Bioinformatics, 37(1), 2021, 29–35
doi: 10.1093/bioinformatics/btaa633
Advance Access Publication Date: 19 July 2020
Original Paper

Sequence analysis

Machine Boss: rapid prototyping of bioinformatic automata

Jordi Silvestre-Ryan , Yujie Wang, Mehak Sharma, Stephen Lin, Yolanda Shen, Shihab Dider and Ian Holmes*

Department of Bioengineering, University of California, Berkeley, CA 94720, USA

*To whom correspondence should be addressed.

Associate Editor: Jinbo Xu

Received on February 18, 2020; revised on June 22, 2020; editorial decision on July 5, 2020; accepted on July 13, 2020

Abstract

Motivation: Many software libraries for using Hidden Markov Models in bioinformatics focus on inference tasks, such as likelihood calculation, parameter-fitting and alignment. However, construction of the state machines can be a laborious task, automation of which would be time-saving and less error-prone.

Results: We present Machine Boss, a software tool implementing not just inference and parameter-fitting algorithms, but also a set of operations for manipulating and combining automata. The aim is to make prototyping of bioinformatics HMMs as quick and easy as the construction of regular expressions, with one-line 'recipes' for many common applications. We report data from several illustrative examples involving protein-to-DNA alignment, DNA data storage and nanopore sequence analysis.

Availability and implementation: Machine Boss is released under the BSD-3 open source license and is available from http://machineboss.org/.

Contact: ihh@berkeley.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Bioinformatics is a field littered with state machines, many of them still functional. The venerable Needleman–Wunsch, Smith–Waterman and Gotoh algorithms from the 1970s and early 1980s can be thought of as aligning pairs of sequences to input–output automata (Gotoh, 1982; Needleman and Wunsch, 1970; Smith and Waterman, 1981). Gribskov's protein profiles of the late 1980s are state machines too (Gribskov *et al.*, 1987). The 1990s saw the probabilistic interpretation of both these kinds of machine as Hidden Markov Models (HMMs); respectively, the 'pair HMM' and the 'profile HMM' (Brown *et al.*, 1993; Durbin *et al.*, 1998). This inspired new applications of HMMs in emerging areas of sequence analysis, such as computational gene prediction (Burge and Karlin, 1997). Further evolution of these ideas including HMMs for multiple sequence alignment (Do *et al.*, 2005; Holmes and Bruno, 2001), comparative genefinding (Alexandersson *et al.*, 2003; Meyer and Durbin, 2004) and phylogenetics (Siepel and Haussler, 2003; Siepel *et al.*, 2006; Suchard and Redelings, 2006) occurred in the 2000s. HMMs (and automata more generally) continue to represent the state of the art for many bioinformatic tasks; for example, when reconstructing the indel histories of ancestral sequences (Holmes, 2017; Löytynoja and Goldman, 2005; Westesson *et al.*, 2012) or aligning protein to DNA (Birney *et al.*, 2004). Meanwhile, the growing field of Deep Learning drew from HMM model-fitting algorithms to train Recurrent Neural Networks (RNNs) with sequential inputs and outputs; specifically, using *Connectionist Temporal Classification* (CTC) which is based on the Forward-Backward algorithm (Graves *et al.*, 2006). Inevitably, RNNs have found application in bioinformatics, sometimes surpassing HMMs; for example, RNN basecallers for nanopore sequence data (Boza *et al.*, 2017) have been shown to outperform the corresponding HMMs (David *et al.*, 2017). Even in cases such as this, where RNNs have displaced HMMs, useful connections to automata theory can sometimes still be made due to the underlying parallels between CTC and HMM dynamic programming (Silvestre-Ryan and Holmes, 2018).

Over this period, a number of software libraries have been developed for parsing, annotating and aligning biological sequences using generic state machines. Examples include Dynamite (Birney and Durbin, 1997), DART (Holmes and Bruno, 2001), GHMM (Schliep *et al.*, 2004), C4 (Slater and Birney, 2005), HMMoC (Lunter, 2007), HMMConverter (Lam and Meyer, 2009), StochHMM (Lott and Korf, 2014), MuxStep (Veličković and Liò, 2016) and ham (Ralph and Matsen, 2016). These various libraries all have slightly different capabilities but typical features include the ability to work with state machines of arbitrary topology, implementations of common dynamic programming algorithms (like the Viterbi and Forward-Backward algorithms) and generation of optimized code implementing those algorithms. Some newer libraries for deep learning that are frequently used in bioinformatics, such as TensorFlow (Abadi *et al.*, 2016), often include implementations of algorithms

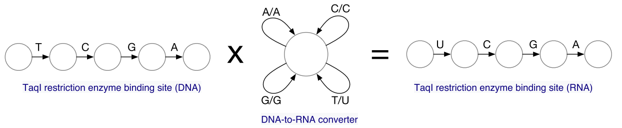
T
C
G
A
TaqI restriction enzyme binding site (DNA)
X
A/A
C/C
G/G
T/U
DNA-to-RNA converter
=
U
C
G
A
TaqI restriction enzyme binding site (RNA)

Fig. 1. A state machine that generates DNA sequences matching the TaqI restriction enzyme binding site (left) can be converted, by multiplication with a DNA-to-RNA conversion machine (center), to a state machine that generates the corresponding RNA motifs (right)

with close state-machine analogs, even though the libraries themselves are not explicitly founded on automata theory. Examples include CTC loss-minimization, or beam search to find the most likely output sequence of a RNN.

Despite all of these libraries for doing automata-based inference on sequences, there are few (if any) general-purpose tools for working with the automata themselves as manipulable mathematical objects. As one illustration of why this is useful, we consider GeneWise, one of the most successful (and elaborate) automata of bioinformatics. The underlying state machine of GeneWise aligns an amino acid sequence to the (unspliced and imperfectly observed) protein-coding genomic DNA. In doing so, it simultaneously models translation, splicing and sequencing errors—as detailed in the GeneWise paper (Birney *et al.*, 2004). The machine developed to do this in full is highly intricate, containing 21 states and 93 transitions when expressed as a Moore (1956) machine; prototyping and subsequent optimization yielded a reduced-size machine with 6 states and 23 transitions. (In using these descriptions to assess the mathematical complexity of GeneWise and the computational complexity of its algorithms, one should note that the transition output labels are strings, not just individual characters, and the transition weights include contributions from all the constituent submodels: translation, splicing and sequencing errors.) GeneWise also includes an algorithm that aligns profile HMMs of protein domains to genomic DNA, accepting HMMER-format profiles as used by the PFAM database (El-Gebali *et al.*, 2019). All these state machines were laboriously developed, validated and debugged by hand (albeit with the help of Dynamite to generate the dynamic programming code once the state machines were specified). In principle—given that the GeneWise model can notionally be 'factorized' into independent component machines for translation, splicing and sequencing error—this approach should be amenable to incremental variations or enhancements; e.g. different profile HMM architectures (as may be found in later releases of HMMER), alternate genetic codes or richer models of the context-dependent error profile of later-emerging sequencing technologies like that of Oxford Nanopore Technologies (ONT) (Jain *et al.*, 2015). In practice, however, because this factorization of the GeneWise state machine into submachines is performed manually, these kinds of upgrade would take a significant amount of work—and quick prototyping would be impossible.

More generally, many bioinformatic automata can be viewed as being derived from simpler state machines by operations such as concatenation, composition (or multiplication), intersection, union (or addition), reversal, complementation, substitution or other well-defined mathematical transformations. This is particularly useful in cases which inherently involve transforming one sequence into another via several steps. GeneWise is one example. Another, very simple, example is the conversion of a DNA motif to an RNA motif, as shown inFigure 1.

A more elaborate example occurs in the context of encoding binary information into DNA as a storage medium: in doing this, it is desirable to avoid repeated nucleotides (which are easily misread by DNA sequencers) and this can be conceived of as converting a binary sequence to a base-3 (ternary) sequence, followed by a conversion from ternary to DNA. Each conversion can be formulated as an input–output state machine (Fig. 2); related coding operations, such as the introduction of parity bits for error correction, can similarly be formulated using state machines (Supplementary Fig. S1).

The approach of formally composing automata is well-documented in other applications of automata in computer science, for example, in linguistics (Mohri *et al.*, 2002). The automata, and the operations to combine or transform them, can be expressed compactly using the notation of linear algebra; in this view, an automaton formally represents an infinite matrix whose rows and columns are indexed by input and output sequences (Bouchard-Côté, 2013). To take one example, the GeneWise combination of three translation, splicing and error submachines corresponds straightforwardly to a three-way matrix multiplication. For some tasks, such as statistical phylogenetic alignment (where such automata generalize the idea of the 'substitution matrix' to whole sequences, allowing indels as well as substitutions), this view of state machines as algebraic objects that can be systematically combined on the branches of a

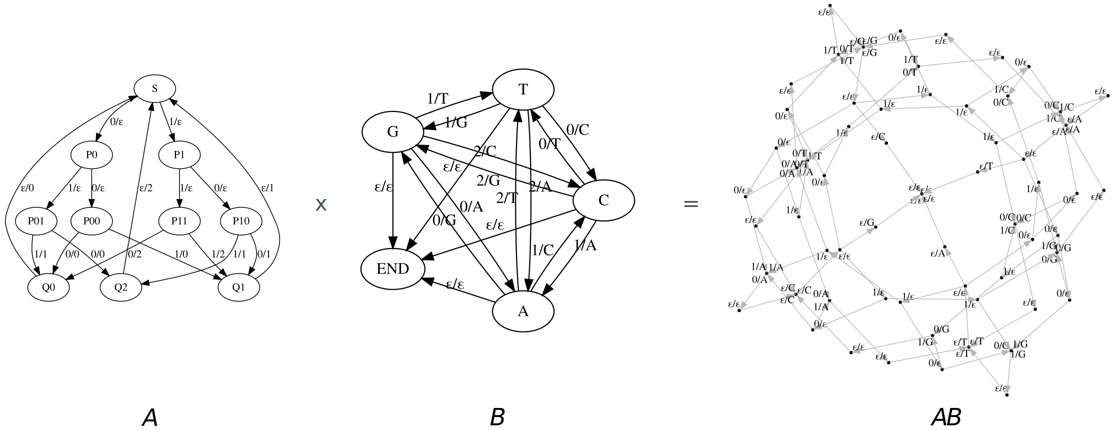

Fig. 2. A non-repeating DNA storage code can be factored into two separate state machines: one for converting binary sequences to ternary, and one for converting ternary to DNA (Goldman *et al.*, 2013). In this diagram, a state machine transition is annotated *x*/*y* if it inputs *x* and outputs *y*; the symbol ϵ denotes the empty string. (**A**) Machine for (imperfectly) converting a binary input sequence into ternary, batching the input into groups of three binary digits and outputting pairs of ternary digits. This machine is inefficient in that the output is $\log(9)/\log(8) \simeq 1.06$ times longer than it would be for a perfect conversion from base 2 to base 3 (because no triplet of input bits is ever converted the pair of output trits '22', which means one of the nine possible output–trit pairs is wasted; more fundamentally, perfect conversion between indivisible integer bases is not possible with a finite state machine). In applications where the length of the input is not known in advance and so must be signaled with an end-of-file character (EOF), the ternary sequence '22' can be used to encode this EOF. (**B**) Machine for converting a ternary input sequence into a non-repeating DNA sequence. The output of this machine is $\log(4)/\log(3) \simeq 1.26$ times longer than the DNA would be if repeated nucleotides are allowed. (**AB**) Machine for a binary input sequence into a non-repeating DNA sequence, obtained by 'multiplying' A and B. The output of this machine is 4/3 times longer than the Shannon limit (obtained by multiplying the inefficiencies of the two constituent machines). The machine diagrams in this figure were generated automatically using Machine Boss and GraphViz

tree is absolutely central to the underlying probabilistic framework (Holmes, 2017; Holmes and Bruno, 2001; Redelings and Suchard, 2005, 2007; Suchard and Redelings, 2006; Westesson *et al.*, 2012). Even without adopting the linear algebraic view, there is clear utility to being able to transform automata by simple operations like reverse-complementation or concatenation. Yet, for all the general-purpose state-machine libraries, this ability to formally operate on the state machines themselves is not generally available. Certainly, most libraries allow state machines to be constructed programmatically, by building appropriate data structures directly in the source code that links to the library. However, this is an intricate and error-prone procedure, and is a far cry from being able to construct state machines from modular components using reliable, general-purpose implementations of elementary operations such as 'multiply', 'concatenate' or 'reverse-complement'.

Motivated by this gap in the bioinformatics tool chain, and finding ourselves repeatedly in need of reference implementations of automata-theoretic algorithms (for prototyping and debugging purposes in both HMM- and RNN-based applications) that allowed for algebraic manipulation of the underlying state machines, we developed Machine Boss, an open source software package that meets this need. In Section 2 and the Supplementary Information, we review the representation of state machines used throughout Machine Boss, and outline its capabilities. In the Results section, we describe non-trivial example applications of Machine Boss to several problems of interest; these include incorporating context-dependent error models (appropriate for nanopore sequencing) into GeneWise-like protein-to-DNA aligners, decoding the output of neural network basecallers for ONT sequencing instruments, and prototyping modular codes for encoding binary information in DNA. In the Discussion, we discuss how this sort of prototyping fits into a bioinformatics tool development workflow, and briefly mention several further applications.

2 Materials and methods

Detailed descriptions of state machines and sequence data may be found in the Supplementary Information to this article.

2.1 Weighted finite-state machines

The following uses notation introduced in the studies by Mohri *et al.* (2002) and Westesson *et al.* (2012).

For our purposes, a *machine* is a tuple $T = (\Omega_I, \Omega_O, \Phi, \tau, \Theta, \Psi, \upsilon)$ where Ω_I is an input alphabet, Ω_O is an output alphabet, Φ is a non-empty ordered list of states (of which the first element is the start state and the last is the end state), $\tau \subseteq \Phi \times (\Omega_I \cup \{\epsilon\}) \times (\Omega_O \cup \{\epsilon\}) \times \Phi$ is a set of transitions between states (labeled with input and/or output symbols), Θ is a set of named parameters (a subset of which are assigned non-negative real values), Ψ is a set of constraints (partitioning Θ into rates, mutually exclusive probability and other parameters) and $\upsilon : \tau \to \Lambda$ is the transition weight function, where Λ represents the set of closed-form differentiable expressions over Θ (with an expression grammar that allows real numbers, arithmetic operators, powers, exponentials and logarithms). Let $\mathcal{M}$ denote the set of all such possible machines.

For a given input sequence $x \in \Omega_I^*$ and output sequence $y \in \Omega_O^*$, let $T_{x,y}$ be the total weight of all paths through the transition graph of T that have input label x and output label y. The sequence weight $T_{x,y}$ can be computed by the Forward algorithm in time $\mathcal{O}(|x| \times |y|)$ and memory $\mathcal{O}(\min(|x|, |y|))$. The derivatives $\frac{\partial T_{x,y}}{\partial \lambda}$ for $\lambda \in \Theta$ can be computed using the Forward–Backward algorithm. (Machine Boss implements this algorithm only for the case when all transition weights can be computed as real values, i.e. all relevant parameters are specified.)

This notation encourages us to think of T as being like an infinite matrix, indexed by sequences, with $T_{x,y}$ being the element in row x and column y. We can then, for example, multiply machines like matrices: if $T, U \in \mathcal{M}$, then we can readily find a machine $TU \in \mathcal{M}$ such that $(TU)_{x,z} = \sum_y T_{x,y} U_{y,z}$. Other matrix expressions such as $T + U$, transpose(T) or αT (for some scalar α) are also straightforward to implement.

A machine for which $\Omega_I = \varnothing$ is called a generator. A machine for which $\Omega_O = \varnothing$ is called a recognizer. We can, for example, think of profile HMMs as generators (because they generate sequences as output) and regular expressions as recognizers (because they accept sequences as input). The labeling of one sequence as 'input' and the other as 'output' is arbitrary and, for many purposes, largely irrelevant. In the linear algebra analogy, the distinction between a generator and a recognizer corresponds to the choice as to whether to represent a vector in row or column form, and exchanging the 'input' and 'output' labels corresponds to taking the transpose.

2.2 Machine boss capabilities

Machine Boss defines a (validatable) JSON format for machines in $\mathcal{M}$, and implements the following operations:

- Matrix-like operations such as multiplication, transposition, addition, intersection (a.k.a. point product), the matrix identity (for a given alphabet) and scalar multiplication;
- String-like operations such as concatenation, reversal, reverse-complement, repetition, Kleene closure, local matching (padding with flanking states),
- HMM transition graph-related operations such as topological sorting, elimination of ϵ-transitions, elimination of redundant or inaccessible states, downsampling, normalization and various probabilistic weightings;
- Constructing generators and recognizers for sequences, elementary patterns (e.g. wildcards) or regular expressions;
- Import of models from various sources such as HMMER files (Eddy, 2009), FASTA, CSV or HTTP fetches from PFAM (El-Gebali *et al.*, 2019) or DFAM (Hubley *et al.*, 2016), and export to GraphViz format;
- Useful built-in 'preset' machines such as probabilistic Smith-Waterman (Bucher and Hofmann, 1996), GeneWise-like models (Birney *et al.*, 2004), DNA storage codes (Goldman *et al.*, 2013), the Jukes-Cantor model (Jukes and Cantor, 1969) and the TKF model (Thorne *et al.*, 1991);
- Implementations of common dynamic programming (Forward, Forward–Backward, Viterbi) and HMM-related algorithms (Baum-Welch and other forms of EM, respecting user-specified constraints), including linear-space Forward/Viterbi (without traceback), and banding heuristics;
- Implementation of search algorithms for finding the highest-weighted input or output sequences, or sampling such sequences probabilistically by weight; including prefix search, beam search, MCMC and simulated annealing;
- Generation of C++ code (32- or 64-bit) and/or JavaScript code;
- Multiple ways to specify input and output sequences (as strings from the command line, JSON arrays or FASTA files);
- A flexible logging system for debugging and progress reports.

To compute the sum over all state paths, the Forward algorithm requires that the 'silent' (i.e. ϵ-labeled) subset of the transition graph is acyclic and topologically sorted (Durbin *et al.*, 1998). Most Machine Boss operations attempt to maintain this property in the state machines that they construct, automatically topo-sorting and eliminating silent cycles by marginalization. However, for large state machines (particularly when the transition weights are expressed symbolically as closed-form algebraic formulae, rather than as real numbers), these operations can become computationally expensive. For such cases, Machine Boss also offers inexact versions of the operations that either attempt to break silent cycles (by deleting silent transitions $i \to j$ where $j < i$, until no cycles remain) or just leave them in place (acknowledging that the Forward algorithm may then give a technically incorrect, albeit stable, result).

With the operations described, prototyping and evaluating new machines with Machine Boss is a relatively quick process that can take place interactively on the command line. In fact, many of these operations can be accessed in multiple ways: from the command line, via the JSON API, or by interfacing directly to the C++ API.

Machine Boss compiles on Apple Mac and Linux systems to a command-line executable with limited dependencies (GSL and SSL if using the network capabilities) and can also be compiled to WebAssembly using emscripten.

3 Results

3.1 Aligning protein sequences to nanopore reads with a context-dependent error model

Our first test of Machine Boss was an experiment to see whether richer error models might benefit a GeneWise-like protein-to-DNA search. Specifically, we sought to prototype an application to search amino acid sequences against individual nanopore reads, to see if they contained coding genes for known *E.coli* proteins. To do this, we combined a protein-to-DNA alignment model with two alternate error models: a 'symmetric context-independent' error model parameterized by a substitution matrix and gap opening/extension parameters, and a richer 'asymmetric context-dependent' error model with separate parameters for insertion and deletion (hence 'asymmetric'), that also allows the error probabilities at a particular position of the genome to depend on the neighboring bases (hence 'context-dependent'). For this application, we were primarily interested in the power of the DNA substitution model; to reduce computation time, we did not incorporate an amino acid substitution model (as the analogous GeneWise algorithm does), but this can easily be incorporated.

The datasets and the construction and parameterization of the constituent state machines are described in more detail in Section 2 and Supplementary Information. The protein-to-DNA model with symmetric context-independent errors has 242 states and 803 transitions (329 IO-conditioned). The model with asymmetric context-dependent errors has 1349 states and 7464 transitions (2558 IO-conditioned).

After constructing the state machines, we used Machine Boss to generate custom C++ code for the Forward algorithm, compiled this code, and ran it to scan for a representative *E.coli* IS26 transposase protein (insB1, 167aa).

Figure 3A shows the results of this experiment. Broadly, both error models performed similarly, scoring on average around 1.4 bits per codon aligned. (This is a rather low alignment score, reflecting the extremely noisy nature of the training alignments.) We observed a small (3%) but significant improvement in log-odds scores for positives when using the asymmetric context-dependent model, with negligible effect on log-odds scores for negatives.

To investigate how much of this improvement arose from the separate insertion and deletion probabilities, we prototyped a third error model, based on the symmetric context-independent model (and having similar state and transition counts), but relaxing the symmetry constraint between insertions and deletions. Results for this model are not shown in Figure 3 but its log-likelihoods for protein-to-DNA alignment generally lie in between the other two models, with a relative improvement of around 1% over the symmetric context-independent model. Thus, the improvement from allowing context-dependence appears to be roughly double the improvement from allowing symmetry-breaking, for this task.

3.2 Aligning protein domain profiles to nanopore reads

In the previous section, we developed a state machine to model nanopore-specific sequencing errors and used that to better align protein sequences to nanopore reads. Machine Boss is able to combine arbitrary state machines, so the previous error models can be easily combined with other bioinformatics state machines. We next sought to investigate whether a richer error model would also benefit a profile HMM search. Instead of aligning a single protein sequence to a nanopore read, we combined a profile HMM with our nanopore-specific error model and aligned it to the same set of nanopore reads from the previous section.

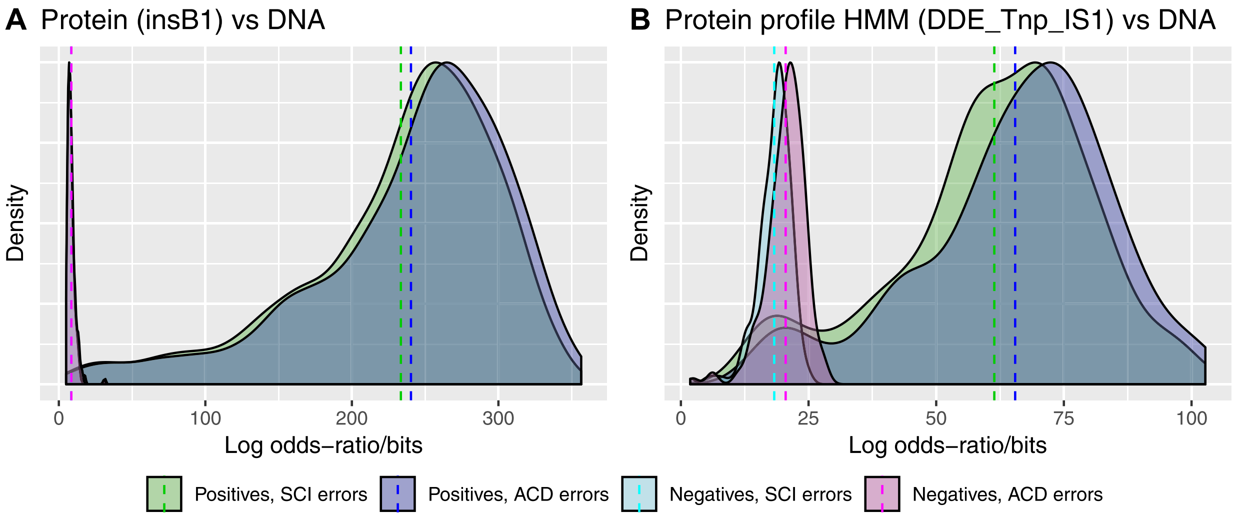

Fig. 3. A richer error model slightly improves the discriminative power of both protein sequence (**A**) and profile HMM (**B**) alignments to noisy sequencing reads. The first plot (A) shows the smoothed density of log-odds ratios of global alignments of *E.coli* protein insB1 to nanopore reads that fully contain a gene for that protein (Positives), versus those that do not contain that protein or close homologs (Negatives). The second plot (B) shows the smoothed density of log-odds ratios of the PFAM domain DDE Tnp IS1 (PF03400) for the IS1 transposase, aligned to nanopore reads that fully contain a gene for the corresponding insB1 protein ('Positives'), versus those that do not contain that protein or close homologs ('Negatives'). These alignments were done using error models with and without insertion/deletion asymmetry and context dependence (SCI = symmetric context-independent; ACD = asymmetric context-dependent). The log-odds ratio for a read is $L = \log_2 \frac{P(\text{read}|\mathcal{H}_1)}{P(\text{read}|\mathcal{H}_0)}$ where $\mathcal{H}_1$ is the hypothesis that the read contains the insB1 gene or domain and $\mathcal{H}_0$ the hypothesis that it does not. The mean of L is indicated for each group with a dashed line. For the sequence-DNA alignment (A), using the asymmetric context-dependent error model increases the mean of the positives ($\Delta L \simeq 6.9$ bits, a relative increase of around 3%) with negligible effect on the negatives ($\Delta L \simeq 0.1$ bits). Similarly, for the profile-DNA alignment (B), using the asymmetric context-dependent error model increases the mean of the positives ($\Delta L \simeq 4.1$ bits) with a smaller effect on the negatives ($\Delta L \simeq 2.2$ bits)

For this experiment, we again focused on the insB1 transposase from the previous example and used the Pfam DDE_Tnp_IS1 domain (accession PF03400), which profiles its catalytic domain. We imported the HMMER-formatted profile HMM from the Pfam database into Machine Boss, and composed it with each of two error models: the symmetric context-independent error model, and the asymmetric context-dependent model. We did not use the asymmetric context-independent model in this experiment. For this experiment, we did not generate custom C++ code and instead relied on Machine Boss's internal Forward algorithm implementation (see Supplementary Information).

Results are shown in Figure 3B. Proceeding from the symmetric context-independent model to the richer asymmetric context-dependent model, we see a relative increase in the log-likelihood of 4.1% for positives and 2.2% for negatives.

3.3 Decoding the most likely output sequence of a neural network basecaller

Our third experiment tests the decoding algorithms used for basecalling on the Oxford Nanopore Technologies (ONT) sequencing platform. As a single strand of DNA (or RNA) passes through the protein nanopore, it perturbs an electrical current signal in a sequence-dependent way. A neural network trained with Connectionist Temporal Classification (CTC, Graves *et al.*, 2006) outputs a probability distribution over sequences, which requires an additional decoding step to find the most likely sequence. CTC was developed for speech recognition and first applied to nanopore sequencing by Chiron (Teng *et al.*, 2018), and was later adopted by various ONT basecallers.

Bonito (https://github.com/nanoporetech/bonito) is ONT's most recent research basecaller: it uses a convolutional architecture based on QuartzNet (Kriman *et al.*, 2020), and is trained with CTC loss. In practice, Bonito uses Viterbi decoding, which simply takes the argmax of the logits and concatenates the resulting nucleotide and gap characters.

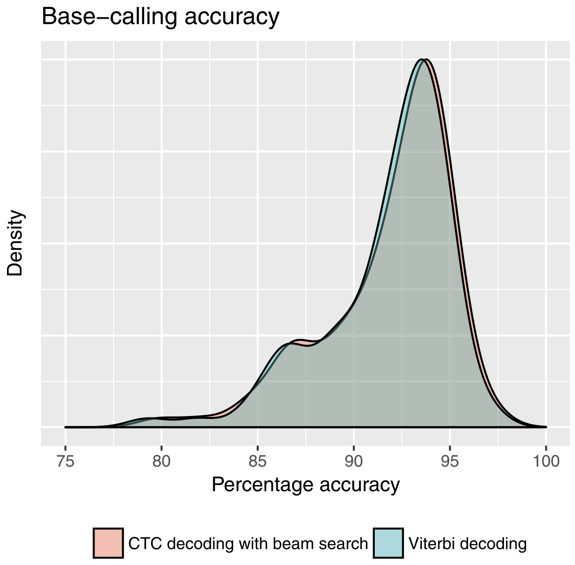

Fig. 4. A beam-search decoding of the maximum likelihood sequence of Oxford Nanopore's Bonito basecaller slightly outperforms a Viterbi best-path decoding on a sample of 100 *Klebsiella pneumoniae* reads. The percent accuracy is defined as the number of identities in the alignment divided by the total alignment length. Median accuracy with Viterbi was 92.8% while beam search yielded a median accuracy of 93.0%. This slight increase in accuracy does incur a computational cost: the beam search (width of 5) takes roughly 1.25 times as long as the Viterbi decoding. We further observe that a bespoke Python implementation of Viterbi decoding (optimized for this model architecture) was roughly five times as fast as Machine Boss's generic C++ implementation of Viterbi decoding (which spends most of its time constructing and topo-sorting the state machine). This reinforces the conclusion that Machine Boss is better suited to development-stage prototyping, than to computationally intensive end-user applications

In this test, we compare the use of a Viterbi decoding scheme, which just finds the single most probable path through the data, with a beam search, a heuristic search algorithm which looks for the best label sequence. The CTC probability outputs are similar to profile HMMs and as such can be interpreted as state machines (Silvestre-Ryan and Holmes, 2018). We evaluated these algorithms on a small sample of 100 reads from a publically available R9.4 *Klebsiella pneumoniae* dataset (Wick *et al.*, 2019). Accuracy was evaluated by aligning basecalled reads with minimap2 (Li, 2018) to a reference genome from the same study. Each read was basecalled with a version of Bonito modified to save the network output, which was then loaded as a state machine into Machine Boss using the recognize-merge-csv option, which constructs a state machine that merges repeated characters in the same manner as the CTC loss, as described in (Graves *et al.*, 2006).

Results are shown in Figure 4. We found that the beam search yielded an increase of 0.2% median accuracy over Viterbi, though in practice its greater computational cost would likely not be worth such a slight improvement. These results were obtained with a beam width of 5; a larger beam size of 50 did not noticeably improve the results.

In addition to the decoding of single reads, more elaborate dynamic programming algorithms for consensus decoding (Silvestre-Ryan and Holmes, 2018) can also easily be implemented and tested in Machine Boss. In this case, performance is too slow for practical application to large datasets, though these reference implementations can be used to debug domain-specific software. In our case, Machine Boss has helped with testing our own consensus basecalling software PoreOver (https://github.com/jordisr/poreover, Silvestre-Ryan and Holmes, 2020).

3.4 Constructing a repeat-avoiding code for DNA data storage

Our last computational experiment is motivated not by sequence analysis, but by analysis of the state machines themselves. We sought to investigate the complexity of error-tolerant codes for storing information in DNA.

We developed a state machine for converting binary information to DNA sequences using, as a starting point, the DNA storage code developed by Goldman *et al.* (2013). The central idea of this code is that DNA homopolymers are often misread by many sequencing technologies, so this class of errors can be avoided altogether by never repeating a base in the encoded sequence. This leaves three nucleotides available for encoding information at any given position in the sequence, which corresponds to a radix-3 (ternary) representation.

We implement this as a multiplication of two machines: one that converts binary to ternary (with some unavoidable inflation of the message size), and a second machine that converts ternary to non-repeating DNA. This configuration is shown in Figure 2.

Using Machine Boss, we were able to successfully prototype this machine and to confirm that it accurately encodes binary messages to non-repeating DNA strings and decodes in the opposite direction, with the ratio of binary to DNA message lengths asymptotically approaching the expected limit of 3/2. This ratio is calculated as follows. The conversion of binary to ternary is approximate, batching the input bits into triplets (with $2^3 = 8$ possibilities per batch) and outputting trits—i.e. ternary digits—in pairs (with $3^2 = 9$ possibilities per batch). Thus, the input sequence has 3/2 as many characters as the output. The ternary-to-DNA conversion converts each ternary digit to a single nucleotide, so the overall input/output ratio for the full binary-to-DNA conversion is also 3/2.

This is slightly wasteful given that the Shannon information content of a non-repeating DNA sequence is $\log_2 3 \simeq 1.58$ bits/symbol, slightly greater than 3/2. The wastage is incurred by the batched binary-to-ternary conversion, since there are more output possibilities than input possibilities for each batch. As can be seen in machine A of Figure 2, there is no triplet of input bits that will ever output the pair of trits '22'. This reflects a more general phenomenon that a finite-state machine cannot perfectly convert a radix-2 input to a radix-3 output (essentially, it can only compute the last digit of this conversion, which amounts to dividing the input by 3 and outputting the remainder; the quotient must then be fed into a similar machine to compute the second-last digit, and so on). It is possible to get quite close to the limit, though, by batching input bits in this way, and a batch size of 3 bits is a reasonably efficient compromise in terms of the number of states required by the machine; no improvement can be gained by improving the batch size until one reaches 11 bits, whereupon the ratio of input/output message lengths is $11/7 \simeq 1.57$, but this requires $\mathcal{O}(2^{11})$ states to track each batch. Such a machine can readily be prototyped with Machine Boss, but the algorithms to manipulate and use the state machines become quite cumbersome for large machines (in addition to the well-known time complexity of dynamic programming to state machines, Machine Boss performs operations like topological sort and state elimination that can be slow for very large machines).

The input/output ratio of 3/2 is approached asymptotically from below, because there is a necessary overhead involved in encoding the message length itself; our machine encodes this using the otherwise-unused pair of output trits '22' as an end-of-message terminator sequence. For simplicity, this mechanism is not included in Figure 2; when it is included, the combined machine for binary-to-non-repeating-DNA conversion has 85 states and 132 transitions (44 IO-conditioned). The component machines were constructed with a short JavaScript program, and are available as presets in Machine Boss.

We can readily extend the above-described approach to study more elaborate DNA storage codes. For example, we can develop a DNA-encoding machine that avoids not just repeated nucleotides in the output, but also avoids certain nucleotide motifs, such as restriction enzyme sites; briefly, the transition graph of such a machine can be found by starting with a de Bruijn graph over k-mers, from which the prohibited k-mers are then deleted. (Of course, restriction enzyme sites that contain repeated nucleotides would already be excluded.) We might also incorporate error-correction units, such as Hamming codes or indel-resistant 'watermarks'. Finally, we can incorporate technology-specific models of sequencing error, such as the nanopore error models described in previous sections, when decoding messages. All these variations can be implemented as modular machines and factored into the 'matrix multiplication' of Figure 2. For example, introducing a Hamming(7,4) error-correcting parity code (Supplementary Fig. S1) to the non-repeating-DNA code (Fig. 2) yields a machine with 1365 states and 1812 transitions (292 IO-conditioned), whose input/output ratio is $\frac{4}{7} \times \frac{3}{2} = \frac{6}{7}$. A deeper exploration of these ideas, using state machines to prototype the codes and investigating their error-correcting properties by simulation, is available in a separate preprint (Holmes, 2016).

4 Discussion

Machine Boss can be useful for prototyping, testing, and theoretical analysis of state machines. In most cases, it is not suitable for developing polished bioinformatics tools, since further heuristic or custom optimizations of the generated state machines and code (beyond Machine Boss's automated capabilities) is often possible.

As an example of this further optimization, our context-dependent error model has 50 states: a start state, an end state, and 48 states which consist of match, insert, and delete states repeated in 16 different flanking contexts. However, it is unnecessary to allocate storage for all 50 states during dynamic programming: the flanking context is always exactly determined by the position in the input genomic sequence, so only 5 states are ever accessible at any position in the dynamic programming matrix. An optimized implementation could make use of this, but Machine Boss currently lacks the sophistication to deduce such optimizations automatically. Rather, Machine Boss can be used (as we have done here) to evaluate whether such development is worthwhile, and to provide a robust reference implementation against which the results of a more optimized version can be checked.

Another application involves nanopore basecalling. The outputs of deep learning basecallers can be interpreted as machine transition weights (Jain *et al.*, 2018; Wick *et al.*, 2019). In building on these results, we have found Machine Boss useful as a debugging and profiling tool (Silvestre-Ryan and Holmes, 2018).

Compared to recent deep learning approaches, automata retain some merits: they are highly interpretable, conceptually straightforward and generally predictable. The interpretability is especially appealing when paths through the automaton have clear meaning—as is the case when state machines are used to represent biological processes such as translation and splicing, information-theoretic processes like radix-based coding, or evolutionary processes such as indels (for which purpose Machine Boss includes a reference implementation of the Thorne–Kishino–Felsenstein model, Thorne *et al.*, 1991). The software development was motivated directly by these cases, but the algorithms implemented are general enough that we have been able to use it for applications in nanopore analysis as well. The README file in the Machine Boss repository describes several further applications, including machines to search for a PROSITE regular expression in a protein sequence and to count copies of this motif in a (translated) DNA sequence. As with the examples in this article, the power of this approach rests on the ability to combine such state machines in a general way, together with new machines as yet undeveloped.

Acknowledgements

The authors thank Ewan Birney and Tim Massingham for informative discussions.

Funding

This work was supported by the National Institutes of Health [U24-CA220441, R01-HG004483 and R01-GM080203], and by a research gift from Oxford Nanopore Technologies.

Conflict of Interest: none declared.

References

Abadi, M. et al (2016) Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467.

Alexandersson,M. *et al.* (2003) SLAM: cross-species gene finding and alignment with a generalized pair hidden Markov model. *Genome Res.*, **13**, 496–502.

Birney,E. and Durbin,R. (1997) Dynamite: a flexible code generating language for dynamic programming methods used in sequence comparison. In: Gaasterland,T. *et al.* (eds.) *Proceedings of the Fifth*. AAAI Press, Menlo Park, CA, pp. 56–64.

Birney,E. *et al.* (2004) GeneWise and GenomeWise. *Genome Res.*, **14**, 988–995.

Bouchard-Côté,A. (2013) A note on probabilistic models over strings: the linear algebra approach. *Bull. Math. Biol.*, **75**, 2529–2550.

Boza,V. *et al.* (2017) DeepNano: deep recurrent neural networks for base calling in MinION nanopore reads. *PLoS ONE*, **12**, e0178751.

Brown,M. *et al.* (1993) Using Dirichlet mixture priors to derive hidden Markov models for protein families. In: Hunter,L. *et al.* (eds.) *Proceedings of the First*. AAAI Press, Menlo Park, CA, pp. 47–55.

Bucher,P. and Hofmann,K. (1996) A sequence similarity search algorithm based on a probabilistic interpretation of an alignment scoring system. In: States,D.J. *et al.* (eds.) *Proceedings of the Fourth*. AAAI Press, Menlo Park, CA, pp. 44–51.

Burge,C. and Karlin,S. (1997) Prediction of complete gene structures in human genomic DNA. *J. Mol. Biol.*, **268**, 78–94.

David,M. *et al.* (2017) Nanocall: an open source basecaller for Oxford Nanopore sequencing data. *Bioinformatics*, **33**, 49–55.

Do,C.B. *et al.* (2005) ProbCons: probabilistic consistency-based multiple sequence alignment. *Genome Res.*, **15**, 330–340.

Durbin,R. *et al.* (1998) *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK.

Eddy,S.R. (2009) A new generation of homology search tools based on probabilistic inference. *Genome Inform.*, **23**, 205–211.

El-Gebali,S. *et al.* (2019) The Pfam protein families database in 2019. *Nucleic Acids Res.*, **47**, D427–D432.

Goldman,N. *et al.* (2013) Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, **494**, 77–80.

Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.

Graves,A. *et al.* (2006) Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, ACM, New York, NY, USA, pp. 369–376.

Gribskov,M. *et al.* (1987) Profile analysis: detection of distantly related proteins. *Proc. Natl. Acad. Sci. USA*, **84**, 4355–4358.

Holmes,I. (2016) Modular non-repeating codes for DNA storage. arXiv:1606.01799.

Holmes,I.H. (2017) Historian: accurate reconstruction of ancestral sequences and evolutionary rates. *Bioinformatics*, **33**, 1227–1229.

Holmes,I. and Bruno,W.J. (2001) Evolutionary HMMs: a Bayesian approach to multiple alignment. *Bioinformatics*, **17**, 803–820.

Hubley,R. *et al.* (2016) The Dfam database of repetitive DNA families. *Nucleic Acids Res.*, **44**, D81–D89.

Jain,M. *et al.* (2015) Improved data analysis for the MinION nanopore sequencer. *Nat. Methods*, **12**, 351–356.

Jain,M. *et al.* (2018) Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nat. Biotechnol.*, **36**, 338–345.

Jukes,T.H. and Cantor,C. (1969) Evolution of protein molecules. In: Munro,H.N. (ed) *Mammalian Protein Metabolism*. Academic Press, New York, pp.21–132.

Kriman,S. *et al.* (2020) Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions. Int. Conf. Acoust. Spee., pp. 6124–6128.

Lam,T.Y. and Meyer,I.M. (2009) HMMCONVERTER 1.0: a toolbox for hidden Markov models. *Nucleic Acids Res.*, **37**, e139.

Li,H. (2018) Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**, 3094–3100.

Lott,P.C. and Korf,I. (2014) StochHMM: a flexible hidden Markov model tool and C++ library. *Bioinformatics*, **30**, 1625–1626.

Löytynoja,A. and Goldman,N. (2005) An algorithm for progressive multiple alignment of sequences with insertions. *Proc. Natl. Acad. Sci. USA*, **102**, 10557–10562.

Lunter,G. (2007) HMMoC—a compiler for hidden Markov models. *Bioinformatics*, **23**, 2485–2487.

Meyer,I.M. and Durbin,R. (2004) Gene structure conservation aids similarity based gene prediction. *Nucleic Acids Res.*, **32**, 776–783.

Mohri,M. *et al.* (2002) Weighted finite-state transducers in speech recognition. *Comput. Speech Lang.*, **16**, 69–88.

Moore,E.F. (1956) Gedanken-experiments on sequential machines. In: Shannon, C. and McCarthy, J. (eds) *Automata Studies*. Princeton University Press, Princeton, NJ, pp. 129–153.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Ralph,D.K. and Matsen,F.A. (2016) Consistency of VDJ rearrangement and substitution parameters enables accurate B cell receptor sequence annotation. *PLoS Comput. Biol.*, **12**, e1004409.

Redelings,B.D. and Suchard,M.A. (2005) Joint Bayesian estimation of alignment and phylogeny. *Syst. Biol.*, **54**, 401–418.

Redelings,B.D. and Suchard,M.A. (2007) Incorporating indel information into phylogeny estimation for rapidly emerging pathogens. *BMC Evol. Biol.*, **7**, 40.

Schliep,A. et al (2004) The general hidden Markov model library: Analyzing systems with unobservable states. Proceedings of the Heinz-Billing-Price, 2004, 121–135.

Siepel,A. and Haussler,D. (2003) Phylogenetic estimation of context-dependent substitution rates by maximum likelihood. *Mol. Biol. Evol.*, **21**, 468–488.

Siepel,A. *et al.* (2006) New methods for detecting lineage-specific selection. In: A. Apostolico,C. et al (eds) *Research in Computational Molecular Biology*. Springer, Berlin, Heidelberg, pages 190–205.

Silvestre-Ryan,J. and Holmes,I. (2018) Consensus Decoding of Recurrent Neural Network Basecallers. In: Jansson,J. et al (eds) Algorithms for Computational Biology. AlCoB 2018. Springer, Cham. pp. 128-139.

Silvestre-Ryan,J. and Holmes,I. (2020) Pair consensus decoding improves accuracy of neural network basecallers for nanopore sequencing. bioRxiv 2020.02.25.956771.

Slater,G.S. and Birney,E. (2005) Automated generation of heuristics for biological sequence comparison. *BMC Bioinformatics*, **6**, 31.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Suchard,M.A. and Redelings,B.D. (2006) BAli-Phy: simultaneous Bayesian inference of alignment and phylogeny. *Bioinformatics*, **22**, 2047–2048.

Teng,H. *et al.* (2018) Chiron: Translating nanopore raw signal directly into nucleotide sequence using deep learning. *GigaScience*, *7(5)* giy037.

Thorne,J.L. *et al.* (1991) An evolutionary model for maximum likelihood alignment of DNA sequences. *J. Mol. Evol.*, **33**, 114–124.

Veličković,P. and Liò,P. (2016) Muxstep: an open-source C ++ multiplex HMM library for making inferences on multiple data types. *Bioinformatics*, **32**, 2562–2564.

Westesson,O. *et al.* (2012) Accurate reconstruction of insertion-deletion histories by statistical phylogenetics. *PLoS One*, **7**, e34572.

Wick,R.R. *et al.* (2019) Performance of neural network basecalling tools for Oxford Nanopore sequencing. *Genome Biol.*, **20**, 129.