9030.
- [132] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, “Markov chains and hidden Markov models,” in *Biological sequence analysis: probabilistic models of proteins and nucleic acids*, Cambridge University Press, 1998, pp. 46–79.
- [133] B.-H. Juang and L. R. Rabiner, “The segmental K-means algorithm for estimating parameters of hidden Markov models,” *IEEE Trans Acoust*, vol. 38, no. 9, pp. 1639–1641, 1990, doi: 10.1109/29.60082.
- [134] T. Y. Lam and I. M. Meyer, “Efficient algorithms for training the parameters of hidden Markov models using stochastic expectation maximization (EM) training and Viterbi training,” *Algorithms for Molecular Biology*, vol. 5, no. 1, p. 38, Dec. 2010, doi: 10.1186/1748-7188-5-38.
- [135] L. J. Rodríguez and I. Torres, “Comparative Study of the Baum-Welch and Viterbi Training Algorithms Applied to Read and Spontaneous Speech Recognition,” in *Pattern Recognition*

- and Image Analysis, *First Iberian Conference*, Mallorca, 2003, pp. 847–857. doi: 10.1007/978-3-540-44871-6_98.
- [136] R. An, Y. Jia, B. Wan, Y. Zhang, P. Dong, J. Li, and X. Liang, “Non-Enzymatic Depurination of Nucleic Acids: Factors and Mechanisms,” *PLoS One*, vol. 9, no. 12, p. e115950, Dec. 2014, doi: 10.1371/journal.pone.0115950.
 - [137] Y. Li, S. Wang, C. Bi, Z. Qiu, M. Li, and X. Gao, “DeepSimulator1.5: a more powerful, quicker and lighter simulator for Nanopore sequencing,” *Bioinformatics*, vol. 36, no. 8, pp. 2578–2580, Apr. 2020, doi: 10.1093/bioinformatics/btz963.
 - [138] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional LSTM and other neural network architectures,” *Neural Networks*, vol. 18, no. 5–6, pp. 602–610, Jul. 2005, doi: 10.1016/j.neunet.2005.06.042.
 - [139] G. Trigeorgis, M. A. Nicolaou, S. Zafeiriou, and B. W. Schuller, “Deep Canonical Time Warping,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016, pp. 5110–5118. doi: 10.1109/CVPR.2016.552.
 - [140] R. Wick, “Badread: simulation of error-prone long reads,” *J Open Source Softw*, vol. 4, no. 36, p. 1316, Apr. 2019, doi: 10.21105/joss.01316.
 - [141] S. Ji, Z. Zhang, S. Ying, L. Wang, X. Zhao, and Y. Gao, “Kullback–Leibler Divergence Metric Learning,” *IEEE Trans Cybern*, vol. 52, no. 4, pp. 2047–2058, Apr. 2022, doi: 10.1109/TCYB.2020.3008248.
 - [142] Oxford Nanopore Technologies plc., “Oxford Nanopore Open Data project.” Accessed: Nov. 04, 2023. [Online]. Available: <https://labs.epi2me.io/dataindex/>
 - [143] R. W. Hamming, “Error Detecting and Error Correcting Codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.
 - [144] C. D. Pilcher, J. K. Wong, and S. K. Pillai, “Inferring HIV Transmission Dynamics from Phylogenetic Sequence Relationships,” *PLoS Med*, vol. 5, no. 3, Mar. 2008, doi: 10.1371/journal.pmed.0050069.
 - [145] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, Feb. 1966.
 - [146] J. Daniels, D. Nye, and G. Hu, “Cluster Analysis for Commonalities Between Words of Different Languages,” in *Applied Computing & Information Technology*, R. Lee, Ed., Okayama: Springer Cham, 2016, pp. 71–84.

- [147] B. Berger, M. S. Waterman, and Y. W. Yu, "Levenshtein Distance, Sequence Comparison and Biological Database Search," *IEEE Trans Inf Theory*, vol. 67, no. 6, pp. 3287–3294, Jun. 2021, doi: 10.1109/TIT.2020.2996543.
- [148] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974, doi: 10.1145/321796.321811.
- [149] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J Mol Biol*, vol. 215, no. 3, pp. 403–410, Oct. 1990, doi: 10.1016/S0022-2836(05)80360-2.
- [150] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Commun ACM*, vol. 7, no. 3, pp. 171–176, Mar. 1964, doi: 10.1145/363958.363994.
- [151] R. A. Wagner and R. Lowrance, "An Extension of the String-to-String Correction Problem," *Journal of the ACM*, vol. 22, no. 2, pp. 177–183, Apr. 1975, doi: 10.1145/321879.321880.
- [152] K. A. Majorek, S. Dunin-Horkawicz, K. Steczkiewicz, A. Muszewska, M. Nowotny, K. Ginalska, and J. M. Bujnicki, "The RNase H-like superfamily: new members, comparative structural analysis and evolutionary classification," *Nucleic Acids Res*, vol. 42, no. 7, pp. 4160–4179, Apr. 2014, doi: 10.1093/nar/gkt1414.
- [153] P. P. M. Liju, "Algorithm to derive shortest edit script using Levenshtein distance algorithm." Accessed: Nov. 23, 2023. [Online]. Available: <https://arxiv.org/abs/2208.08823>
- [154] E. W. Myers, "AnO(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1–4, pp. 251–266, Nov. 1986, doi: 10.1007/BF01840446.
- [155] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A Greedy Algorithm for Aligning DNA Sequences," *Journal of Computational Biology*, vol. 7, no. 1–2, pp. 203–214, Feb. 2000, doi: 10.1089/10665270050081478.
- [156] J. Devore and N. Farnum, *Applied statistics for engineers and scientists*, 2nd ed. Belmont: Thomson Brooks.Cole, 2005.
- [157] "Measures of Skewness and Kurtosis," NIST/SEMATECH e-Handbook of Statistical Methods. Accessed: Feb. 03, 2024. [Online]. Available: <https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm>
- [158] D. Zwillinger and S. Kokoska, "Measures of skewness," in *CRC Standard Probability and Statistics Tables and Formulae*, Florida: CRC Press LLC, 2000.
- [159] H. D. Kambezidis, "The Solar Resource," in *Comprehensive Renewable Energy*, Elsevier, 2012, pp. 27–84. doi: 10.1016/B978-0-08-087872-0.00302-4.

- [160] The MathWorks Inc., “MATLAB.” Accessed: Nov. 03, 2023. [Online]. Available: <https://www.mathworks.com/products/matlab.html>
- [161] G. Van Rossum, “Python tutorial.” CWI, Amsterdam, 1995.
- [162] “The Python Standard Library.” Accessed: Oct. 27, 2023. [Online]. Available: <https://docs.python.org/3/library/index.html>
- [163] P. E. Barret, J. Hunter, J. T. Miller, J.-C. Hsu, and P. Greenfield, “matplotlib - A Portable Python Plotting Package,” *Astronomical Data Analysis Software and Systems XIV*, vol. 347, Dec. 2025.
- [164] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, A. Vijaykumar, A. Pietro Bardelli, A. Rothberg, A. Hilboll, A. Kloeckner, A. Scopatz, A. Lee, A. Rokem, C. N. Woods, C. Fulton, C. Masson, C. Häggström, C. Fitzgerald, D. A. Nicholson, D. R. Hagen, D. V. Pasechnik, E. Olivetti, E. Martin, E. Wieser, F. Silva, F. Lenders, F. Wilhelm, G. Young, G. A. Price, G.-L. Ingold, G. E. Allen, G. R. Lee, H. Audren, I. Probst, J. P. Dietrich, J. Silterra, J. T. Webber, J. Slavič, J. Nothman, J. Buchner, J. Kulick, J. L. Schönberger, J. V. de Miranda Cardoso, J. Reimer, J. Harrington, J. L. C. Rodríguez, J. Nunez-Iglesias, J. Kuczynski, K. Tritz, M. Thoma, M. Newville, M. Kümmerer, M. Bolingbroke, M. Tartre, M. Pak, N. J. Smith, N. Nowaczyk, N. Shebanov, O. Pavlyk, P. A. Brodtkorb, P. Lee, R. T. McGibbon, R. Feldbauer, S. Lewis, S. Tygier, S. Sievert, S. Vigna, S. Peterson, S. More, T. Pudlik, T. Oshima, T. J. Pingel, T. P. Robitaille, T. Spura, T. R. Jones, T. Cera, T. Leslie, T. Zito, T. Krauss, U. Upadhyay, Y. O. Halchenko, and Y. Vázquez-Baeza, “SciPy 1.0: fundamental algorithms for scientific computing in Python,” *Nat Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, doi: 10.1038/s41592-019-0686-2.
- [165] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.
- [166] The MathWorks Inc., “MATLAB.” Accessed: Nov. 03, 2023. [Online]. Available: <https://www.mathworks.com/help/matlab/>

- [167] The MathWorks Inc., “Text Analytics Toolbox.” Accessed: Nov. 03, 2023. [Online]. Available: <https://www.mathworks.com/help/textanalytics/>
- [168] The Python Software Foundation, “Python 3.9.17 documentation.” Accessed: Nov. 03, 2023. [Online]. Available: <https://docs.python.org/3.9/>
- [169] J. Hunter, D. Dale, and M. Droettboom, “Matplotlib 3.8.1 documentation.” Accessed: Nov. 03, 2023. [Online]. Available: <https://matplotlib.org/stable/>
- [170] “Numpy and Scipy Documentation,” SciPy.org. Accessed: Feb. 03, 2024. [Online]. Available: <https://docs.scipy.org/doc/>
- [171] “NumPy v1.21 Manual.” Accessed: Nov. 03, 2023. [Online]. Available: <https://numpy.org/doc/1.21/>
- [172] T. E. Oliphant, *Guide to Numpy*, 2nd ed. CreateSpace Independent Publishing Platform, 2015.
- [173] M. Bachmann, “Welcome to Levenshtein’s documentation!” Accessed: Nov. 03, 2023. [Online]. Available: <https://maxbachmann.github.io/Levenshtein/>
- [174] M. Bachmann, “Levenshtein module.” Accessed: Nov. 03, 2023. [Online]. Available: <https://maxbachmann.github.io/Levenshtein/levenshtein.html#distance>
- [175] The Python Software Foundation, “statistics — Mathematical statistics functions.” Accessed: Nov. 03, 2023. [Online]. Available: <https://docs.python.org/3/library/statistics.html>
- [176] The MathWorks Inc., “editDistance.” Accessed: Nov. 03, 2023. [Online]. Available: <https://www.mathworks.com/help/textanalytics/ref/editdistance.html>
- [177] The Python Software Foundation, “random — Generate pseudo-random numbers.” Accessed: Nov. 03, 2023. [Online]. Available: <https://docs.python.org/3/library/random.html>
- [178] “Julia 1.9 Documentation.” Accessed: Nov. 03, 2023. [Online]. Available: <https://docs.julialang.org/en/v1/>
- [179] The Python Software Foundation, “itertools — Functions creating iterators for efficient looping.” Accessed: Nov. 03, 2023. [Online]. Available: <https://docs.python.org/3/library/itertools.html>
- [180] “scipy.stats.pearsonr,” SciPy v1.12.0 Manual. Accessed: Feb. 03, 2024. [Online]. Available: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html>

- [181] “scipy.stats.skew,” SciPy v1.12.0 Manual. Accessed: Feb. 03, 2024. [Online]. Available: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html>
- [182] “math — Mathematical functions,” Python Software Foundation. Accessed: Feb. 03, 2024. [Online]. Available: <https://docs.python.org/3/library/math.html>
- [183] S. J. D. Prince, *Understanding Deep Learning*. The MIT Press, 2023.

APPENDIX A LEVENSHTein DISTANCE ERROR IDENTIFICATION ALGORITHM

The edit script algorithm based on the Levenshtein Distance from [153] is described below.

A and B represent the two strings to be compared. A and B have lengths m and n respectively. The strings contain characters from the alphabet S . An edit matrix D of size $(m + 1) \times (n + 1)$ is used. Each index of D is a dictionary that contains two key value pairs. The keys refer to the edit distance and edit script values, and are therefore named *edit_distance* and *edit_script*. The pseudocode follows.

Levensthein Distance Error Identification Algorithm

function editScript (A, B):

declare D of size $m + 1 \times n + 1$

set each value of D to {edit_distance : 0, edit_script: []}

for $1 \leq j \leq m$:

$D[0, m] = \{ \text{edit_distance} : j, \text{edit_script}: [\text{Delete } A[k] \text{ at position } j \text{ for } 1 \leq k \leq j] \}$

For $1 \leq i \leq n$:

$D[i, 0] = \{ \text{edit_distance} : i, \text{edit_script}: [\text{Insert } A[k] \text{ at position } i \text{ for } 1 \leq k \leq i] \}$

for $0 \leq i \leq n-1$:

for $0 \leq j \leq m-1$:

if $A[j] == B[i]$:

$D[i + 1, j + 1] = D[i, j]$

else:

distance = minimum of

$D[i, j + 1] + 1$ // represents deletion

$D[i + 1, j] + 1$ // represents insertion

$D[i, j] + 1$ // represents substitution

```

        if D[i, j + 1]{edit_distance} == D[i + 1, j ] {edit_distance}:    //
substitution
        D[i + 1, j + 1] = {edit_distance: distance, edit_script:
            D[i][j]{edit_script} + [Substitute A[j] with B[i] at position i]}

    else if D[i, j + 1]{edit_distance} < D[i + 1, j ] {edit_distance}: // insertion
        D[i + 1, j + 1] = {edit_distance: distance, edit_script:
            D[i][j]{edit_script} + [Insert B[i] at position i]}

    else if D[i, j + 1]{edit_distance} < D[i + 1, j ] {edit_distance}: // deletion
        D[i + 1, j + 1] = {edit_distance: distance, edit_script:
            D[i][j]{edit_script} + [Insert A[j] at position j]}

return D[m, n]{edit_script}

```

APPENDIX B ALGORITHM FOR POPULATING STRINGS OF OUTCOMES WITH BASES

The algorithm to populate a string of outcomes with bases requires two input parameters, namely the string of outcomes O and an input string of bases A . Additionally, two auxiliary functions are used to select a base to insert or substitute according to the probability distributions for those errors of the dataset. The function to substitute a base requires the base B in the original string as input.

The probability distribution to insert a base is given by a dictionary in the form:

$$Pr_{ins} = \{A: Pr_A, C: Pr_C, G: Pr_G, T: Pr_T\}$$

Pr_A , Pr_C , Pr_G , and Pr_T represent the probability to insert an A, C, G or T respectively, as calculated from the training set.

The probability distribution to substitute a base is given by a nested dictionary in the form:

$$Pr_{sub} = \{A: Pr_S_A, C: Pr_S_C, G: Pr_S_G, T: Pr_S_T\}$$

Pr_S_A , Pr_S_C , Pr_S_G , and Pr_S_T are dictionaries that represent the probability to replace the given base with each other base, for the current base A, C, G or T respectively. These dictionaries are given below.

$$Pr_S_A = \{C: Pr_C, G: Pr_G, T: Pr_T\}$$

$$Pr_S_C = \{A: Pr_A, G: Pr_G, T: Pr_T\}$$

$$Pr_S_G = \{A: Pr_A, C: Pr_C, T: Pr_T\}$$

$$Pr_S_T = \{A: Pr_A, C: Pr_C, G: Pr_G\}$$

Pr_A , Pr_C , Pr_G , and Pr_T represent the probability to replace the given base with the base A, C, G or T respectively.

The pseudocode for the functions is given below.

Insert base

function insertBase:

randVal = random number

totProb = 0

for key in [A, C, G, T]:

totProb = totProb + Pr_ins{key}

if randval < totProb:

return key

Substitute base

function substituteBase (B):

randVal = random number

totProb = 0

keyList = keys from Pr_sub{B}

for key in keyList:

totProb = totProb + Pr_sub{B}{key}

if randval < totProb:

return key

Populate string

function populateString (O, A):

generatedString = []

i = length of O

k = 0

for $0 \leq j < i$:

if O[j] == 't':

generatedString = generatedString + A[k]

```
        k = k + 1

    else if O[j] == 'i':
        generatedString = generatedString + insertBase()

    else if O[j] == 's':
        generatedString = generatedString + substituteBase(A[k])
        k = k + 1

return generatedString
```

APPENDIX C PYTHON IMPLEMENTATION OF THE MEMORY MODEL

The Python implementation for the Memory model [24], [105], as summarised in Section 4.11, is given in pseudocode below. Let Y represent the output string generated by the model. x_t represents the t^{th} base of the input string, while N represents the length of the input string.

Memory model implementation

```
// Initialize
Y = []

// First position of string
Select  $e_t$  according to  $P(e_t)$  for  $t = 1$ 
if  $e_t == \text{Insertion}$ :
    Select  $L_t$  according to  $P(L_t|e_t = \text{Insertion})$  for  $t = 1$ 
    for  $i = 1, \dots, L_t$ :
        Select  $B$  according to  $P(B|e_t = \text{Insertion})$  for  $t = 1$ 
        Append to  $Y$ :  $B$ 
else if  $e_t == \text{Substitution}$ :
    Select  $B_t$  according to  $P(B_t|e_t = \text{Insertion})$  for  $t = 1$ 
    Append to  $Y$ :  $B_t$ 
else if  $e_t == \text{Deletion}$ :
    // Do nothing
else: // Transmission
    Append to  $Y$ :  $x_1$ 

// Positions smaller than 6
for  $t = 2, 3, 4, 5, 6$ 
    Select  $e_t$  according to  $P(e_t|x_t)$ 
    if  $e_t == \text{Insertion}$ :
        Select  $L_t$  according to  $P(L_t|x_t, e_t = \text{Insertion})$ 
        for  $i = 1, \dots, L_t$ :
            Select  $B$  according to  $P(B|e_t = \text{Insertion})$ 
            Append to  $Y$ :  $B$ 
```

```

else if  $e_t == \text{Substitution}$ :
    Select  $B_t$  according to  $P(B_t|x_t, e_t = \text{Insertion})$ 
    Append to  $Y$ :  $B_t$ 
else if  $e_t == \text{Deletion}$ :
    // Do nothing
else: // Transmission
    Append to  $Y$ :  $x_t$ 

// Positions before the end of the string
for  $t = 7, 8, \dots, N - 1$ 
    Select  $e_t$  according to  $P(e_t|kmer_t)$ 
    if  $e_t == \text{Insertion}$ :
        Select  $L_t$  according to  $P(L_t|kmer_t, e_t = \text{Insertion})$ 
        for  $i = 1, \dots, L_t$ :
            Select  $B$  according to  $P(B|e_t = \text{Insertion})$ 
            Append to  $Y$ :  $B$ 
    else if  $e_t == \text{Substitution}$ :
        Select  $B_t$  according to  $P(B_t|kmer_t, e_t = \text{Insertion})$ 
        Append to  $Y$ :  $B_t$ 
    else if  $e_t == \text{Deletion}$ :
        // Do nothing
    else: // Transmission
        Append to  $Y$ :  $x_t$ 

// Final position of the string
Select  $e_t$  according to  $P(e_t)$  for  $t = N$ 
if  $e_t == \text{Insertion}$ :
    Select  $L_t$  according to  $P(L_t|e_t = \text{Insertion})$  for  $t = N$ 
    for  $i = 1, \dots, L_t$ :
        Select  $B$  according to  $P(B|e_t = \text{Insertion})$  for  $t = N$ 
        Append to  $Y$ :  $B$ 
else if  $e_t == \text{Substitution}$ :
    Select  $B_t$  according to  $P(B_t|e_t = \text{Insertion})$  for  $t = N$ 
    Append to  $Y$ :  $B_t$ 
else if  $e_t == \text{Deletion}$ :

```

```

    // Do nothing
else: // Transmission
    Append to  $Y$ :  $x_N$ 

// Insertions after the final position of the string
Select  $e_t$  according to  $P(e_t = \text{Insertion})$  for  $t = N + 1$ 
if  $e_t == \text{Insertion}$ :
    Select  $L_t$  according to  $P(L_t | e_t = \text{Insertion})$  for  $t = N + 1$ 
    for  $i = 1, \dots, L_t$ :
        Select  $B$  according to  $P(B | e_t = \text{Insertion})$  for  $t = N + 1$ 
        Append to  $Y$ :  $B$ 

```

APPENDIX D ALGORITHM FOR DETECTING ERROR RUNS

The function to detect error runs requires a string of outcomes O as input. The value of **check** can be set to different values to find runs of different types of errors:

1. *not (out == 't') // Runs consisting of errors of any type*
2. *if out == 'i': // Runs of insertion errors*
3. *if out == 'd': // Runs of deletion errors*
4. *if out == 's': // Runs of substitution errors*

The algorithm output is a list. The value at each index of the list indicates the number of runs with a run-length matching the index. The pseudocode of the algorithm follows:

Get error runs

```
function getRuns (O):  
  
    declare R of size 500     // Algorithm detects runs up to length 500  
    set each value of R to 0  
  
    runLength = 0  
    for out in O:  
        if check:  
            runLength = runLength + 1  
        else:  
            R[runLength] = R[runLength] + 1  
            runLength = 0  
    R[runLength] = R[runLength] + 1 // Runs at the end of the string  
  
    return R
```

APPENDIX E ALGORITHM FOR DETECTING ERROR BURSTS

The function to detect error bursts requires a string of outcomes O as input. A threshold T is also given by the user, representing the maximum number of correct transitions between two errors for it to be considered a burst.

The algorithm outputs the list *bursts*. Each entry in *bursts* represents a burst in the following format:

[Length of burst, number of errors in burst, starting position of burst]

The pseudocode of the algorithm follows:

Get error bursts

function getBursts (O, T):

bursts = []

burstLength = 0

burstErrs = 0

burstLoc = 0

burstStart = False

correct = 0

k = length of O

for $0 \leq i < k$:

if *burstStart* and $O[i] == 't'$:

correct = correct + 1

burstLength = burstLength + 1

if $correct \geq T$ and *burstStart* == *True* and *burstErrs* > 1:

bursts = bursts + [burstLength - correct, burstErrs, i -burstLengths + 1]

if correct $\geq T$:

burstStart = False

if not (O[i] == 't'):

if burstStart == False:

burstLength = 1

burstErrs = 1

burstStart = True

else

burstLength = burstLength + 1

burstErrs = burstErrs + 1

correct = 0

return bursts

APPENDIX F ALGORITHM FOR DETECTING ERROR-FREE RUNS

The function to detect error-free runs requires a string of outcomes O as input. The algorithm output is a list. The value at each index of the list indicates the number of error-free runs with a run-length matching the index. The pseudocode for the function is given below.

Get error-free runs

function *getErrorFreeRuns* (O):

declare R of size 500 *// Algorithm detects error-free runs up to length 500*

set each value of R *to* 0

$runLength = 0$

for out in O :

if $out == 't'$:

$runLength = runLength + 1$

else:

$R[runLength] = R[runLength] + 1$

$runLength = 0$

$R[runLength] = R[runLength] + 1$ *// Runs at the end of the string*

return R

APPENDIX G ALGORITHM FOR DETECTING ERRORS IN HOMOPOLYMER RUNS

The function to find a homopolymer run requires two inputs, namely a string *center* and the position to start searching *i*. The function then determines the length of the homopolymer run starting at *i*, and returns this length *k*.

The function to find errors within homopolymer runs within a string requires a string *center* as input. It requires a second input *errors*, that is a list of all the positions of errors within that string. The function returns a list *polymers* as output. Each entry of the list is in the following form:

[Length of homopolymer run,
base of homopolymer run,
number of errors in run,
number of errors right before the run,
number of errors right after the run]

The pseudocode for both these algorithms are given below.

Get homopolymer runs

function *getHomopolymers* (*center*, *k*):

k = 1

j = *length of center*

 while *i* + *k* < *j*

 if not (*center*[*i*] == *center*[*i* + *k*]):

 return *k*

k = *k* + 1

 return *k*

Get errors in homopolymer runs

```
function getErrorsInRuns (center, errors):  
    polymers = []  
    k = length of center  
    for  $0 \leq i < k - \text{minSize} - 1$ :  
        currentLength = getHomopolymers(center, i)  
        if currentLength > minSize - 1:  
            temp = []  
            temp = [temp + currentLength]  
            temp = [temp + center[i]]  
            errors = 0  
            for  $0 \leq j < \text{currentLength}$ :  
                for error in errorList:  
                    if error == i + j:  
                        errors = errors + 1  
  
            temp = [temp + errors]  
  
            before = 0  
            after = 0  
            for error in errorList:  
                if error == i + currentLength:  
                    after = after + 1  
                else if error == i - 1:  
                    before = before + 1  
  
            temp = [temp + before]
```

temp = [temp + after]

polymers = [polymers + temp]

skip iterations of i: currentLength - 1

return polymers

APPENDIX H EXAMPLE STRING FROM THE DATASET USED TO VERIFY THE LEVENSHTSTEIN DISTANCE ERROR IDENTIFICATION ALGORITHM

Legend									
Deletion					Insertion				
Substitution					Swap				

Position in original string										0	1	2	3	4	5	6	7	8	
Base in original string										A	C	C	A	T	A	A	T	G	...
Base in noisy string										A	C	C	A	T	A	A	T	G	...
Position in noisy string										0	1	2	3	4	5	6	7	8	

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
C	G	T	G	G	G	G	C	C	G	A	C	C	T	C	G	G	...
C	G	T	G	G	G	G	C	T	G	A	C	C	T	C	G	G	...
9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	

26	27	28	29	30		31	32	33	34	35	36	37	38	39	40		
A	A	T	G	C		G	G	T	C	T	C	C	A	T	G	...	
A	A	T	G	C	G	T	G	G	T	C	T	C	C	A	T	G	...
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	

41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	
C	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T	...
C	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T	...
43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	

58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	
A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C	...
A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C	...
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	

75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	
A	T	T	G	G	A	C	C	T	C	T	G	A	T	G	G	C	...
A	T	T		G	A	C	C	T	C	T	G	A	T	G	G	C	...
77	78	79		80	81	82	83	84	85	86	87	88	89	90	91	92	

92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A
G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A
93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110

Supplementary Figure H-1: Evaluating the error identification algorithm on an example from the dataset

APPENDIX I EXAMPLE STRING TO VERIFY THE GENERATION OF STRINGS OF OUTCOMES

Legend																											
Deletion					Insertion					Substitution																	
Position in original string										0	1	2	3	4	5	6	7	8									
Base in original string										A	C	C	A	T	A	A	T	G									
Base in noisy string										A	C	C	A	T	A	A	T	G									
Position in noisy string										0	1	2	3	4	5	6	7	8									
String of outcomes										t	t	t	t	t	t	t	t	t									
										9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
										C	G	T	G	G	G	G	C	C	G	A	C	C	T	C	G	G	
										C	G	T	G	G	G	G	C	T	G	A	C	C	T	C	G	G	
										9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
										t	t	t	t	t	t	t	t	s	t	t	t	t	t	t	t		
										26	27	28	29	30			31	32	33	34	35	36	37	38	39	40	
										A	A	T	G	C			G	G	T	C	T	C	C	A	T	G	
										A	A	T	G	C	G	T	G	G	T	C	T	C	C	A	T	G	
										26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	
										t	t	t	t	t	i	i	t	t	t	t	t	t	t	t	t	t	
										41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	
										C	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T	
										C	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T	
										43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	
										t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t		
										58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	
										A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C	
										A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C	
										60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	
										t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t		
										75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	
										A	T	T	G	G	A	C	C	T	C	T	G	A	T	G	G	C	
										A	T	T		G	A	C	C	T	C	T	G	A	T	G	G	C	
										77	78	79		80	81	82	83	84	85	86	87	88	89	90	91	92	
										t	t	t	d	t	t	t	t	t	t	t	t	t	t	t	t		
										92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
										G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A
										G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A
										93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110
										t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	

Supplementary Figure I-1: Generated string of outcomes compared to original and noisy DNA strings

APPENDIX J EXAMPLE STRING TO VERIFY THE GENERATION OF STRINGS OF OUTCOMES FOR INDEPENDENT GILBERT-ELLIOT MODELS

Legend																	
Deletion						Insertion						Substitution					
String of outcomes									t	t	t	t	t	t	t	t	t
String representing insertions									1	1	1	1	1	1	1	1	1
String representing deletions									1	1	1	1	1	1	1	1	1
String representing substitutions									1	1	1	1	1	1	1	1	1
									...								
t									t	t	t	t	t	t	t	t	t
1									1	1	1	1	1	1	1	1	1
1									1	1	1	1	1	1	1	1	1
1									2	1	1	1	1	1	1	1	1
									...								
t									t	t	t	t	t	t	t	t	t
1									2	2	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
									...								
t									t	t	t	t	t	t	t	t	t
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
									...								
t									t	t	t	t	t	t	t	t	t
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
									...								
t									t	t	t	t	t	t	t	t	t
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
									...								
t									t	t	t	t	t	t	t	t	t
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
									...								
t									t	t	t	t	t	t	t	t	t
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
									...								

Supplementary Figure J-1: Strings of outcomes for each error type compared to original string of outcomes

APPENDIX K ERRORS IN EXAMPLE STRINGS USED TO VERIFY THE ALGORITHM TO CALCULATE STATISTICS

['i', 0, 0, 'C']

['i', 1, 0, 'C']

['i', 2, 0, 'G']

['s', 0, 0, 'A', 'G']

['s', 1, 1, 'A', 'C']

['i', 6, 6, 'A']

['i', 7, 6, 'A']

['i', 8, 6, 'A']

['s', 4, 4, 'G', 'A']

['s', 5, 5, 'T', 'A']

['s', 6, 6, 'T', 'A']

['i', 0, 0, 'A']

['i', 2, 1, 'T']

['s', 7, 5, 'T', 'G']

['s', 8, 6, 'T', 'G']

['d', 9, 7, 'A']

['s', 17, 17, 'C', 'T']

['i', 31, 31, 'G']

['i', 32, 31, 'T']

['d', 80, 78, 'G']

APPENDIX L EXAMPLES FOR VERIFYING THE CALCULATION OF ERROR RATES AT EACH STRING POSITION

The example test set used for Section 5.10.5 as well as the tables summarizing the results are given in this appendix. An example test set consisting of six strings is used. The following errors are found in the respective strings of the example set:

1. ['s', 17, 17, 'C', 'T'], ['i', 31, 31, 'G'], ['i', 32, 31, 'T'], ['d', 80, 78, 'G']
2. ['d', 51, 51, 'C'], ['d', 68, 69, 'T'], ['d', 69, 71, 'G'], ['s', 101, 104, 'C', 'T'], ['i', 102, 105, 'G']
3. ['i', 24, 24, 'G']
4. ['d', 12, 12, 'G'], ['s', 23, 24, 'G', 'A']
5. ['i', 27, 27, 'G'], ['s', 36, 35, 'T', 'A'], ['i', 94, 93, 'C'], ['i', 95, 93, 'T'], ['i', 96, 93, 'T'],
['i', 97, 93, 'A'], ['d', 113, 108, 'A']
6. ['i', 21, 21, 'T'], ['d', 23, 22, 'T'], ['d', 48, 48, 'C'], ['d', 77, 78, 'G'], ['d', 96, 98, 'A']

The number of errors at each position can then be tallied, as well as the number of each type of error at each position. The results are summarised in Supplementary Table L-1. Positions without any errors are not shown in the table.

The overall error rate, as well as the rate of each type of error, can then be calculated at each position. Since there are six strings in the example set, each position occurs six times. The number of errors at each position can be divided by the number of strings in order to find the error rate at each position. The process can be repeated for each type of error. The results are summarised in Supplementary Table L-2. Positions without any errors are excluded from the table.

Supplementary Table L-1: Number of errors at different positions within the example data set

Position	Number of errors	Number of insertions	Number of deletions	Number of substitutions
12	1	0	1	0
17	1	0	0	1
21	1	1	0	0
22	1	0	1	0
24	2	1	0	1
27	1	1	0	0
31	2	2	0	0
35	1	0	0	1
48	1	0	1	0
51	1	0	1	0
69	1	0	1	0
71	1	0	1	0
78	2	0	2	0
93	4	4	0	0
98	1	0	1	0
104	1	0	0	1
105	1	1	0	0
108	1	0	1	0

Supplementary Table L-2: Error rate at different positions within the example data set

Position	Insertion rate	Deletion rate	Substitution rate	Error rate
12	0	0.167	0	0.167
17	0	0	0.167	0.167
21	0.167	0	0	0.167
22	0	0.167	0	0.167
24	0.167	0	0.167	0.333
27	0.167	0	0	0.167
31	0.333	0	0	0.333
35	0	0	0.167	0.167
48	0	0.167	0	0.167
51	0	0.167	0	0.167
69	0	0.167	0	0.167
71	0	0.167	0	0.167
78	0	0.333	0	0.333
93	0.667	0	0	0.667
98	0	0.167	0	0.167
104	0	0	0.167	0.167
105	0.167	0	0	0.167
108	0	0.167	0	0.167

APPENDIX M SUMMARY OF GC CONTENT ANALYSIS FOR A SAMPLE STRING

Legend																								
Deletion					Insertion					Substitution														
Position in original string										0	1	2	3	4	5	6	7	8						
Base in original string										A	C	C	A	T	A	A	T	G						
Outcome										t	t	t	t	t	t	t	t	t	...					
Number of errors										0					0									
GC content										2					3									
9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25								
C	G	T	G	G	G	G	C	C	G	A	C	C	T	C	G	G								
t	t	t	t	t	t	t	t	s	t	t	t	t	t	t	t	t	...							
					1					0					0									
					6					4					3									
26	27	28	29	30			31	32	33	34	35	36	37	38	39	40								
A	A	T	G	C			G	G	T	C	T	C	C	A	T	G								
t	t	t	t	t			t	t	t	t	t	t	t	t	t	t	...							
					2					0					...									
					4					4														
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57								
	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T								
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...							
					0					0					0									
					3					4					2									
58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74								
A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C								
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...							
					0					0					0									
					4					3					1									
75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91								
A	T	T	G	G	A	C	C	T	C	T	G	A	T	G	G	C								
t	t	t	d	t	t	t	t	t	t	t	t	t	t	t	t	t	...							
					1					0					0									
					4					3					4									
92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109							
G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A							
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...						
					0					0					NA									
					1					5					NA									

Supplementary Table M-1: Summary of the error rate of substrings with different GC contents in the example string

GC content of substring	Percentage GC content	Number of occurrences	Number of bases	Number of errors	Error rate
0	0.0	0	0	0	0
1	16.7	2	12	0	0
2	33.3	2	12	0	0
3	50.0	5	30	0	0
4	66.7	7	42	3	0.071
5	83.3	1	6	0	0
6	100.0	1	6	1	0.167

APPENDIX N SUMMARY OF RDS ANALYSIS FOR A SAMPLE STRING

Legend																	
Deletion						Insertion						Substitution					
Position in original string						0						8					
Base in original string						A						T					
Outcome						t						t					
Running digital sum						-1						-3					
9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
C	G	T	G	G	G	G	C	C	G	A	C	C	T	C	G	G	...
t	t	t	t	t	t	t	t	s	t	t	t	t	t	t	t	t	...
-2	-1	-2	-1	0	1	2	3	4	5	4	5	6	5	6	7	8	
26	27	28	29	30				31	32	33	34	35	36	37	38	39	40
A	A	T	G	C				G	G	T	C	T	C	C	A	T	G
t	t	t	t	t	i	i	t	t	t	t	t	t	t	t	t	t	...
7	6	5	6	7	8	8	8	9	8	9	8	9	10	9	8	9	
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	
C	G	C	G	T	T	T	C	C	T	C	C	A	A	C	C	T	...
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...
10	11	12	13	12	11	10	11	12	11	12	13	12	11	12	13	12	
58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	
A	A	G	G	T	A	G	C	C	T	G	T	A	G	T	T	C	...
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	...
11	10	11	12	11	10	11	12	13	12	13	12	11	12	11	10	11	
75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	
A	T	T	G	G	A	C	C	T	C	T	G	A	T	G	G	C	...
t	t	t	d	t	t	t	t	t	t	t	t	t	t	t	t	t	...
10	9	8	9	10	9	10	11	10	11	10	11	10	9	10	11	12	
92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
G	C	T	T	A	T	A	G	A	A	A	C	C	G	G	G	A	A
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t
13	14	13	12	11	10	9	10	9	8	7	8	9	10	11	12	11	10

Supplementary Figure N-1: RDS analysis for a sample string

Supplementary Table N-1: Summary of RDS analysis for a sample string

RDS	Number of occurrences	Number of errors	Error rate
-5	0	0	NA
-4	1	0	0.000
-3	2	0	0.000
-2	3	0	0.000
-1	4	0	0.000
0	3	0	0.000
1	2	0	0.000
2	1	0	0.000
3	1	0	0.000
4	2	1	0.500
5	4	0	0.000
6	4	0	0.000
7	4	0	0.000
8	8	2	0.250
9	12	1	0.083
10	17	0	0.000
11	19	0	0.000
12	15	0	0.000
13	7	0	0.000
14	1	0	0.000
15	0	0	NA

APPENDIX O EXAMPLES FOR PROBLEMATIC SUBSTRING ANALYSIS

Supplementary Table O-1: Analysis of substrings of length 2 for example strings

Substring	Number of occurrences	Total number of bases	Errors in substring	Error rate
AA	11	22	9	0.409
AC	5	10	4	0.400
AT	2	4	1	0.250
AG	1	2	0	0
CA	2	4	1	0.250
CC	3	6	1	0.167
CT	2	4	0	0
CG	6	12	6	0.500
GA	2	4	0	0
GC	4	8	1	0.125
GT	4	8	2	0.250
GG	3	6	2	0.333
TA	2	4	2	0.500
TC	3	6	0	0
TT	2	4	2	0.500
TG	3	6	1	0.167

Supplementary Table O-2: Summary of errors at the first position of substrings of length 2 for example strings

Substring	Insertions	Deletions	Substitutions	Errors
AA	2	1	2	5
AC	0	1	0	1
AT	0	0	0	0
AG	0	0	0	0
CA	0	1	0	1
CC	0	0	0	0
CT	0	0	0	0
CG	3	0	0	3
GA	0	0	0	0
GC	0	0	0	0
GT	0	1	0	1
GG	2	0	0	2
TA	0	0	1	1
TC	0	0	0	0
TT	0	0	1	1
TG	0	1	0	1

Supplementary Table O-3: Summary of error rates at the first position of substrings of length 2 for example strings

Substring	Insertion rate	Deletion rate	Substitution rate	Error rate
AA	0.182	0.091	0.182	0.455
AC	0.000	0.200	0.000	0.200
AT	0.000	0.000	0.000	0.000
AG	0.000	0.000	0.000	0.000
CA	0.000	0.500	0.000	0.500
CC	0.000	0.000	0.000	0.000
CT	0.000	0.000	0.000	0.000
CG	0.500	0.000	0.000	0.500
GA	0.000	0.000	0.000	0.000
GC	0.000	0.000	0.000	0.000
GT	0.000	0.250	0.000	0.250
GG	0.667	0.000	0.000	0.667
TA	0.000	0.000	0.500	0.500
TC	0.000	0.000	0.000	0.000
TT	0.000	0.000	0.500	0.500
TG	0.000	0.333	0.000	0.333

Supplementary Table O-4: Summary of errors at the second position of substrings of length 2 for example strings

Substring	Insertions	Deletions	Substitutions	Errors
AA	1	1	2	4
AC	2	1	0	3
AT	0	1	0	1
AG	0	0	0	0
CA	0	0	0	0
CC	0	0	1	1
CT	0	0	0	0
CG	2	1	0	3
GA	0	0	0	0
GC	0	1	0	1
GT	0	0	1	1
GG	0	0	0	0
TA	0	1	0	1
TC	0	0	0	0
TT	0	0	1	1
TG	0	0	0	0

Supplementary Table O-5: Summary of error rates at the second position of substrings of length 2 for example strings

Substring	Insertion rate	Deletion rate	Substitution rate	Error rate
AA	0.091	0.091	0.182	0.364
AC	0.400	0.200	0.000	0.600
AT	0.000	0.500	0.000	0.500
AG	0.000	0.000	0.000	0.000
CA	0.000	0.000	0.000	0.000
CC	0.000	0.000	0.333	0.333
CT	0.000	0.000	0.000	0.000
CG	0.333	0.167	0.000	0.500
GA	0.000	0.000	0.000	0.000
GC	0.000	0.250	0.000	0.250
GT	0.000	0.000	0.250	0.250
GG	0.000	0.000	0.000	0.000
TA	0.000	0.500	0.000	0.500
TC	0.000	0.000	0.000	0.000
TT	0.000	0.000	0.500	0.500
TG	0.000	0.000	0.000	0.000

Supplementary Table O-6: Summary of problematic substring analysis of substrings of length 6 for an example string

Substring	Number of occurrences	Number of errors	Error rate
ACCTCG	1	0	0.000
AATGCG	1	3	0.500
ATGCGG	1	3	0.500
CGGTCT	1	2	0.333
CGGAAT	1	1	0.167
CCGACC	1	1	0.167
CGACCT	1	1	0.167
CCTCGG	1	0	0.000
CTCGGA	1	0	0.000
GAATGC	1	1	0.167
GGCCGA	1	1	0.167
GCCGAC	1	1	0.167
GACCTC	1	0	0.000
GGAATG	1	1	0.167
GCGGTC	1	2	0.333
GGTCTC	1	2	0.333
GTCTCC	1	0	0.000
TCTCCA	1	0	0.000
TGCGGT	1	3	0.500
TCGGAA	1	0	0.000
AAAAAC	0	0	NA
AAAAAA	0	0	NA
⋮			

APPENDIX P TRANSITION AND EMISSION MATRICES OF DIFFERENT MARKOV MODELS

All values in this appendix are rounded to two decimal places.

Supplementary Table P-1: Transition matrix of GE model pertaining insertion errors

State	1	2
1	0.98	0.02
2	0.81	0.19

Supplementary Table P-2: Transition matrix of GE model pertaining deletion errors

State	1	2
1	0.98	0.02
2	0.81	0.19

Supplementary Table P-3: Transition matrix of GE model pertaining substitution errors

State	1	2
1	0.99	0.01
2	0.86	0.14

Supplementary Table P-4: Emission matrix of the three GE models

State	Transition	Error
1	1	0
2	0	1

Supplementary Table P-5: Transition matrix of 7MMLS model

State	1	2	3	4	5	6	7
1	0.94	0.02	0.01	0.01	0.02	0.00	0.00
2	0.96	0.02	0.01	0.00	0.00	0.01	0.00
3	0.03	0.11	0.19	0.16	0.24	0.20	0.07
4	0.41	0.29	0.00	0.20	0.00	0.09	0.01
5	0.42	0.08	0.00	0.00	0.33	0.17	0.00
6	0.05	0.01	0.27	0.38	0.13	0.17	0.00
7	0.12	0.06	0.00	0.61	0.00	0.19	0.02

Supplementary Table P-6: Emission matrix of 7MMLS model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	0.17	0.00	0.00	0.83
4	0.42	0.00	0.58	0.00
5	0.39	0.61	0.00	0.00
6	1.00	0.00	0.00	0.00
7	0.02	0.00	0.98	0.00

Supplementary Table P-7: Transition matrix of 7MMIT model

State	1	2	3	4	5	6	7
1	0.91	0.05	0.01	0.01	0.01	0.00	0.01
2	0.94	0.02	0.00	0.03	0.00	0.00	0.00
3	0.12	0.10	0.16	0.00	0.16	0.28	0.17
4	0.01	0.66	0.00	0.33	0.00	0.00	0.00
5	0.14	0.41	0.00	0.00	0.21	0.24	0.00
6	0.11	0.03	0.19	0.10	0.15	0.31	0.11
7	0.00	0.43	0.00	0.31	0.00	0.21	0.06

Supplementary Table P-8: Emission matrix of 7MMIT model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	0.00	0.00	0.00	1.00
4	0.52	0.00	0.48	0.00
5	0.06	0.94	0.00	0.00
6	1.00	0.00	0.00	0.00
7	0.00	0.00	1.00	0.00

Supplementary Table P-9: Transition matrix of 7MM model

State	1	2	3	4	5	6	7
1	0,91	0.05	0.00	0.01	0.02	0.00	0.01
2	0,97	0.03	0.0	0.00	0.00	0.00	0.00
3	0,02	0.01	0.32	0.00	0.00	0.04	0.61
4	0,15	0.08	0.08	0.17	0.19	0.15	0.18
5	0.24	0.33	0.07	0.00	0.25	0.11	0.00
6	0.04	0.01	0.00	0.36	0.32	0.26	0.01
7	0.49	0.21	0.06	0.00	0.00	0.04	0.20

Supplementary Table P-10: Emission matrix of 7MM model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	1.00	0.00	0.00	0.00
4	0.00	0.00	0.00	1.00
5	0.21	0.79	0.00	0.00
6	1.00	0.00	0.00	0.00
7	0.03	0.00	0.97	0.00

Supplementary Table P-11: Transition matrix of 4MM model

State	1	2	3	4
1	0.95	0.01	0.02	0.02
2	0.61	0.37	0.02	0.00
3	0.01	0.19	0.50	0.30
4	0.62	0.00	0.01	0.37

Supplementary Table P-12: Emission matrix of 4MM model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	0.44	0.56	0.00	0.00
3	0.56	0.05	0.00	0.39
4	0.45	0.00	0.55	0.00

Supplementary Table P-13: Transition matrix of 6MM model

State	1	2	3	4	5	6
1	0.94	0.02	0.01	0.01	0.02	0.00
2	0.91	0.02	0.05	0.00	0.01	0.00
3	0.01	0.01	0.34	0.30	0.28	0.06
4	0.39	0.21	0.00	0.33	0.00	0.07
5	0.38	0.07	0.00	0.00	0.40	0.15
6	0.06	0.02	0.43	0.44	0.04	0.01

Supplementary Table P-14: Emission matrix of 6MM model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	0.42	0.00	0.00	0.58
4	0.39	0.00	0.61	0.00
5	0.48	0.52	0.00	0.00
6	1.00	0.00	0.00	0.00

Supplementary Table P-15: Transition matrix of 9MM model

State	1	2	3	4	5	6	7	8	9
1	0.95	0.01	0.01	0.01	0.00	0.00	0.00	0.01	0.01
2	0.96	0.01	0.00	0.00	0.00	0.02	0.00	0.00	0.01
3	0.21	0.23	0.01	0.00	0.00	0.03	0.28	0.00	0.24
4	0.11	0.04	0.00	0.25	0.02	0.13	0.29	0.14	0.02
5	0.37	0.11	0.00	0.00	0.44	0.08	0.00	0.00	0.00
6	0.02	0.01	0.04	0.31	0.00	0.14	0.18	0.08	0.22
7	0.30	0.11	0.16	0.00	0.00	0.17	0.00	0.00	0.26
8	0.04	0.01	0.00	0.00	0.72	0.23	0.00	0.00	0.00
9	0.96	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.03

Supplementary Table P-16: Emission matrix of 9MM model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	0.31	0.00	0.69	0.00
4	0.25	0.00	0.00	0.75
5	0.66	0.34	0.00	0.00
6	1.00	0.00	0.00	0.00
7	0.22	0.00	0.78	0.00
8	0.01	0.69	0.00	0.30
9	0.57	0.00	0.43	0.00

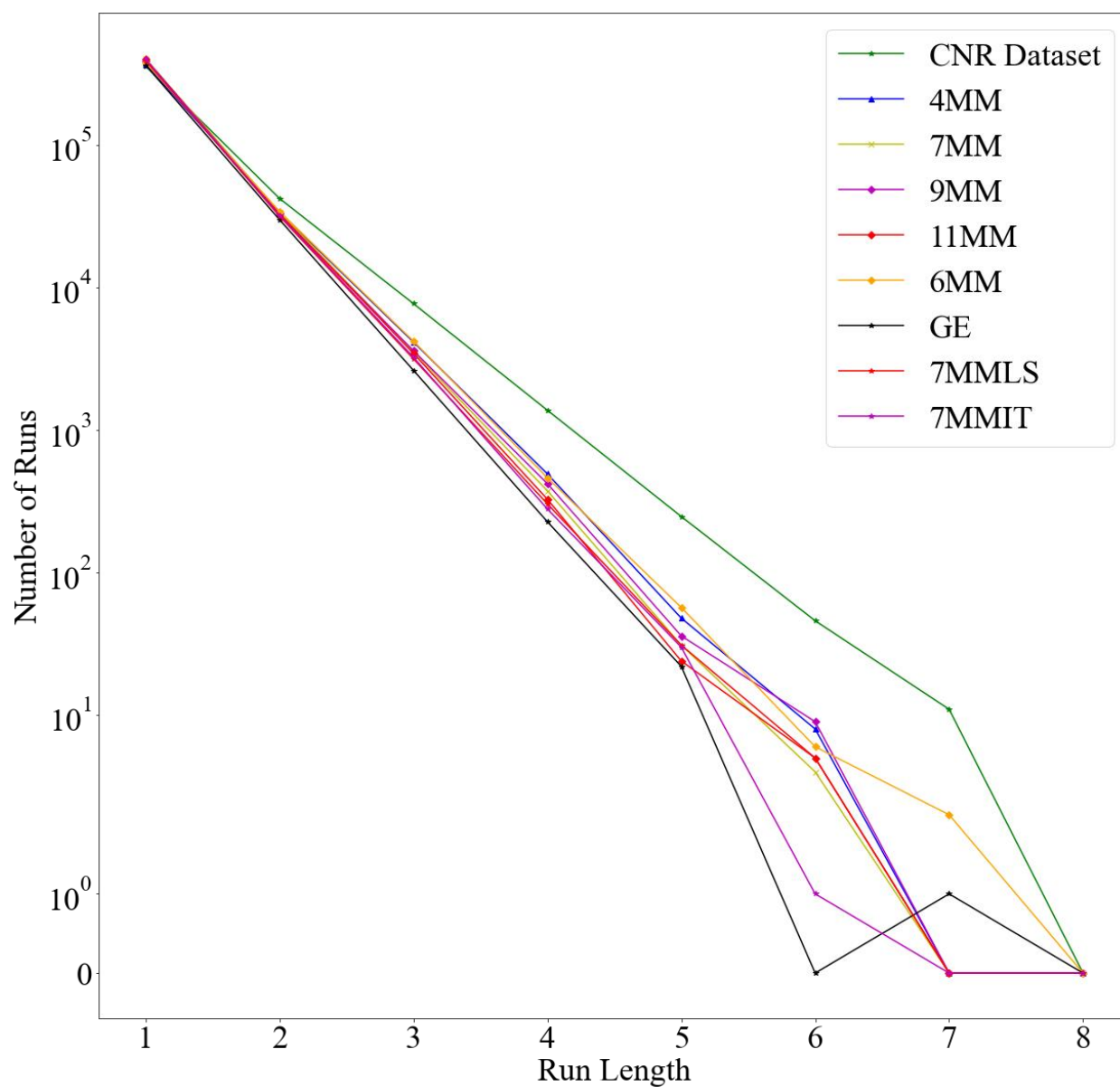
Supplementary Table P-17: Transition matrix of 11MM model

State	1	2	3	4	5	6	7	8	9	10	11
1	0.94	0.01	0.01	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.01
2	0.98	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
3	0.15	0.15	0.02	0.00	0.00	0.03	0.24	0.03	0.05	0.00	0.33
4	0.02	0.01	0.01	0.18	0.29	0.10	0.36	0.00	0.00	0.02	0.01
5	0.35	0.10	0.00	0.00	0.35	0.03	0.00	0.03	0.14	0.00	0.00
6	0.03	0.01	0.00	0.71	0.20	0.04	0.00	0.00	0.00	0.01	0.00
7	0.37	0.12	0.18	0.00	0.00	0.05	0.00	0.08	0.06	0.00	0.14
8	0.07	0.01	0.00	0.01	0.18	0.06	0.40	0.04	0.07	0.12	0.04
9	0.07	0.02	0.00	0.01	0.15	0.05	0.24	0.04	0.05	0.11	0.26
10	0.11	0.03	0.01	0.01	0.02	0.10	0.70	0.00	0.00	0.01	0.01
11	0.85	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.12

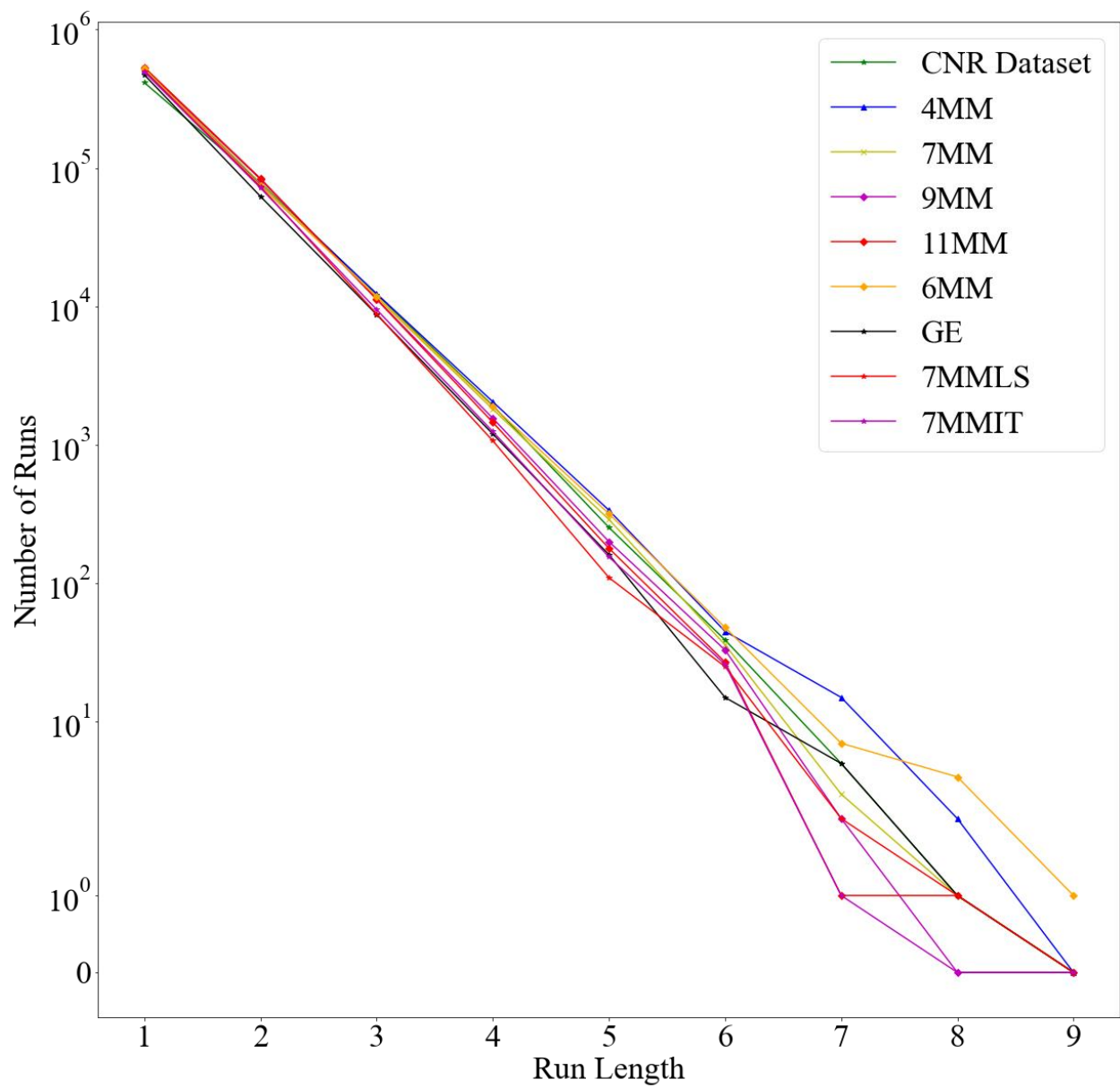
Supplementary Table P-18: Emission matrix of 11MM model

State	Transition	Insertion	Deletion	Substitution
1	1.00	0.00	0.00	0.00
2	1.00	0.00	0.00	0.00
3	0.23	0.00	0.77	0.00
4	0.04	0.00	0.00	0.96
5	0.45	0.55	0.00	0.00
6	1.00	0.00	0.00	0.00
7	0.55	0.00	0.45	0.00
8	1.00	0.00	0.00	0.00
9	1.00	0.00	0.00	0.00
10	0.52	0.00	0.00	0.48
11	0.41	0.00	0.59	0.00

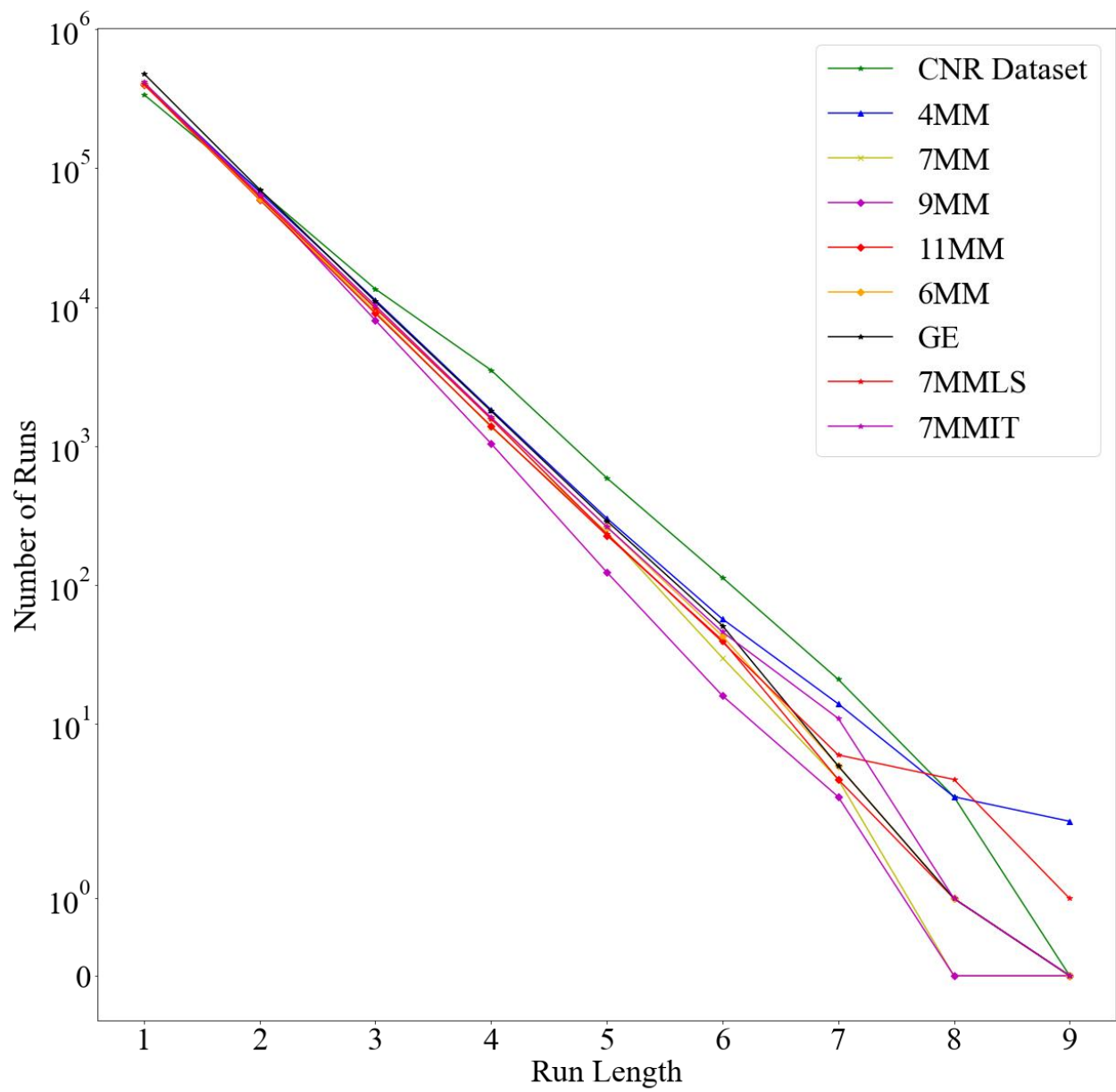
APPENDIX Q COMPARISON OF MARKOV MODELS



Supplementary Figure Q-1: Comparison of run lengths for runs of substitution errors produced by different Markov models

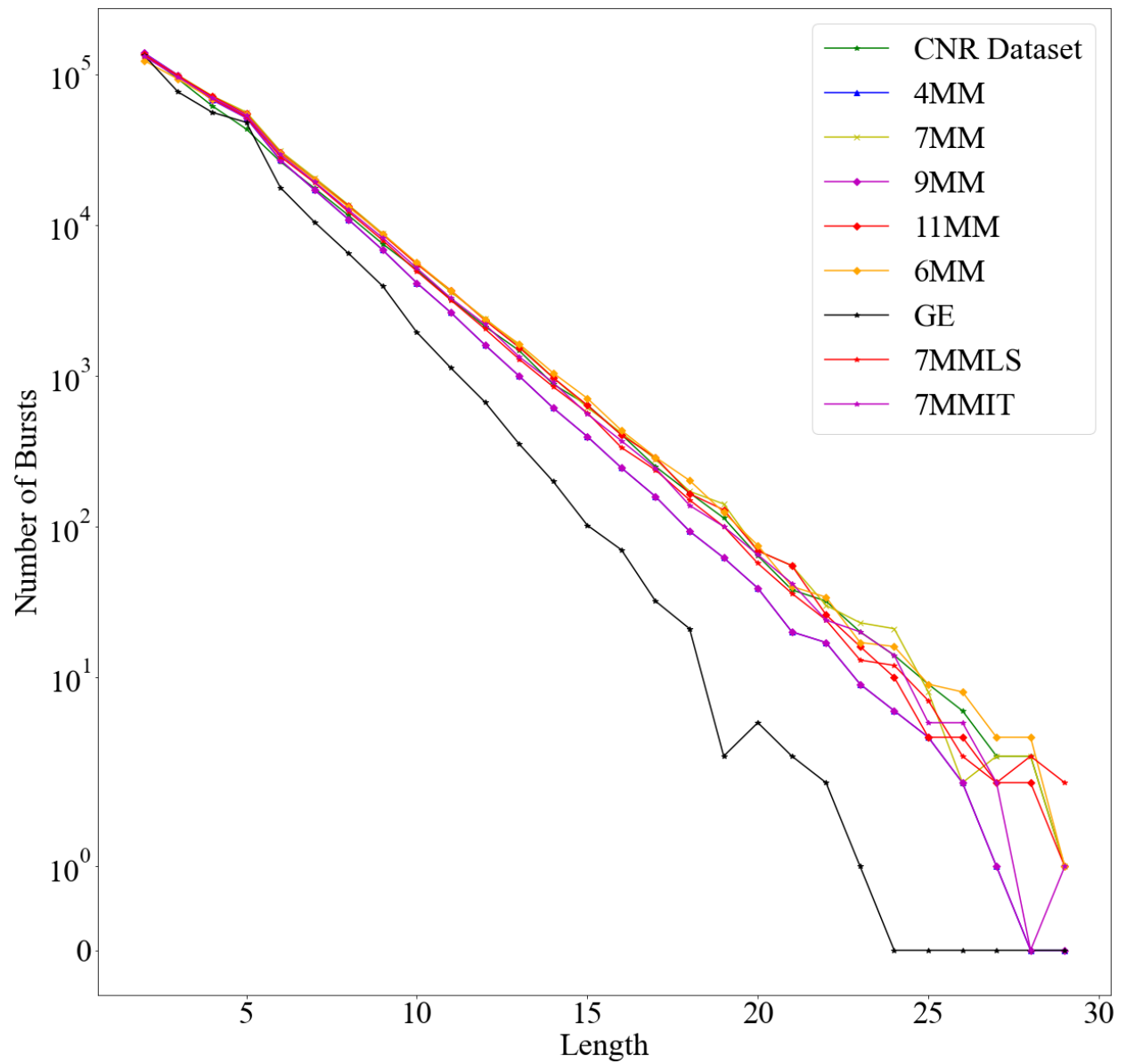


Supplementary Figure Q-2: Comparison of run lengths for runs of deletion errors produced by different Markov models



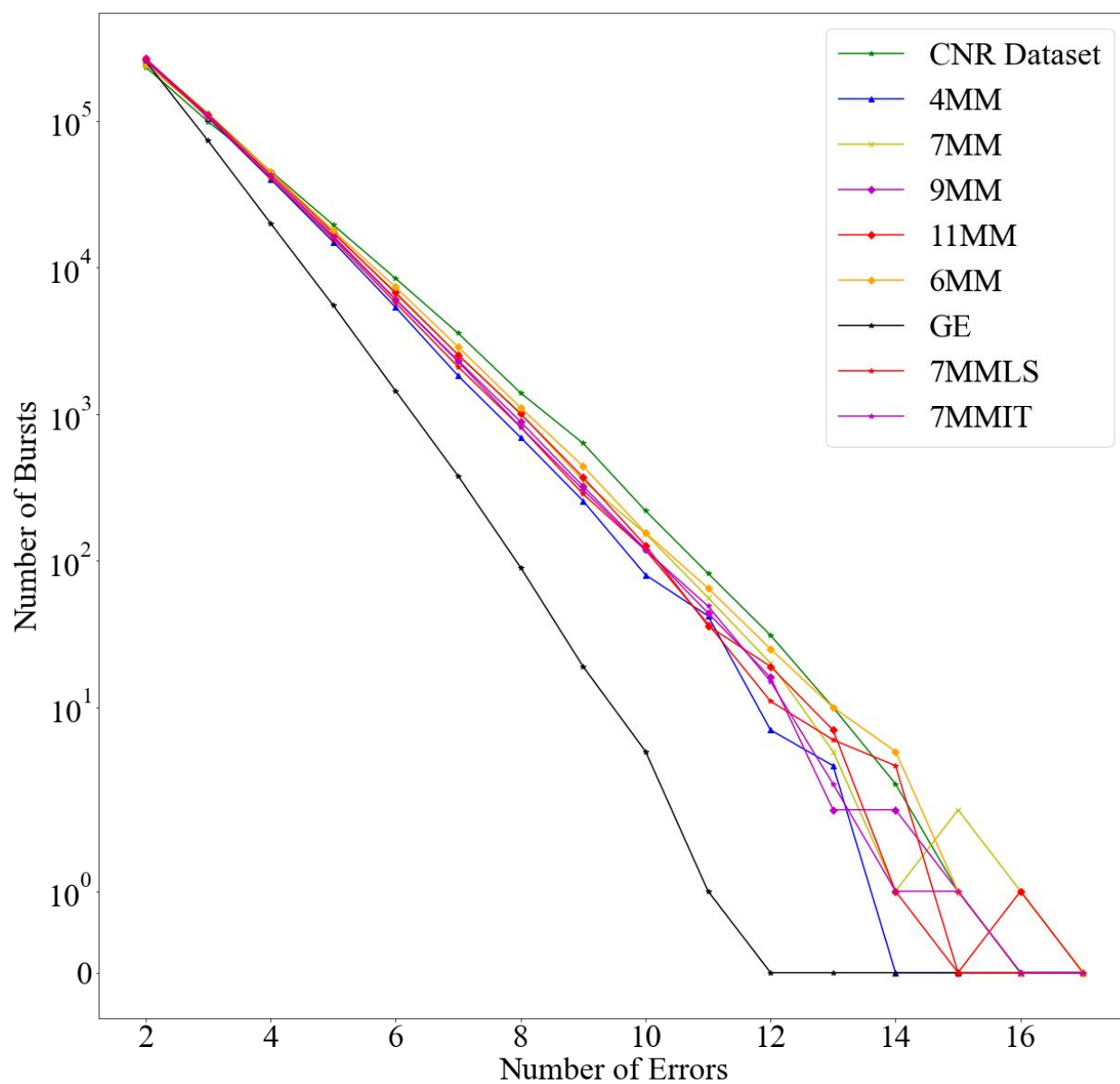
Supplementary Figure Q-3: Comparison of run lengths for runs of insertion errors produced by different Markov models

In Supplementary Figure Q-4, the number of error bursts of different lengths generated by each model is depicted. This length includes successful transitions between the errors within a burst.

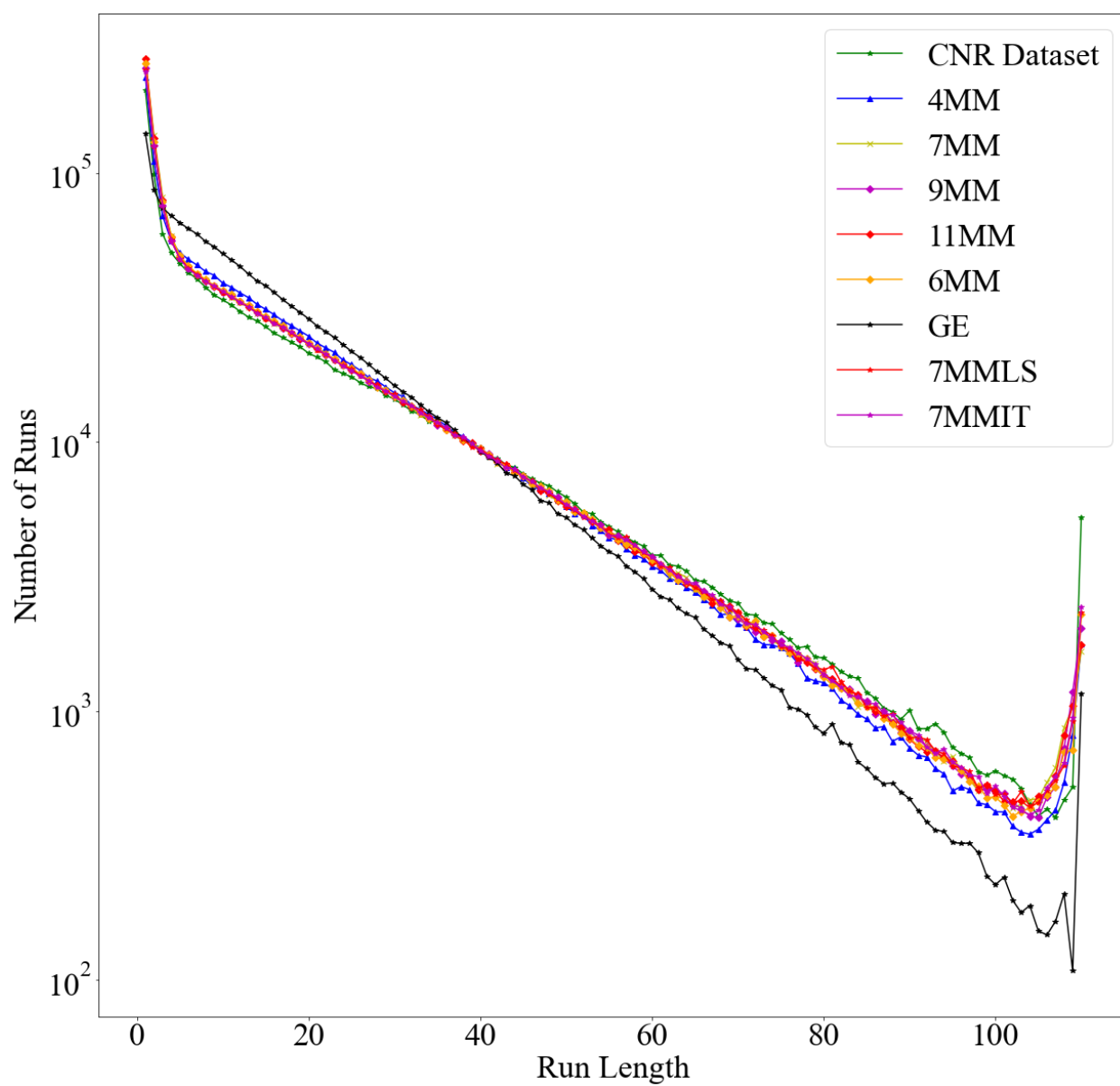


Supplementary Figure Q-4: Comparison of error burst lengths produced by different Markov models

In Supplementary Figure Q-5, the number of bursts with different amounts of errors generated by each model is depicted. This does not include successful transitions between the errors within a burst, but only the number of errors within the bursts.

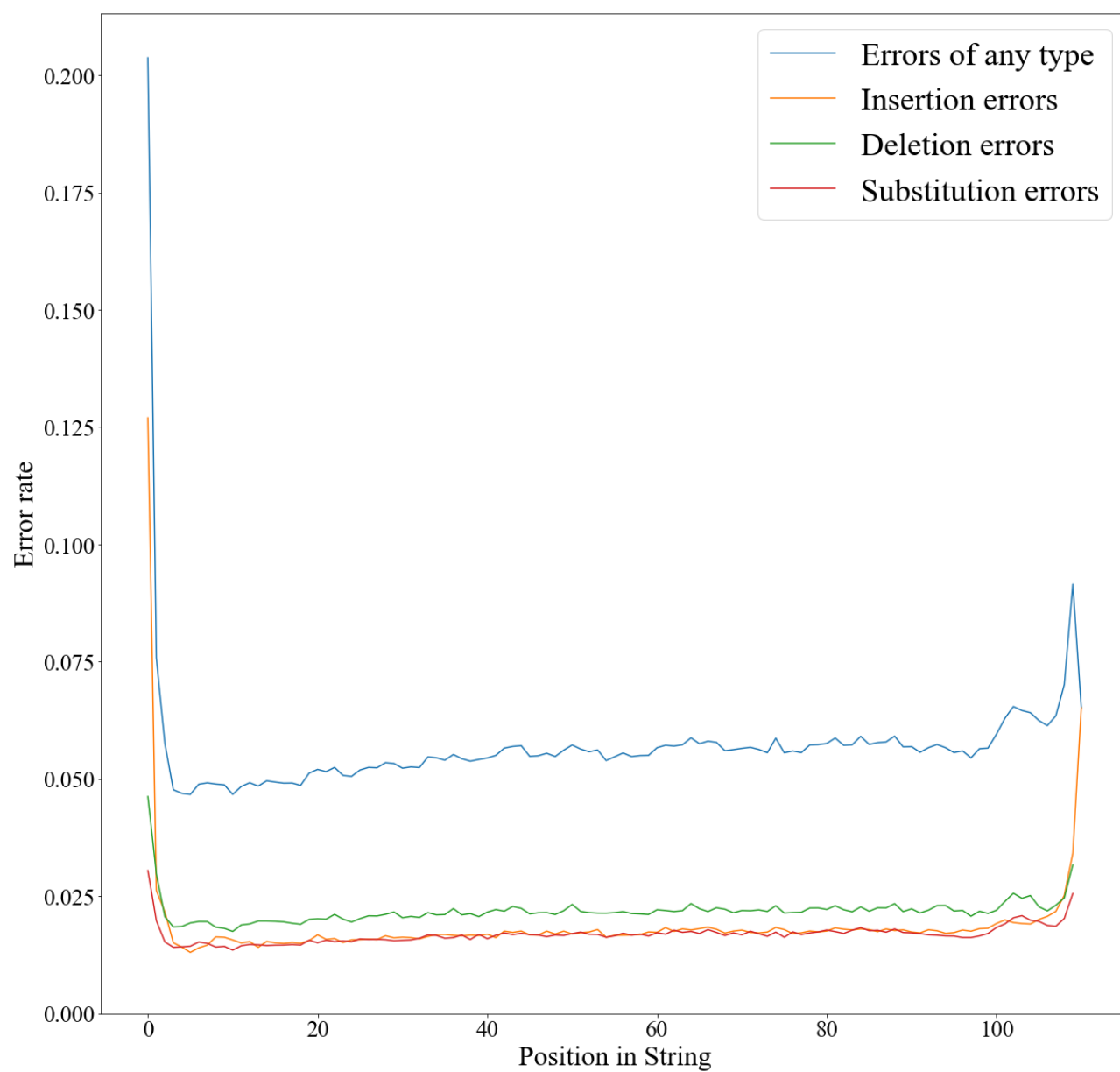


Supplementary Figure Q-5: Comparison of errors per burst produced by different Markov models

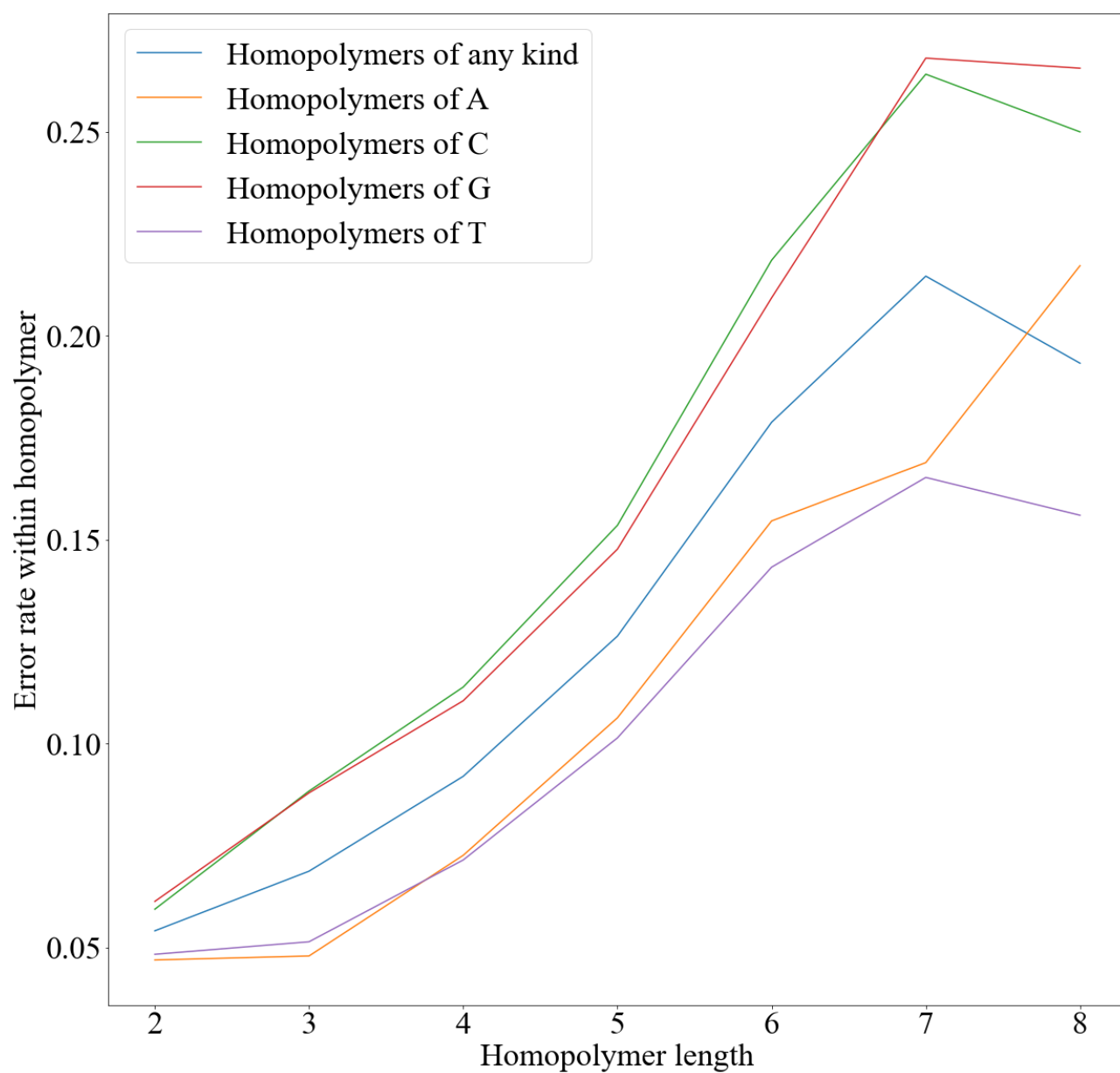


Supplementary Figure Q-6: Comparison error-free run lengths produced by different Markov models

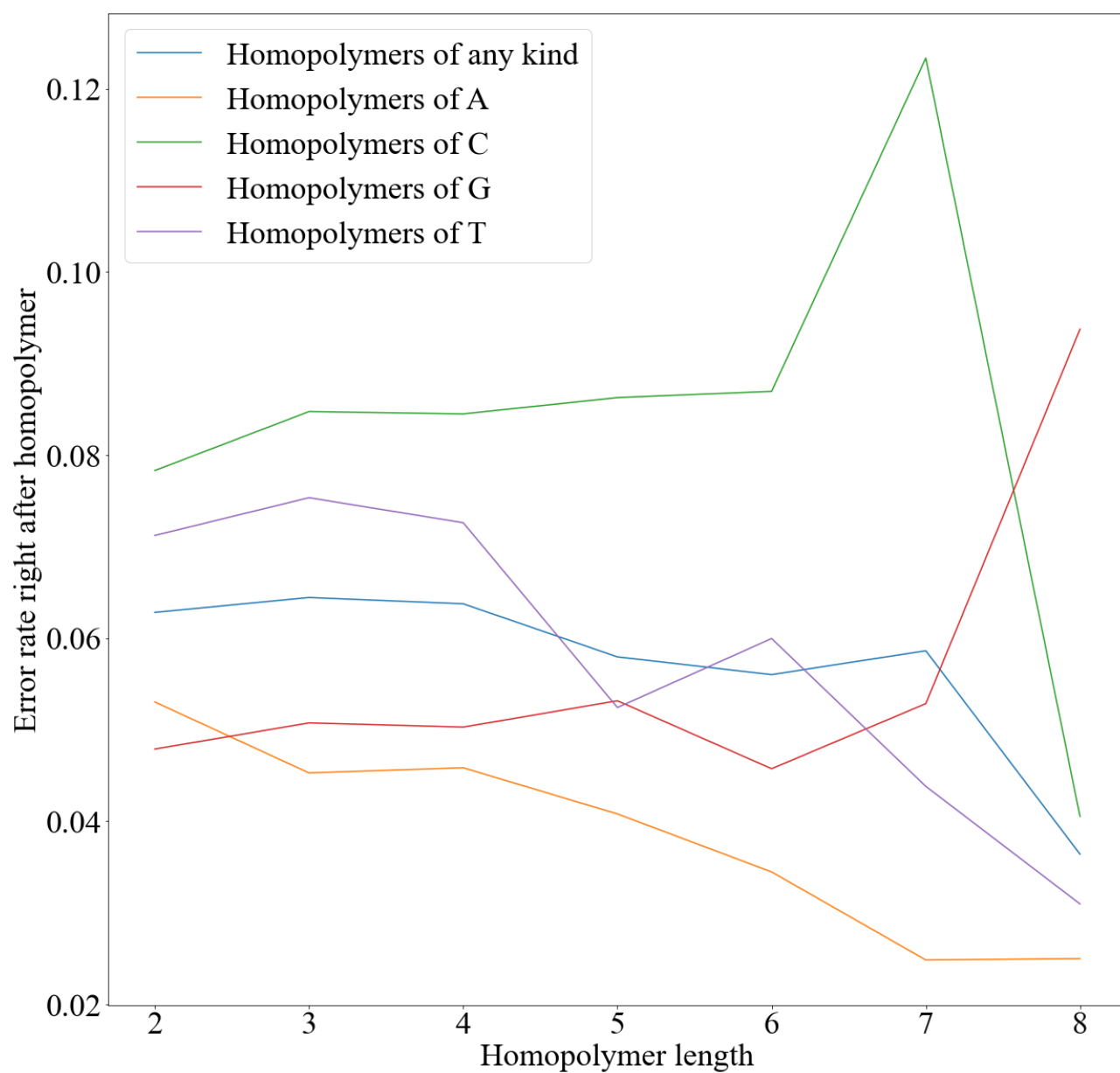
APPENDIX R ANALYSIS OF THE DATASET



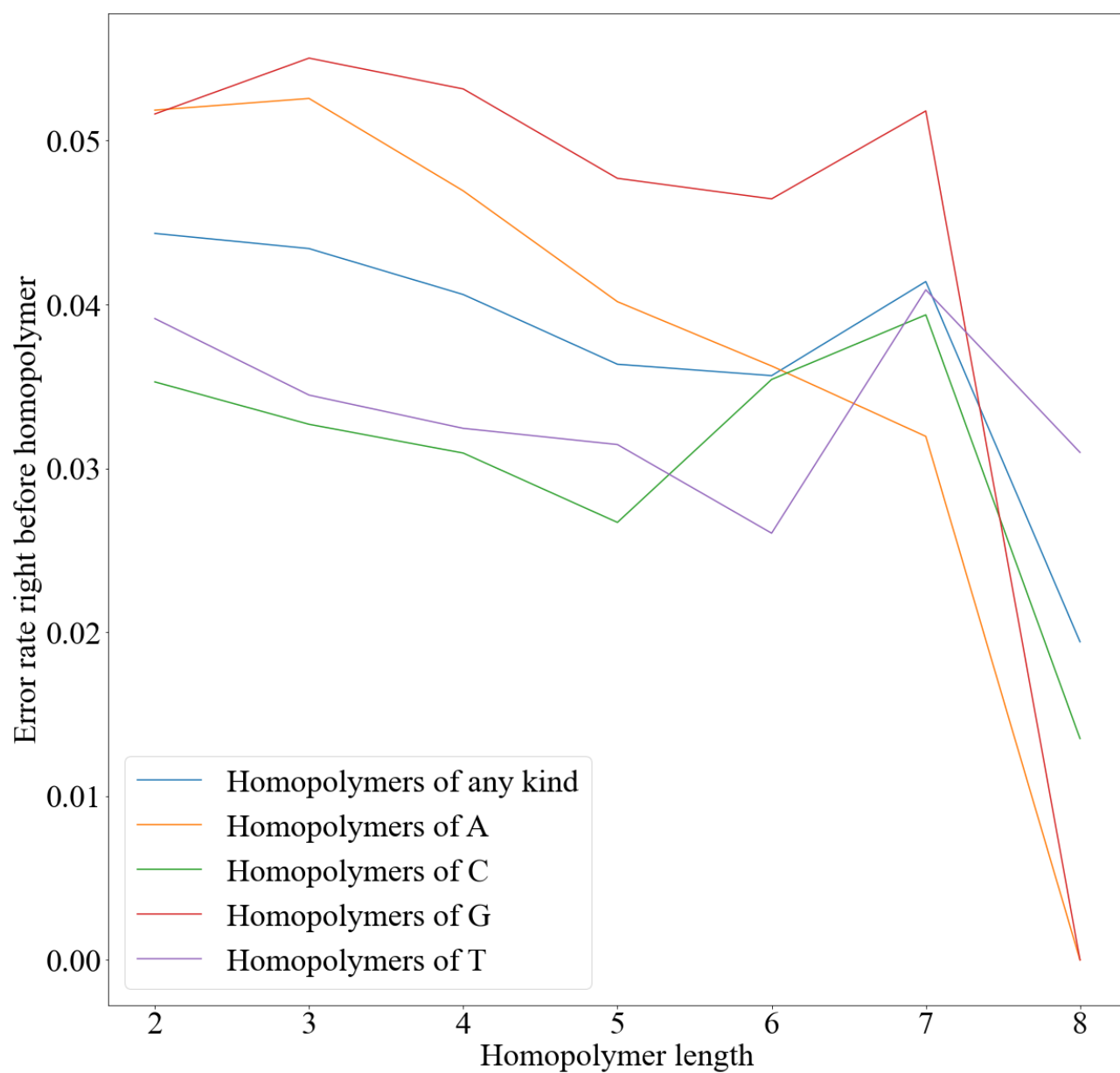
Supplementary Figure R-1: Error rates at different positions of the dataset



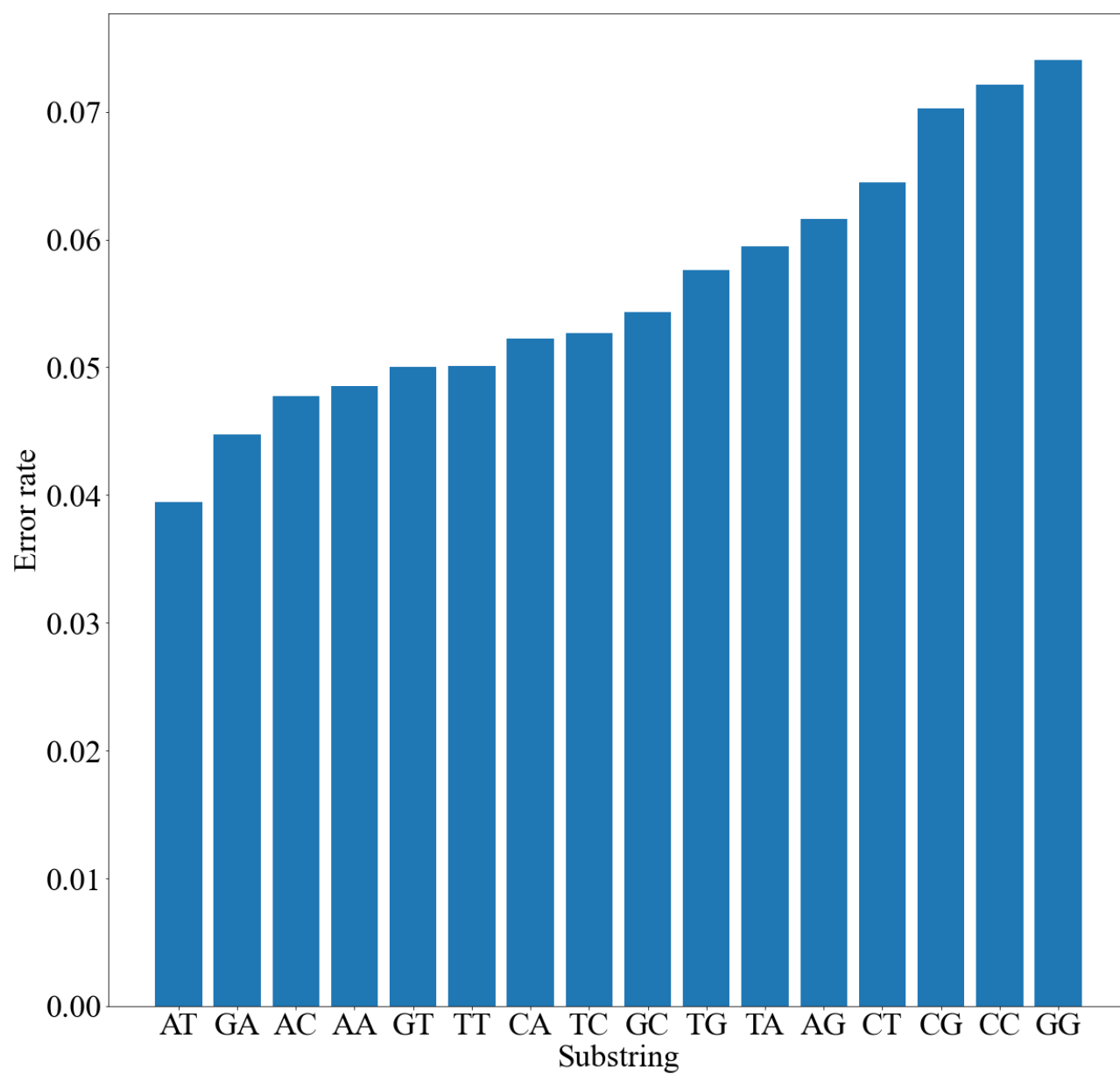
Supplementary Figure R-2: Error rate within homopolymers of different lengths in the CNR dataset



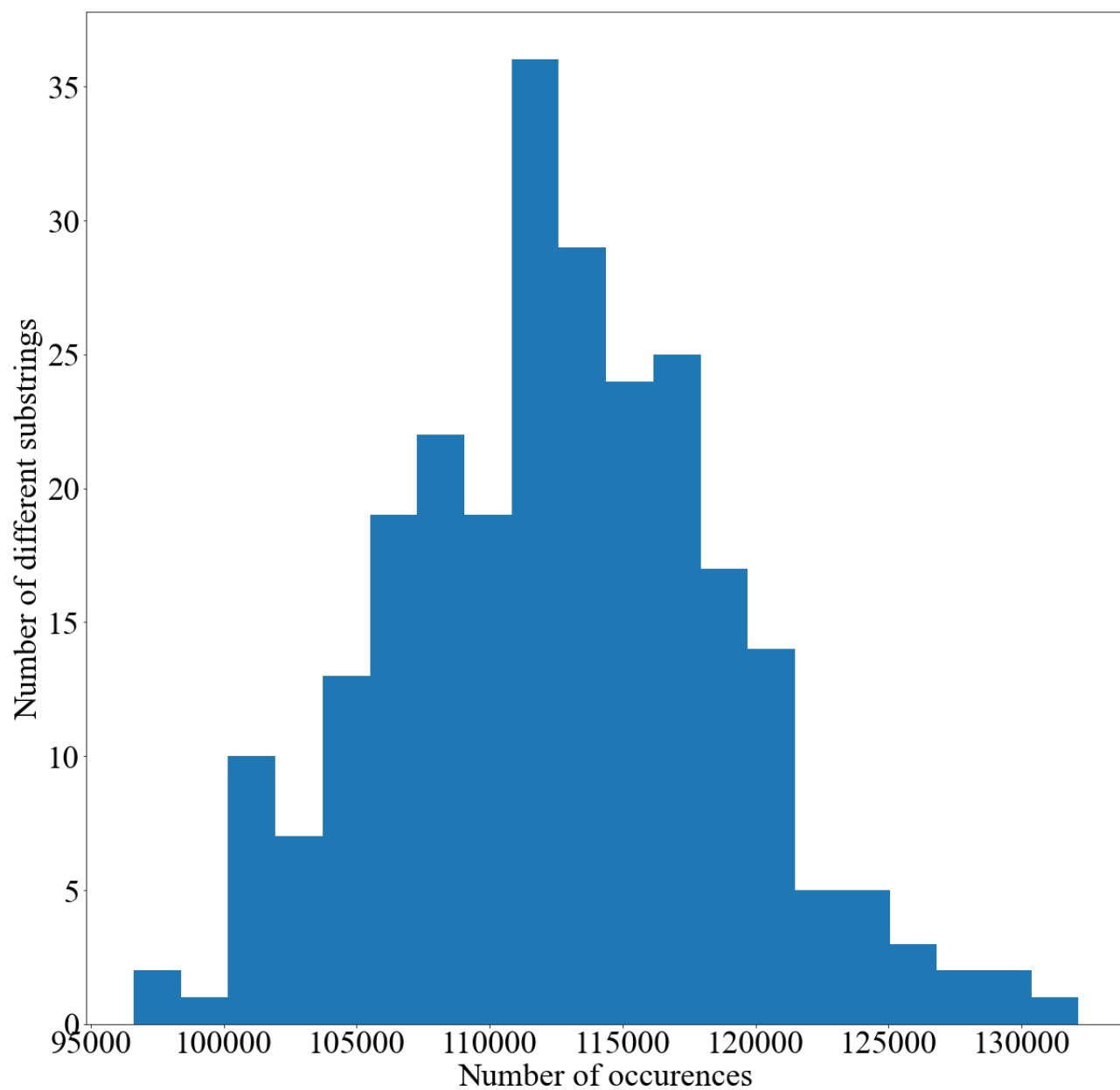
Supplementary Figure R-3: Error rate at the position right after homopolymers of different lengths in the CNR dataset



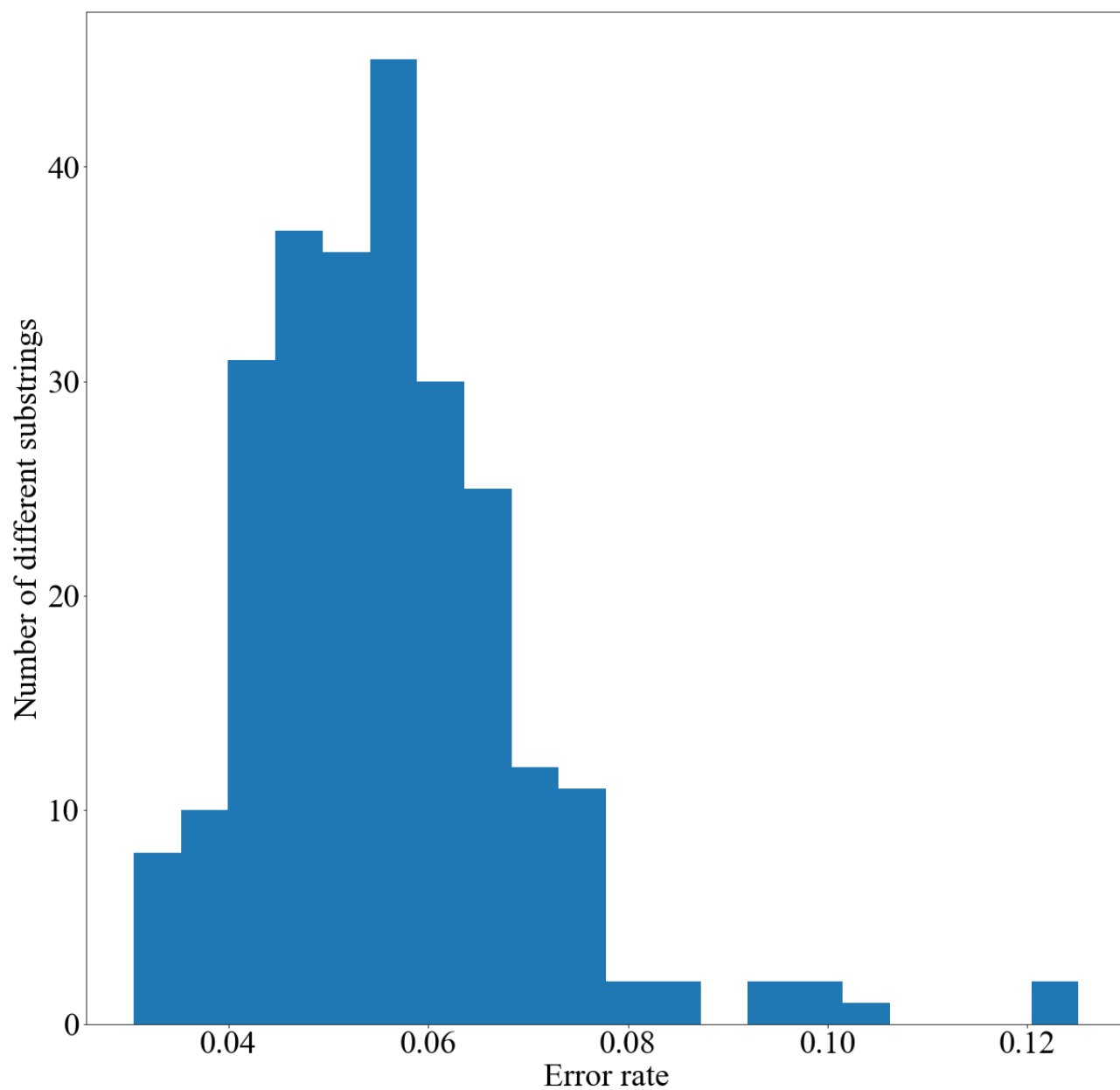
Supplementary Figure R-4: Error rate at the position right before homopolymers of different lengths in the CNR dataset



Supplementary Figure R-5: Error rate of each substring of length 2 in the dataset



Supplementary Figure R-6: Distribution of number of occurrences of different substrings of length 4 in the dataset



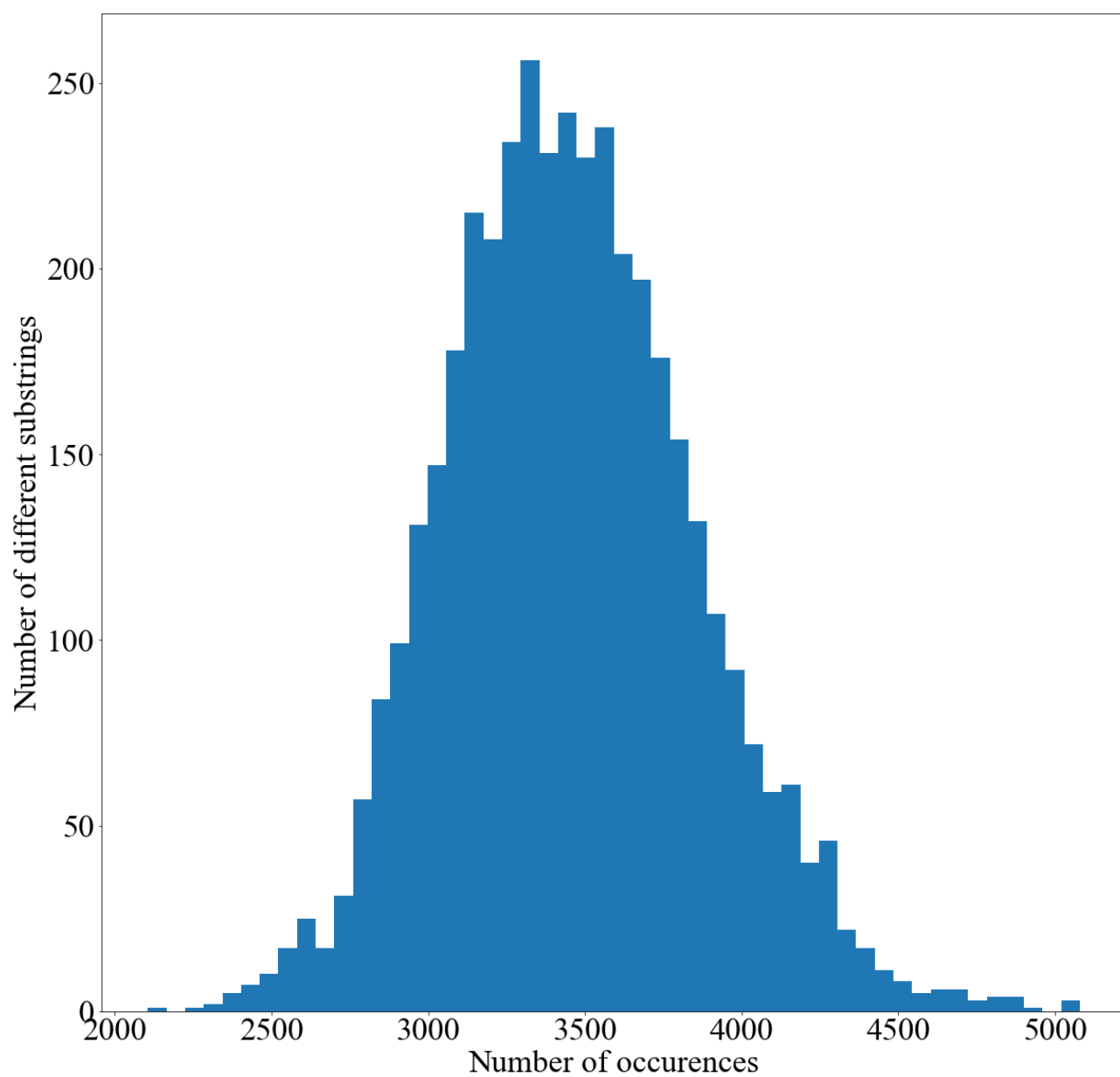
Supplementary Figure R-7: Distribution of error rates between different substrings of length 4 in the dataset

Supplementary Table R-1: Length 4 substrings in the dataset with the highest error rates

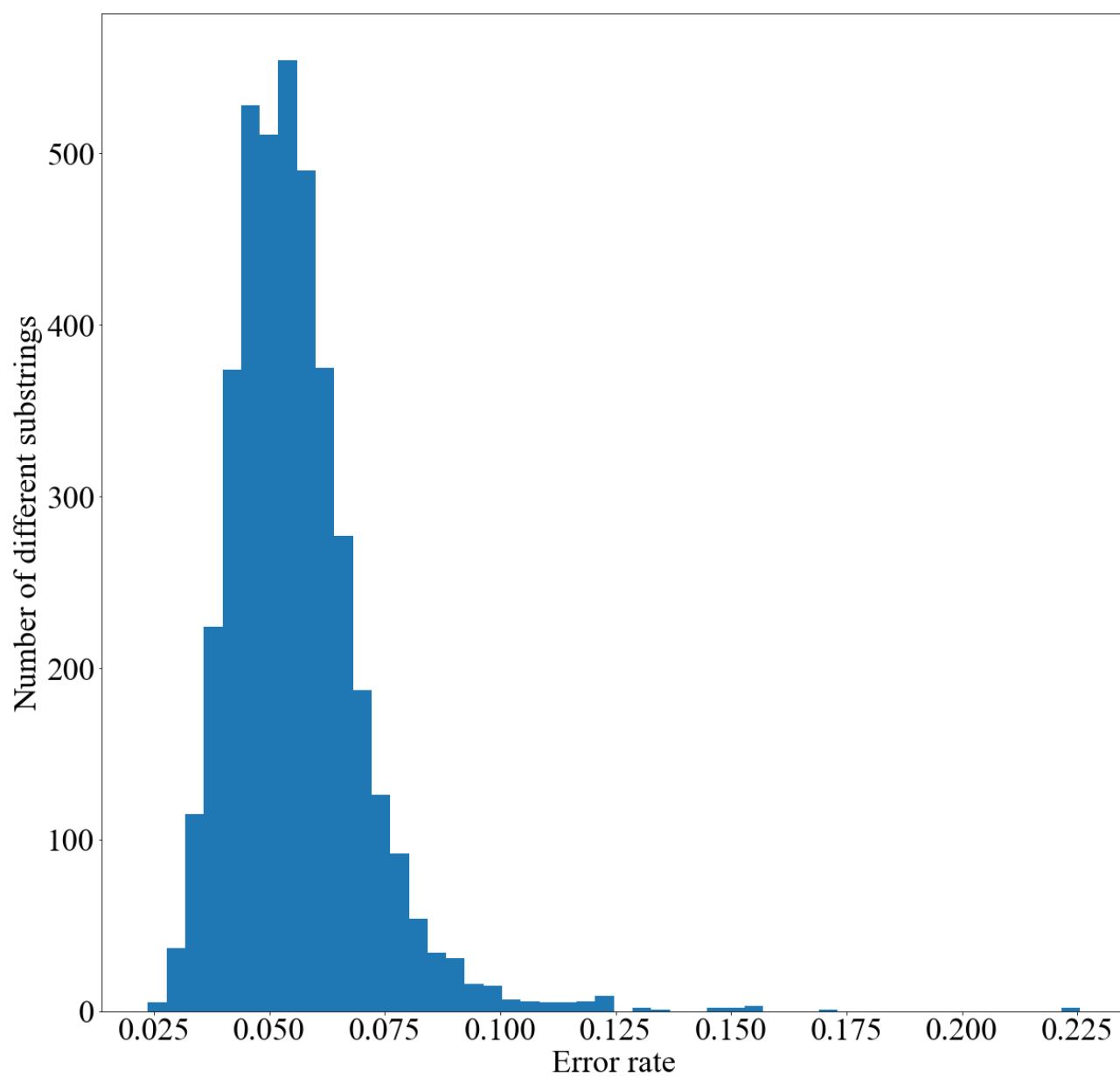
Substring	Error rate
CCCC	0.125
GGGG	0.121
CGGG	0.104
AGGG	0.101
CCCG	0.097
CCCT	0.093
TCCC	0.092
ACCC	0.085
TAGG	0.084
CCGG	0.080

Supplementary Table R-2: Length 4 substrings in the dataset with the lowest error rates

Substring	Error rate
GATC	0.031
GATG	0.031
AGAT	0.031
CATC	0.032
ATCA	0.034
GATA	0.034
GATT	0.034
GAAC	0.035
TGAT	0.035
AAAC	0.036



Supplementary Figure R-8: Distribution of number of occurrences of different substrings of length 6 in the dataset



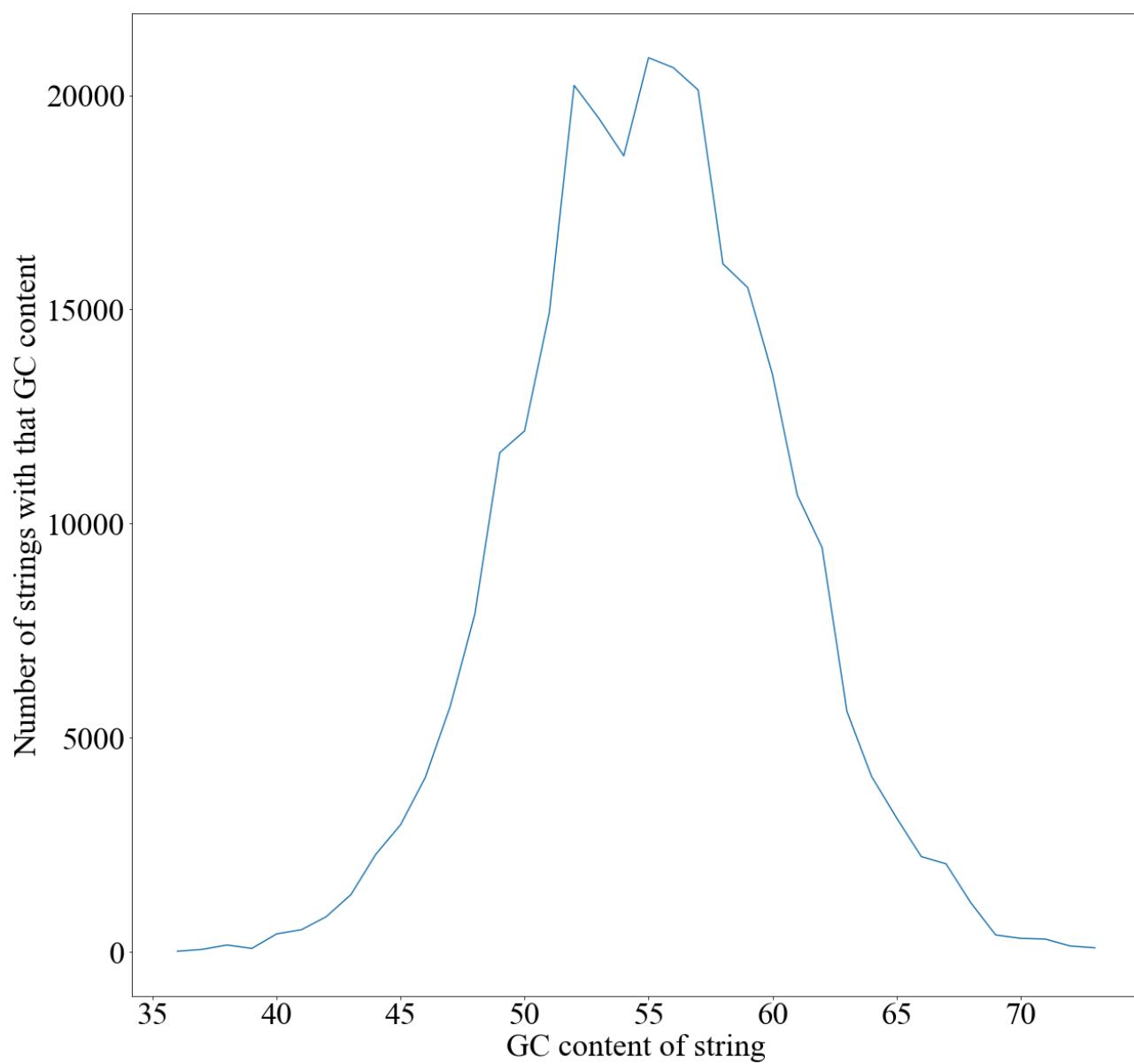
Supplementary Figure R-9: Distribution of error rates between different substrings of length 6 in the dataset

Supplementary Table R-3: Length 6 substrings in the dataset with the highest error rates

Substring	Error rate
CCCCCC	0.226
GGGGGG	0.221
CGGGGG	0.171
TCCCCC	0.153
AGGGGG	0.152
ACCCCC	0.151
CCCCCT	0.150
CCCCCG	0.149
AAAAAA	0.148
GCCCCC	0.144

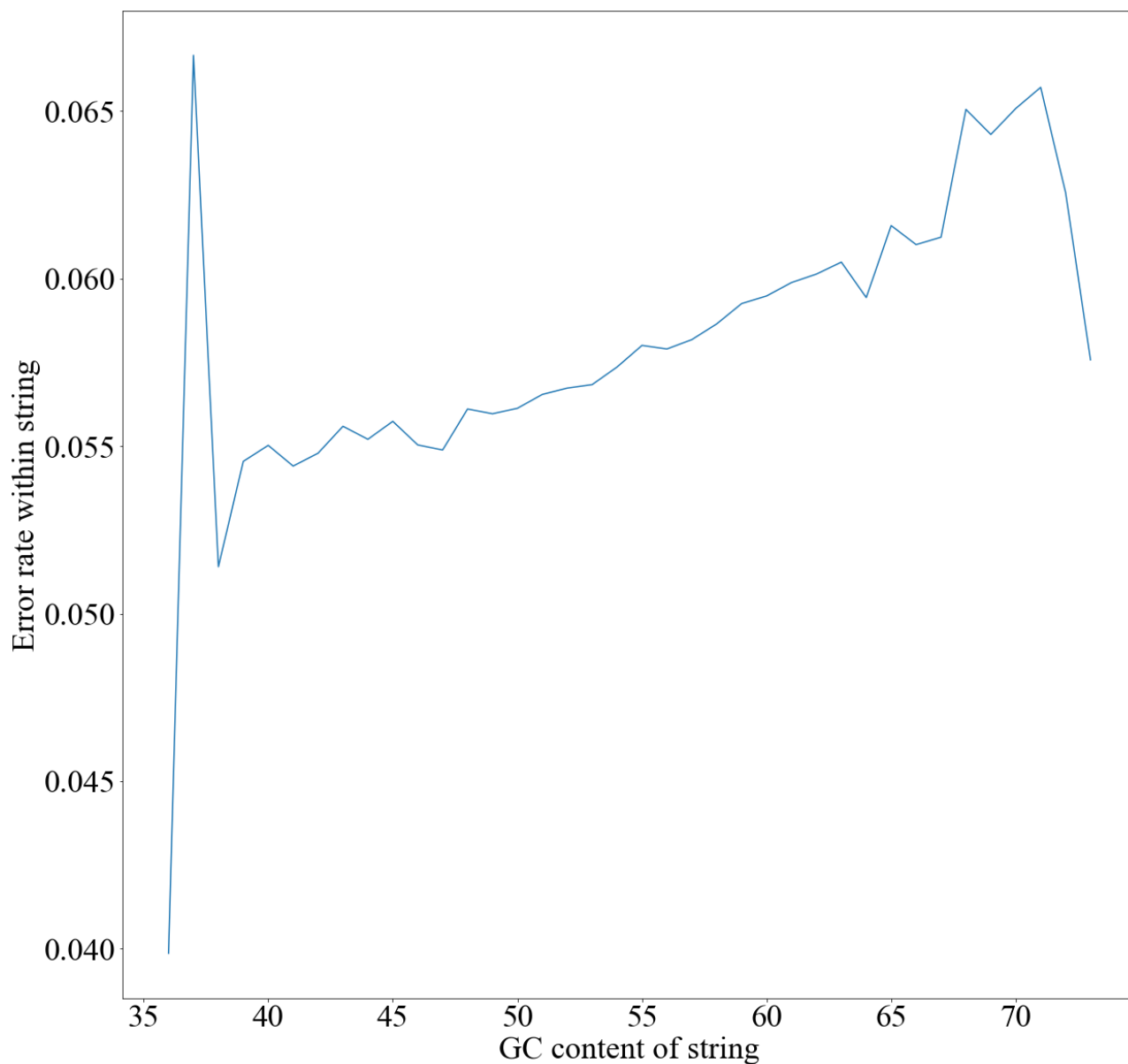
Supplementary Table R-4: Length 6 substrings in the dataset with the lowest error rates

Substring	Error rate
GATGAT	0.024
GATGGA	0.026
ATCCAT	0.026
GTCCAT	0.027
GATGAA	0.028
AACGAT	0.028
ATCATC	0.028
GATATC	0.028
GATATT	0.028
AAGATG	0.029



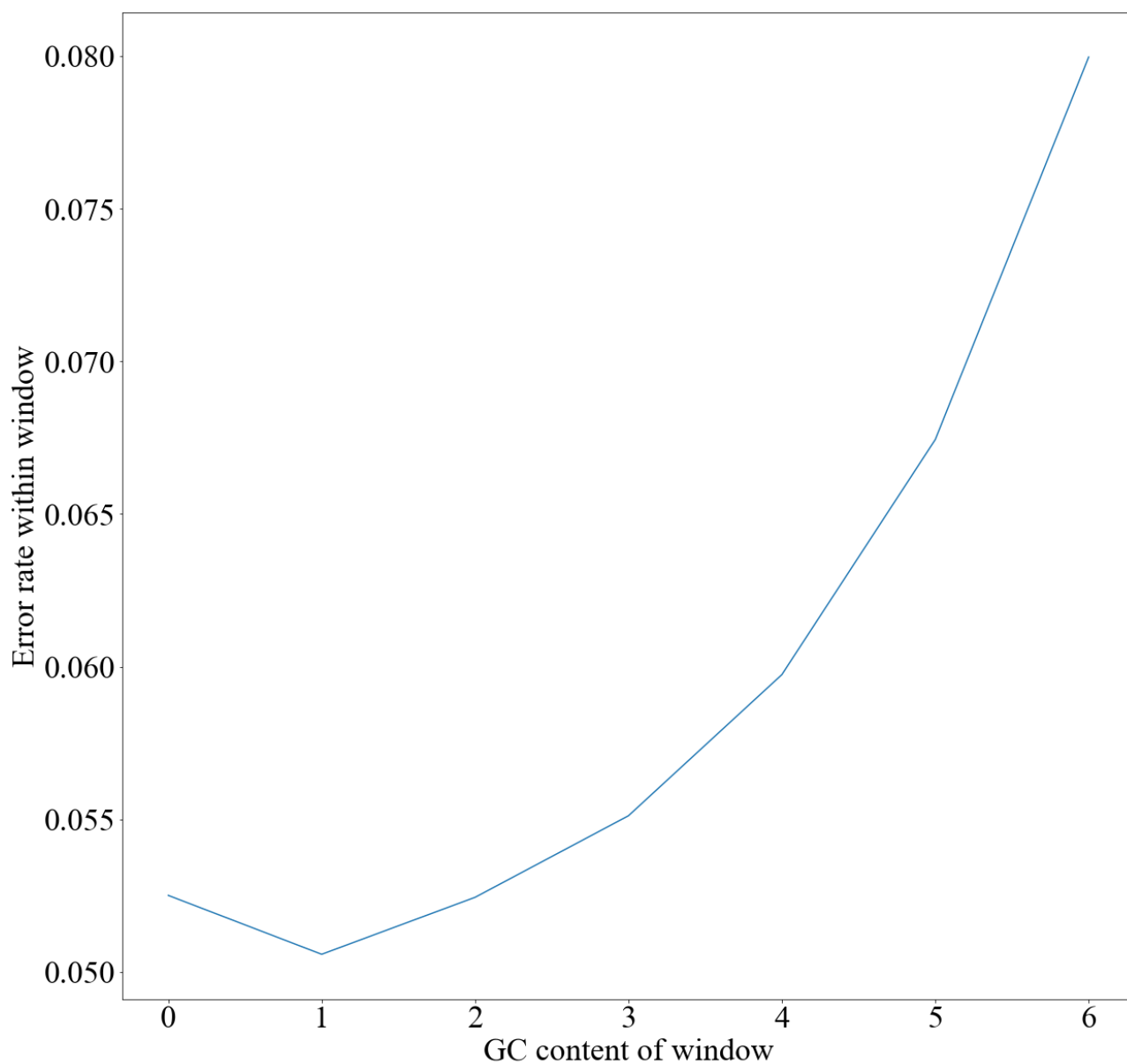
Supplementary Figure R-10: Distribution of GC content of different strings in the dataset

The average error rate within strings with different GC contents is shown in Supplementary Figure R-11. Only GC content values that occur within the dataset is depicted. As seen in Supplementary Figure R-10, there are not many occurrences of GC content values above 70 and below 40, therefore outliers may skew the results at the edge of the graph.



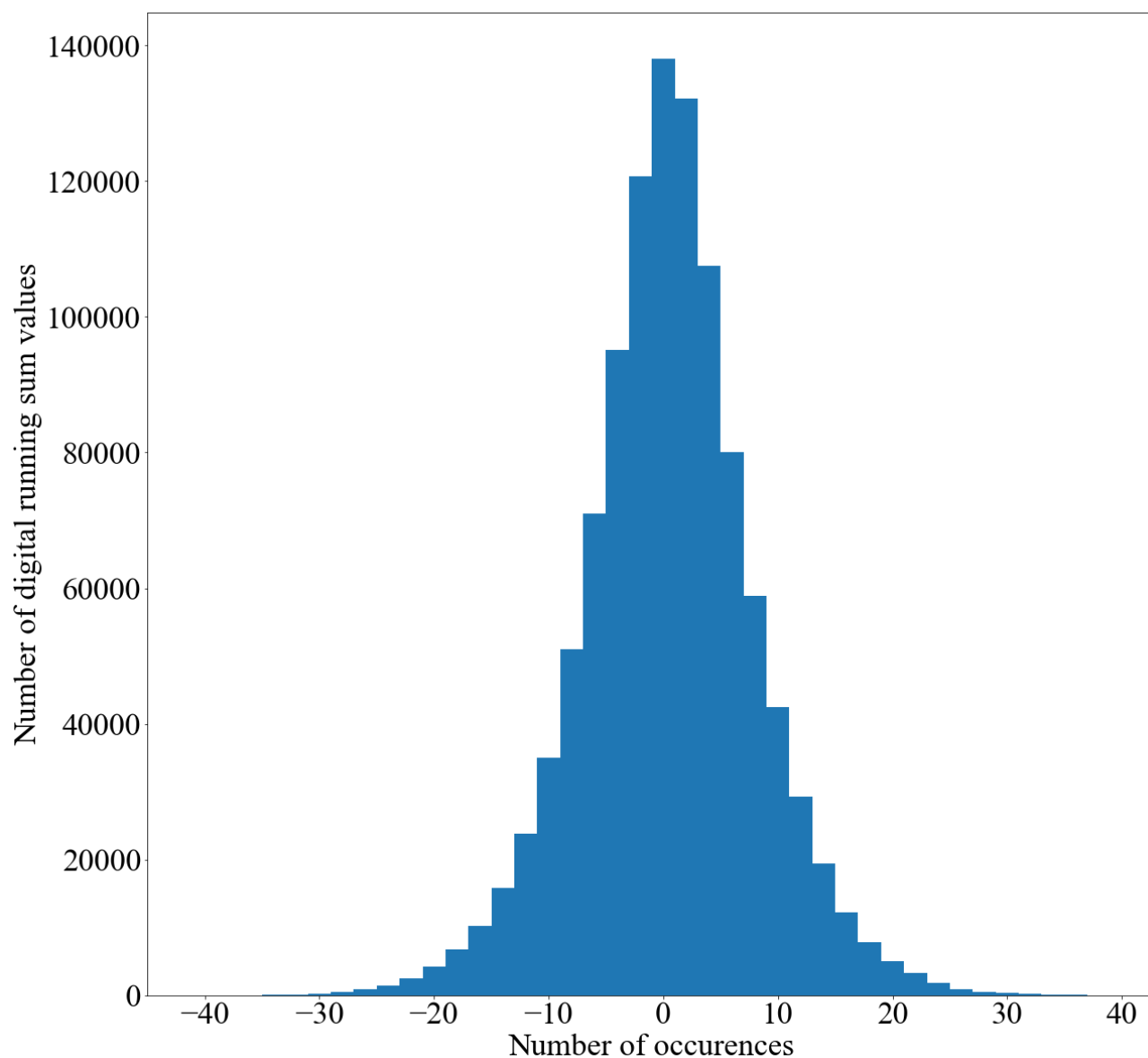
Supplementary Figure R-11: Comparison of error rates in strings of different GC contents in the dataset

The error rate of substrings with different GC contents is plotted in Supplementary Figure R-12. The x-axis represents the GC content of the substring, while the y-axis represents the average error rate for substrings with that GC content value. Substrings of length 6 were analysed.

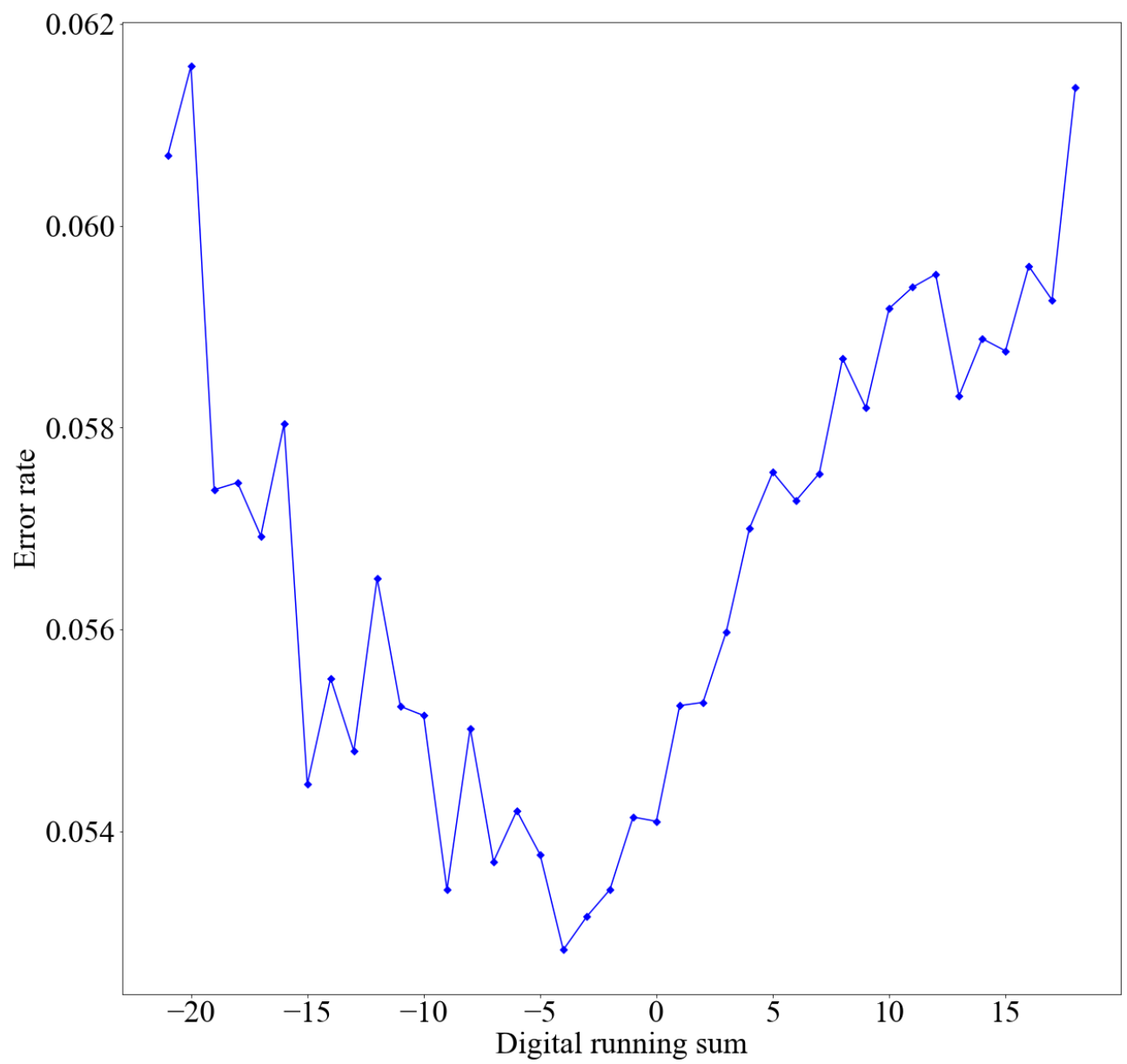


Supplementary Figure R-12: Comparison of GC content of windows and error rate for windows of length 6 in the dataset

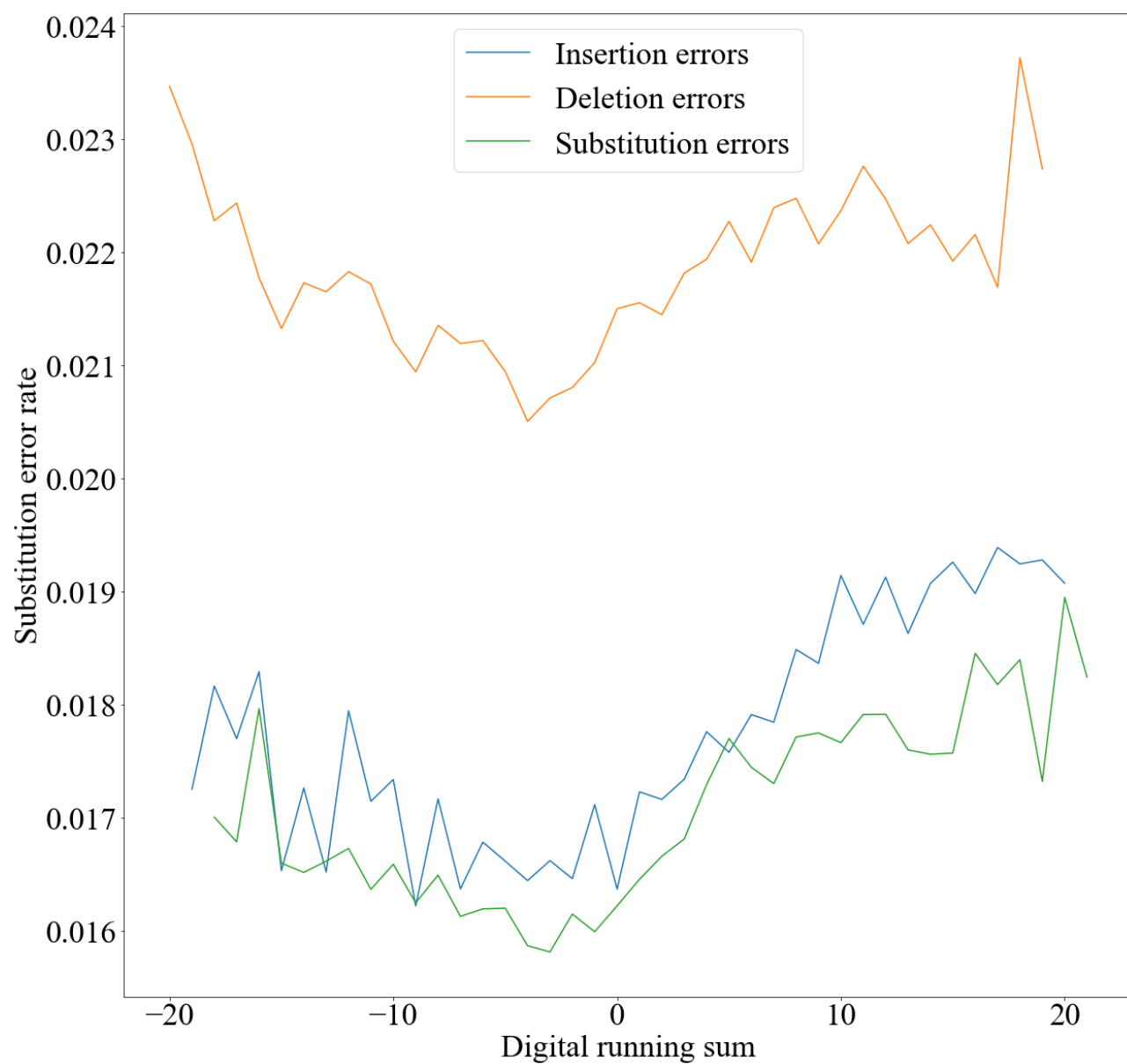
The distribution of number of occurrences of bases with different RDS values is plotted in Supplementary Figure R-13. The x-axis represents the RDS at a certain position, while the y-axis represents the number of occurrences.



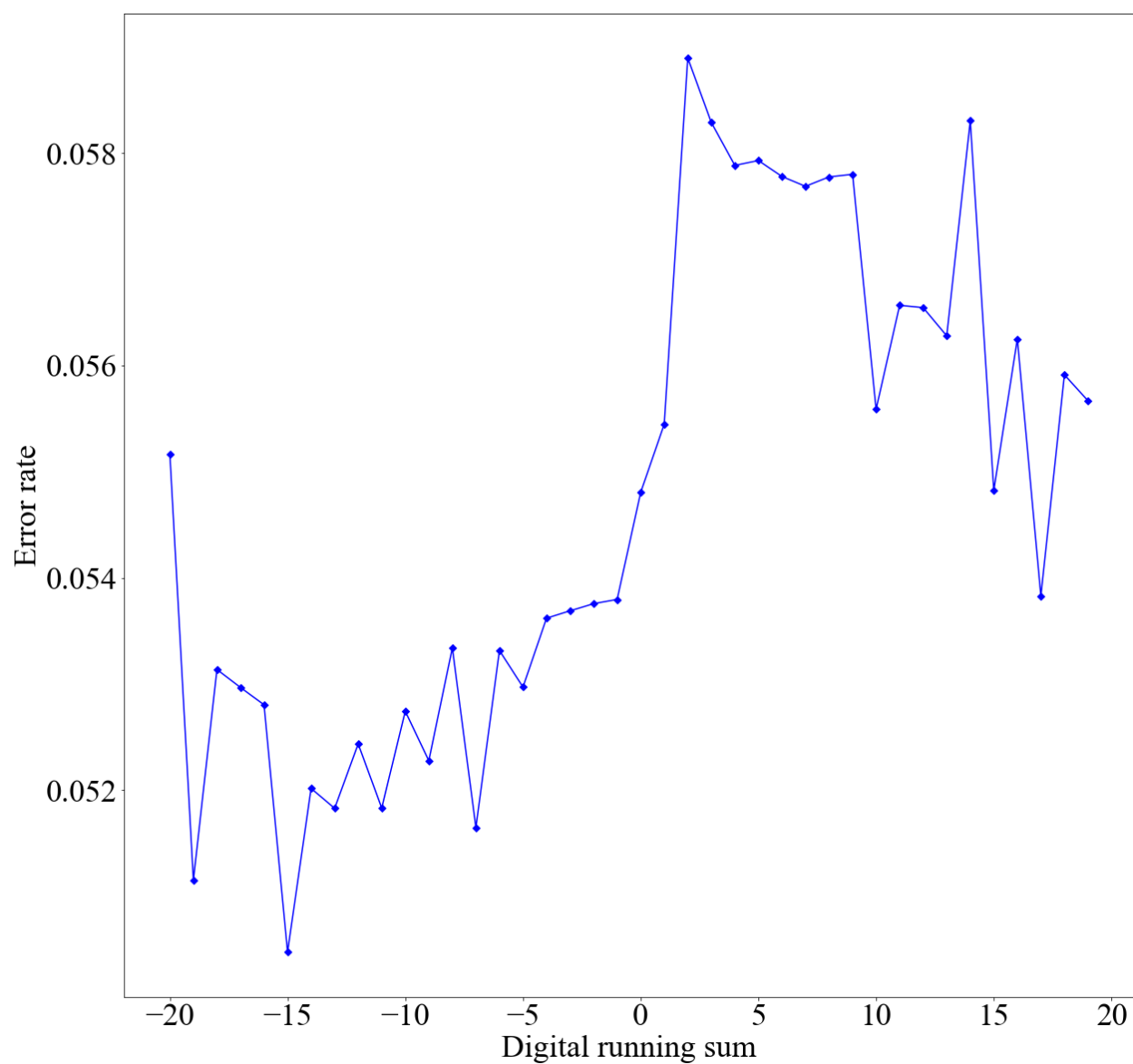
Supplementary Figure R-13: Distribution of different RDS values in the dataset



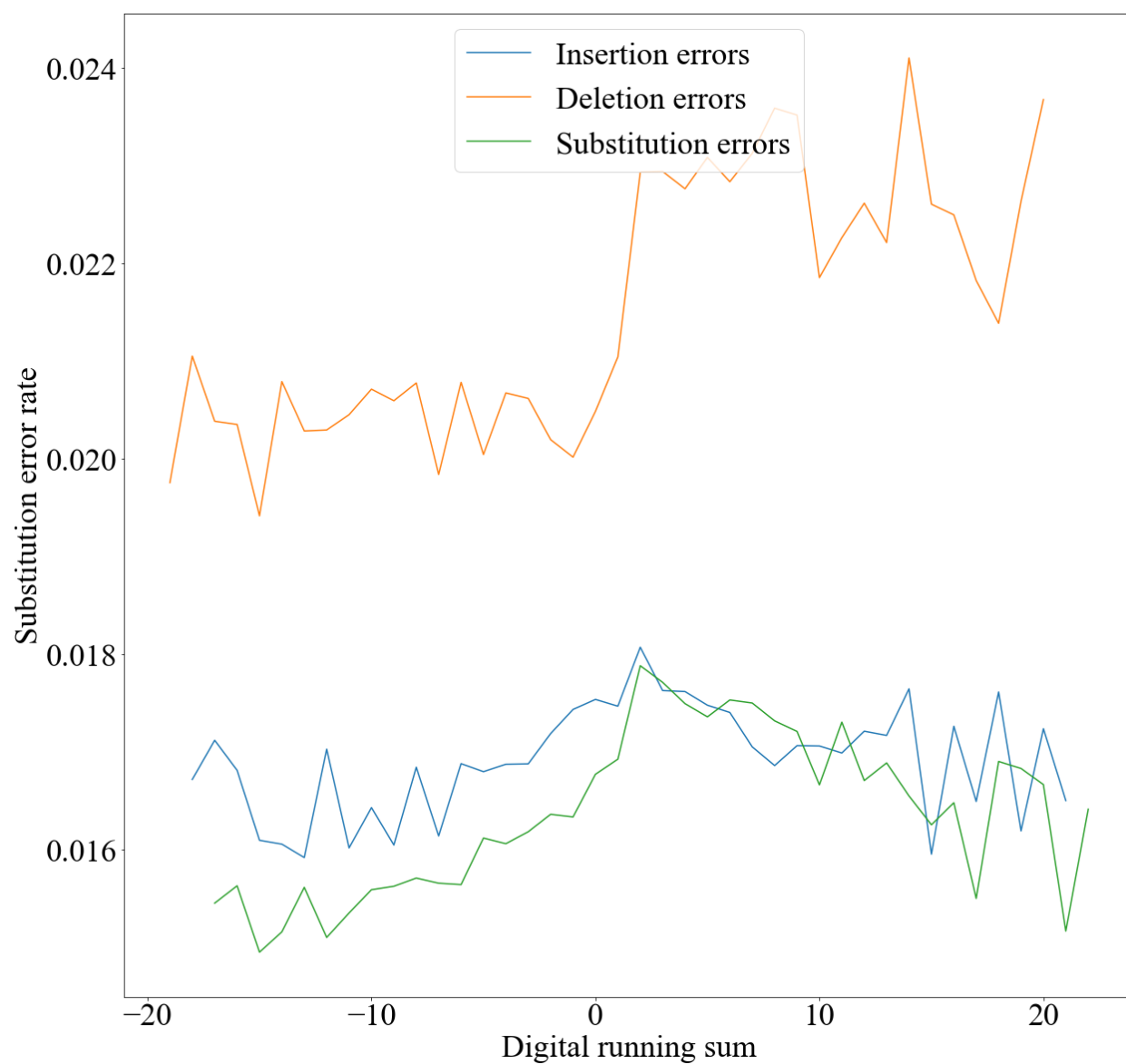
Supplementary Figure R-14: Comparison of RDS values and error rates for the dataset



Supplementary Figure R-15: Comparison of RDS values and error rates of different types for the dataset



Supplementary Figure R-16: Comparison of RDS values calculated from the back of the string and error rates for the dataset



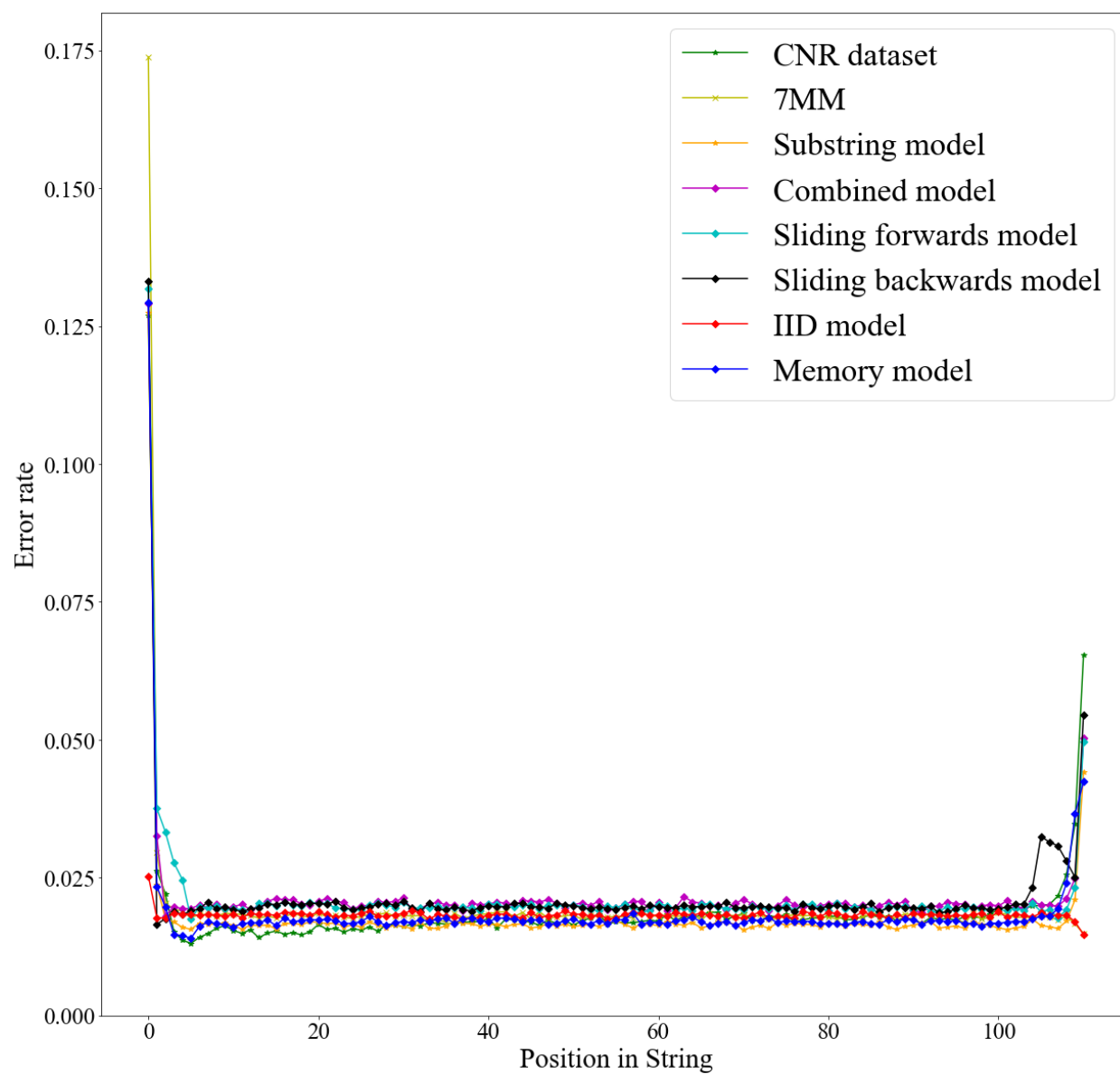
Supplementary Figure R-17: Comparison of RDS values calculated from the back of the string and error rates of different types for the dataset

APPENDIX S GENERAL STATISTICS OF DIFFERENT MODELS

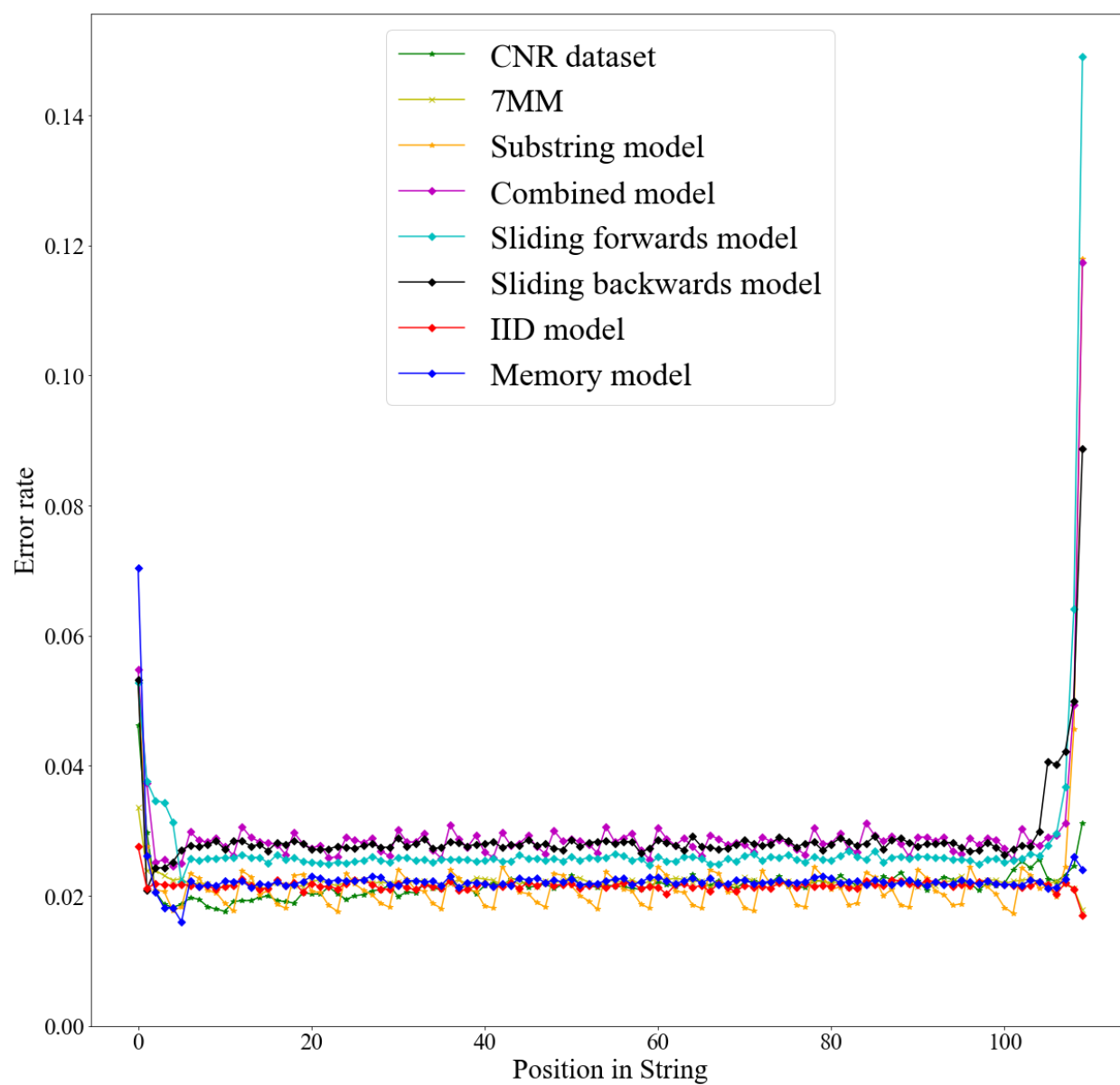
Supplementary Table S-1: Distribution of error types produced by each model

Model	Insertions	Deletions	Substitutions
CNR Dataset	33%	38%	29%
7MM	35%	39%	27%
Substring model	31%	39%	30%
Combined model	31%	41%	28%
Sliding forwards model	31%	41%	28%
Sliding backwards model	30%	42%	29%
IID model	32%	37%	31%
Memory model	33%	40%	27%

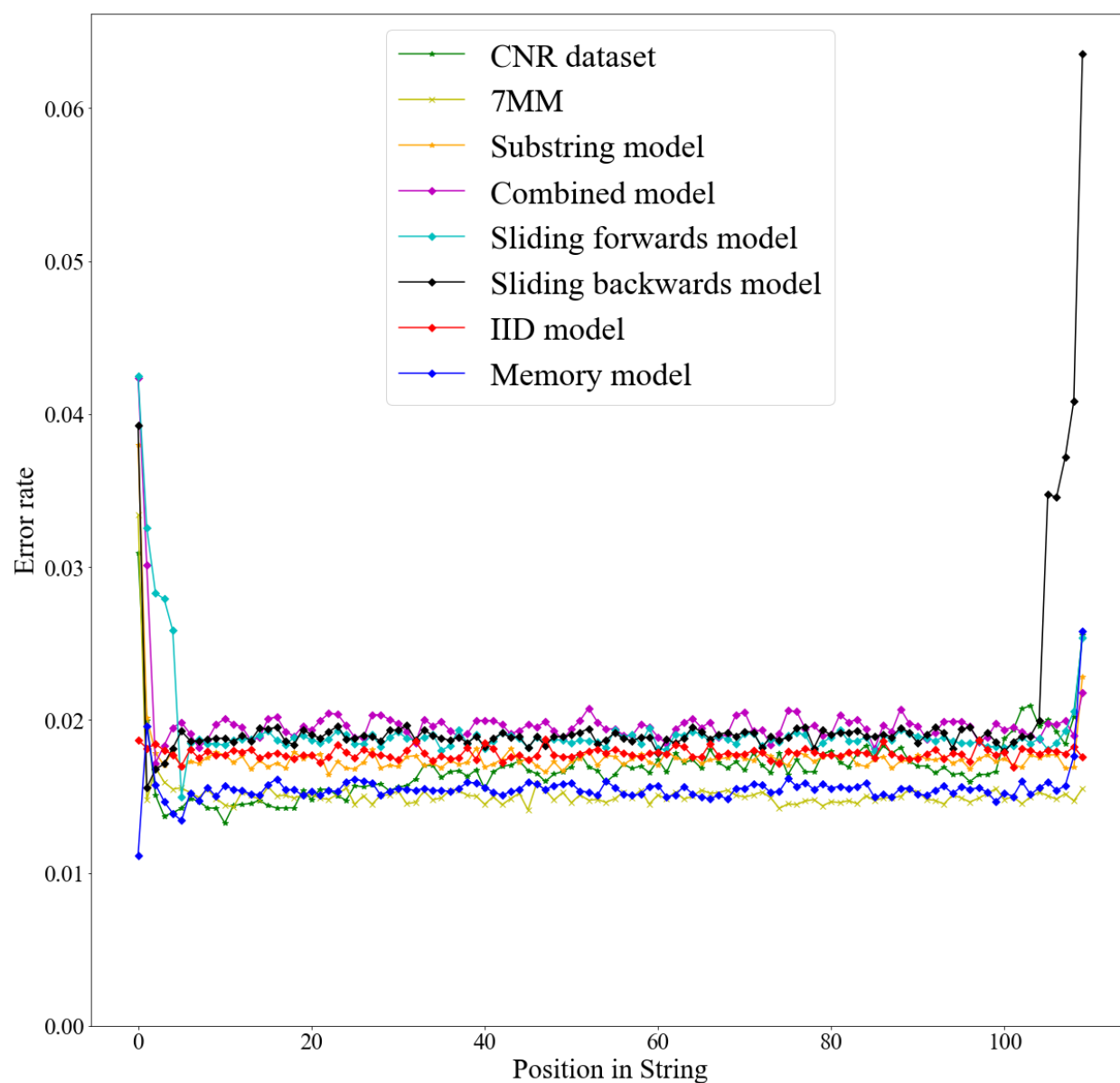
APPENDIX T ERROR RATES PRODUCED AT DIFFERENT POSITIONS



Supplementary Figure T-1: Comparison of insertion rates produced at each position by each model

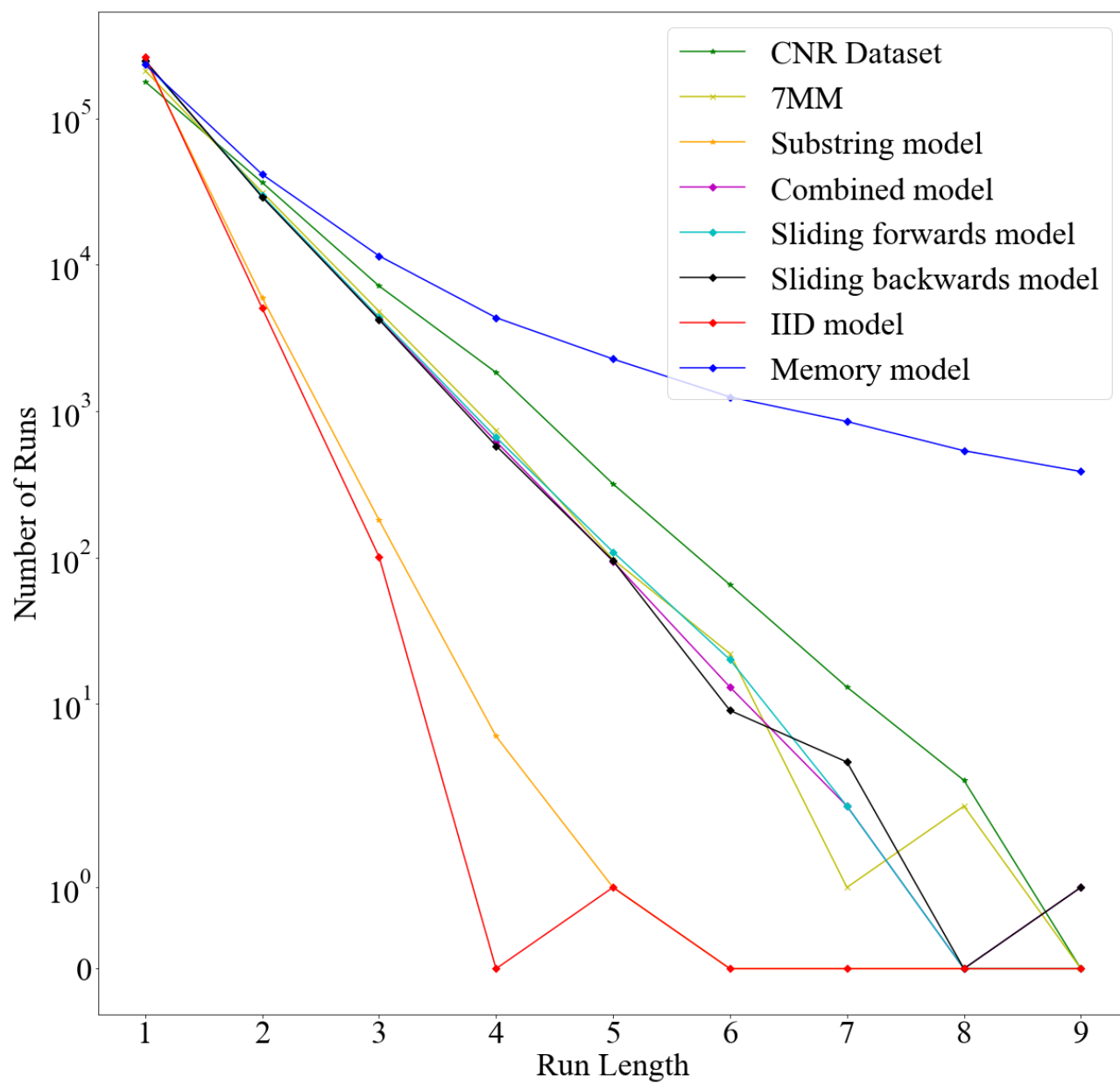


Supplementary Figure T-2: Comparison of deletion rates produced at each position by each model

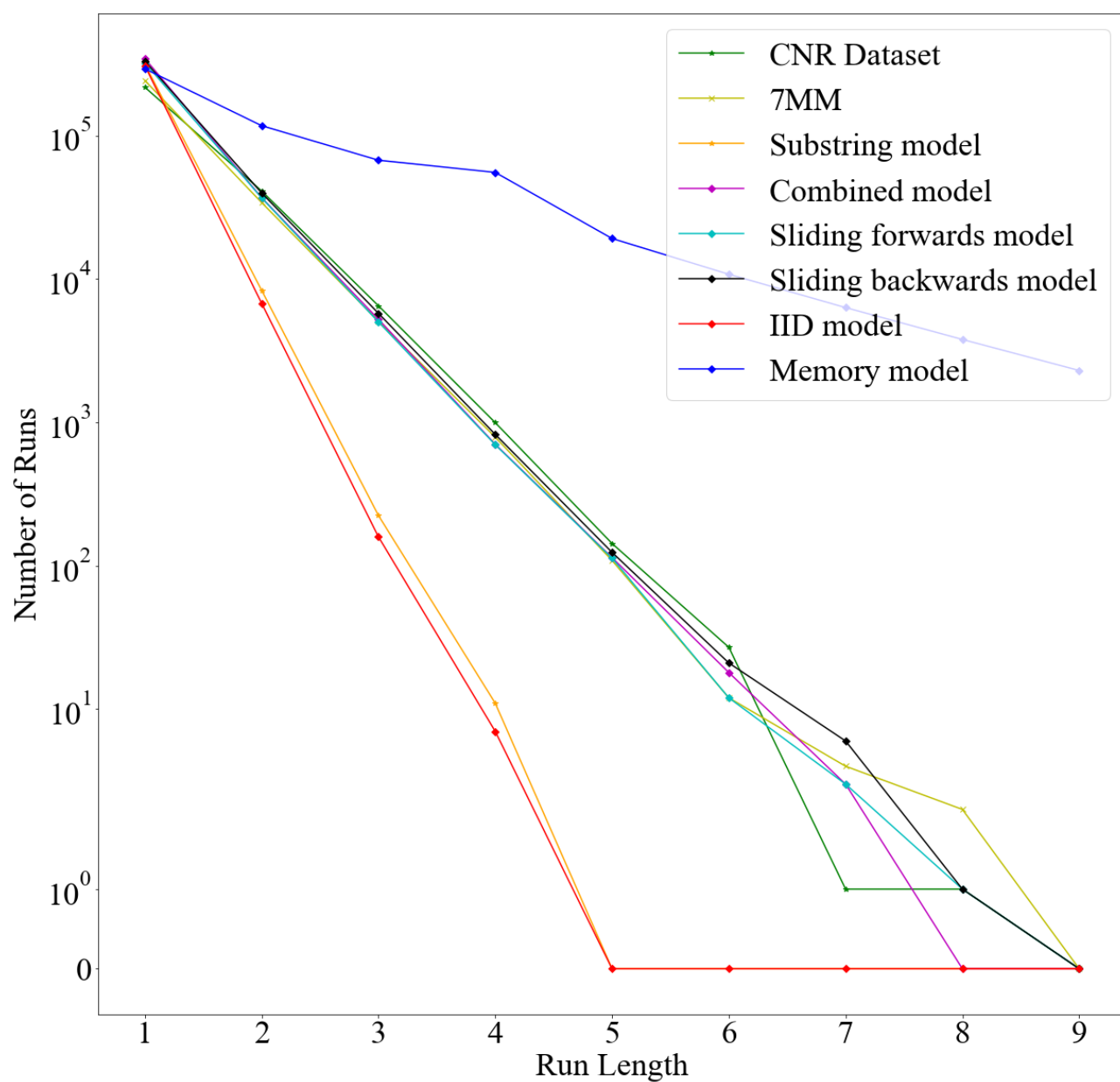


Supplementary Figure T-3: Comparison of substitution rates produced at each position by each model

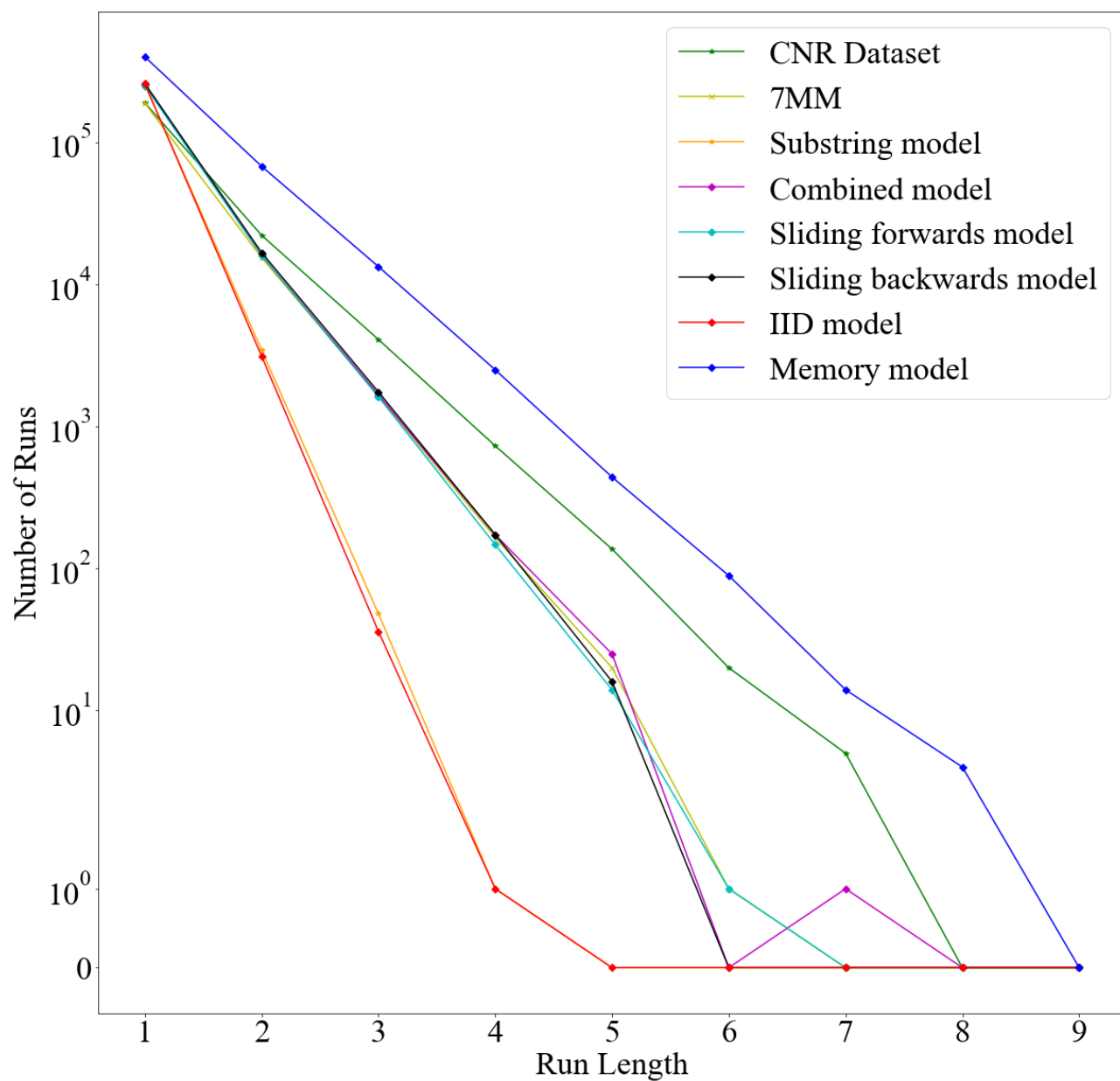
APPENDIX U ANALYSIS OF ERROR RUNS



Supplementary Figure U-1: Comparison of insertion error runs produced by each model

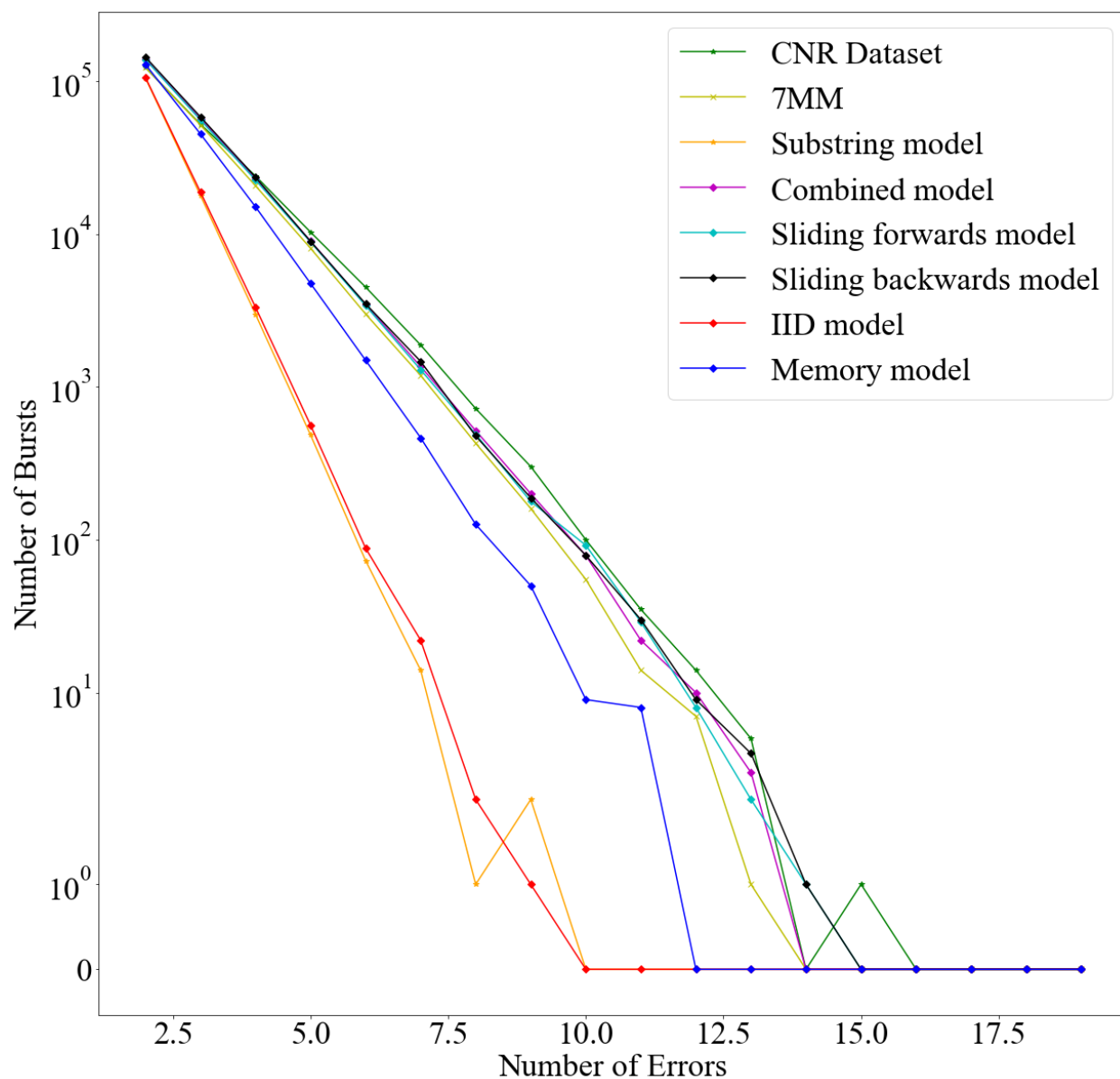


Supplementary Figure U-2: Comparison of deletion error runs produced by each model

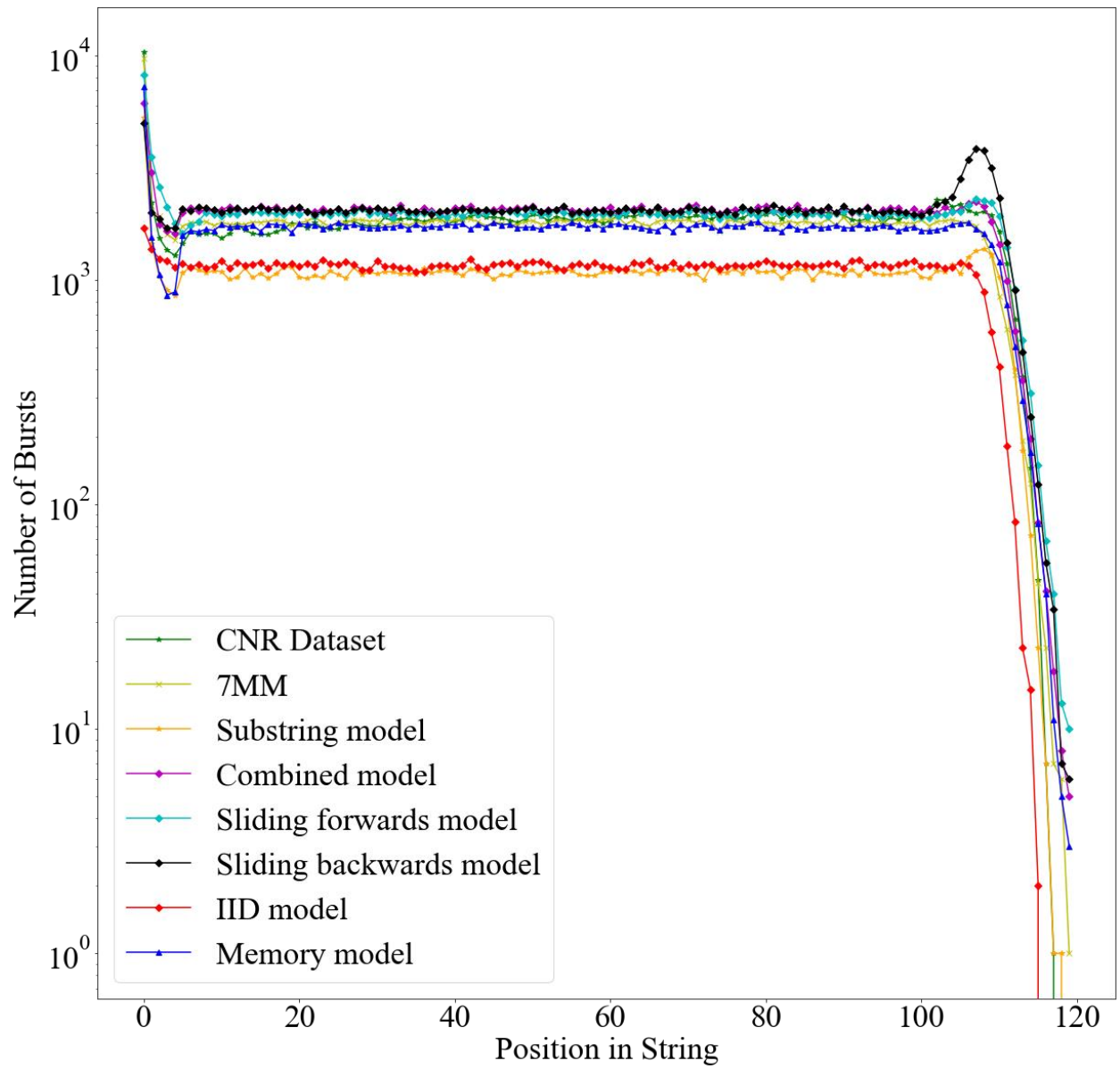


Supplementary Figure U-3: Comparison of substitution error runs produced by each model

APPENDIX V ANALYSIS OF ERROR BURSTS

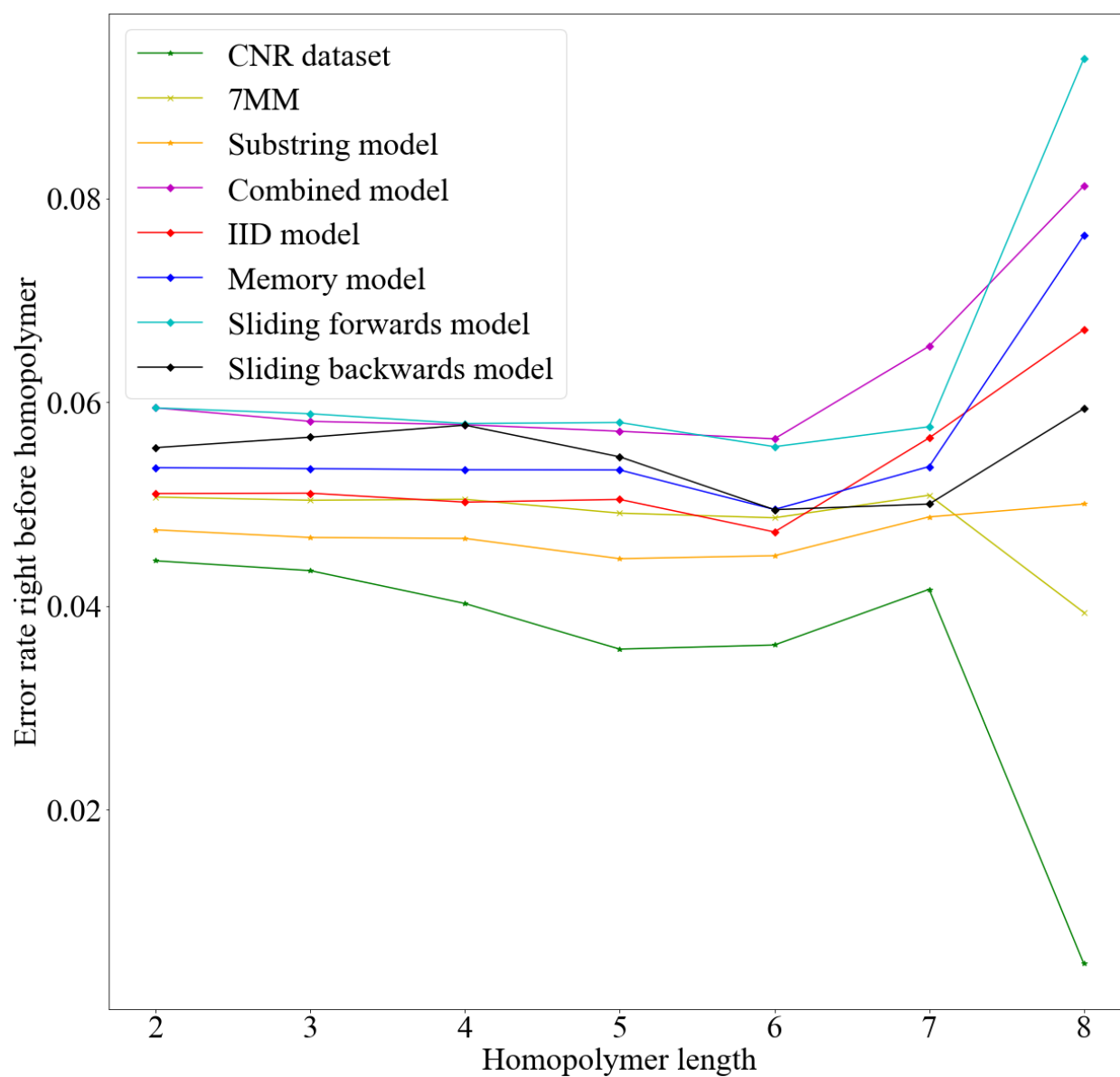


Supplementary Figure V-1: Comparison of errors per burst produced by each model

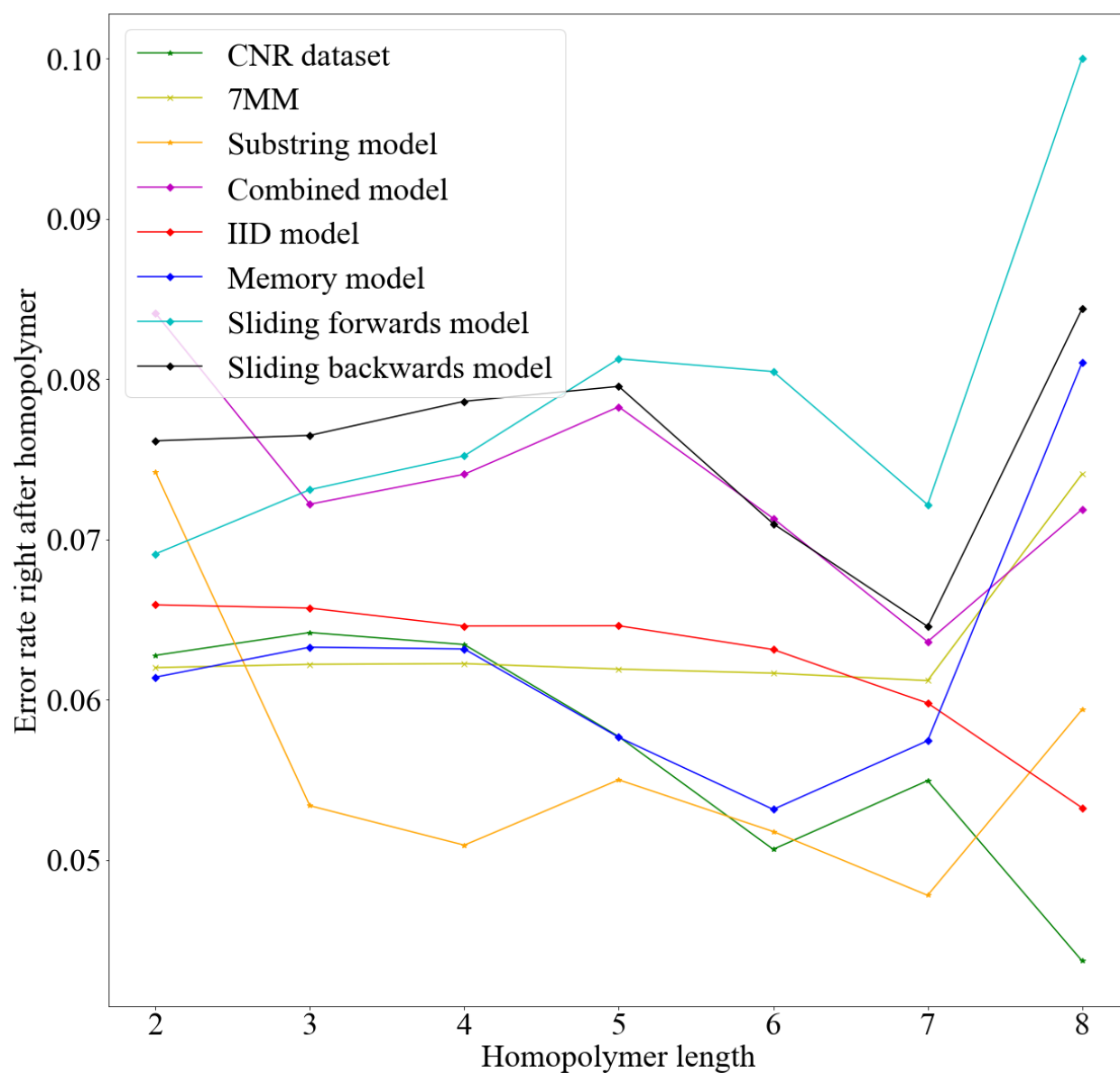


Supplementary Figure V-2: Start positions of bursts produced by each model

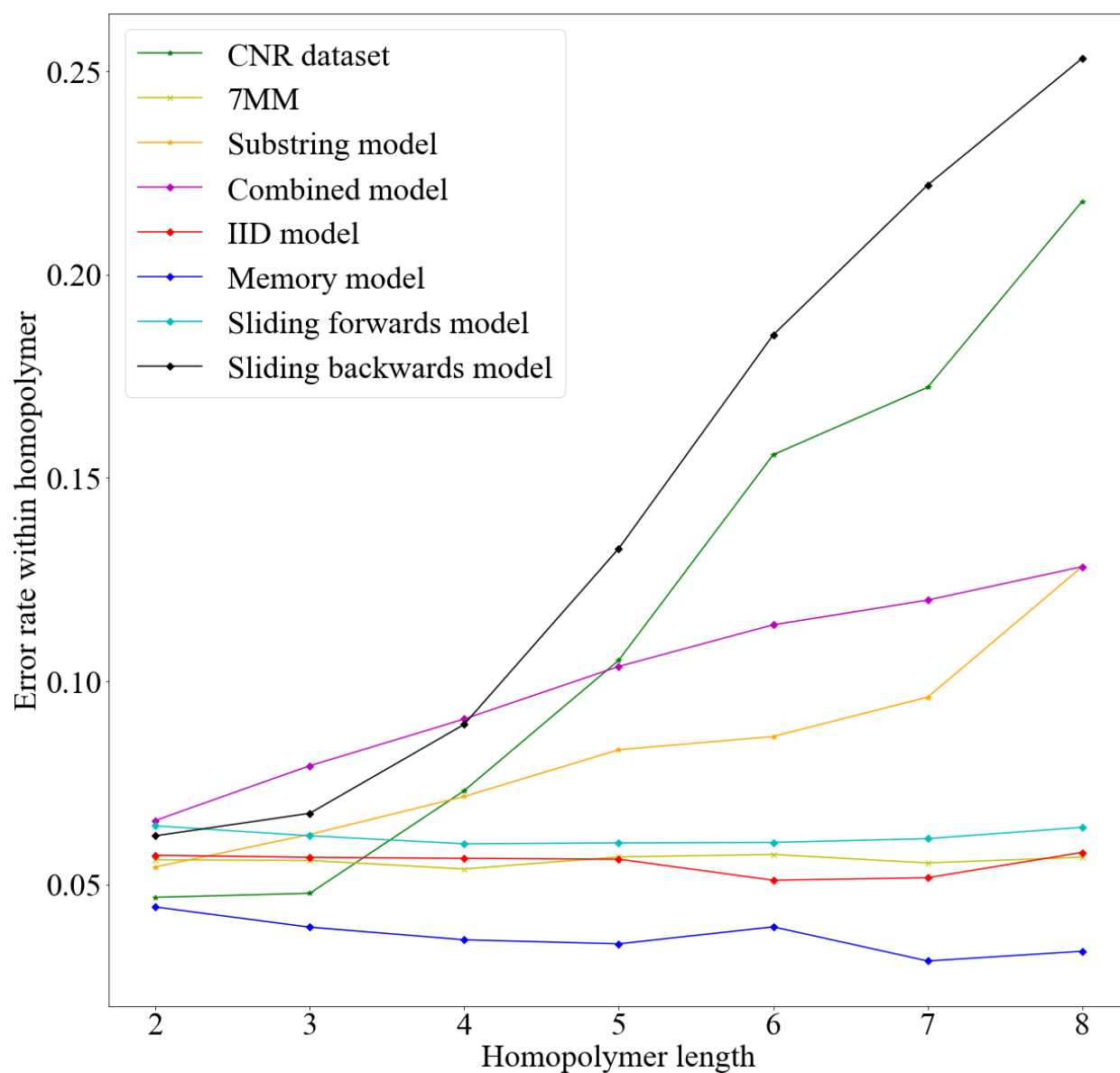
APPENDIX W ANALYSIS OF HOMOPOLYMERS



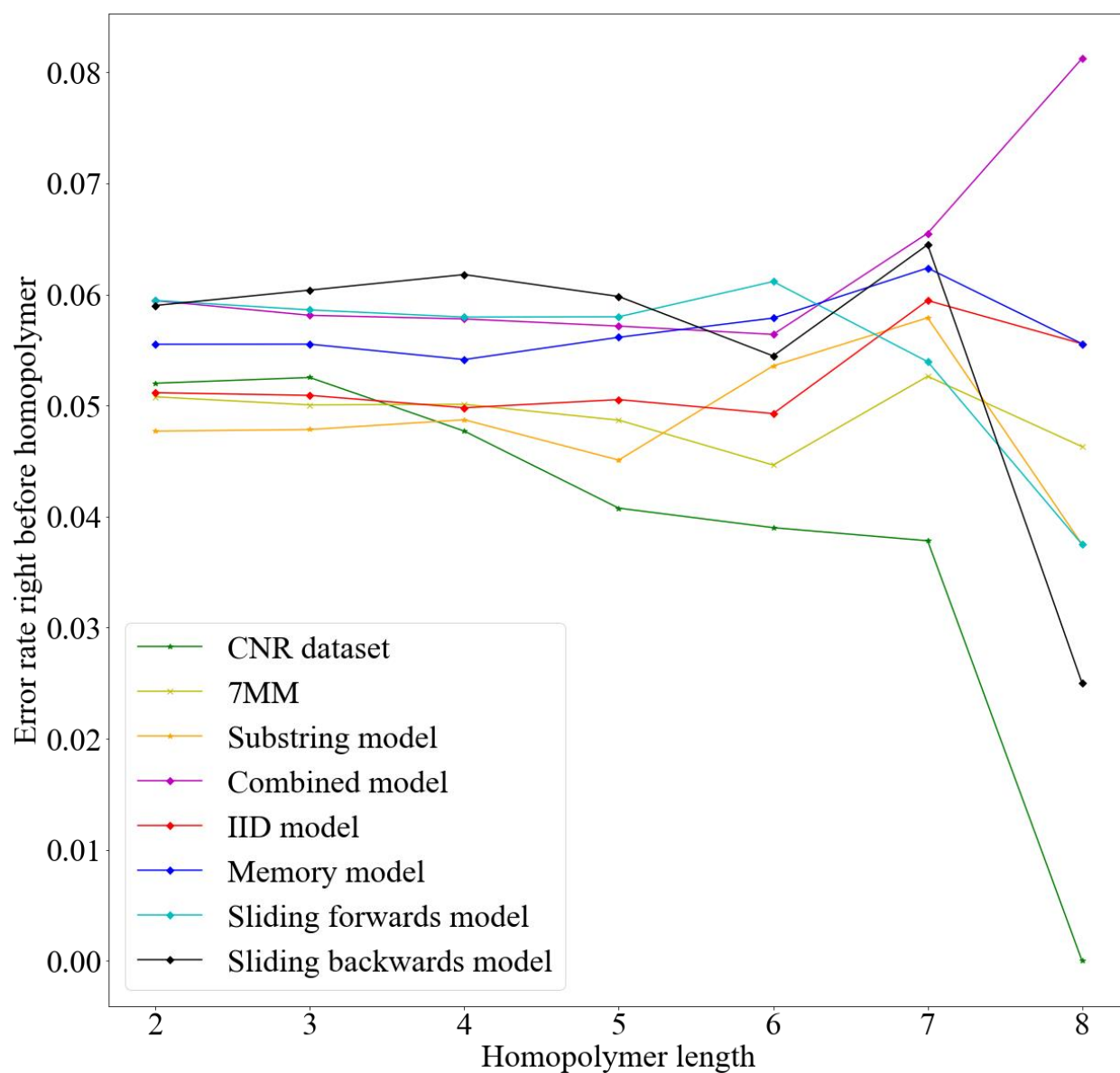
Supplementary Figure W-1: Comparison of error rates produced before homopolymers by each model



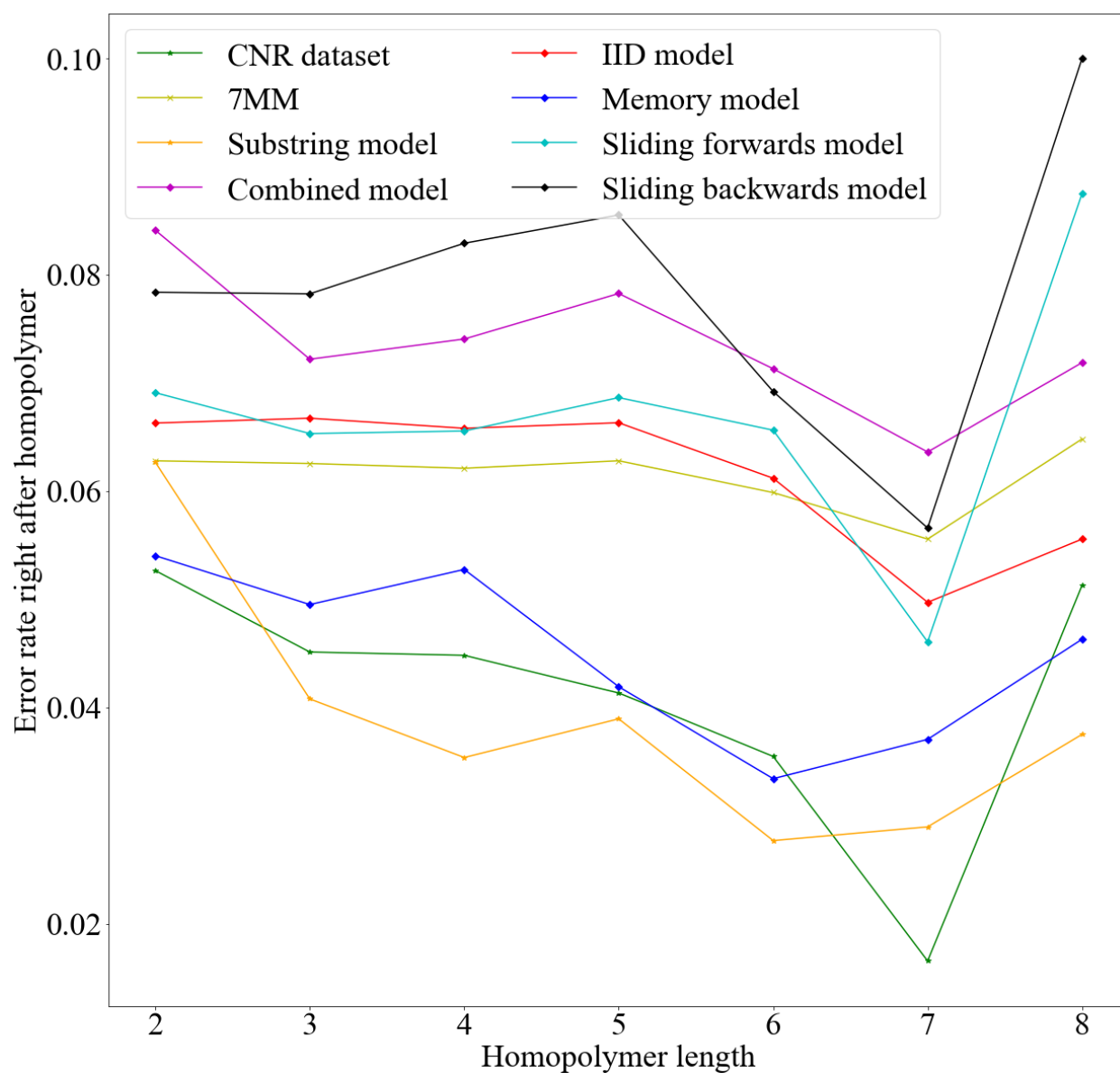
Supplementary Figure W-2: Comparison of error rates produced after homopolymers by each model



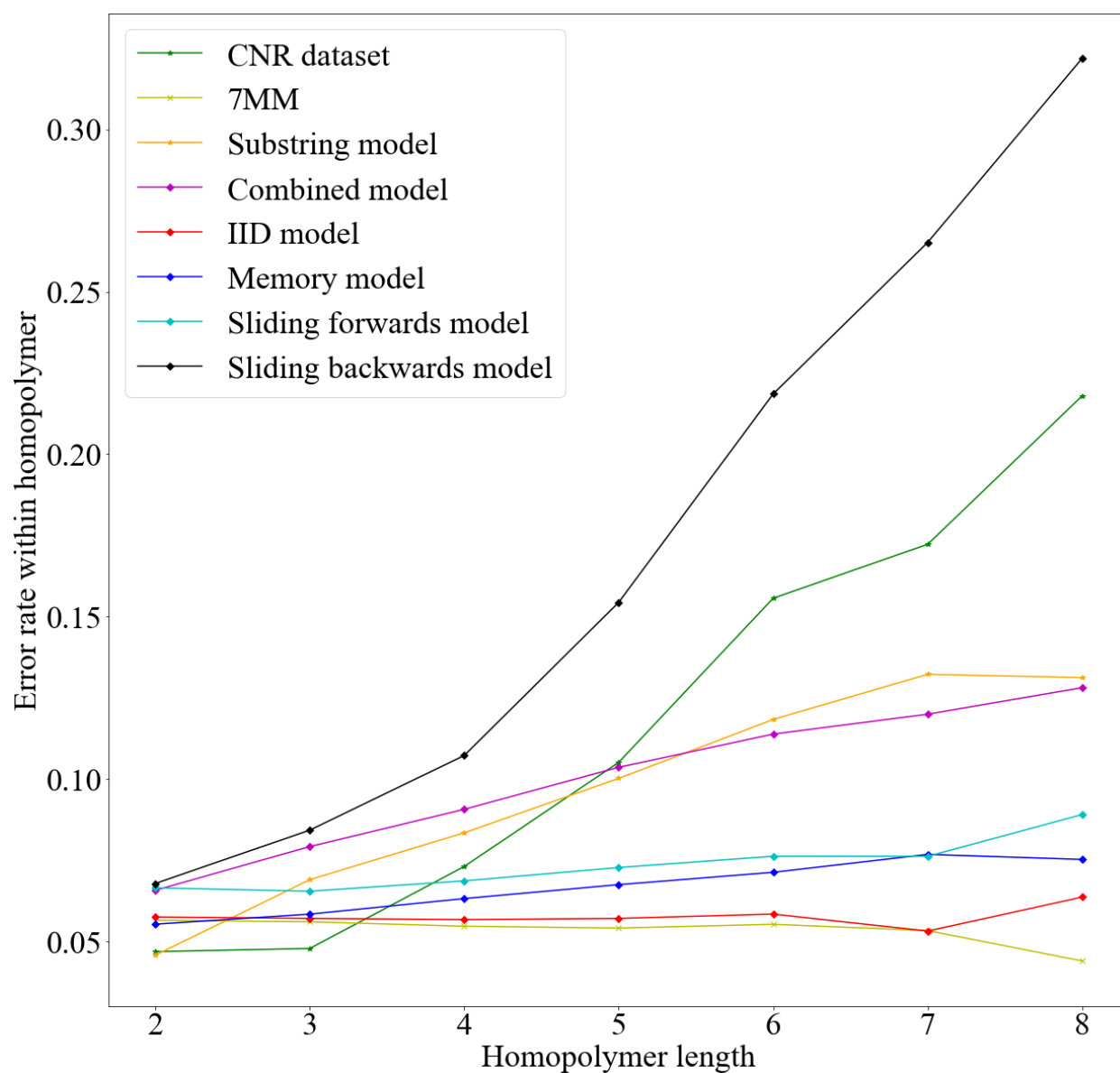
Supplementary Figure W-3: Comparison of error rates produced within homopolymers of A by each model



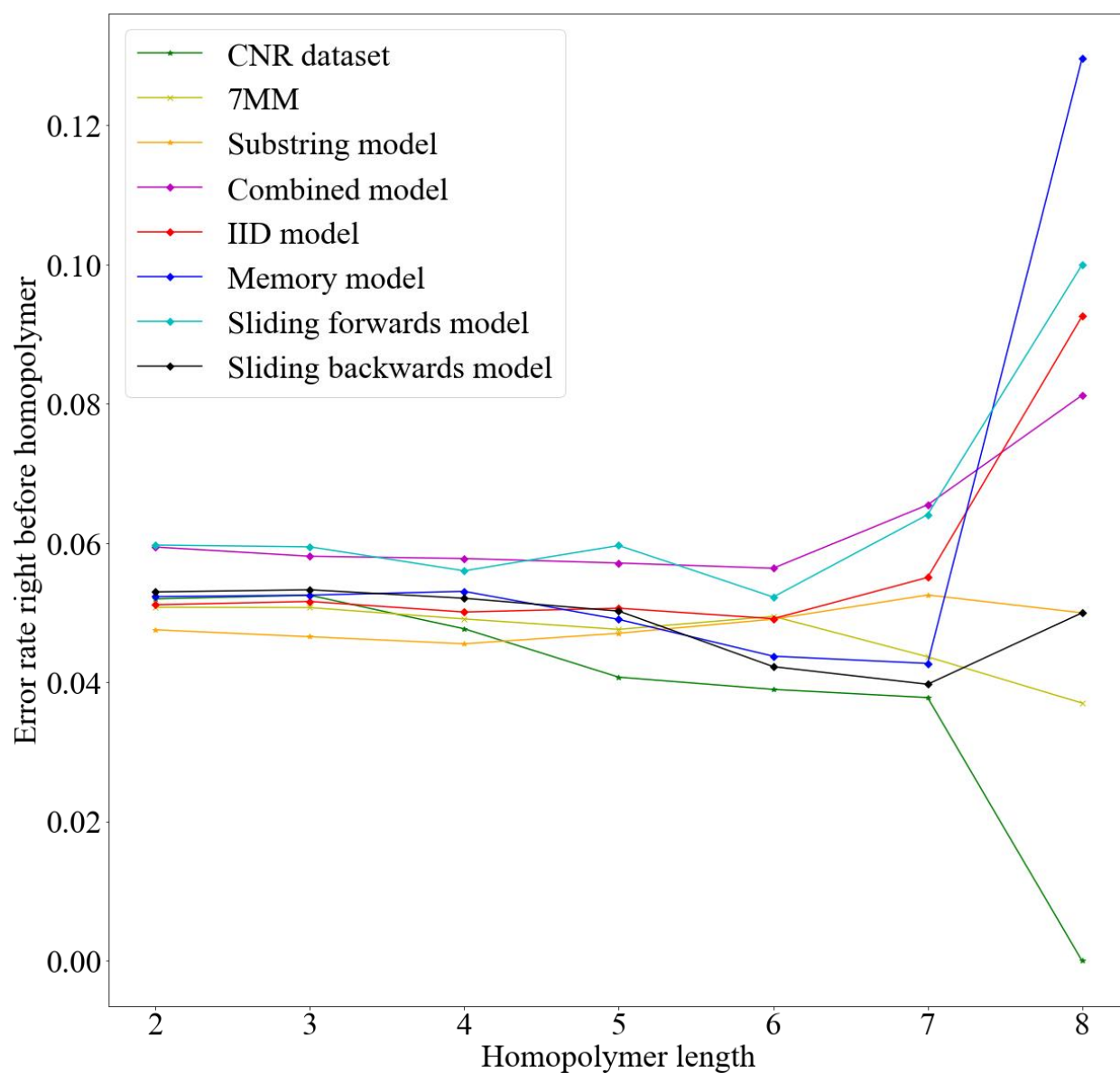
Supplementary Figure W-4: Comparison of error rates produced before homopolymers of A by each model



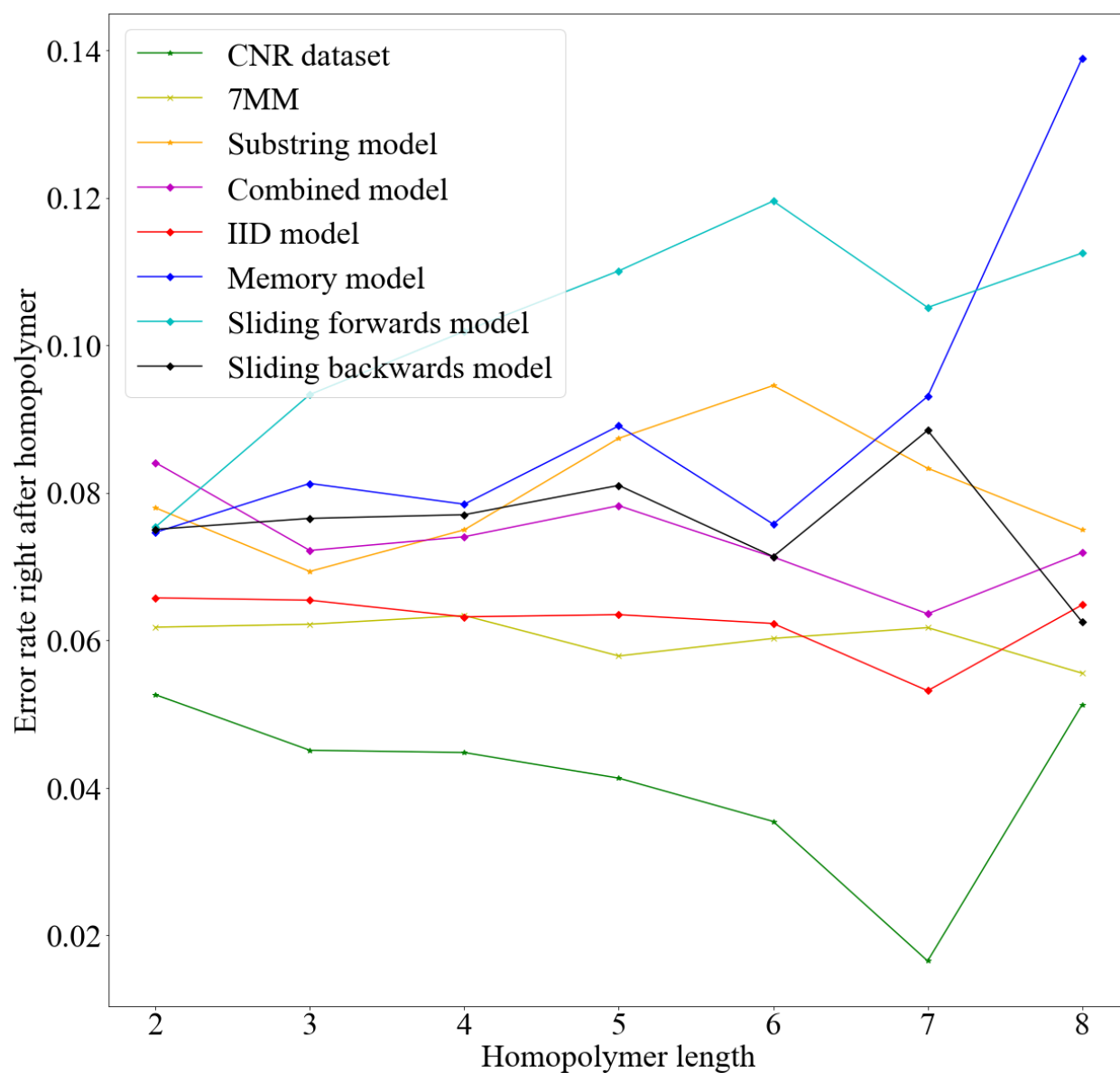
Supplementary Figure W-5: Comparison of error rates produced after homopolymers of A by each model



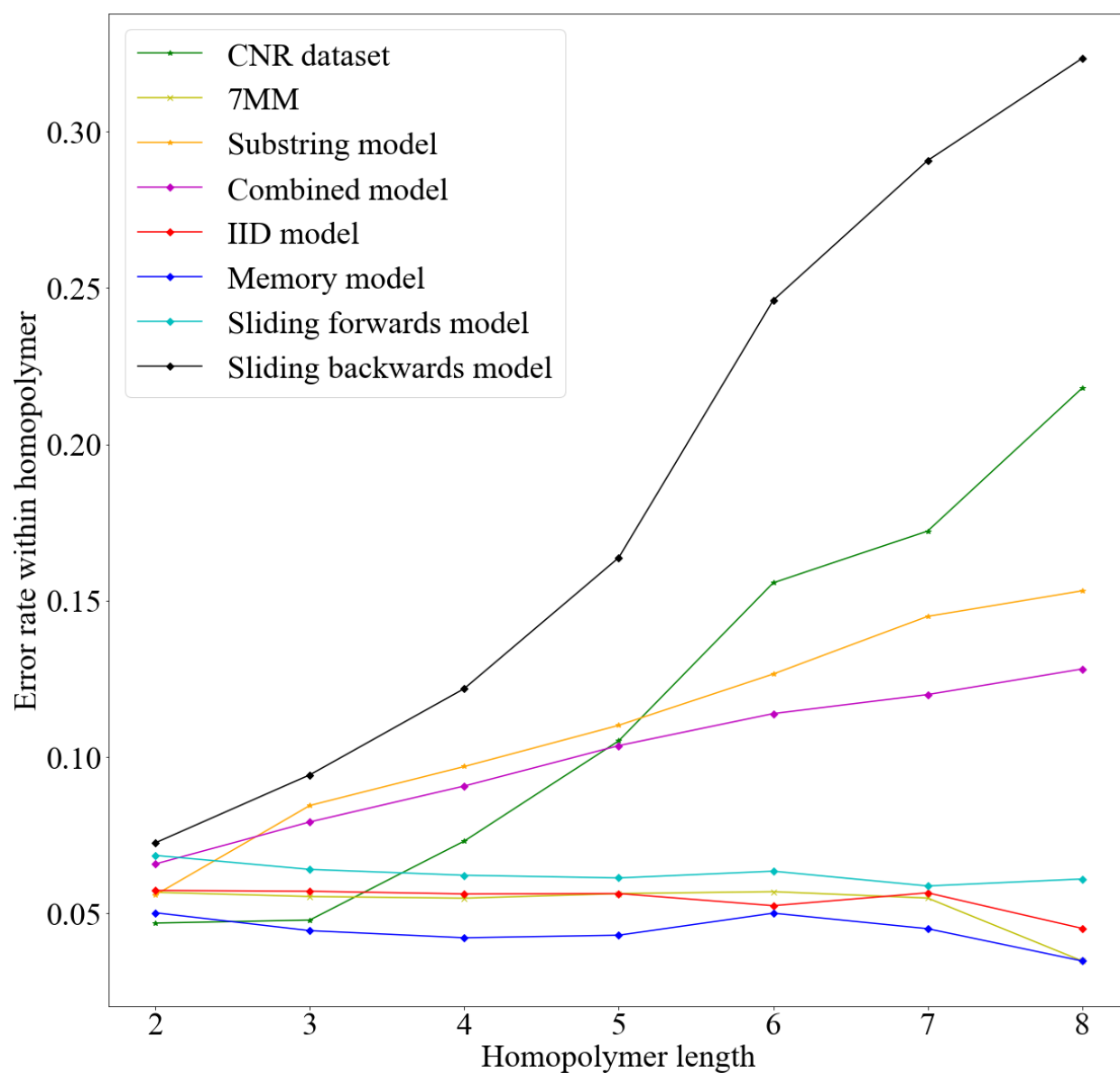
Supplementary Figure W-6: Comparison of error rates produced within homopolymers of C by each model



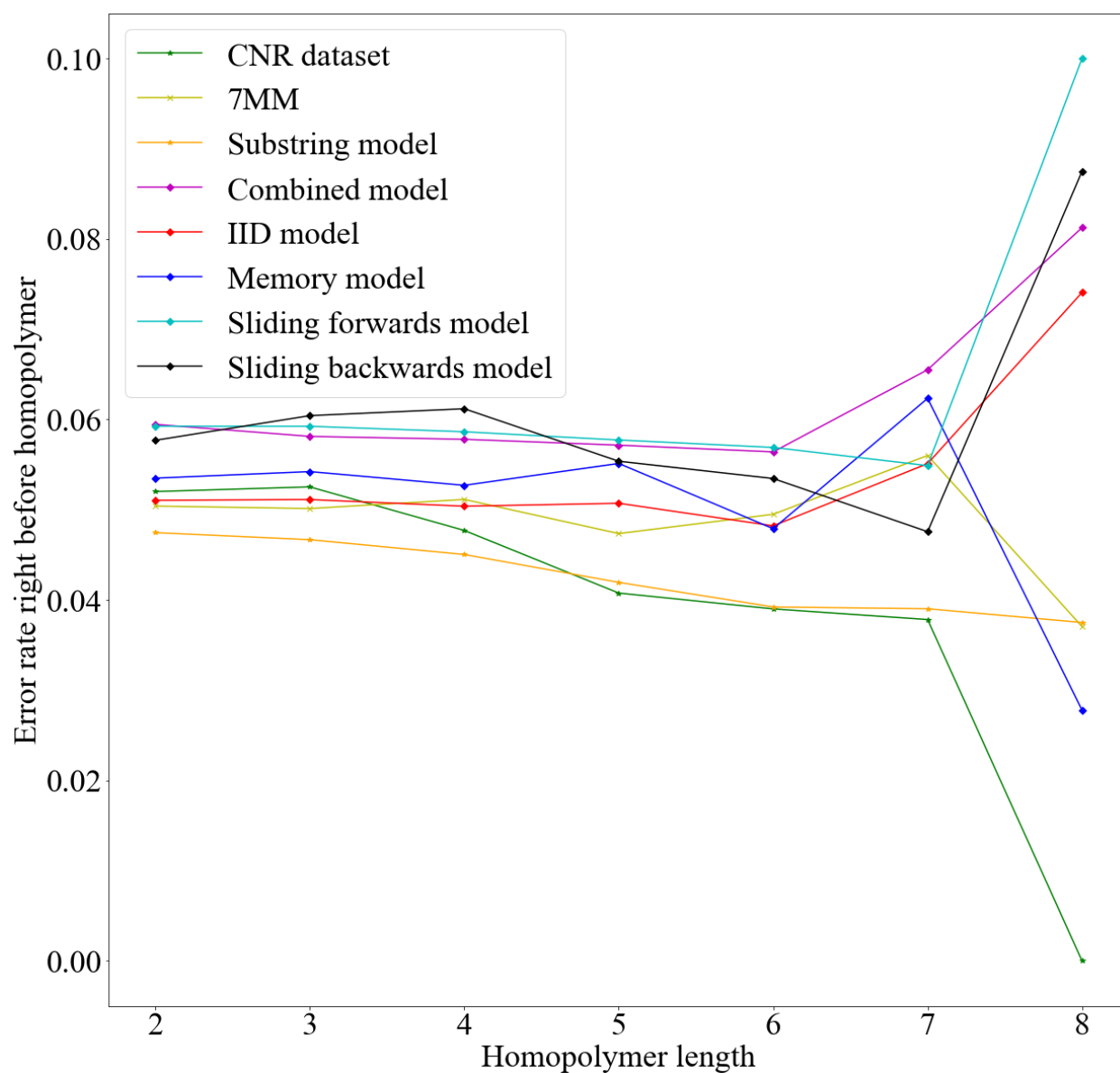
Supplementary Figure W-7: Comparison of error rates produced before homopolymers of C by each model



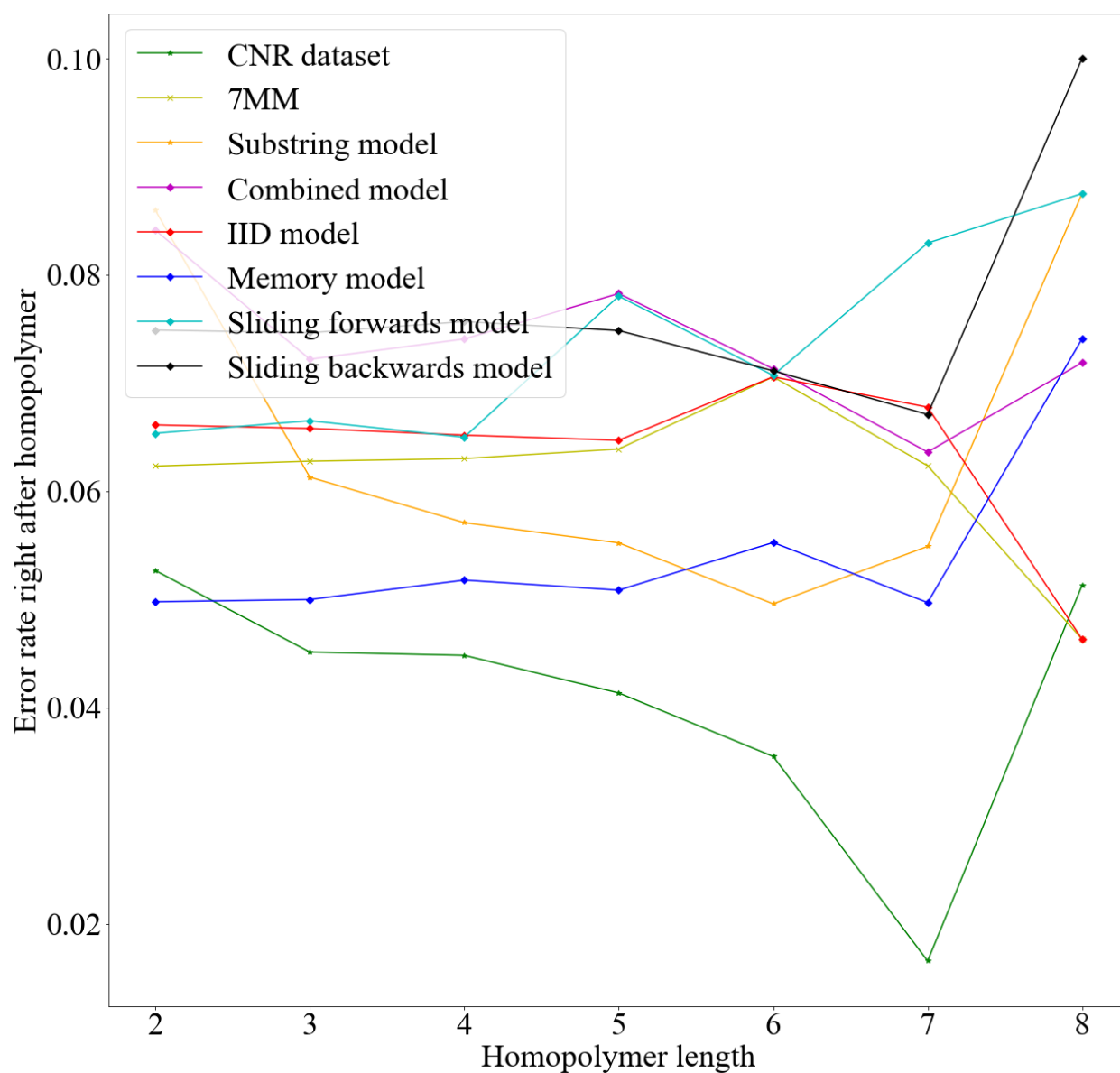
Supplementary Figure W-8: Comparison of error rates produced after homopolymers of C by each model



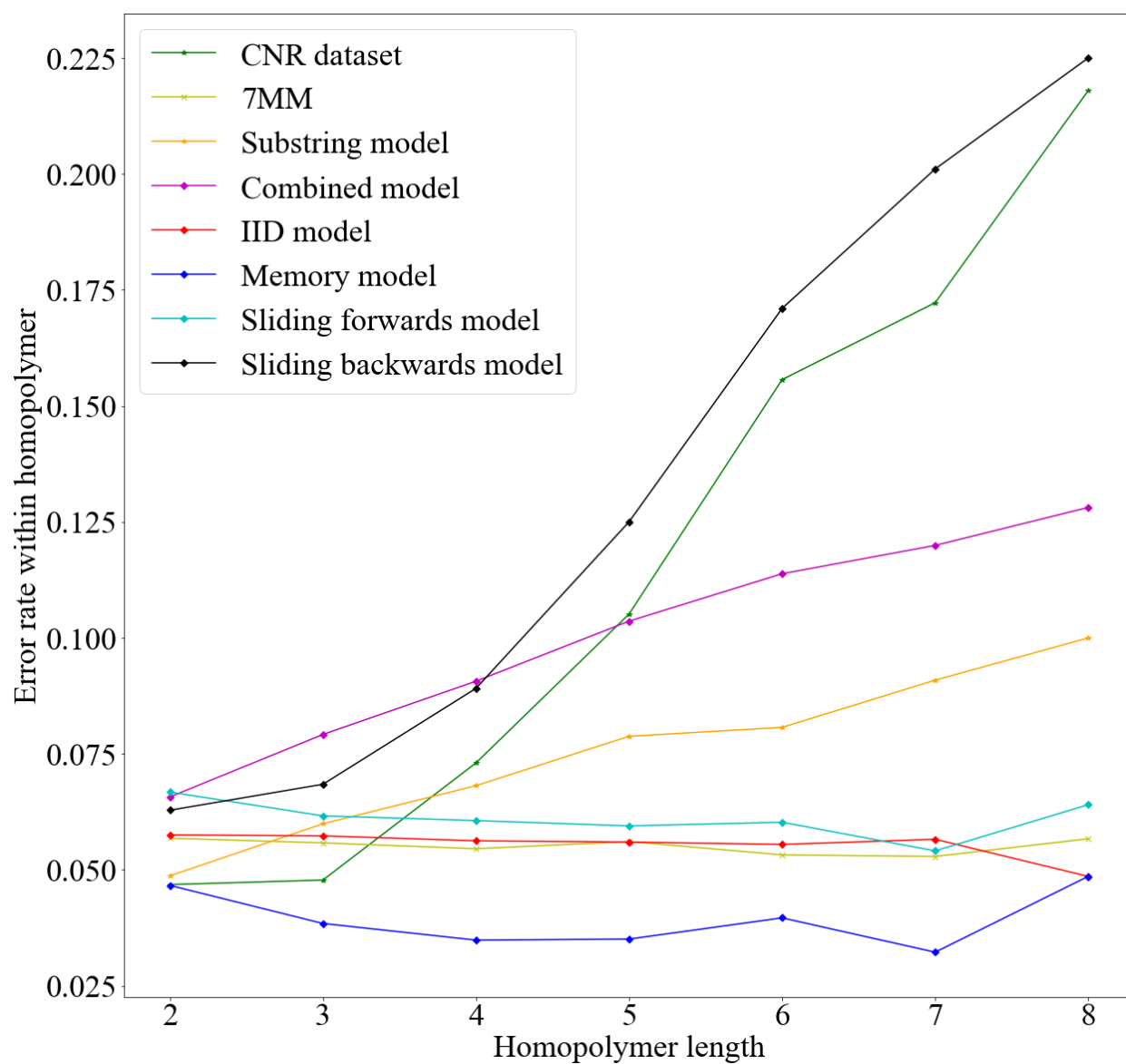
Supplementary Figure W-9: Comparison of error rated produced within homopolymers of G by each model



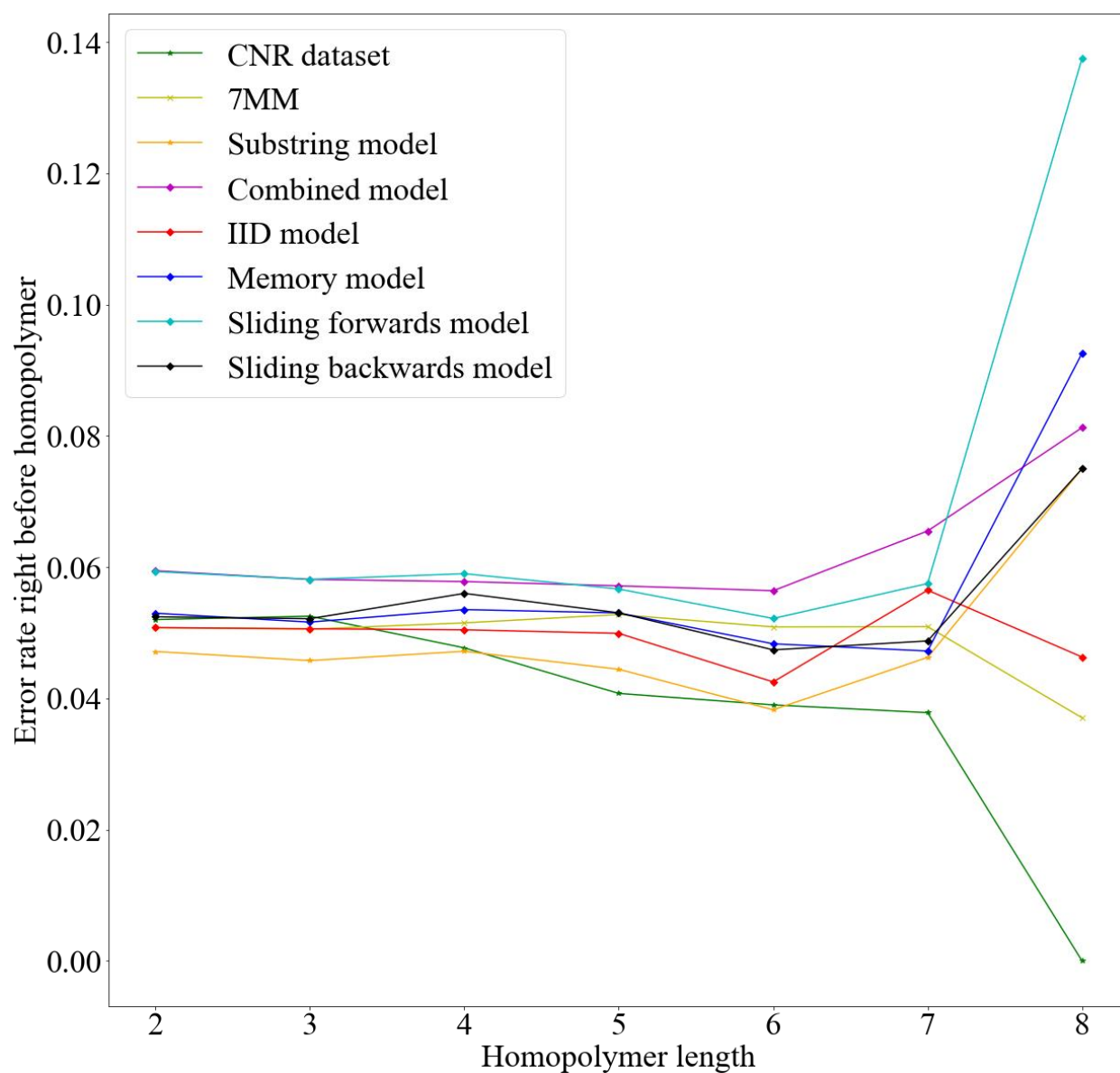
Supplementary Figure W-10: Comparison of error rates produced before homopolymers of G by each model



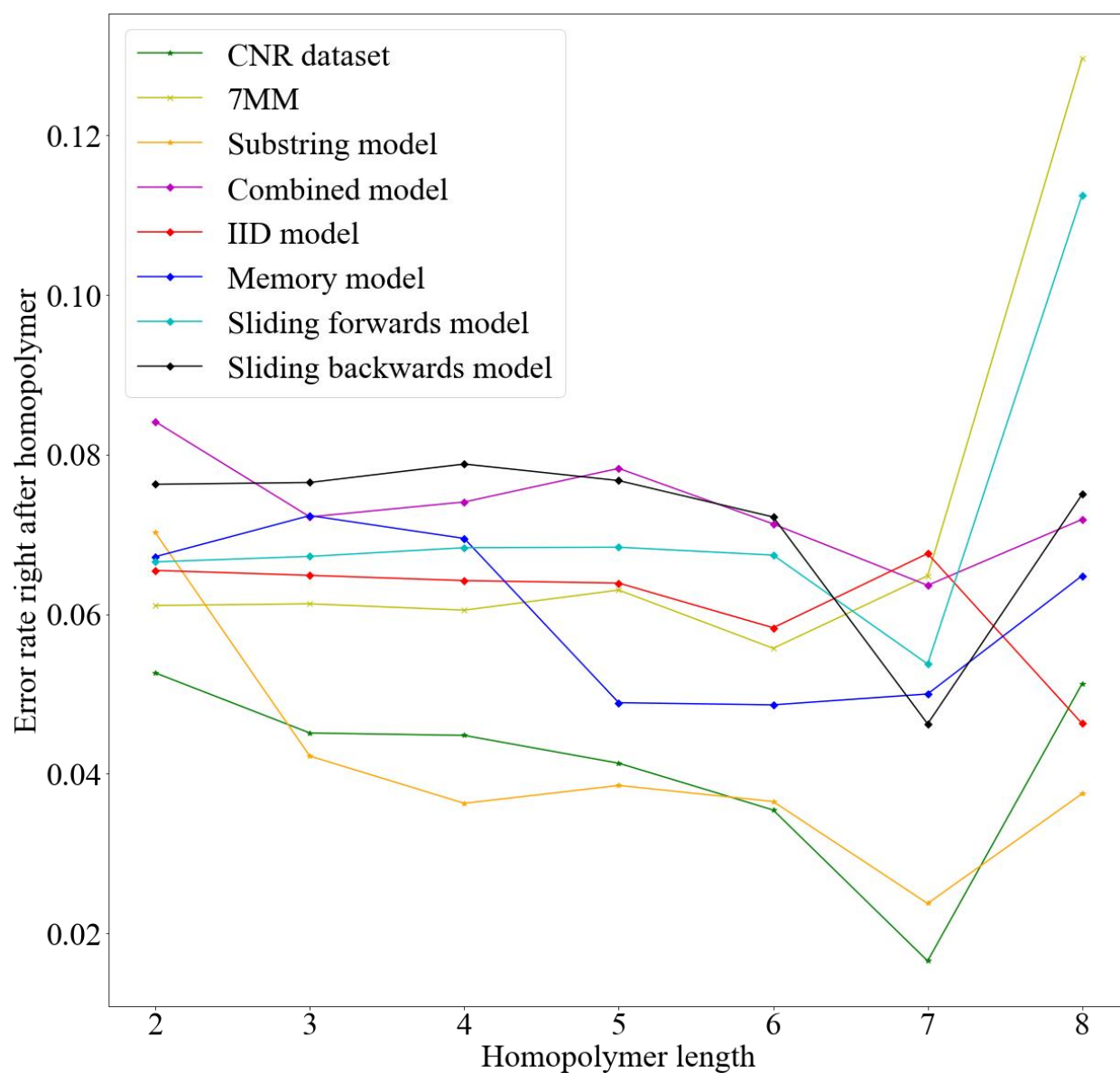
Supplementary Figure W-11: Comparison of error rates produced after homopolymers of G by each model



Supplementary Figure W-12: Comparison of error rates produced within homopolymers of T by each model

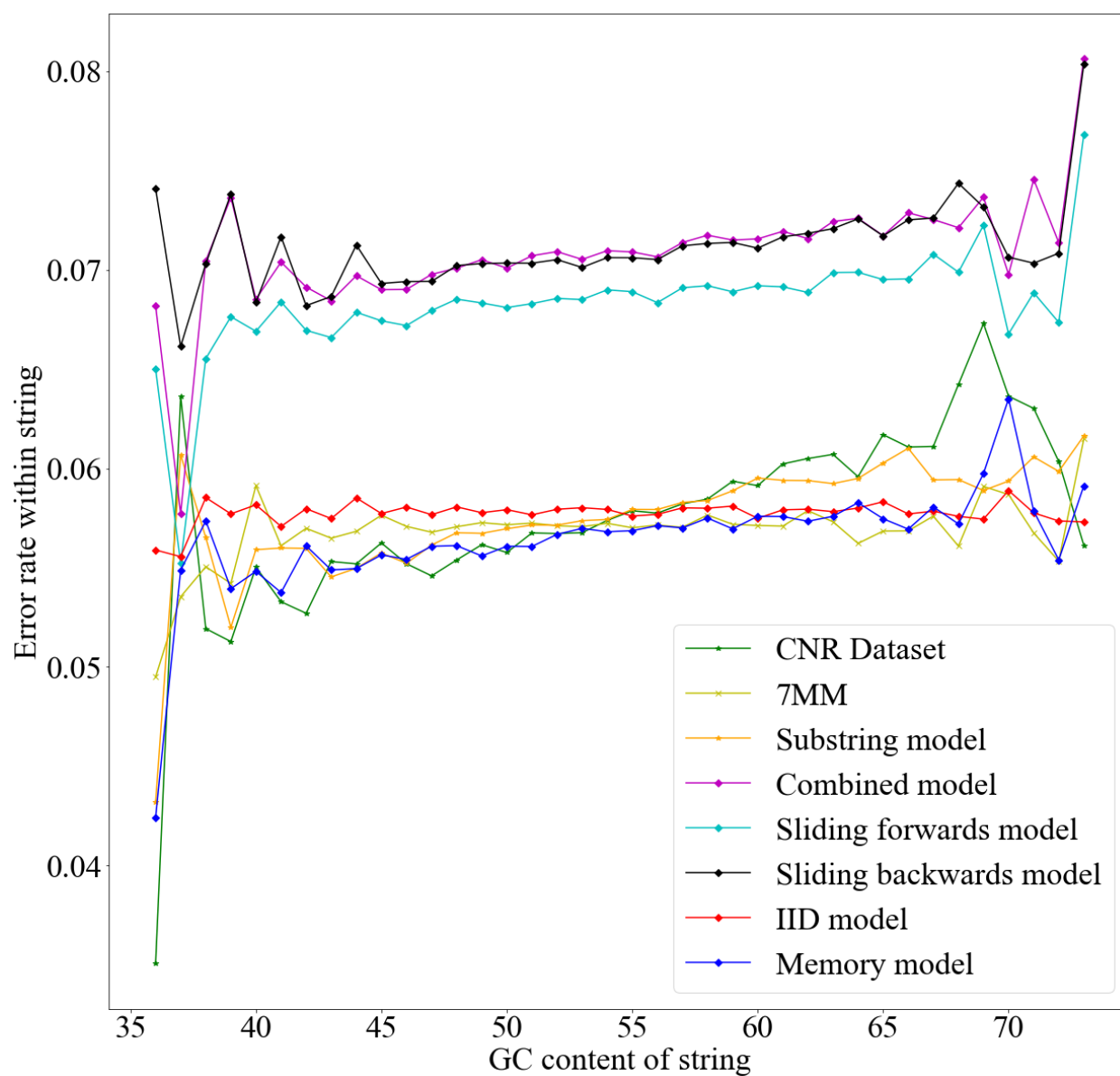


Supplementary Figure W-13: Comparison of error rates produced before homopolymers of T by each model



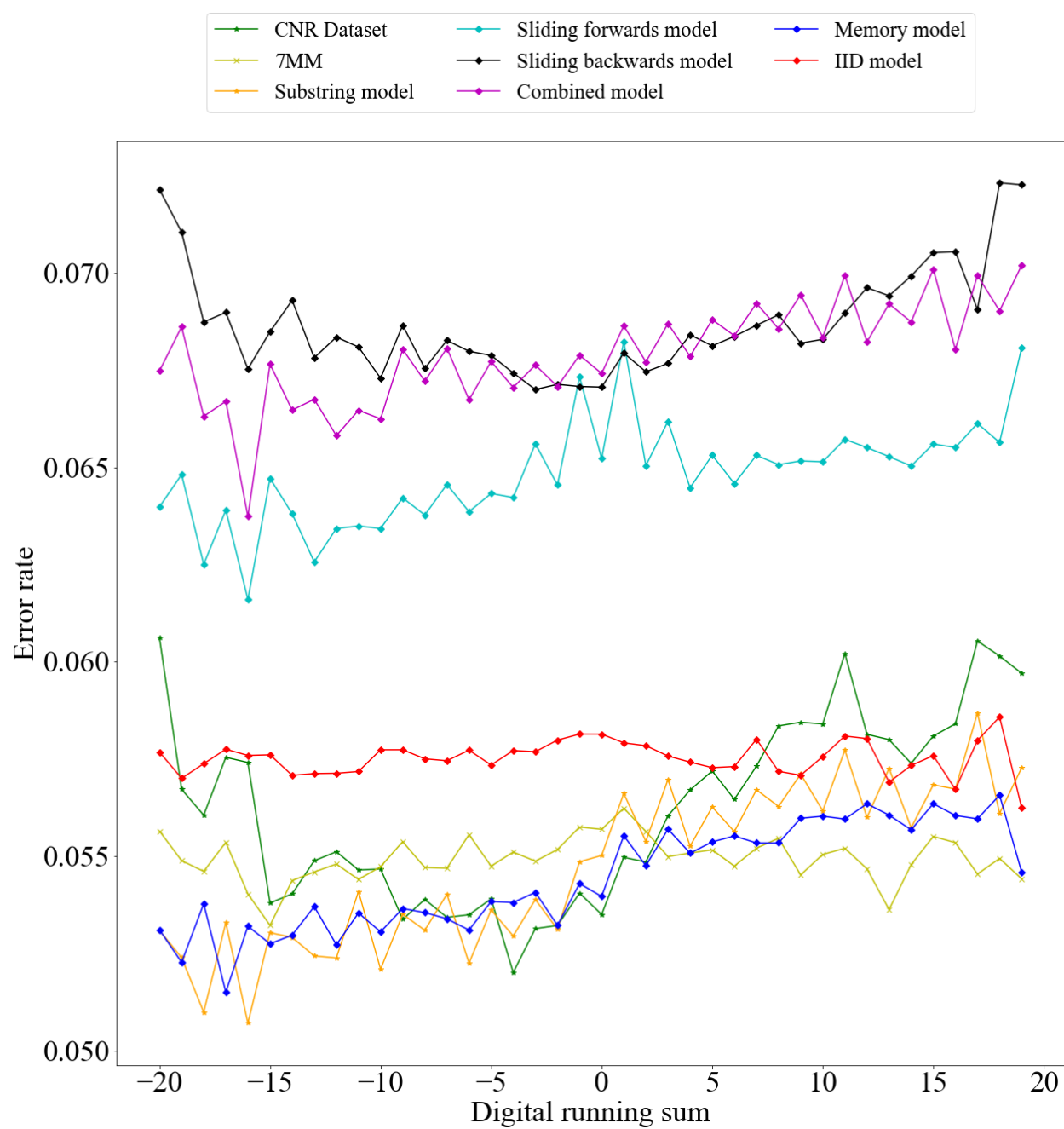
Supplementary Figure W-14: Comparison of error rates produced after homopolymers of T by each model

APPENDIX X ANALYSIS OF GC CONTENT

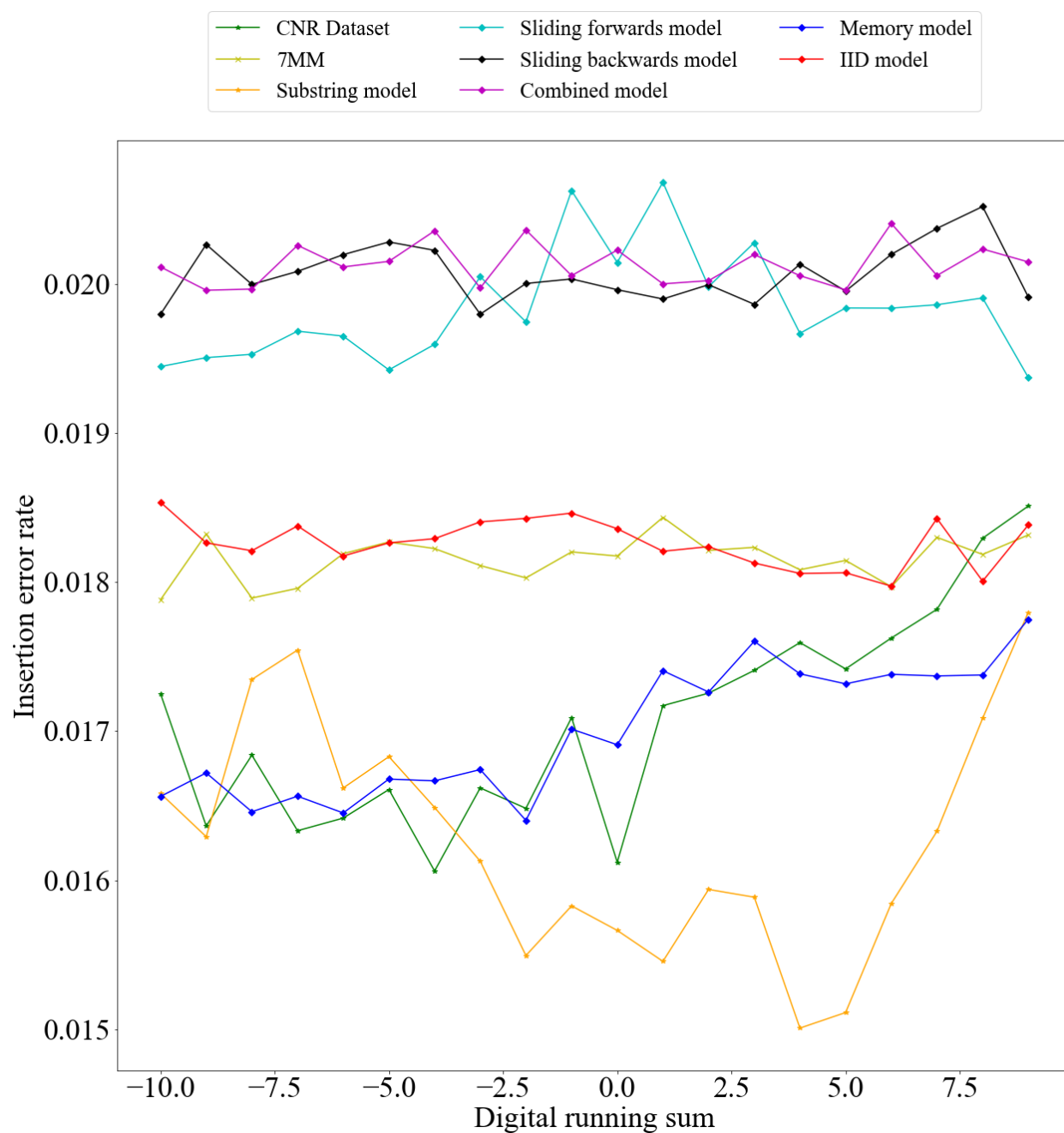


Supplementary Figure X-1: Comparison of error rates produced by each model for strings of different GC contents

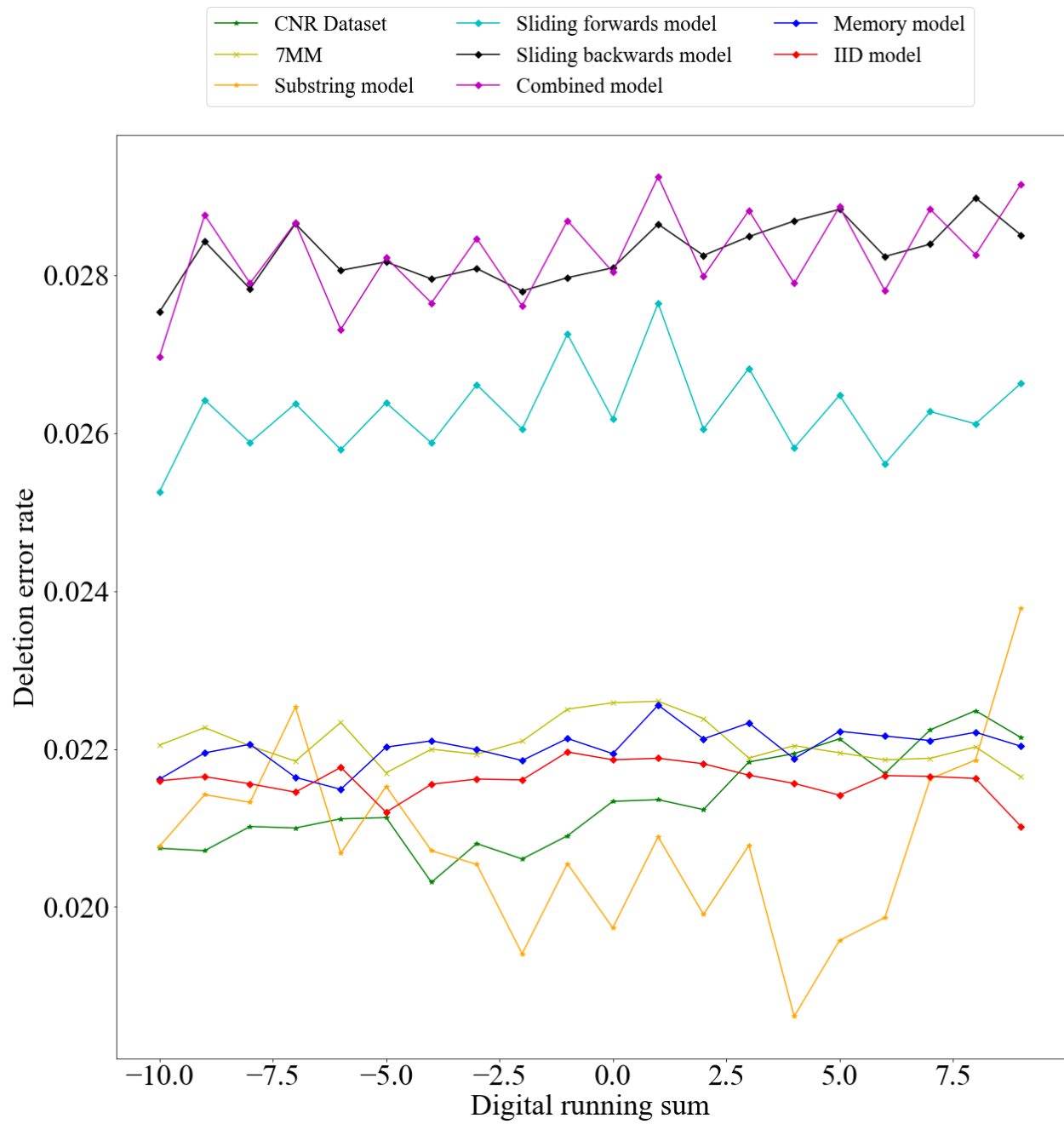
APPENDIX Y RUNNING DIGITAL SUM ANALYSIS



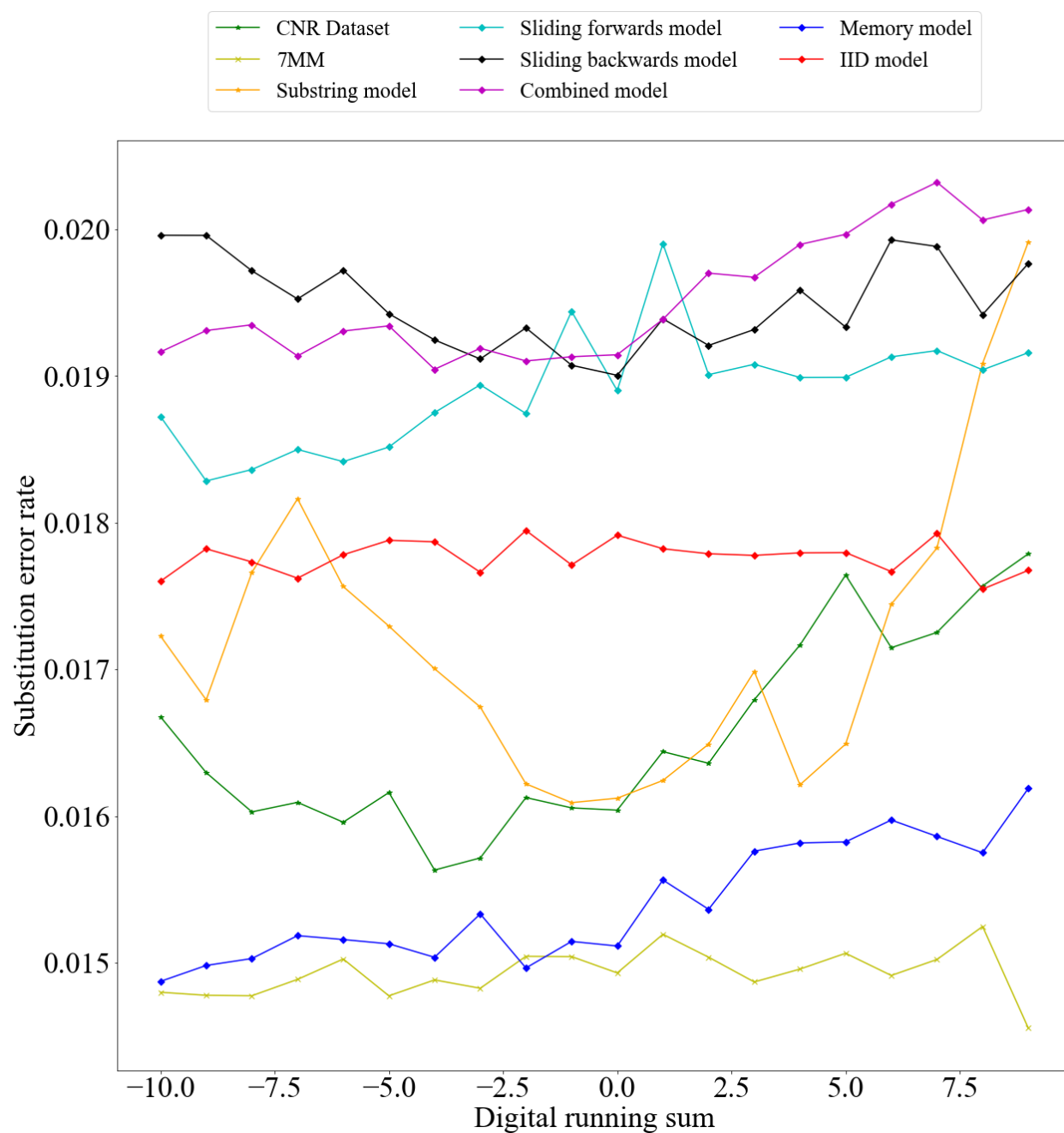
Supplementary Figure Y-1: Comparison of error rates produced by each model for different RDS values calculated from the front of the string



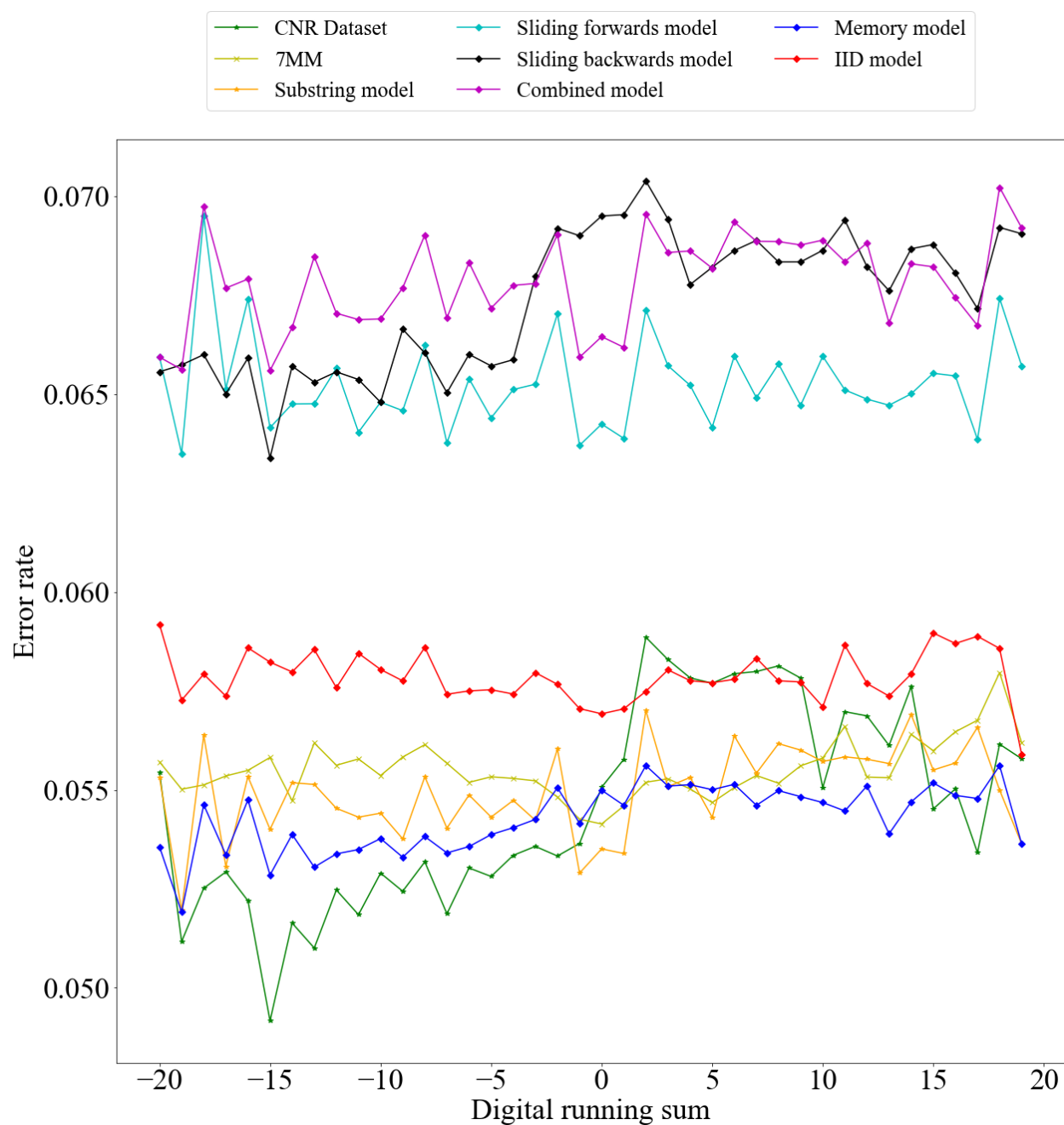
Supplementary Figure Y-2: Comparison of insertion rates produced by each model for different RDS values calculated from the front of the string



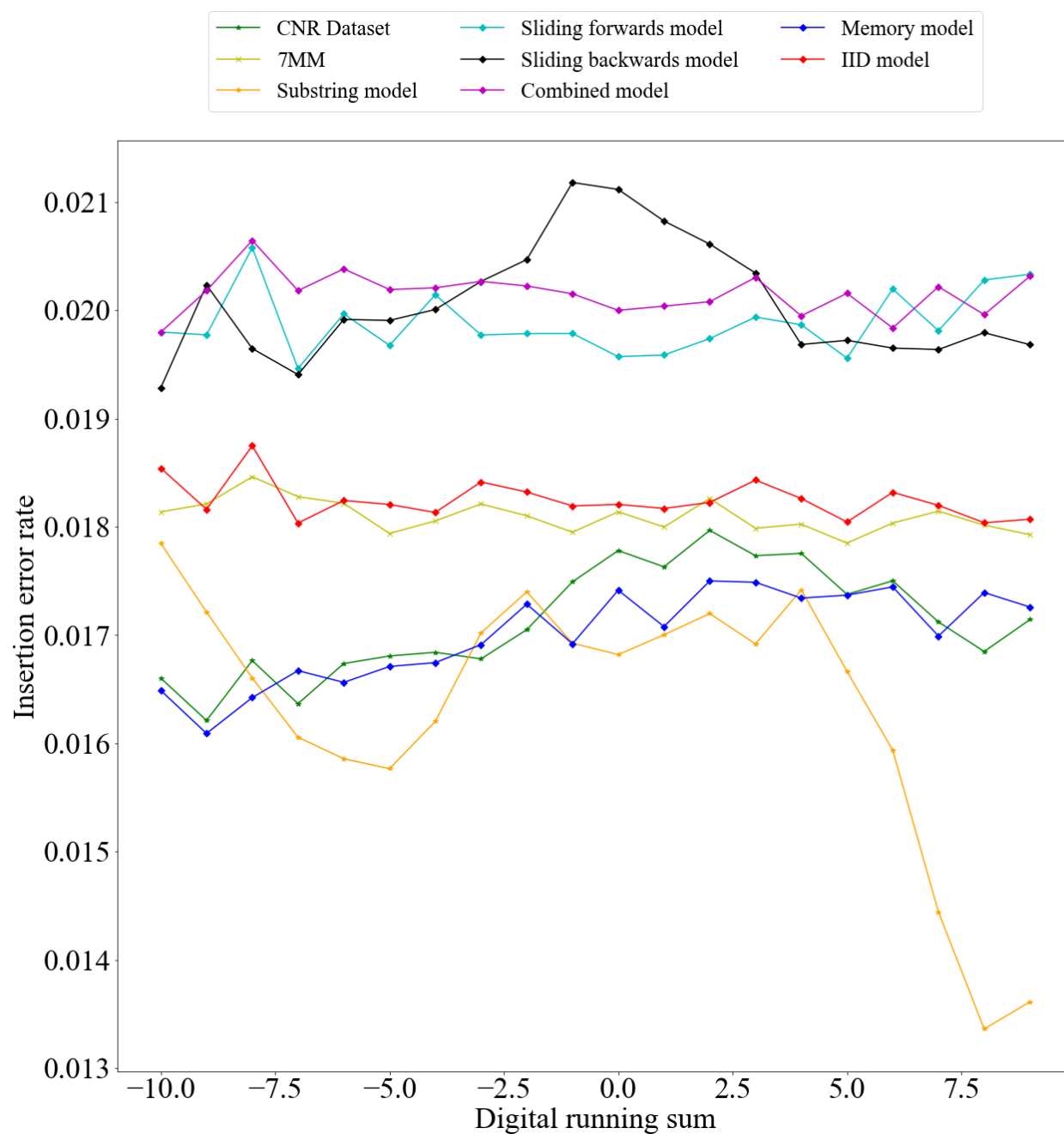
Supplementary Figure Y-3: Comparison of deletion rates produced by each model for different RDS values calculated from the front of the string



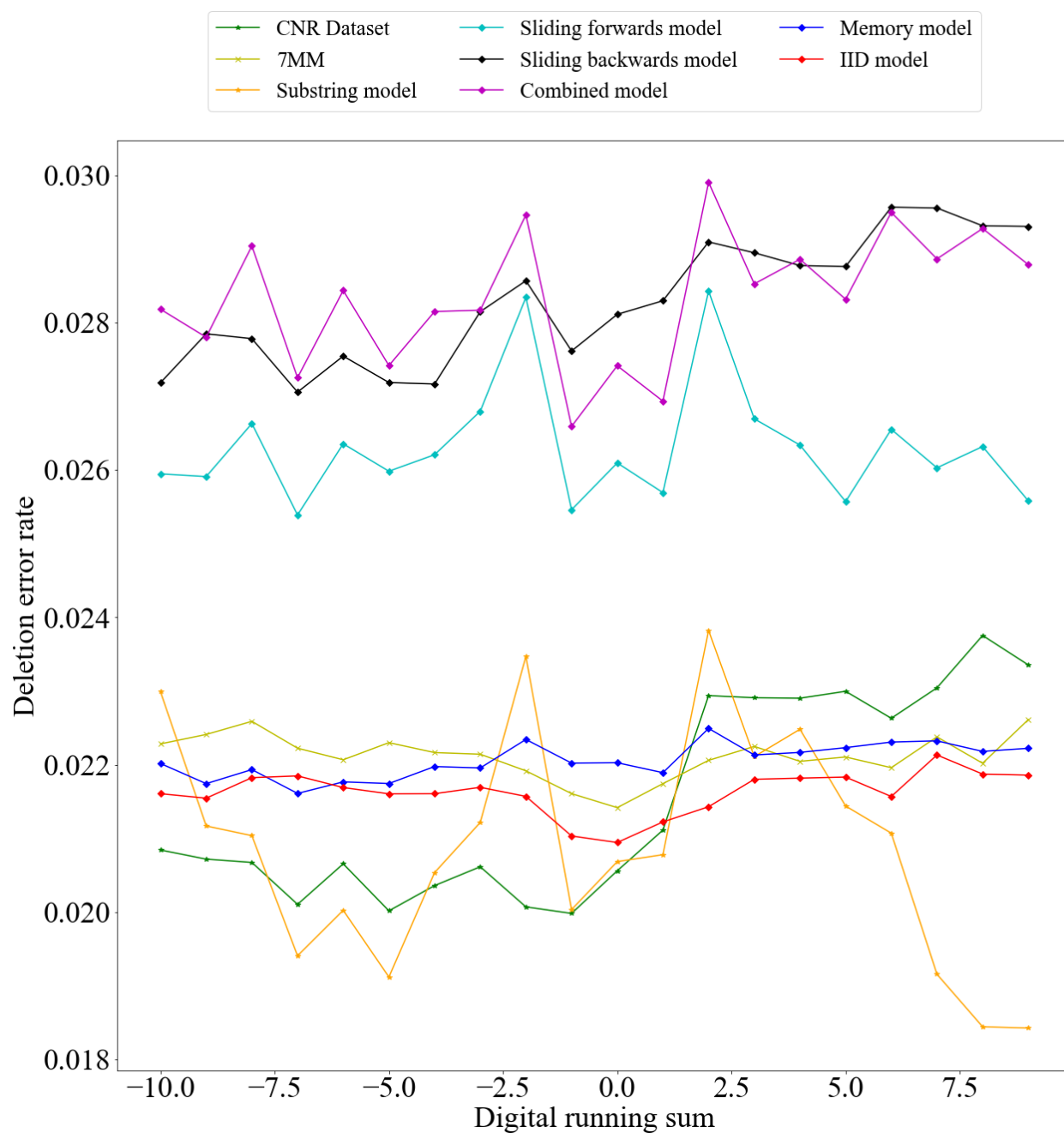
Supplementary Figure Y-4: Comparison of substitution rates produced by each model for different RDS values calculated from the front of the string



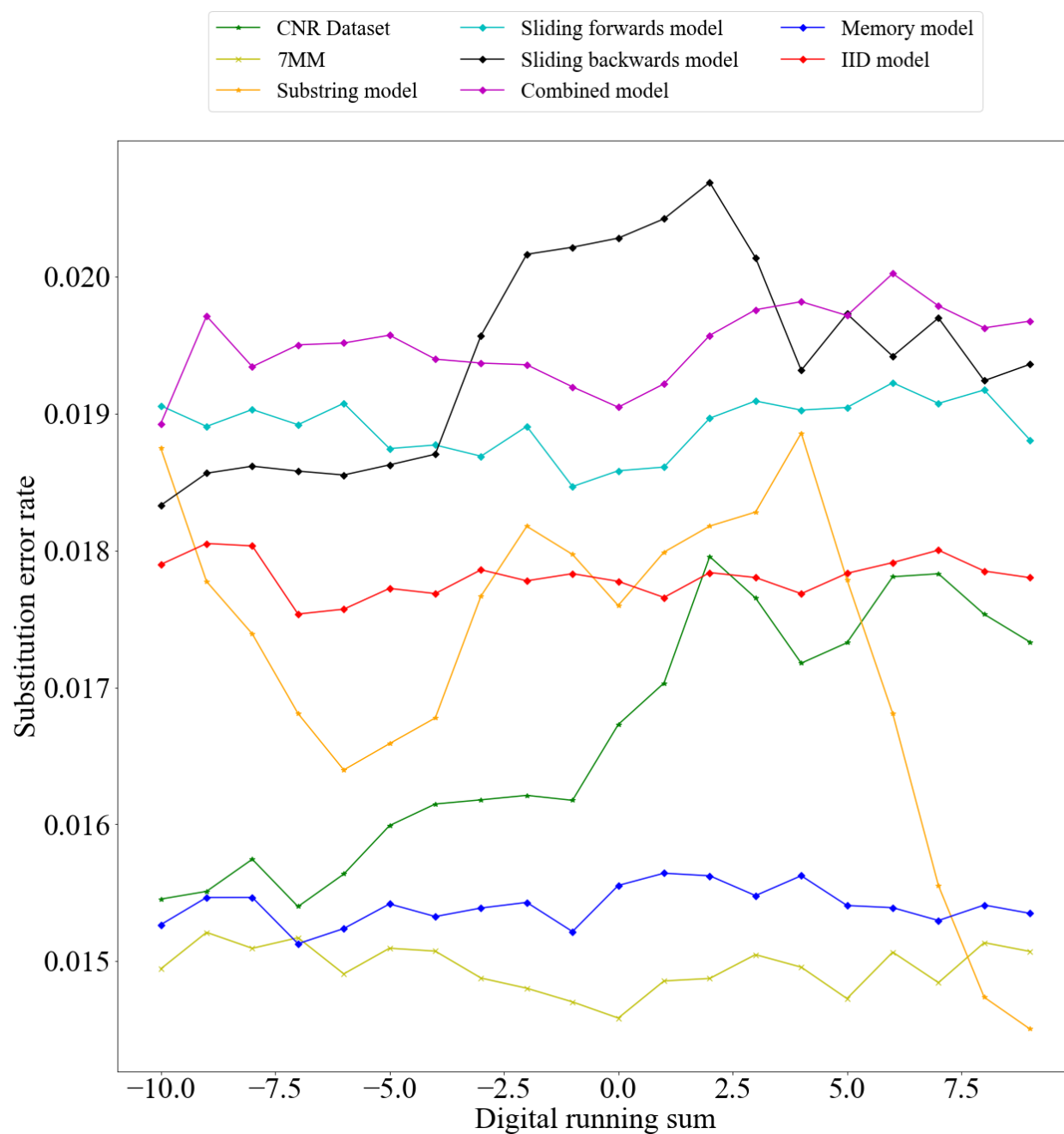
Supplementary Figure Y-5: Comparison of error rates produced by each model for different RDS values calculated from the back of the string



Supplementary Figure Y-6: Comparison of insertion rates produced by each model for different RDS values calculated from the back of the string



Supplementary Figure Y-7: Comparison of deletion rates produced by each model for different RDS values calculated from the back of the string



Supplementary Figure Y-8: Comparison of substitution rates produced by each model for different RDS values calculated from the back of the string

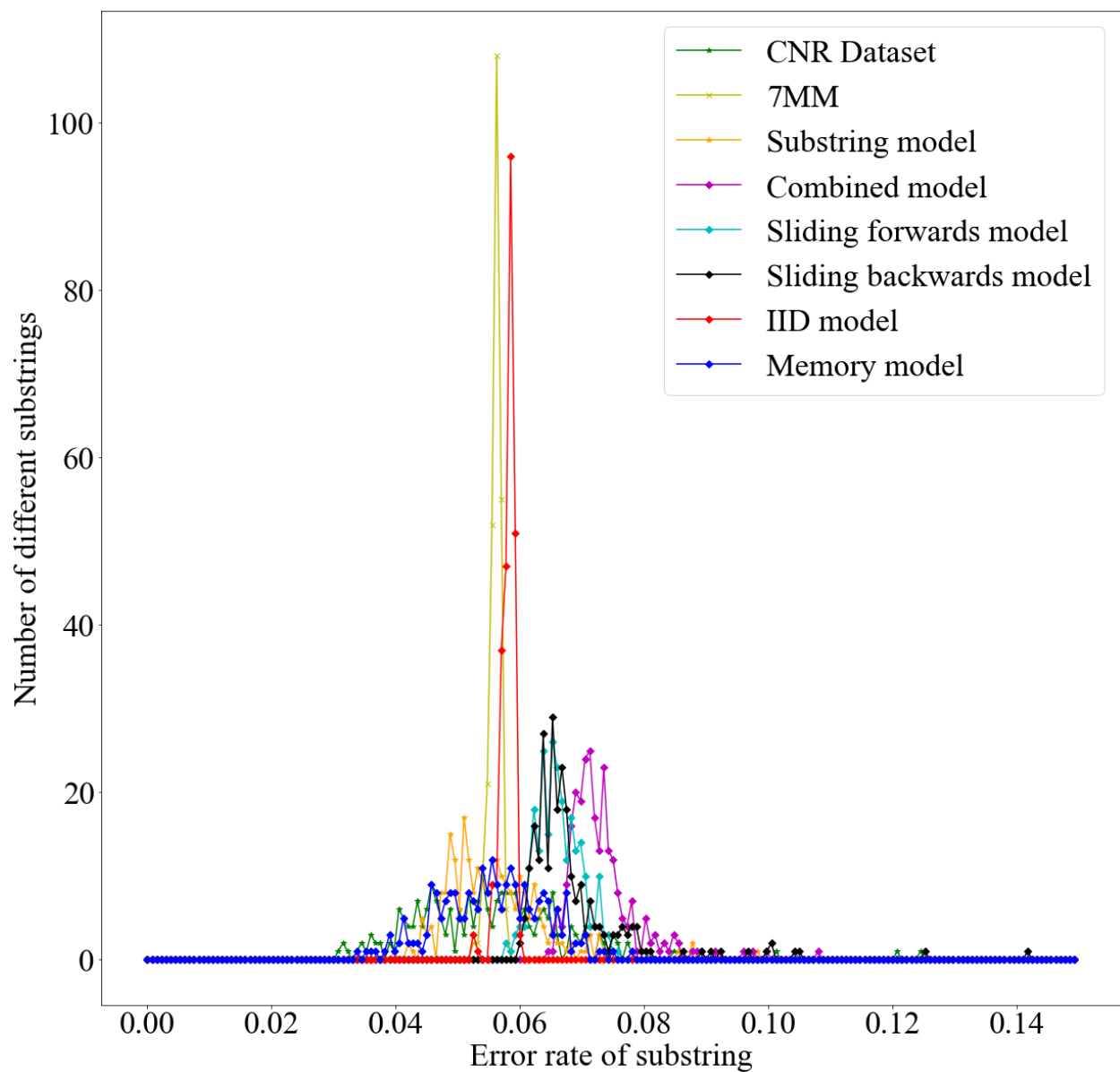
APPENDIX Z PROBLEMATIC SUBSTRING ANALYSIS

Supplementary Table Z-1: Error rate of each length 2 substring produced by each model

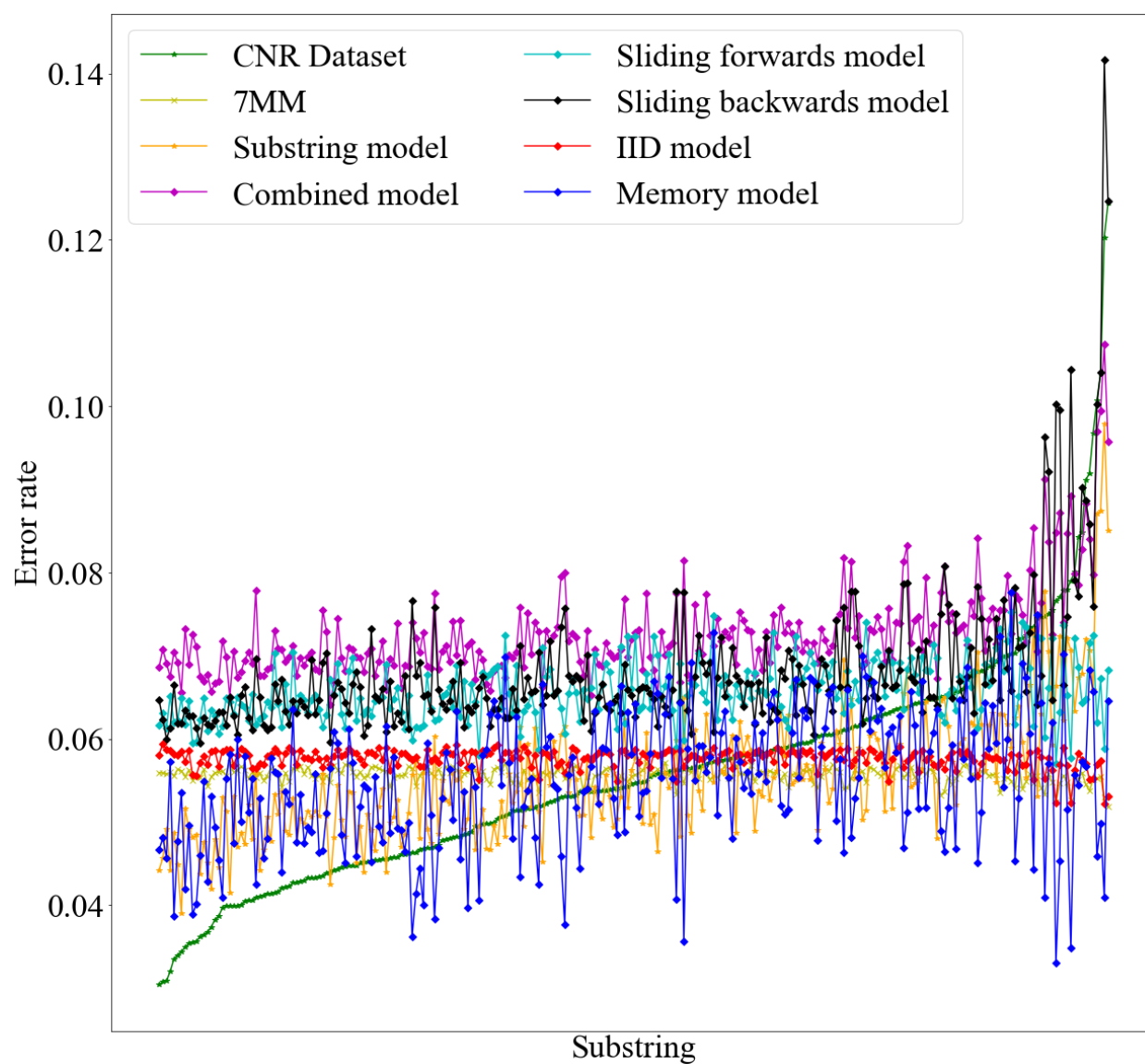
Substring	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
AA	0.048	0.054	0.057	0.077	0.062	0.068	0.054	0.041
AC	0.048	0.057	0.051	0.074	0.066	0.068	0.059	0.055
AG	0.062	0.056	0.063	0.078	0.066	0.070	0.058	0.055
AT	0.039	0.058	0.050	0.073	0.067	0.065	0.059	0.050
CA	0.052	0.057	0.053	0.073	0.069	0.066	0.059	0.060
CC	0.072	0.054	0.055	0.075	0.065	0.077	0.055	0.056
CG	0.070	0.057	0.055	0.074	0.071	0.074	0.059	0.068
CT	0.064	0.056	0.055	0.073	0.069	0.063	0.059	0.070
GA	0.045	0.056	0.063	0.079	0.067	0.066	0.058	0.050
GC	0.055	0.057	0.059	0.077	0.069	0.069	0.059	0.060
GG	0.074	0.054	0.067	0.082	0.065	0.085	0.054	0.046
GT	0.050	0.058	0.059	0.077	0.069	0.065	0.059	0.058
TA	0.059	0.057	0.053	0.072	0.067	0.066	0.058	0.057
TC	0.053	0.056	0.052	0.072	0.068	0.065	0.059	0.061
TG	0.058	0.057	0.055	0.074	0.068	0.070	0.059	0.058
TT	0.050	0.055	0.052	0.072	0.063	0.068	0.055	0.041

Supplementary Table Z-2: Statistics for error rates produced by each model within length 2 substrings

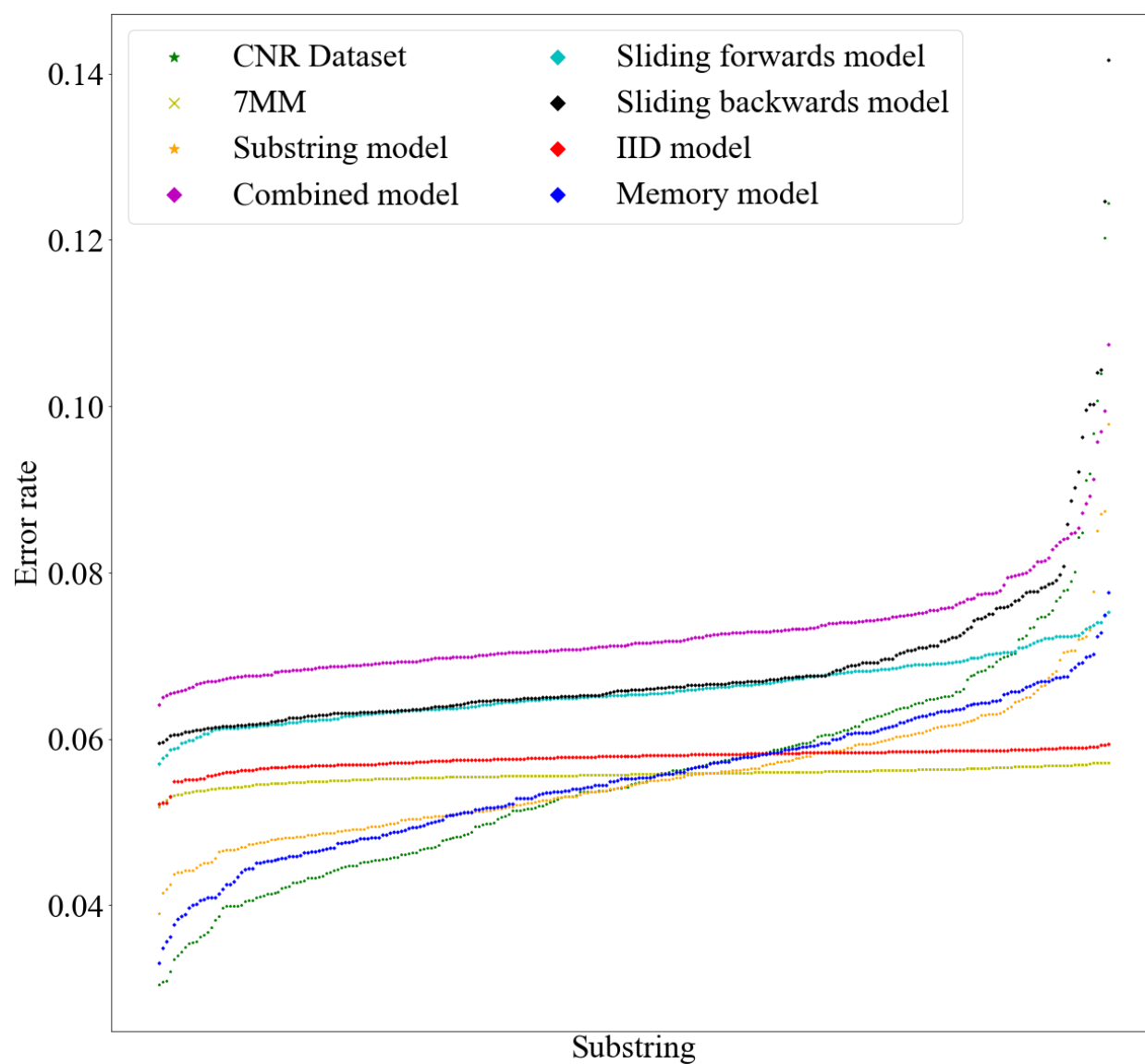
Model	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
Maximum error rate	0.074	0.057	0.067	0.082	0.071	0.084	0.059	0.070
Minimum error rate	0.039	0.054	0.050	0.072	0.062	0.063	0.054	0.041
Mean error rate	0.056	0.056	0.056	0.075	0.067	0.069	0.058	0.056
Median error rate	0.054	0.056	0.055	0.074	0.067	0.068	0.059	0.057
Standard deviation	0.010	0.001	0.005	0.003	0.002	0.005	0.002	0.008
Skew	0.342	−0.192	0.866	0.944	−0.333	1.748	−1.076	−0.157



Supplementary Figure Z-1: Comparison of distribution of error rates between different substrings of length 4 produced by different models



Supplementary Figure Z-2: Comparison of error rates produced by each model for each substring of length 4 ordered according to error rate in the dataset



Supplementary Figure Z-3: Comparison of error rates produced by each model for each substring of length 4 ordered according to error rate produced by the model

For Supplementary Table Z-3 to Supplementary Table Z-6, in each table the strings that are present in the model outputs as well as the dataset, are highlighted.

Supplementary Table Z-3: Length 4 substrings with the highest error rate produced by each model

Number	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
1	CCCC	TCGT	GGGG	GGGG	CTCG	GGGG	GATC	CTCG
2	GGGG	GCAC	CGGG	CGGG	CTCA	CCCC	TCAC	ACTC
3	CGGG	CGGT	AGGG	AGGG	CCTC	AAAA	GCGT	CTCA
4	AGGG	TAAG	CCCC	CCCC	CCTG	CGGG	CAGT	CTCT
5	CCCG	ATCG	GGGA	GGGA	CTAG	TTTT	TAGC	CTAG
6	CCCT	GCAT	GGGT	AAAA	CCTA	AGGG	ATGT	TACT
7	TCCC	CGTG	TGGG	TCCC	CTCT	TGGG	TGGT	TCGT
8	ACCC	GTAT	CTTT	TGGG	CCGT	GGGA	AGCT	CACT
9	TAGG	ATTA	TCCC	GGGT	TCGT	GCCC	ACGT	CCTC
10	CCGG	CGTA	CCCT	TTTT	CCCG	ACCC	GACA	CGTG

Supplementary Table Z-4: Length 4 substrings with the lowest error rate produced by each model

Number	CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
1	GATG	CCCC	ATCA	TTCA	TTTT	AATC	GGGG	TTTT
2	GATC	TTTT	ATCG	ATCG	AAAA	TTCA	AAAA	AAAA
3	AGAT	AAAA	ATCT	TTCT	TTTA	AGAT	TTTT	GAAA
4	CATC	GGGG	TTCA	ACTA	TTTG	AACT	CCCC	ATTT
5	GATT	GAGG	AATC	ATCA	GGGG	CTTC	AAAG	GGAA
6	GATA	AAAG	TCAT	ATCT	AAAG	GTCA	CCCA	GTTT
7	ATCA	GGGA	AACT	TCTG	AAAC	CACT	TTTC	GATT
8	GAAC	CTCT	AACG	TCAT	AAAT	CTCA	TTTA	AAAC
9	TGAT	TTTG	GATG	ATTC	ATTT	TCAT	GGGT	AATT
10	AAAC	GGAG	CATC	ACTG	GAAA	TTCT	CCCT	GGTT

Supplementary Table Z-5: Length 6 substrings with the highest error rate produced by each model

CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
CCCCCC	CACCAT	GGGGGG	GGGGGG	CCCCGT	GGGGGG	GTAGCC	CACTCG
GGGGGG	GTATCG	CCCCCC	CGGGGG	CTTCGA	CCCCCC	CATGGT	ATACTC
CGGGGG	CGTATC	CGGGGG	AGGGGG	TACTCA	AAAAAA	CAGAAG	ACTCGT
AGGGGG	TAAGCC	AGGGGG	CCCCCC	CACTCG	CGGGGG	GTAGCG	TACTCG
ACCCCC	GTTTAA	TCCCCC	TCCCC	ACTCGC	TTTTTT	GTTGAC	CTACTC
AAAAAA	CTAACC	CCCCCT	CCCCCT	TACTCT	TGGGGG	GTCCGA	TACTCA
TCCCCC	GCCAGT	GGGGGC	GCCCCC	CCCCCA	AGGGGG	GACAGT	ACTTCG
CCCCCG	CGACTA	TGGGGG	TGGGGG	TACTCG	TCCCCC	GTAACCT	GCACTC
CCCCCT	CAGTAA	ACCCCC	GGGGGC	ACTCAT	GGGGGA	ACCTAG	GTACTC
GCCCCC	TGACCG	GCCCCC	GGGGGA	GCCTAG	CCCCCT	CATTAT	ACTCAT

Supplementary Table Z-6: Length 6 substrings with the lowest error rate produced by each model

CNR dataset	7MM model	Substring model	Combined model	Sliding forwards model	Sliding backwards model	IID model	Memory model
GATGAT	AAAGGA	ATCATC	ATATCG	TTTTG	CTAATC	AAAAAA	GAAAAA
GATGGA	GAGGAG	ATCAAC	TATTCT	AGAGAT	AAATTC	GGGGGG	GGAAAA
GATATT	TTTCAA	AAATGC	ATATCA	GGGAAG	CAAATC	GGGAGG	ATTTTT
ATCCAT	CCTCTA	TCATCA	AATGCG	TTTTTT	GCATCT	AAAAAC	AGAAAA
GATATC	CCCCCC	ATCAAT	TTAATC	AAAGGA	TCTCCA	TTTTTT	GGGAAA
ATCGTC	GAAGAG	AATCTG	TTCATG	GAAAAT	AATTCA	AAAATA	GGATTT
ATCGTC	CCTCCC	GATATT	GATATA	TTTGGA	AACATC	TTTTAA	GGTTTT
AACGAT	TCCGTT	AACATC	ATTAAT	AAAAGA	ACTTCA	TTTTTG	GATTTT
ATCATC	GGGCCG	CATCAT	ATCAAT	TTTATT	GACACT	CCCCTG	GAAAAC
GATGTC	GGCCCC	AATCAT	TTGTCG	ACAATT	TTCAAT	TTTCAA	AAAAAC

APPENDIX AA MODELLING ERROR RUNS IN DNA DATA STORAGE USING A MARKOV MODEL

The paper “Modelling Error Runs in DNA Data Storage Using a Markov Model” [36], that was produced during the course of this study, follows.

Modelling Error Runs in DNA Data Storage Using a Markov Model

Danie Walker, Albert Helberg

School of Electrical, Electronic and Computer Engineering, North-West University, Potchefstroom Campus, South Africa

¹26846179@mynwu.ac.za

²albert.helberg@nwu.ac.za

Abstract—DNA data storage offers high-density storage but synthesis and sequencing cause high error rates. Accurate models of the errors and error distributions are required due to the high synthesising cost. Currently, available channel models fail to represent all aspects of the errors introduced during the DNA data storage process. This paper proposes a new model capable of representing DNA data storage channel error bursts. A Markov model trained on a MinIon dataset is created and compared to currently available models. We show that this model represents error runs, bursts, and error-free runs better than similar existing models.

Keywords— Channel Modelling, DNA Data Storage, Markov Models

I. INTRODUCTION

Due to the large amounts of data constantly produced in the current technological era, traditional storage media are hard-pressed to keep up with growing demands. Many corporations, such as Microsoft, are investigating new storage methods, including DNA data storage systems [1]. The advantages of DNA data storage systems include the high data density provided by DNA, with a theoretical limit of $\sim 4.5 \times 10^7$ GB/g [2].

DNA is a molecule consisting of four bases or nucleotides: Adenine, Cytosine, Guanine and Thymine (A, C, G and T). Data can then be stored on a four-letter alphabet, represented by these four bases. DNA data storage consists of encoding data to this alphabet and synthesising the corresponding DNA strings. Polymerase chain reaction (PCR) is used to amplify DNA strands, i.e. creating more copies of each strand before the data can be read. In order to read the data, the DNA strings should then be sequenced, whereafter the data can be decoded. Synthesis and PCR occur via various chemical reactions and are usually done in labs. Sequencing is done via specialised machinery, and available sequencing technologies include Illumina and Oxford Nanopore (MinIon) sequencing [3].

During the various stages of the DNA data storage process, different types of errors can occur. Deletion errors may occur when one or more nucleotides are deleted from the string, i.e. not written during synthesis or not read during sequencing. Insertion errors may occur when nucleotides are added to the string in erroneous locations during the synthesis process or read in an erroneous location during sequencing. Lastly, substitution errors may occur when a nucleotide is substituted for a different nucleotide. For example, a C can be read as a T [3]. Burst errors (multiple errors in close proximity to each other) are also common [4].

Current technologies make the DNA synthesis and sequencing processes prohibitively expensive. In 2021, the cost of DNA storage was estimated at \$1 300 000/ TB [5]. Although the costs are predicted to decrease in the coming years, it is still a barrier to entry for prospective researchers. Therefore, channel models are required to represent the data storage channel, making research on the topic accessible.

In this paper, we propose an improved method for modelling error bursts created by MinIon technology. Section II describes existing channel models and section III describes the methods used to create the proposed model. Finally, section IV compares the proposed model with two existing models.

II. EXISTING CHANNEL MODELS

Many current models assume that errors are independent and identically distributed (IID) throughout the DNA string [6], [7]. On the other hand, research indicates that errors do not occur uniformly, but certain areas of a DNA string may see more errors [8], [9]. Areas with higher GC-content may see an increase in errors [9]. Long runs of homopolymers (repetitions of the same base) can also lead to increased errors [4]. Furthermore, errors may be more likely to occur at the beginning or end of a DNA string than at the middle [8]. Existing models have been developed using statistical and machine learning techniques to account for these factors.

The most well-known machine learning model is DeepSimulator which simulates the electrical current produced by Nanopore devices [10]. This current can then be decoded to simulate the sequencing process. The use of artificial intelligence allows the model to consider many factors when making a prediction, including preceding bases and the position within the string. A shortcoming of DeepSimulator is that the same kind of error regularly appears in the same position in different sequences, which is not representative of an actual DNA storage channel [11].

Another study has used a Markov chain to model the occurrence of errors [11] to improve on the DeepSimulator model. The dataset used to train the model was created using ThermoFisher's chemical synthesis followed by MinIon sequencing. The model considers the previous event and uses the previous six bases within the DNA string to predict the following event. By taking the previous six bases into account, the model attempts to take attributes of the input DNA string, such as homopolymer runs, into account. This method requires a very large dataset to implement. The data used to train the model did not contain every possible combination of

previous bases and events. Therefore, probabilities for unobserved combinations were estimated by averaging over the observed combinations. Although this model has shown some improvements over DeepSimulator, it fails to properly account for long-run errors [11]. This model shall be referred to as the Memory Model throughout the rest of this paper.

III. METHOD

The proposed model consists of a Markov model trained using the Clustered Nanopore Reads Dataset published by Microsoft [1]. The dataset consists of 10 000 DNA strings of length 110 bases. These strings were synthesised by Twist Bioscience and sequenced using Minlon technology, resulting in a total of 269 709 noisy reads. The reads are arranged into clusters corresponding to the input strings.

In order to detect errors within the DNA strings, we compared the noisy strings to the original strings via Gestalt pattern matching [12]. The algorithm starts by matching the longest common substrings. Gaps between the substrings are then identified as errors. For example, let *AACTGTA* be the original string and *CACGTAG* the noisy read. The algorithm would detect a substitution error at position 0, a deletion error at position 3 and an insertion error after position 6 of the original string. The specific bases substituted, inserted or deleted were then identified according to the positions of the errors.

A subset of the original data containing 1349 DNA strings was sampled uniformly from the dataset to train the Markov model, leaving a test set of 268 360 strings. Four observations can be constructed for each string in the training set using the detected errors. The observations consist of transmissions, indicating that the corresponding base was transmitted correctly, as well as insertions, deletions and substitutions, representing the various types of errors. The observations are represented by the following set: {'t', 'i', 'd', 's'}. For the example above, the observations in order are *sttdtiti*.

The Markov model comprises seven states and the four observations described above. The model was trained using the Baum-Welch algorithm as described in [13]. The algorithm was run for 1000 iterations for each string in the training set. The trained model can then predict a string of observations when given an input DNA string. Note that the set of observations given by the model only indicates the occurrences of different types of errors and not the specific nucleotide bases for which the error was made. For example, the Markov model can predict the occurrence of an insertion error, but not the type of base inserted. Therefore, an additional layer was added to the model to predict the probable base.

Using the training set, the probability of insertion for each base was calculated. The model then calls on these probabilities when inserting a base to determine which base to insert. Furthermore, the probability for every type of substitution was also calculated, for example, the probability of substituting an 'A' if the current base is a 'C'. When the model predicts a substitution, it looks up the current base and chooses a base to substitute according to the established

probability distribution. When a deletion is predicted, the model deletes the base at that position without the need to consult additional probability distributions.

Deletion and substitution errors are handled as occurring at the current position. Insertion errors are handled as occurring before the current position in the input string. Therefore, if an insertion is predicted, it is added, and the current position in the input string does not change. In order to allow insertions to be added at the end of the input string, the model predicts an additional base after reaching the end of the input string. If a substitution or deletion is predicted, it is ignored. If an insertion is predicted, it is added to the end of the string, and the model continues predicting until it predicts something other than an insertion. This condition allows the model to represent runs of insertion errors at the end of the DNA string.

Lastly, the probabilities for different observations at the first position of a string were calculated for the training set in order to choose the initial state for the model. Of the seven states, only the four with the highest probability for each observation were considered. The probability for each state was calculated in order to mimic the probability of each observation in the training set.

The model chooses a starting state when an input string is given. After that, a string of observations is created by the Markov model. Afterwards, these predictions are translated into a string of specific bases. If the prediction is a transmission, the corresponding base of the input string is added to the output. If the prediction is a deletion, the corresponding base of the input string is left out. If the prediction is an insertion or a substitution, the model uses the calculated error probabilities to determine which base to add to the output string.

IV. RESULTS

The ability of the proposed model to generate errors in a similar manner as the actual synthesis and sequencing system was compared to the commonly used IID model [6, 7], and the Memory Model from [11]. Although the Memory Model was created using a different dataset than the proposed model, Minlon technology was also used, allowing the models to be compared. Each of the three models was used to generate 268 360 output strings, corresponding to the number of strings in the test set. The same error detection algorithm used to analyse the Microsoft dataset was used to find the errors for each model output. These three models are compared to the Microsoft dataset below.

A. Error Statistics

In order to compare the accuracy of each model, various statistics were calculated using the output of each model and the test set.

Table I shows that both the IID model and the proposed Markov model have similar error rates to the Microsoft dataset. The Memory Model has a much larger error rate. Furthermore, the original data has a number of error-free strings. The proposed Markov model produced a smaller

number of error-free strings, while the Memory Model produced no error-free strings.

Additionally, the average string length was examined. This gives an indication of the number of each error type, as more deletions than insertions would result in a shorter output string. It is seen that the IID model and the proposed Markov model produced string lengths similar to the original data, while the Memory Model produced shorter strings on average.

TABLE I
COMPARISON OF ERROR STATISTICS FOR EACH MODEL

Metric	Microsoft Dataset	Proposed Markov Model	Memory Model	IID Model
Error rate (%)	5.77	5.70	15.17	5.79
Number of error-free strings	4983	3328	0	377
Average string length (bases)	109.69	109.63	103.69	109.69

Table II describes the distribution of different types of errors. It is seen that deletion errors are the most prevalent in all models. The IID and proposed Markov models have similar rates for each type of error when compared to the test set. The Memory Model produced a much larger deletion rate and fewer insertions, confirming the shorter string length produced by this model.

TABLE II
DISTRIBUTION OF ERROR TYPES FOR EACH MODEL

Type of Error	Microsoft Dataset	Proposed Markov Model	Memory Model	IID Model
Insertion	33 %	35 %	18 %	32 %
Deletion	38 %	39 %	55 %	37 %
Substitution	29 %	26 %	26 %	31 %

B. Error Distribution

Fig. 1 represents the position of insertion, deletion and substitution errors of the first 500 strings for the test dataset, the Memory Model and our proposed model. The distribution of errors within the strings can be examined using these plots. The IID model is not shown as it has evenly distributed errors throughout each string, as expected.

At the right of each plot, it can be seen that only insertion errors are present at the last position of each string. This is due to the fact that insertions can occur after the last base of the input string.

It is clear that the Microsoft dataset has an increase in errors, especially insertions, at the beginnings and ends of the strings. The proposed Markov model shows a corresponding increase in errors at the beginnings of strings but fails to model the increase at the ends accurately. The Memory

Model shows an overwhelming increase in deletion errors at the beginnings of strings, while insertion errors dominate the ends of strings.

Various bursts of errors can be observed in the Microsoft dataset and the proposed Markov model in the form of multiple errors close to each other. The Memory Model shows more bursts than the test data.

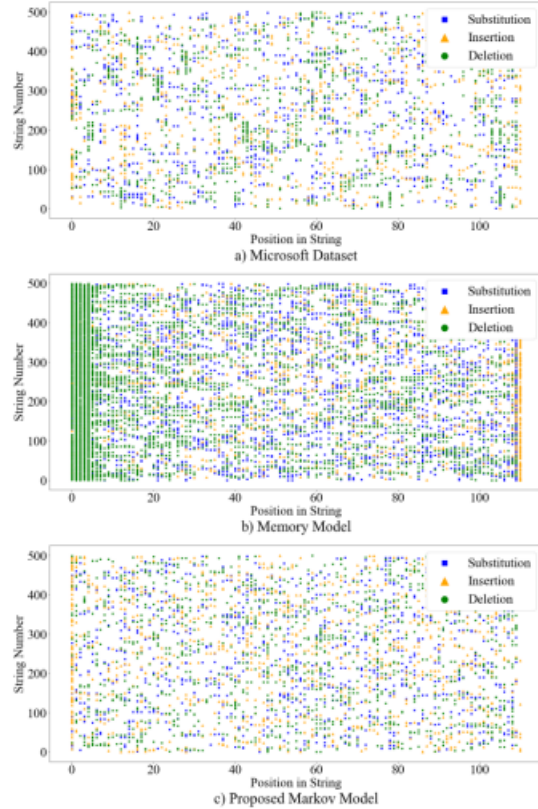


Figure 1: Scatterplots representing error positions in the first 500 strings produced by each model – a) Microsoft dataset, b) Memory Model and c) proposed Markov model

Since the first 500 strings produced by each model are plotted, some successive output strings correspond to the same input string. Numerous errors in close proximity in successive strings (represented on the y-axis) represent errors in the model outputs in the same position for the same input string. Such errors can indicate aspects such as homopolymer runs or other attributes of the input string, which leads to an increase in errors at that position. The Microsoft dataset has several indications of such dependencies, but the other models do not produce as many similar cases. Our proposed Markov model does not inspect individual bases of the input string when producing error positions and does not model these dependencies accurately.

Fig. 2 depicts the total number of errors each model generates at each position within the input strings. An

increase in errors of each type at the beginning of the string is observed in the Microsoft dataset and mimicked by the proposed Markov model. The Microsoft data shows increased errors at the end of the strings, which none of the models reproduce accurately. The Memory Model shows a marked increase in errors at each position throughout the strings. The IID model shows an even distribution of errors throughout the strings.

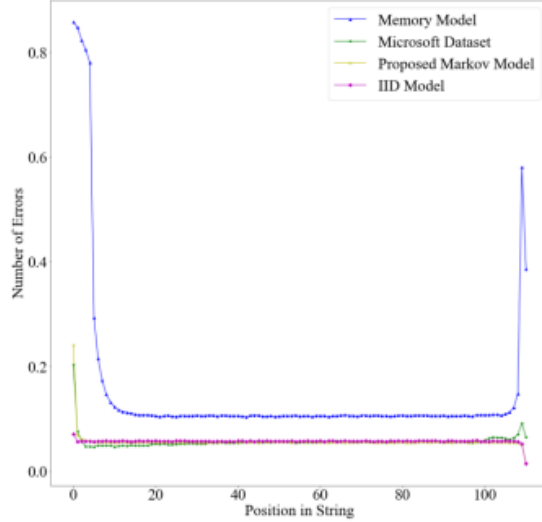


Figure 2: Comparison of error rate produced by each model at each position

C. Error Bursts

Error bursts are areas where errors occur at a high density [14]. In order to compare the bursts of errors generated by each model, a specific density was defined. Therefore, a burst was defined as two errors separated by three or fewer error-free bases. By this definition, each burst has a length of at least two and contains at least two errors. The bursts generated by each model are compared in Fig. 3, 4 and 5.

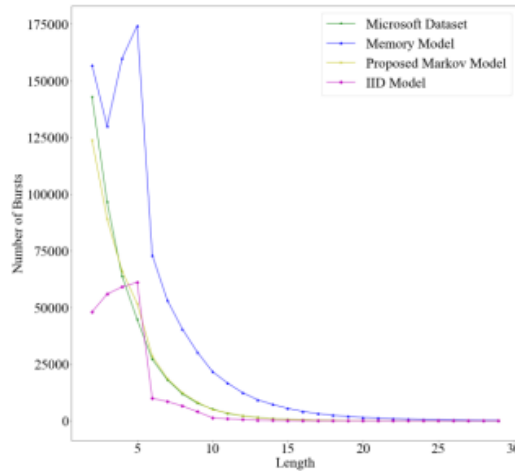


Figure 3: Comparison of burst lengths produced by each model

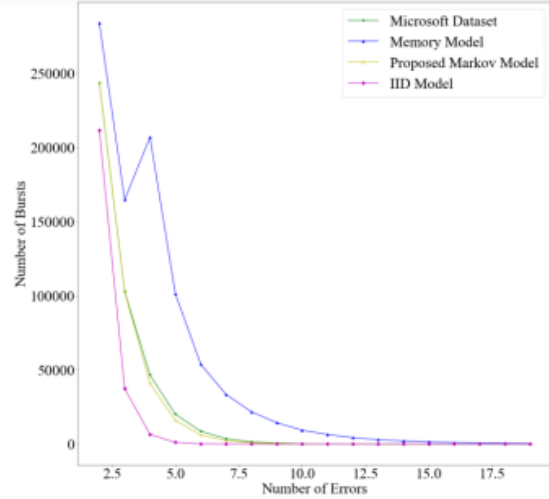


Figure 4: Comparison of errors per burst produced by each model

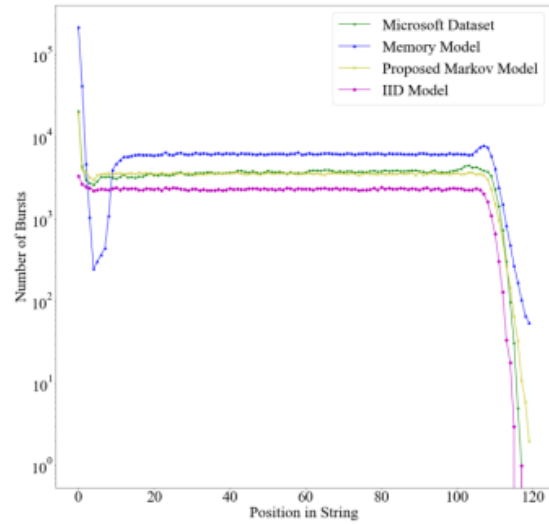


Figure 5: Start positions of bursts produced by each model

In Fig. 3, 4 and 5, it is seen that the proposed Markov model closely mimics the test set when it comes to burst length, errors per burst and burst position. Just after the start of the string, the Memory Model shows fewer bursts than the Microsoft Dataset. This may be due to the large number of bursts predicted by the model at the start of the string. As these bursts continue through the positions just after the start, fewer new bursts can start at these positions. The Memory Model shows a much larger number of bursts throughout the other positions in the strings.

The IID model has fewer bursts as errors are independently distributed in this model. The fact that the Microsoft data has

more bursts than the IID model confirms that errors are not independent.

D. Error Runs

Error runs are defined as multiple errors directly after one another. A run with a length of one indicates a single error. The error runs produced by the different models are compared in Fig. 6, while runs for each type of error are shown in Fig. 7, 8 and 9.

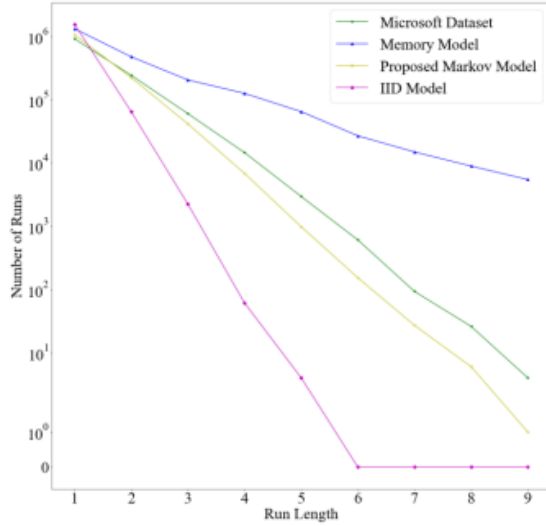


Figure 6: Comparison of error runs produced by each model

In Fig. 6, it is seen that the proposed Markov model mimics the Microsoft dataset closely when producing error runs, although the model produces slightly fewer runs. This is the case for combined errors as well as individual types of errors, as seen in Fig. 7, 8 and 9.

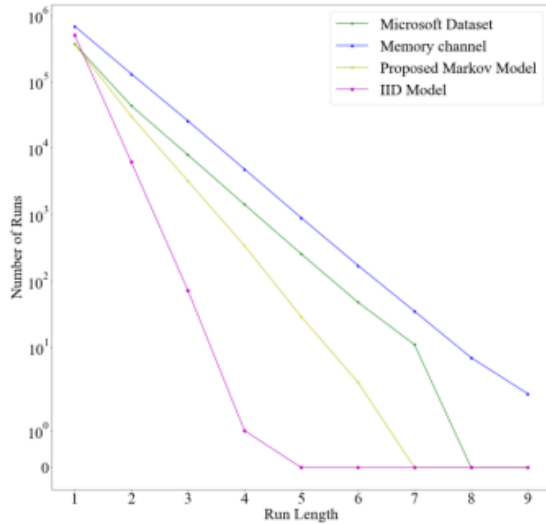


Figure 7: Comparison of substitution error runs produced by each model

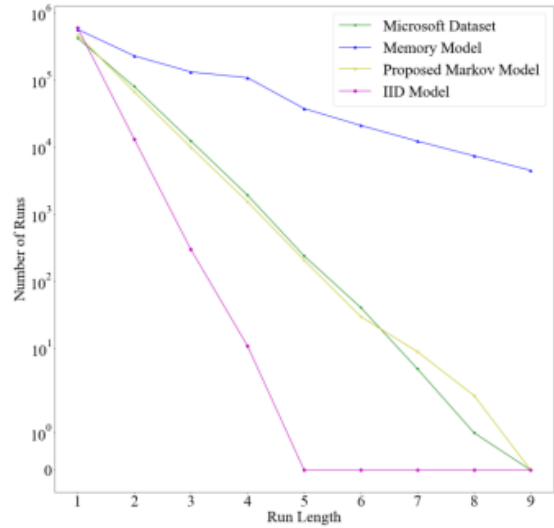


Figure 8: Comparison of deletion error runs produced by each model

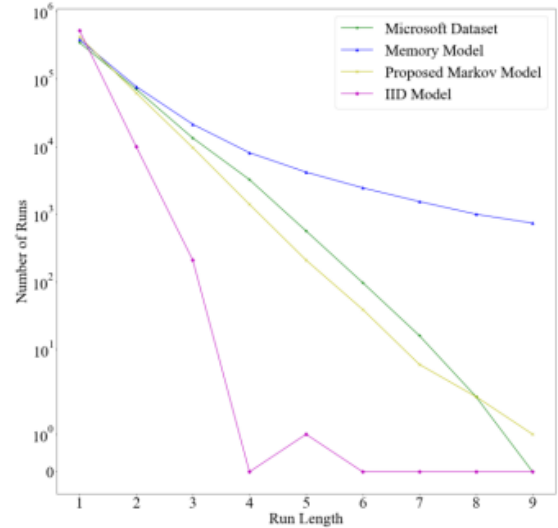


Figure 9: Comparison of insertion error runs produced by each model

Both the IID model and Memory Model produced more single errors. The IID model produced much fewer runs and more individual errors as errors are independently distributed in this model. The Memory Model produced more runs than the testing set, especially runs of longer length. This correlates with the increased error rate of this model.

E. Error Free Runs

Error-free runs are defined as multiple errorless bases directly after one another. A run of length one indicates a single error-free base between two errors.

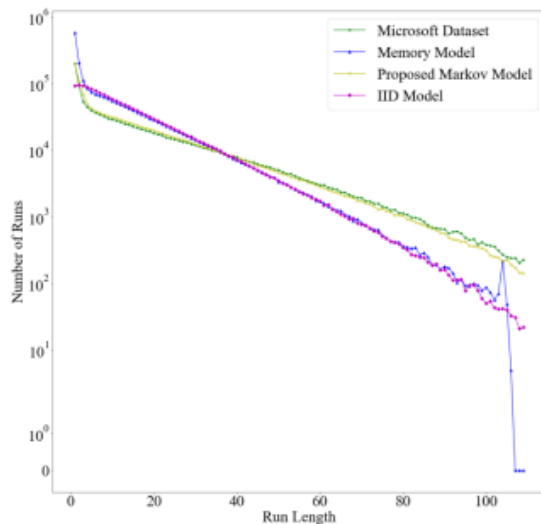


Figure 10: Comparison of error-free run lengths produced by each model

In Fig. 10, it is seen that the proposed Markov model has a similar amount of error-free runs of each length when compared to the Microsoft dataset. There are slightly fewer long runs. This confirms the slightly smaller number of error-free strings produced by the model than there are in the testing data.

It is seen that both the IID model and Memory Model have many more short error-free runs and fewer long error-free runs than the testing set. This confirms the much smaller number of error-free strings produced by the IID model and the zero error-free strings produced by the Memory Model. The IID model has fewer single error-free bases than the Microsoft data, while the Memory Model has more.

V. CONCLUSION

The proposed Markov model provided an error rate and output string lengths similar to the testing set. The distributions of the errors are similar to the Microsoft data set, thereby producing error-free runs and bursts similar to the test set.

The widely used IID model produced much fewer runs and bursts, confirming that errors in DNA strings are not independent of each other.

The complexity of the Memory Model requires large amounts of data to train. Furthermore, the Memory Model does not represent long error runs correctly. Due to its simplicity, the proposed Markov model requires a fraction of the data to implement, while remaining able to produce results closely mimicking the testing set, including long error runs.

While the proposed Markov model closely models many aspects of the Microsoft dataset, future work may include taking into account homopolymer runs and GC-content of the input string.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the financial support of this study by the Telkom CoE at the NWU and Hensoldt GEW.

REFERENCES

- [1] S. R. Srinivasavaradhan, S. Gopi, H. D. Pfister, and S. Yekhanin, "Trellis BMA: Coded Trace Reconstruction on IDS Channels for DNA Storage," in *IEEE International Symposium on Information Theory - Proceedings*, Institute of Electrical and Electronics Engineers Inc., Jul. 2021, pp. 2453–2458. doi: 10.1109/ISIT45174.2021.9517821.
- [2] L. Organick *et al.*, "Probing the physical limits of reliable DNA data retrieval," *Nat Commun*, vol. 11, no. 1, Dec. 2020, doi: 10.1038/s41467-020-14319-8.
- [3] R. Heckel, G. Mikutis, and R. N. Grass, "A Characterisation of the DNA Data Storage Channel," *Sci Rep*, vol. 9, no. 1, Dec. 2019, doi: 10.1038/s41598-019-45832-6.
- [4] S. M. H. T. Yazdi, H. M. Kiah, E. Garcia-Ruiz, J. Ma, H. Zhao, and O. Milenkovic, "DNA-Based Storage: Trends and Methods," *IEEE Trans Mol Biol Multiscale Commun*, vol. 1, no. 3, pp. 230–248, Sep. 2015, doi: 10.1109/TMBMC.2016.2537305.
- [5] T. Coughlin, "DNA Storage Update," <https://www.forbes.com/sites/tomcoughlin/2021/10/28/dna-storage-update/?sh=4fa56c5b2fb7>, Oct. 28, 2021.
- [6] E. G. S. Antonio, T. Heinis, L. Carteron, M. Dimopoulou, and M. Antonini, "Nanopore Sequencing Simulator for DNA Data Storage," in *2021 International Conference on Visual Communications and Image Processing, VCIP 2021 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/VCIP53242.2021.9675388.
- [7] L. Yuan, Z. Xie, Y. Wang, and X. Wang, "DeSP: a systematic DNA storage error simulation pipeline," *BMC Bioinformatics*, vol. 23, no. 1, Dec. 2022, doi: 10.1186/s12859-022-04723-w.
- [8] C. Delahaye and J. Nicolas, "Sequencing DNA with nanopores: Troubles and biases," *PLoS One*, vol. 16, no. 10 October, Oct. 2021, doi: 10.1371/journal.pone.0257521.
- [9] M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum and D. B. Jaffe, "Characterising and measuring bias in sequence data," *Genome Biol*, vol. 14, no. 5, May 2013, doi: 10.1186/gb-2013-14-5-r51.
- [10] Y. Li, R. Han, C. Bi, M. Li, S. Wang, and X. Gao, "DeepSimulator: A deep simulator for Nanopore sequencing," *Bioinformatics*, vol. 34, no. 17, pp. 2899–2908, Sep. 2018, doi: 10.1093/bioinformatics/bty223.
- [11] B. Hamoun, E. Dupraz, L. Conde-Cancencia, and D. Lavenier, "Channel Model with Memory for DNA Data Storage with Nanopore Sequencing," in *2021 11th International Symposium on Topics in Coding, ISTC 2021*, Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/ISTC49272.2021.9594243.
- [12] D. Metzner and J. Ratcliff, "Pattern matching: The gestalt approach," *Dr. Dobbs's Journal*, vol. 7, p. 46, Jul. 1988.
- [13] W. H. Tranter, *Principles of communication systems simulation with wireless applications*. Prentice Hall, 2004. ISBN: 0134947908
- [14] W. Turin, *Performance Analysis and Modeling of Digital Transmission Systems*. in Information Technology: Transmission, Processing and Storage. Boston, MA: Springer US, 2004. doi: 10.1007/978-1-4419-9070-9.

Danie Walker received his B.Eng degree in Computer Electronic Engineering in 2022 at the North-West University and is currently a first-year M.Eng student. His research interests include DNA data storage and channel modelling using traditional and machine learning methods.