Efficient Mixed-Order Hidden Markov Model Inference

Ludwig Schwardt

Dissertation presented for the degree of Doctor of Philosophy (Electronic Engineering) at the University of Stellenbosch

Promoter: Prof. J.A. du Preez

December 2007

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: . . . .   Date: . . . .
L.C. Schwardt

Abstract

Higher-order Markov models are more powerful than first-order models, but suffer from an exponential increase in model parameters with order, which leads to data scarcity problems during training. A more efficient approach is to use mixed-order Markov models, which model data sequences with contexts of different lengths.

This study proposes two algorithms for inferring mixed-order Markov chains and hidden Markov models (HMMs), respectively. The basis of these algorithms is the prediction suffix tree (PST), an efficient representation of a mixed-order Markov chain.

The smallest encoded context tree (SECT) algorithm constructs PSTs from data, based on the minimum description length principle. It has no user-specifiable parameters to tune, and will expand the depth of the resulting PST as far as the data set allows it, making it a self-bounded algorithm. It is also faster than the original PST inference algorithm.

The hidden SECT algorithm replaces the underlying Markov chain of an HMM with a prediction suffix tree, which is inferred using SECT. The algorithm is efficient and integrates well with standard techniques.

The properties of the SECT and hidden SECT algorithms are verified on synthetic data. The hidden SECT algorithm is also compared with a fixed-order HMM training algorithm on an automatic language recognition task, where the resulting mixed-order HMMs are shown to be smaller and train faster than the fixed-order models, for similar classification accuracies.

Opsomming

Hoër-orde Markov-modelle is kragtiger as eerste-orde modelle, maar hul parametertelling neem eksponensieel met hul orde toe, wat tot probleme met dataskaarsheid lei tydens afrigting. 'n Meer effektiewe benadering is om gemengde-orde Markov-modelle aan te wend, wat datasekwensies modelleer met kontekste van verskillende lengtes.

Hierdie studie stel twee nuwe algoritmes voor vir die afrigting van gemengde-orde Markov-kettings en verskuilde Markov-modelle (HMM's), onderskeidelik. Die basis van hierdie algoritmes is die probabilistiese agtervoegsel-boom (PST), 'n effektiewe voorstelling van 'n gemengde-orde Markov-ketting.

Die kleinste geënkodeerde konteksboom (SECT) algoritme konstrueer PST's vanaf data, gebaseer op die beginsel van minimum beskrywingslengte. Dit het geen gebruiker-verstelbare parameters nie en is self-begrens, aangesien dit die diepte van die uiteindelike PST uitbrei so ver as wat die data dit toelaat. Dit is ook vinniger as die oorspronklike PST afrigtingsmetode.

Die verskuilde SECT algoritme vervang die onderliggende Markov-ketting van 'n HMM met 'n PST wat afgerig word met die SECT algoritme. Die algoritme is effektief en integreer goed met bestaande metodes.

Die eienskappe van die SECT en verskuilde SECT algoritmes word bevestig op sintetiese data. Die verskuilde SECT algoritme word ook vergelyk met 'n vaste-orde HMM afrigtingsmetode op 'n outomatiese taalherkenningstaak, waar die resulterende gemengde-orde HMM's kleiner is en vinniger afrig as die vaste-orde modelle, vir soortgelyke klassifikasie-akkuraathede.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- Johan du Preez, for supervising me all throughout three degrees and guiding me with patience and insight;
- The rest of my colleagues at the Electrical & Electronic Engineering Department, for being such nice people and creating a stimulating environment;
- The students of the DSP Lab, for being such a unique and fun bunch;
- My Aikido, photography and climbing buddies, for providing much-needed distraction;
- My friends, who goaded me on and kept the faith when I didn't feel up to the challenge, and patiently listened to me saying, "I'm nearly done"—many thanks to Albert, Casper, Chris, Christelle, Colin, Dirk, Emli-Mari, Emmerentia, Gerhard, Goof, Jaco Badsout, Jaco Klim, Jaco Satelliet, Jan, Jobie, Juan, Konrad, Knut, Naomi, Nelius, Nicky, Robert, Rudolph, Sintiche, Tammy, Theron, Werner, Zoë, and the rest that slipped my mind;
- Ilze, for her love and endless patience;
- Wilhelm and Eckhard, for being great brothers and finishing before me, even though I started first;
- My parents, for their love and support all these years, even when they are not sure what I'm up to.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sequential data are ubiquitous in many problem domains, including speech processing, natural language processing, bioinformatics and financial time-series modelling. One of the most successful statistical models of this type of data is the Markov model, in which the probability of observing an element or symbol in the sequence only depends on a finite number of preceding symbols. While this assumption restricts its modelling capabilities, it also makes the Markov model highly tractable and efficient. The standard Markov model for discrete-valued data is the Markov chain [1], while the hidden Markov model (HMM) [2] is a more powerful extension that is especially useful for modelling discrete-time sequences of continuous-valued data.

The number of preceding symbols that influence the probability of the next symbol is known as the *order* of the Markov model. While Markov models are typically first-order in practice, higher-order Markov chains are also found in various guises, such as the n-gram language model [3] of natural language processing. Training these models from data is straightforward, as the model structure is fixed once the model order is selected. On the other hand, these models are generally very large, because the number of parameters increases exponentially with the order, and thus require large amounts of training data. They are also structurally poor [4], which means that their complexity increases in large discrete steps with increasing order.

These defects are addressed by *mixed-order* Markov chains, which allow the model order to vary, depending on the specific symbol sequence (or *context*) preceding the symbol to be predicted. While the contexts in a standard (fixed-order) Markov chain all have the same length, which is equal to the model order, the mixed-order Markov chain has contexts of different lengths. This frequently allows the mixed-order model to be more compact for similar performance, as many data sets contain mixed-order dependencies. A good example of such a data set is natural language text. The disadvantage of mixed-order models is that they require more complex training algorithms to infer their model structures from data. Mixed-order Markov models are also referred to as *variable-order*, *variable-memory-length* or *variable-length* models, and we will use these terms interchangeably.

Variable-order Markov chains have been successfully used for text compression [5, 6], natural language modelling [7, 8, 9, 10, 11], reinforcement learning [12], motion tracking from video and human gesture recognition [13, 14], analysing user navigation on the web [15], high-frequency financial time-series modelling [16, 17, 18], and bioinformatics [19, 20, 21].

In this study we examine the prediction suffix tree (PST) [22], a mixed-order Markov model that explicitly maintains a tree of variable-length symbol contexts, which allows efficient training from data. Since its inception in [23], many PST inference algorithms have been proposed [24, 25, 26, 20, 27], which testify to its popularity. Other examples of mixed-order Markov chains (not considered in this study) include[1] the variable-length Markov chain (VLMC) [4, 32], context tree weighting (CTW) [33, 6], multigrams [34, 35], utile suffix memory [36], hierarchical sparse n-grams [37], and fractal prediction machines [38].

Hidden Markov models have the same issues with model order as Markov chains do. Most HMMs are first-order in practice, and the literature on higher-order HMMs is sparse [39]. Mixed-order HMMs have received even less attention. They promise to provide more compact higher-order models with similar performance to fixed-order HMMs. As the model order is an aspect of the topology of the HMM, algorithms for HMM topology inference, such as [40, 41, 42, 43], are also relevant when considering mixed-order models.

[1]The *mixed-memory Markov model* (or mixture transition distribution model) of [28, 29, 30] approximates an Lth-order Markov model as a weighted mixture of L non-adjacent first-order models, which drastically reduces the number of model parameters. It is also referred to as a *mixed-order Markov model* in [31], in reference to the weighted mixture in the model, but it is a constrained fixed-order Markov chain according to our definitions.

1.1 Research objectives

This work aims

- to improve existing inference algorithms for mixed-order Markov chains,
- to find an efficient inference algorithm for mixed-order HMMs,
- and to demonstrate the advantages of mixed-order HMMs over fixed-order models on a non-trivial problem.

1.2 Overview of research

The focus of this study is on inference algorithms for mixed-order Markov chains and HMMs. This section summarises the algorithms and experiments that are described in more detail in the rest of this dissertation.

1.2.1 The smallest encoded context tree (SECT) algorithm

The original PST inference algorithm, Learn-PSA [22], has several drawbacks. It has five user-specifiable parameters that control the expansion of the context tree, of which one is a limit on the model order (or tree depth). If the problem domain does not suggest suitable values for these parameters, they are typically determined using an expensive cross-validation step on held-out data. While the algorithm is memory-efficient, its computational complexity can also be improved.

The *smallest encoded context tree* (SECT) algorithm proposed in this study is one of the first improvements on Learn-PSA. It is introduced in [44] and described in detail in Chapter 4.

The algorithm removes the need for tunable parameters by invoking the *minimum description length* (MDL) principle [45, 46, 47]. This principle states that the model that results in the smallest combined description length of the model itself and a data set encoded in terms of the model is to be preferred, as it represents a good compromise between the complexity of the model and the accuracy with which it describes the data set. It is therefore a mathematical rendition of the scientific principle of *Occam's razor*, which states that models should only be as complex as is required to explain some phenomenon, and no more. The description length is frequently expressed in bits, as the MDL approach is closely related to the information-theoretic concept of lossless compression. MDL is also related to maximum a posteriori (MAP) estimation, where the model description length serves as a model prior.

The SECT algorithm uses a heuristic compact encoding for the PST structure, while the description length of the data set in terms of the PST is provided by the standard negative log probability (or Shannon information content) of the data. The amount of data required to infer a given PST structure is minimised by making the encoding of the PST as compact as possible while still ensuring unique decodability. The algorithm also does not impose a prior limit on the model order or PST depth, which will expand as far as the data set allows it.

SECT uses an efficient recursive procedure to infer the PST. It starts with an empty tree and recursively adds nodes that can potentially improve the code length provided by their parent nodes. The tree expansion policy is liberal, because while a specific node might not be an improvement on its parent, some of its children might. A subtree is only kept if it results in a smaller code length for the data associated with it than the code length provided by its parent node, even after including the overhead of specifying the subtree structure.

The sufficient statistics driving most PST inference algorithms are the counts of symbols following particular context strings in the training data sequence. These *next-symbol counts* are used to measure the similarity between parent and child nodes, and to estimate the next-symbol probability distributions in the final model. Obtaining these counts also represents the major computational hurdle in most of these algorithms. SECT speeds up the counting process by sorting a set of symbol pointers according to the symbol contexts preceding each symbol, as part of the recursive tree-building procedure. When counting the symbols following a specific symbol context, only the symbols that actually follow that context have to be considered. This improves on the computational complexity of Learn-PSA, while remaining reasonably memory-efficient.

1.2.2 The hidden SECT algorithm

Many algorithms for inferring the topology of a hidden Markov model (for example [40, 41, 42, 43]) merge or split its states in a greedy iterative fashion until some measure of model fitness stops increasing. This study follows a more global approach, by focusing on the Markov chain underlying the HMM instead. As explained in Chapter 2, this Markov chain models sequences of observation density indices, or *symbols*. The HMM topology is directly reflected in the structure of this Markov chain, which can be learnt from an estimated symbol sequence using any variable-length Markov chain inference algorithm. A suitable symbol sequence can be obtained from the training data using the Viterbi algorithm [48, 49].

In this context, we propose the *hidden smallest encoded context tree* (hidden SECT) algorithm, which is introduced in [50] and described in detail in Chapter 5. It replaces the underlying Markov chain of an HMM with a prediction suffix tree, which has equivalent modelling capabilities but is more amenable to structure learning. This PST is inferred by the SECT algorithm from the optimal (Viterbi) symbol sequence associated with the training observation sequence.

The hidden SECT algorithm is closely related to standard Viterbi re-estimation [51]. It simply adds a structure learning step to the maximisation ("M") step of this algorithm, while the expectation ("E") step and the estimation of the observation densities and transition probability values are performed in the usual way. The algorithm therefore has little computational overhead compared to standard training methods, as the E step dominates the computational complexity in typical problems. It can be seen as approximate maximum a posteriori (MAP) estimation of the HMM, where the prior on model structure is provided by the MDL terms in the SECT algorithm.

Hidden SECT can be provided with a training schedule that gradually increases the maximum model order during the iterations of Viterbi re-estimation. By first training models constrained to lower orders and then using them to initialise the training of higher-order models, the optimisation process is hopefully steered towards better local optima. This is reminiscent of the FIT algorithm for training high-order HMMs [39], and deterministic annealing [52].

1.2.3 Experiments

1.2.3.1 Synthetic experiments

In order to gain a better understanding of the properties of the SECT and hidden SECT algorithms, the techniques are first applied to synthetic data generated from known Markov models. In the case of the SECT algorithm, a true PST with specified parameters generates a symbol sequence, and SECT is used to infer a PST from this sequence. The structure of the inferred PST can be directly verified against that of the true PST. The true and inferred PSTs are also compared on other criteria, such as the number of states and the (true) Kullback-Leibler divergence rate between the models, which can be determined analytically for ergodic PSTs [53]. To our knowledge, this is the first study that compares mixed-order Markov models using this analytic expression of the true divergence rate. Various experiments examine the effect of the parameters of the true PST on its inference, by modifying the perplexity of the next-symbol distributions and the number of symbols in the alphabet, and corrupting the symbols before inference.

In the case of the hidden SECT algorithm, a simple hidden Markov model with one-dimensional Gaussian observation densities and a specified ergodic structure generates an observation sequence. The hidden SECT algorithm infers an HMM from this sequence, which is compared to the true HMM on a variety of criteria. The HMM structure cannot be verified directly, as the true and inferred HMMs will in effect have different alphabets if their observation densities differ. Similar problems arise when considering the Kullback-Leibler divergence rate between two HMMs. This is estimated instead as the average log-likelihood ratio of the two models calculated on a large test set generated by the true HMM. The number of states in the two models are also compared, which serves as a simple indication of their structure. Another criterion is the ability of the inferred model to correct the symbol errors introduced by the true model during the generation process, as a result of its hidden nature. Various experiments measure the performance of different training schedules for the hidden SECT algorithm, and the influence of the number of observation densities and their overlap on HMM topology inference.

1.2.3.2 Language recognition experiments

The final set of experiments compare mixed-order HMMs with fixed-order models on the real-world problem of language recognition. Automatic language recognition (ALR) [2] algorithms attempt to determine the language that a person speaks, based on recordings of her speech. This can be used to drive dialogues in interactive voice response (IVR) systems or to automatically channel callers to the appropriate consultant in call centres. A more useful application is as an aid to searching in large speech databases typically containing recorded telephone conversations. There has recently been a resurgence of interest in this field, as the language recognition evaluations [54, 55] of the National Institute of Standards and Technology (NIST) attest.

The current state-of-the-art ALR systems (for example [56, 57]) fuse the outputs of many subsystems to achieve their high performance. Some of these subsystems focus on the *acoustic-phonetic* aspects of a language, which describe the set of basic sounds or phonemes found in the spoken language. The sounds are typically characterised by cepstral features, which are subsequently modelled by Gaussian mixture models (GMMs) or support vector machines (SVMs). Other subsystems model *phonotactic* constraints, which describe the phoneme sequences that are allowed in the language, and which differ between languages. These subsystems typically rely on a set of parallel phone recognisers followed by an n-gram-based language model [58] to capture the phonotactic constraints. The subsystems are finally fused together in a combined classifier that performs better than any of the individual subsystems. The interested reader can find an overview of earlier work in [59, 58], while many of the latest advances are described in publications related to the NIST language recognition evaluations.

High-order HMMs can also be used to model phonotactics, as was shown in [39]. Instead of explicitly recognising phones and then modelling their interactions on a symbolic level, high-order HMMs model phonotactics implicitly, by describing longer-range dependencies between regions in acoustic feature space. These feature space regions are determined by unsupervised clustering and are shared among the languages, but do not directly correspond to phones. However, they can be trained on untranscribed speech databases, which benefits

[2] ALR is also known as language identification (LID).

languages for which extensive transcribed data do not exist. The cost of creating transcriptions for training phone recognisers can otherwise be prohibitive. Furthermore, high-order HMMs avoid the errors introduced by the hard classification step of phone recognisers, by combining the steps of phone recognition and language modelling into one.

This study recreates the experimental setup of [39], but uses a much larger speech database for training. Mixed-order HMMs trained with the hidden SECT algorithm are compared with fixed-order HMMs trained with the FIT algorithm of [39], and are shown to be smaller and faster for similar classification accuracy.

It should be emphasised that these HMMs are not intended to be full-fledged ALR systems in their own right. The experiment merely serves as a useful test case for higher-order HMMs. However, because of their alternative approach to phonotactic modelling, these models may still be useful subsystems that improve the classification accuracy of a language recogniser after fusion with other more accurate subsystems.

1.3 Related work

This section contrasts the SECT and hidden SECT algorithms with related methods for inferring PSTs and HMMs, which are discussed in greater detail in Chapter 3. Many of these methods were introduced after SECT and hidden SECT.

Since the introduction of the prediction suffix tree in [23], many variants of the basic PST inference algorithm (Learn-PSA) have been proposed. Some variants [24, 25, 20] address the quadratic computational complexity of Learn-PSA, improving it to become linear in the training sequence length. In the case of [24], this is achieved at the expense of increased memory usage. Other variants [26, 20, 27] reduce the number of user-specifiable parameters, and allow trees of unbounded depth, where the data set itself determines the optimal model order. Bejerano's variant [26, 20] is particularly relevant to this study, as it is also based on the minimum description length principle.

The SECT algorithm has a computational complexity of $O(LT \log T)$ for a sequence length of T and maximum tree depth of L, which improves on the (worst-case) $O(LT^2)$ complexity of standard Learn-PSA. While its complexity is not linear in T, it is more memory-efficient than the linear-time version of Learn-PSA proposed in [22]. It also requires less memory than the method in [24], especially for highly variable context lengths.

In addition, SECT has no user-specifiable parameters, which removes the need for an expensive cross-validation step, and it is self-bounded. In this respect it is closely related to Bejerano's MDL algorithm [26, 20]. SECT predates this method, however, and is faster and has no restrictions on the types of PST it can infer (see Section 4.6 for details).

While there are several studies devoted to mixed-order Markov chains, there is surprisingly little literature on mixed-order hidden Markov models. This is related to the lack of studies dealing with high-order HMMs in general, as noted in [39]. As the order of an HMM is an aspect of its topology, research that focuses on the inference of HMM topology is also relevant to this study. This includes state merging methods such as [40, 41, 60], state splitting methods [42, 43], Brand's entropic prior approach [61] and the higher-order HMM training approach of [39]. Many of these methods only apply to discrete observation sequences, while the hidden SECT algorithm can also handle continuous observations.

The algorithm of Sage et al. [62] bears the closest relation to hidden SECT. It also combines a PST with a continuous HMM to obtain a mixed-order model, and derives the PST from soft symbol counts, which is still an unexplored avenue for hidden SECT. On the other hand, the derivation in [62, 63] does not explore the full generality of mixed-order HMMs, and some aspects of the training algorithm are customised for a specific application (stochastic trajectory generation for motion tracking). The development of hidden SECT also preceded that of the algorithm in [62] by several years.

1.4 Contributions of this work

1. The SECT algorithm is an improvement on many existing algorithms for learning mixed-order Markov chains. It has no tunable parameters and trains models with a potentially unlimited model order that is bounded by the data set itself. It is also faster than the original PST algorithm. This is shown in Chapter 4.

2. Mixed-order hidden Markov models are still uncommon in the literature, and this study provides insight in the training of these models. It points out the connection between mixed-order HMMs and mixed-order Markov chains in Chapter 2. Furthermore, the formulation of a higher-order HMM in terms of symbols rather than states simplifies its notation and directly suggests training algorithms such as hidden SECT, as shown in Chapter 5.

3. The hidden SECT algorithm is a flexible and efficient way to train the topology of a continuous mixed-order HMM from data. It focuses exclusively on topology inference, and allows the use of standard techniques for updating the observation densities and transition probability values. The algorithm is described in Chapter 5.

4. Mixed-order HMMs are shown to outperform fixed-order models in terms of speed and model size on a non-trivial task of language recognition, as discussed in Section 6.3.

1.5 Organisation of dissertation

Chapter 2 introduces various models, algorithms and concepts that serve as background knowledge for the rest of the dissertation. This includes the minimum description length principle, Markov chains, prediction suffix trees and hidden Markov models. It also establishes the mathematical notation used in the following chapters. The literature study of Chapter 3 focuses on algorithms that infer PSTs and HMM topology, which are thereby directly comparable to the SECT and hidden SECT algorithms, respectively. Special attention is paid to the computational complexity of each method.

The two main algorithms of this study are introduced in Chapters 4 and 5. The chapters describe the algorithms in detail, and also discuss their computational complexity. Chapter 6 describes the experiments done in this study, and presents and discusses the results of each experiment. This includes both experiments on synthetic data and language identification experiments. Chapter 7 provides some concluding remarks, and also recommends various improvements to the SECT and hidden SECT algorithms.

Chapter 2

Background

This chapter introduces some basic concepts that serve as background for the rest of the dissertation. The focus is on Markovian models, including Markov chains, prediction suffix trees and hidden Markov models. A good overview of these models can be found in [2, 22]. Minimum description length and related topics from information theory are also introduced. The chapter aims to be self-contained and may be skipped by readers familiar with these topics. It also serves as a reference for the notation used in the rest of the dissertation.

2.1 Information Theory

Information theory originated with the seminal paper of Shannon [64], who first showed the intimate connection between the transfer of information and probability theory. Subsequently, many texts have been published on the subject, of which [65] is a very readable (and freely available) introduction and [66] is a standard textbook. This section provides the theoretical background of data compression and the principle of minimum description length.

2.1.1 Data compression

Consider a finite set or *alphabet* of M symbols $\Sigma = \{s_1, s_2, ..., s_M\}$. These symbols can for example represent various messages to be received, or the possible answers to a question. A random variable X is now defined on Σ, so that each symbol s_i has a probability $P(X = s_i) = p_i$ to occur, where $p_i \geq 0$ and $\sum_{i=1}^{M} p_i = 1$. The *Shannon information content* [65] or *self-information* of an individual outcome $X = s_i$ is given by

$$h(s_i) = \log_2 \frac{1}{p_i}$$

in *bits*, and the average information content of the random variable X is known as the *entropy*

$$H(X) = \mathrm{E}[h(X)] = \sum_{i=1}^{M} p_i \log_2 \frac{1}{p_i},$$

where a zero probability $p_i = 0$ contributes zero to the sum. The entropy ranges over $0 \leq H(X) \leq \log_2 M$, where $H(X) = 0$ implies one of the probabilities $p_i = 1$ and the rest are zero (i.e. X is deterministic), and $H(X) = \log_2 M$ implies all $p_i = 1/M$ (i.e. X is uniform). The entropy will also be indicated as $H(\boldsymbol{p})$, where $\boldsymbol{p}$ is the vector of probabilities $\{p_1, p_2, ..., p_M\}$.

A sequence of symbols of length T is indicated by $\boldsymbol{x} = x_1^T = \{x_1, x_2, ..., x_T\}$, or $x_1 x_2 ... x_T$ if there is no ambiguity. The set of all length-T sequences of symbols from Σ is Σ^T, while Σ^+ is the set of all finite-length sequences of symbols from Σ (excluding the empty string λ). A variable-length binary *symbol code* $c : \Sigma \to \{0, 1\}^+$ is a function that assigns a variable-length string of zeroes and ones, or *codeword*, $c(s_i)$ to each symbol s_i. The length of codeword $c(s_i)$ will be indicated by $l(s_i) = l_i$. The expected length of the code is $L(c, X) = \sum_{i=1}^{M} p_i l_i$. The *extended code* $c^+ : \Sigma^+ \to \{0, 1\}^+$ maps strings of symbols to strings of binary digits or *encodings*, by concatenating the codewords for each symbol as

$$c^+(x_1 x_2 ... x_T) = c(x_1) c(x_2) ... c(x_T).$$

An example [65, Example 5.10] of a random variable X and symbol code c is shown in Table 2.1. The entropy of X, as well as the expected length of the code $L(c, X)$, is 1.75 bits. The symbol string $\boldsymbol{x} =$ `acdbac` is encoded as $c^+(\boldsymbol{x}) =$ `0110111100110`.

For a symbol code to be useful, it has to be uniquely and easily decodable. A code c is *uniquely decodable* if no two distinct symbol strings have the same

Table 2.1: An example of a symbol code c for random variable X. From left to right, the columns show the symbol, symbol probability, symbol information content, codeword and codeword length, respectively.

s_i	p_i	$h(s_i)$	$c(s_i)$	$l(s_i)$
`a`	1/2	1.0	`0`	1
`b`	1/4	2.0	`10`	2
`c`	1/8	3.0	`110`	3
`d`	1/8	3.0	`111`	3

encoding, i.e. if

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \Sigma^+, \quad \boldsymbol{x} \neq \boldsymbol{y} \Longrightarrow c^+(\boldsymbol{x}) \neq c^+(\boldsymbol{y}).$$

A code is easy to decode if it is a *prefix-free code*,[1] which means that no codeword is a prefix[2] of any other codeword. This allows the ends of codewords to be recognised without having to look ahead in the encoding. Prefix-free codes are also uniquely decodable. The code in Table 2.1 is prefix free, and the encoded string $c^+(\boldsymbol{x}) =$ `0110111100110` can easily be decoded as $\boldsymbol{x} =$ `acdbac`.

The codeword lengths l_i of any uniquely decodable binary symbol code satisfy the *Kraft-McMillan inequality* [67, 68]

$$\sum_{i=1}^{M} 2^{-l_i} \leq 1. \tag{2.1.1}$$

Conversely, if a set of codeword lengths satisfy this inequality, there exists a prefix-free code with these codeword lengths. A uniquely decodable binary code can therefore always be converted to a prefix-free code. If a uniquely decodable code satisfies the Kraft-McMillan inequality with equality, the code is called *complete*.

An optimal code for a random variable X should be uniquely decodable and have the minimum expected length $L(c, X)$, thereby achieving the best possible

[1] A prefix-free code is also known as a prefix code, instantaneous code or self-punctuating code.

[2] A string $\boldsymbol{p}$ is a *prefix* of another string $\boldsymbol{x}$ if a third (possibly empty) string $\boldsymbol{s}$ can be found such that $\boldsymbol{ps} = \boldsymbol{x}$. Similarly, a string $\boldsymbol{s}$ is a *suffix* of string $\boldsymbol{x}$ if $\boldsymbol{ps} = \boldsymbol{x}$ for some $\boldsymbol{p}$. A *proper* prefix or suffix cannot be the string $\boldsymbol{x}$ itself.

compression in the long run. This occurs if and only if its codeword lengths are equal to the Shannon information contents of the random variable, that is

$$l_i = \log_2 \frac{1}{p_i}. \tag{2.1.2}$$

In this case, the code is prefix free and complete, and its expected length is equal to the entropy $H(X)$. The entropy of X is therefore the smallest average size to which symbols occurring with the distribution of X can be compressed.

Any code which does not satisfy (2.1.2) will have an expected codelength $L(c, X) > H(X)$. For example, the *raw* code associated with alphabet Σ of size $M = |\Sigma|$ has M codewords with identical lengths $l_i = \log_2 M$, which represents the default encoding of symbols from Σ, even before a probability distribution is assigned to them. An example of such a raw encoding is ASCII, which assigns eight bits to each character, regardless of their distribution. The raw code is only optimal if the symbols are uniformly distributed. Since typical text strings have a non-uniform character distribution, text compression algorithms can improve on ASCII and encode text files to less than eight bits per character.

The *relative entropy*, *cross-entropy*, *Kullback-Leibler divergence* [69] or simply *divergence* quantifies the optimality of a code. If a code that is optimal for distribution $\boldsymbol{r} = \{r_1, r_2, ..., r_M\}$, with codeword lengths $l_i = -\log_2 r_i$, encodes symbols with a probability distribution $\boldsymbol{p} = \{p_1, p_2, ..., p_M\}$ instead, the excess expected codelength is given by the *divergence*

$$\begin{aligned} \mathrm{D}(\boldsymbol{p}\|\boldsymbol{r}) &= \sum_{i=1}^{M} p_i l_i - H(\boldsymbol{p}) \\ &= \sum_{i=1}^{M} p_i \log_2 \frac{p_i}{r_i}. \end{aligned} \tag{2.1.3}$$

This can be interpreted as a similarity measure between the two discrete probability distributions $\boldsymbol{p}$ and $\boldsymbol{r}$. The divergence satisfies *Gibbs' inequality* [65][3]

$$\mathrm{D}(\boldsymbol{p}\|\boldsymbol{r}) \geq 0, \tag{2.1.4}$$

with equality only if $\boldsymbol{p} = \boldsymbol{r}$. This makes the divergence appear like a metric on the space of probability distributions, but it is not symmetric: in general, $\mathrm{D}(\boldsymbol{p}\|\boldsymbol{r}) \neq \mathrm{D}(\boldsymbol{r}\|\boldsymbol{p})$. It also does not satisfy the triangle inequality. Note that $\mathrm{D}(\boldsymbol{p}\|\boldsymbol{r}) = \infty$ if $p_i > 0$ and $r_i = 0$ for some i. In this case, the code is forced to encode a symbol s_i that it did not expect at all.

2.1.2 Minimum description length

The minimum description length (MDL) principle originated in the 1960s, following independent studies by Solomonoff [70], Kolmogorov [71, 72, 73], Chaitin [74], Wallace [45] and Rissanen [46]. The original formulation of Kolmogorov, Solomonoff and Chaitin used *Kolmogorov* or *algorithmic complexity* [75] as description length, which is defined as the length of the shortest computer program that can generate a given data sequence. The subsequent work of Wallace and Rissanen defined description length in information-theoretic terms, as the length of a message that can transmit the data sequence without error. Since then, it has found widespread application in the fields of data compression [76, 77], statistical inference [78], model comparison [79], clustering [80, 81] and classification [45, 82]. Good tutorial introductions can be found in [47, 83, 84], while [85] is a collection of recent research results.

The two most practical manifestations of the MDL principle are Wallace's minimum message length (MML) [45, 86] and Rissanen's MDL[4] [46, 87, 47]. A discussion of the differences between MML and Rissanen's MDL can be found

[3]Gibbs' inequality is an application of *Jensen's inequality* [65], which states that

$$\mathrm{E}[f(x)] \geq f(\mathrm{E}[x])$$

for any random variable x and any convex function f, where $\mathrm{E}[\cdot]$ denotes expectation with respect to the density of x. Since the logarithm is concave, it can be used to prove (2.1.4).

[4]Strictly speaking, MDL refers to Rissanen's approach [46]. Nevertheless, all the approaches mentioned so far share the same basic idea, of which Rissanen's version is the most well-known. Modern textbooks such as [65] refer to the collective concept as the *MDL principle*. We follow this convention, and will refer to the original MDL as Rissanen's MDL. Discussions on these naming conventions can also be found in [84].

in [75, 85]. While they differ philosophically, their practical results are frequently similar. A good introduction to MML can be found in [88, 89, 86], while [87] is a good explanation of the philosophy behind Rissanen's MDL. An understanding of the basic idea of MDL will suffice for this study.

The MDL principle starts off with a data set D and a set of models $\mathcal{M} = \{M_1, M_2, \ldots\}$ which will be used to explain the data. A two-part message is formed, consisting of a description of a model M_i and of the data D in terms of this model, usually in the form of a binary string based on algorithmic complexity or information content. According to MDL, the best model in $\mathcal{M}$ is the one that allows the shortest total encoding of model and data, where the length is measured in bits.

The encoding process follows the basic principles of Shannon's information theory: if an event x has a probability of occurrence of $P(x)$, it requires a message length of $L(x) = -\log_2 P(x)$ bits. The model part of the message typically contains descriptions of the model structure and parameter values stored to a certain precision. The data part of the message contains the data set encoded by an optimal code based on the probability distribution provided by the model. MDL focuses solely on the message lengths involved and is not concerned with the actual implementation of an encoder and decoder, as long as the overall encoding is uniquely decodable.

In the case of a finite number of models, the MDL principle is related to the Bayesian *maximum a posteriori* (MAP) approach. Bayes' rule states that

$$P(M_i|D) = \frac{P(M_i)\,P(D|M_i)}{P(D)},$$

and the MAP model M_{MAP} obtained by maximising the posterior model probability is the same[5] as the model found by minimising the message length as

$$\begin{aligned} M_{\text{MAP}} &= \arg\max_i P(M_i|D) \\ &= \arg\min_i \left[-\log_2(P(M_i)\,P(D|M_i))\right] \\ &= \arg\min_i \left[L(M_i) + L(D|M_i)\right]. \end{aligned}$$

[5]The connection between MDL and MAP is less direct than is implied here, and becomes even more tenuous for a continuum of models—see [90, 89] for more details.

MDL therefore provides a practical way to compute model priors $P(M_i)$, by encoding the relevant model information and calculating its codelength.

The MDL principle is useful because of its generalisation ability and robustness. Since the total complexity of model and data is minimised, MDL penalises both overly simple models that explain the data poorly and highly complex models that fit the data perfectly, thereby performing the role of Occam's razor [65, 91]. It avoids overfitting, which improves prediction and classification of unseen (test) data.

2.2 Markov Chains

A Markov chain (MC) [1, 92, 93] is a statistical model that describes sequential data. The data is in the form of a sequence of *symbols* $s_1^T = \{s_1, s_2, ..., s_T\}$, where each symbol is an element of a set Σ called the *alphabet*. The symbols are assumed to be completely distinguishable and the alphabet is assumed to be finite with size $M = |\Sigma| < \infty$. Examples of such symbol sequences include written text, phoneme strings, DNA sequences, and many more.

The model provides a probability distribution over the set Σ^T of all symbol sequences of a fixed but arbitrary length T. The probability[6] of a specific sequence, $P(s_1^T)$, can be factored by the standard chain rule of probability theory [94] as

$$P\left(s_1^T\right) = P(s_1) \prod_{t=2}^{T} P\left(s_t \middle| s_1^{t-1}\right). \tag{2.2.1}$$

The conditional probability $P(s_t|s_1^{t-1})$ can be thought of as a prediction of the next symbol, given the past symbols (or string prefix). This form of $P(s_1^T)$ is intractable in general, since the complexity of the conditional probability $P(s_t|s_1^{t-1})$ increases with t. The Markov chain therefore introduces a simplifying assumption, the so-called *Markov property*, which states that the conditional probability of symbol s_t, given the values of preceding symbols s_1^{t-1}, only depends on the

[6]To be precise, each symbol s_t is replaced by a random variable S_t, and the random sequence S_1^T then has the joint probability distribution $P(S_1^T = s_1^T) = P(S_1 = s_1, S_2 = s_2, ..., S_T = s_t)$. For simplicity, we will use the notation $P(x)$ to mean the probability $P(X = x)$ that the random variable X takes the value x, unless it leads to ambiguity.

values of a finite number L of them, i.e.

$$P\left(s_t \middle| s_1^{t-1}\right) = P\left(s_t \middle| s_{t-L}^{t-1}\right),$$

where L is known as the *order* of the model. This conditional independence property allows the joint distribution of a sequence to be decomposed as

$$P\left(s_1^T\right) = P(s_1) \prod_{t=2}^{T} P\left(s_t \middle| s_{t-L}^{t-1}\right),$$

which makes the model less expressive, but highly tractable. Stated informally, the defining feature of a Markov model is that it forgets everything except the recent past. In the case of a first-order Markov chain, the distribution simplifies even further to

$$P\left(s_1^T\right) = P(s_1) \prod_{t=2}^{T} P(s_t|s_{t-1}),$$

which is completely specified by the *initial symbol probabilities* $P(s_1)$ and *symbol transition probabilities* $P(s_t|s_{t-1})$.

A very useful concept in Markov models is that of a *state variable*, which represents all information at a given point in the sequence relevant to the distribution of the next symbol. In the case of an Lth-order Markov chain (denoted by L-MC), the relevant information at time t is summarised by the state variable[7]

$$q_t = s_{t-L+1}^{t},$$

where this sequence of L symbols is also referred to as the *context string* associated with the state. The symbol associated with the state is the last symbol in the context string. For first-order models, the state simplifies to $q_t = s_t$, and states and symbols turn out to be equivalent. The state variable q_t takes on values from the set $\mathcal{Q} = \Sigma^L$, which is also known as the *state space*. The state space is finite for

[7]This definition of the state variable q_t differs from the standard approach in pattern recognition texts such as [49] and [2], which instead choose to call the symbols s_t *states*. It is closely related to the definition of probabilistic suffix automata (PSAs) in [22]. The two approaches are equivalent for first-order models, which is by far the most prevalent case. The approach taken by this text and [22] has the advantage of simpler notation in the case of higher-order Markov models.

a finite alphabet and order, with a fixed size of $N = |\mathcal{Q}| = M^L$. This underlines the tractability of the Markov chain, as the general sequential model of (2.2.1) has a state space that grows unbounded over time.

Using the state notation, the sequence distribution is transformed to

$$P\left(s_1^T\right) = P(q_1) \prod_{t=2}^{T} P(s_t | q_{t-1}) ,$$

where $P(q_1)$ is the *initial state distribution* and $P(s_t|q_{t-1})$ is referred to as the *next-symbol distribution*. Instead of indexing the states with symbol strings, it is possible to relabel them with numerical indices by enumerating the N elements of the finite state space. The symbol s_t can likewise be replaced by the state $q_t = s_{t-L+1}^t$, by prepending its length-$(L-1)$ historical context. This transforms the Lth-order Markov chain into an equivalent first-order Markov chain based on a sequence of states with the state space Σ^L as alphabet, and sequence distribution given by

$$P\left(q_1^T\right) = P(q_1) \prod_{t=2}^{T} P(q_t | q_{t-1}) ,$$

where $P(q_t|q_{t-1})$ is known as the *state transition probability function*.

The two formulations are equivalent in that there is a one-to-one correspondence between symbol sequences s_1^T and state sequences q_1^T, and $P(s_1^T) = P(q_1^T)$ for each corresponding pair.[8] The higher-order relations between the symbols are encoded in the first-order model as constraints on the state transitions. Although there are M^L states, each state has at most M outbound transitions, and the context strings of connected states have to match up appropriately. If there is a transition from state q_1 to state q_2, the last $L-1$ symbols of the context string of q_1 has to match the first $L-1$ symbols of the context string of q_2. For example, a transition can be made from state `abaab` to state `baabc` but not to state `ababa`.

[8]This correspondence is not guaranteed if the Markov chain had been constructed with an arbitrary state topology and assignment of symbols to states for orders $L > 1$, in which case the model would turn into a *hidden* Markov model or general probabilistic finite-state automaton. Higher-order Markov chains have a highly constrained state topology by design, which is why it is important to distinguish between symbols and states. Each state in a properly constructed higher-order Markov chain can always be associated with a unique symbol context.

2.2.1 Initial and final states

In the initial part of the symbol sequence, for $t < L$, the context strings are shorter than L symbols, which require special treatment. There are two ways to handle this. A special initial symbol[9] $\wedge$ can be added to the alphabet (i.e. $\Sigma' = \Sigma \cup \{\wedge\}$), so that $s_t = \wedge$ for $t \leq 0$. This allows the use of length-L context strings throughout the model, and the state sequence is started off with $q_0 = \wedge \wedge ... \wedge$. Another option is to add all context strings of lengths 0 to $L-1$ as *start-up* states, with the length-0 empty string indicated by λ (i.e. $\mathcal{Q}' = \cup_{m=1}^{L} \Sigma^m \cup \{\lambda\}$). These states serve to connect the new initial state $q_0 = \lambda$ to the states with length-L contexts, and cannot be reached after $t = L-1$. This is the approach followed in this study. With this approach, the initial state distribution $P(q_1)$ becomes a set of state transition probabilities, given by $P(q_1) = P(q_1|q_0)$, which unifies the parameters of the model.

The standard Markov chain provides a probability distribution normalised over Σ^T, the set of all length-T strings over Σ. Variable-length strings are modelled by a distribution over the set Σ^* of all possible strings over Σ instead. This distribution is normalised over Σ^*, i.e. $\sum_{x \in \Sigma^*} P(x) = 1$. The Markov chain can be extended to handle variable-length strings [95], by adding the concept of a *final* state. A special final symbol or end-of-string event \$ can be added to the alphabet, while a final state $q_\$$ with context string \$ is added to the state space. The model is forced to end up in $q_\$$, by requiring that $q_{T+1} = q_\$$. The probability of a sequence therefore becomes

$$P\left(s_1^T\right) = P(q_1) \left[\prod_{t=2}^{T} P(s_t|q_{t-1}) \right] P(\$|q_T) \, .$$

When generating strings with the model, the generation process only stops when $q_\$$ is reached. The sequence length is modelled by the amount of probability mass "leaked" into the final symbol in each next-symbol distribution.

The inclusion of a final state is important when the model is used to compare the probabilities of strings of different lengths. In typical problems involving sequence classification and verification, however, a single length-T sequence is presented to multiple models in order to choose the model with the best fit to the

[9]The symbols for initial and final contexts are inspired by Posix regular expressions.

data. This allows the use of distributions normalised over Σ^T, such as Markov chains without a final state.

2.2.2 Definitions

A *homogeneous* or time-invariant Markov chain has state transition probabilities that do not change with time. This simplification is frequently required in practice to enable reliable estimation from data, and all Markov chains will henceforth be considered to be homogeneous. This allows a simpler notation for the probability that the state with index i makes a transition to state j, given by

$$a_{ij} = P(q_t = j | q_{t-1} = i) , \qquad 1 \leq i, j \leq N,$$

since $P(q_t | q_{t-1})$ does not depend on absolute time t, but only on the actual states involved. Alternatively, the transition probability can be indicated in terms of symbols, by appending the symbol associated with state j (the last symbol of its context string) to the context string of state i. For example the probability of a transition between state `abaab` and state `baabc` is a_{abaabc}.

A homogeneous Markov chain has two very useful representations. It can be seen as a graph, with the states as nodes and transitions between states as edges, like the first-order model shown in Figure 2.1. It can also be represented by an $N \times N$ *transition matrix* $\boldsymbol{A}$ with elements a_{ij}, where $N = |\mathcal{Q}|$. The key properties of the chain is reflected in the properties of this matrix. The rows of $\boldsymbol{A}$ are normalised, i.e. $\sum_j a_{ij} = 1$.

Formally, therefore, an Lth-order homogeneous Markov chain is a 4-tuple $(\Sigma, \mathcal{Q}, \tau, \eta)$, where Σ is a finite alphabet, $\mathcal{Q} = \cup_{m=1}^{L} \Sigma^m \cup \{\lambda\}$ is the state space, $\tau : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the state transition function, and $\eta : \mathcal{Q} \times \Sigma \to [0, 1]$ is the next-symbol probability distribution. The state transition function $r = \tau(q, s)$ creates the context string of the next state r by appending its associated symbol s to the context string of q, and removing the first symbol in the string if its length becomes larger than L. The next-symbol distribution is normalised so that $\sum_{s \in \Sigma} \eta(q, s) = 1$. The initial state is always taken to be $q_0 = \lambda$. Alternatively, the functions τ and η can be replaced by the transition matrix $a : \mathcal{Q} \times \mathcal{Q} \to [0, 1]$, with the appropriate constraints on state transitions as imposed by τ.

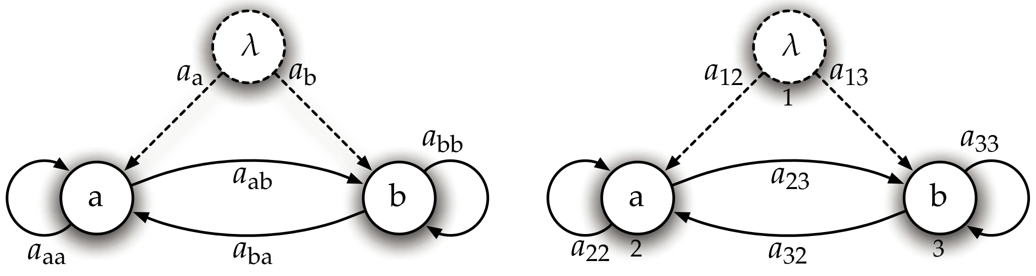


Figure 2.1: A first-order Markov chain with $\Sigma = \{a,b\}$. The left-hand version indexes transitions using symbols, while the right-hand version numbers each state and indicates transitions in a_{ij} format. The start-up state is drawn in dashed lines.

It is very useful to think of a Markov chain in terms of a state machine. The state is initialised to $q_0 = \lambda$, and at each time step, the state advances randomly to a new state according to the transition matrix $\boldsymbol{A}$. The *state probability distribution* at time t is a vector $\boldsymbol{a}_t = \{P(q_t = 1), P(q_t = 2), ..., P(q_t = N)\}$, which is advanced to the next time step by multiplication by the transition matrix, giving

$$\boldsymbol{a}_t = \boldsymbol{a}_0 \boldsymbol{A}^t$$

with initial value $\boldsymbol{a}_0 = \{1, 0, 0, ..., 0\}$ (assuming the initial state has index 1). A state of an MC is said to be *persistent* if the state machine will ultimately return to it in finite time with probability one; otherwise, it is called *transient*. Persistent states typically occur perpetually, while transient states typically occur only once.

An *ergodic* Markov chain [1] contains a single special group of states that

- are all persistent,
- can all reach each other through the appropriate state transitions,
- cannot reach any states outside the group,
- and does not contain any deterministic loops in the transitions within the group (also known as *periodic* states).

In other words, the states of an ergodic MC all keep recurring randomly (except possibly for a finite number of transient start-up states). In this case, the state probabilities converge over time to a *long run distribution*[10]

$$\boldsymbol{\pi} = \lim_{t\to\infty} \boldsymbol{a}_t.$$

This implies that $\boldsymbol{\pi}\boldsymbol{A} = \boldsymbol{\pi}$, which provides a simple way to obtain $\boldsymbol{\pi}$ as the eigenvector of $\boldsymbol{A}$ associated with the (largest) eigenvalue of one, normalised so that $\sum_i \pi_i = 1$. The long run distribution is useful in various calculations involving the MC, such as determining its entropy. Ergodic models are generally used to model sequential data with a recurring nature, such as natural languages.

An example of a generic second-order Markov chain on the binary alphabet $\Sigma = \{\text{a,b}\}$ is shown in Figure 2.2. Assume its transition matrix is given by

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0.8 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.3 & 0.7 \end{bmatrix}.$$

The matrix $\boldsymbol{A}$ is very sparse, since each state can have at most two outbound transitions. The model is ergodic, as all allowed connections are non-zero (it is *fully connected*). The start-up states $\{1, 2, 3\}$ are transient, while states $\{4, 5, 6, 7\}$ form the persistent subset. The long run distribution is $\boldsymbol{\pi} = \{0, 0, 0, 0.41, 0.16, 0.16, 0.27\}$. The symbol sequence $s_1^8 = \texttt{abaababb}$ has an associated state sequence of $q_0^8 = \{1, 2, 5, 6, 4, 5, 6, 5, 7\}$ and would be assigned a probability of

$$\begin{aligned} P(\text{abaabaab}) &= a_{\text{a}} \cdot a_{\text{ab}} \cdot a_{\text{aba}} \cdot a_{\text{baa}} \cdot a_{\text{aab}} \cdot a_{\text{aba}} \cdot a_{\text{bab}} \cdot a_{\text{abb}} \\ &= 0.5 \cdot 0.2 \cdot 0.5 \cdot 0.5 \cdot 0.2 \cdot 0.5 \cdot 0.5 \cdot 0.5 = 0.000625. \end{aligned}$$

Besides scoring sequences as in the example above, Markov chains are also efficient generators of symbol sequences. The model is initialised in state $q_0 = \lambda$. Thereafter, a random transition is made according to the state transition distribu-

[10]The long run distribution is also known as the stationary, equilibrium or invariant probability distribution for the Markov chain.

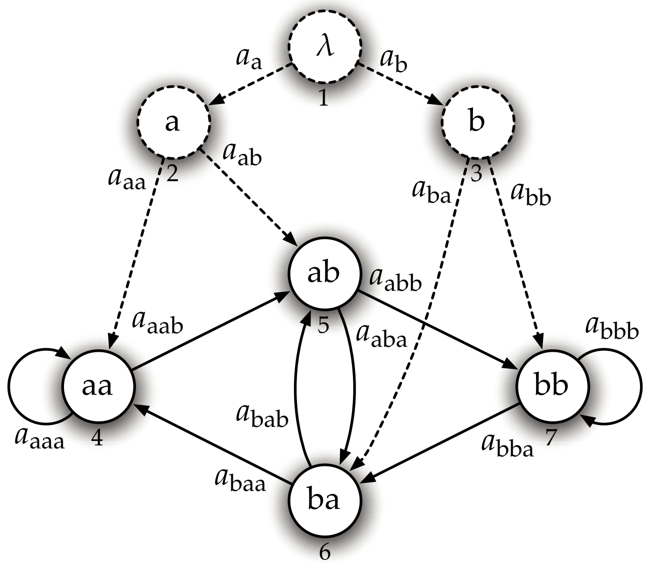


Figure 2.2: A generic second-order Markov chain on the binary alphabet $\Sigma = \{a,b\}$. The start-up states are indicated with dashed lines.

tion (i.e. the appropriate transition matrix row), and the symbol associated with the new state is emitted. This process is repeated until T symbols are generated. This use of Markov chains is the basis of the popular Markov chain Monte Carlo (MCMC) approach to the evaluation of statistical expectations [96, 65], which uses ergodic MCs to achieve a desired long run distribution.

Markov chains have the useful property that models of different orders form a natural hierarchy, with expressive power increasing with order. An L-MC can be turned into an $(L-1)$-MC by constraining the next-symbol probabilities of all states with the same length-$(L-1)$ suffix of their context strings to be equal, i.e. by letting $P(s_t|s_{t-L}^{t-1}) = P(s_t|s_{t-L+1}^{t-1})$. The reverse is not possible, hence 1-MC $\subset$ 2-MC $\subset$ ... This provides a natural way to smooth the estimates of higher-order transition probabilities, by *backing off* to models of lower order [97].

2.2.3 Learning MCs from data

It is straightforward to infer a Markov chain from training data. Since the structure of an L-MC is fixed, all that remains is to estimate the next-symbol probabilities (or state transition probabilities) from the data. The next-symbol probabilities of the start-up states of an MC can only be reliably estimated from multiple training sequences, while the next-symbol probabilities of the persistent states of an ergodic MC can be efficiently estimated from a single (typically long) training sequence. The maximum likelihood estimate of $\eta(q,s)$ is given by

$$\hat{\eta}(q,s) = \frac{\#(q \cdot s)}{\#(q*)}, \tag{2.2.2}$$

where $q \cdot s$ indicates string concatenation, *string count* $\#(x)$ is the number of (possibly overlapping) occurrences of the string x in the training data, and *non-suffix string count*

$$\#(q*) = \sum_{s\in\Sigma} \#(q \cdot s) \tag{2.2.3}$$

is the number of occurrences of q followed by any symbol, therefore disregarding occurrences of q right at the end of training sequences. While $\#(q*)$ is usually equal to $\#(q)$, its use in (2.2.2) ensures that $\hat{\eta}(q,s)$ remains normalised.

2.3 Prediction Suffix Trees

The prediction suffix tree[11] (PST) is introduced in [23, 98, 99, 22], along with the related probabilistic suffix automaton, which will be explained first.

2.3.1 Probabilistic suffix automata

A probabilistic suffix automaton (PSA) is essentially a variable-order Markov chain. As with Markov chains, each state in the PSA is associated with a finite-length symbol sequence or context string. The only difference between a PSA and an MC is that the lengths of the context strings may vary in a PSA, while the context strings of an L-MC are all the same length L (ignoring start-up states).

A PSA is therefore a 4-tuple $(\Sigma, \mathcal{Q}, \tau, \eta)$, where Σ is a finite alphabet, $\mathcal{Q} \subset \Sigma^*$ is a set of finite-length strings (including the empty string λ) serving as state space, $\tau : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the state transition function, and $\eta : \mathcal{Q} \times \Sigma \to [0,1]$ is the next-symbol probability distribution. The next-symbol distribution is nor-

[11]PSTs are also referred to as *probabilistic suffix trees*.

malised so that $\sum_{s \in \Sigma} \eta(q,s) = 1$, and the initial state is taken to be $q_0 = \lambda$, as before.

The state transition function $r = \tau(q,s)$ has to be carefully designed to ensure that the destination state of any possible transition (with $\eta(q,s) > 0$) exists and is unique. It selects the next state r as the one whose context string is the longest suffix of the string $q \cdot s$ formed by appending the symbol s to the context string of q (hence the name of the model). The set of all suffixes of a length-T string s_1^T is given by $\mathcal{S}(s_1^T) = \{s_i^T | 1 \leq i \leq T\} \cup \{\lambda\}$. The transition function $r = \tau(q,s)$ therefore chooses r to be the longest string in $\mathcal{S}(q \cdot s) \cap \mathcal{Q}$. This set will never be empty, as the empty string λ is a state and also a suffix of any string. The state transition function is therefore well defined.

This definition of the PSA differs from the definition found in [22], which allows r to be any suffix and instead requires the state set to be suffix free to ensure a unique value for r. A set of strings S is called *suffix free* if $\forall s \in S$, no proper suffixes of s are in S. This prevents the inclusion of start-up states (and even λ) in the state set, which is handled by an additional initial state distribution instead. The new definition places no restrictions on the state space (except that it should contain λ), while still ensuring a well-defined state transition function. It also simplifies the relationship between PSAs and PSTs, and lifts the restriction in [22] that PSAs are ergodic (which is used to prove the equivalence of PSTs and PSAs).

The subclass of PSAs in which the maximum context string length is L is denoted by L-PSA, for any $L \geq 0$. An L-PSA is equivalent to an L-MC, since any context string shorter than L can be extended to a length-L string without affecting the probabilities assigned to symbol sequences. That is, the equivalent L-MC contains all length-L strings as states, and the next-symbol distribution of MC state q is equal to that of the PSA state whose context string is the longest suffix of q. Similarly, any L-MC can be represented as an L-PSA, by forcing the PSA state space to contain all length-L strings. Although these models are equivalent in expressive power, the L-PSA is potentially much more efficient and compact, as the L-MC may have many more states if the PSA context strings tend to be much shorter than L. Since the prediction of the next symbol with a PSA depends on varying context lengths, it can be seen as a Markov chain with variable order, mixed order or variable memory.

Even though the transition function is well defined, it is possible that certain states in $\mathcal{Q}$ can never be reached from the initial state λ, even though their context strings appear in the symbol sequence. For example, a PSA with states $\mathcal{Q} = \{\lambda, \texttt{aa}, \texttt{aba}, \texttt{bba}, \texttt{b}\}$ can never reach state bba, as the symbol sequence $s_1^3 = \texttt{bba}$ has a corresponding state sequence $q_0^3 = \{\lambda, \texttt{b}, \texttt{b}, \lambda\}$ according to the longest-suffix rule.

The problem is solved by adding all prefixes of PSA states to the state space. The set of all prefixes of a length-T string s_1^T is given by $\mathcal{P}(s_1^T) = \{s_1^i | 1 \leq i \leq T\} \cup \{\lambda\}$. The state space is therefore extended to $\mathcal{Q}' = \cup_{q \in \mathcal{Q}} \mathcal{P}(q)$, where the additional *glue states* $\mathcal{Q}' - \mathcal{Q}$ ensure that all states in $\mathcal{Q}$ are reachable. The next-symbol distribution of glue state q' is equal to that of the longest suffix of q' in $\mathcal{Q}$. The glue states therefore do not introduce new prediction capabilities of their own, but serve as context memory to enable longer significant contexts to be reached. In the previous example, the glue states $\{\texttt{a}, \texttt{ab}, \texttt{bb}\}$ would be added, and the symbol sequence $s_1^3 = \texttt{bba}$ now has a corresponding state sequence $q_0^3 = \{\lambda, \texttt{b}, \texttt{bb}, \texttt{bba}\}$. The resulting PSA is shown in Figure 2.5. The number of glue states in an L-PSA has an upper bound of $L \cdot |Q|$, given that the longest context string in $\mathcal{Q}$ has at most L prefixes to be added.

It is straightforward to learn an L-MC from data, as the state space is known beforehand and the state transition function is also fixed, leaving the next-symbol distribution as the only unknown to be estimated. The learning of a PSA is much more involved, as the state space has to be determined as well. In order to identify the significant contexts in the data, it is useful to recast the PSA in a different representation that is more amenable to the learning process. This representation is the prediction suffix tree.

2.3.2 Definition of prediction suffix tree

A prediction suffix tree (PST) is a triple $(\Sigma, \mathcal{Q}, \eta)$, where Σ is a finite alphabet, $\mathcal{Q} \subset \Sigma^*$ is a set of finite-length strings (including the empty string λ), which will be referred to as *coding contexts*, and $\eta : \mathcal{Q} \times \Sigma \rightarrow [0, 1]$ is the next-symbol probability distribution associated with each coding context. If the longest context string in $\mathcal{Q}$ has length L, the PST will be referred to as an L-PST. The next-symbol distribution is normalised so that $\sum_{s \in \Sigma} \eta(q, s) = 1$ for all $q \in \mathcal{Q}$. Its definition

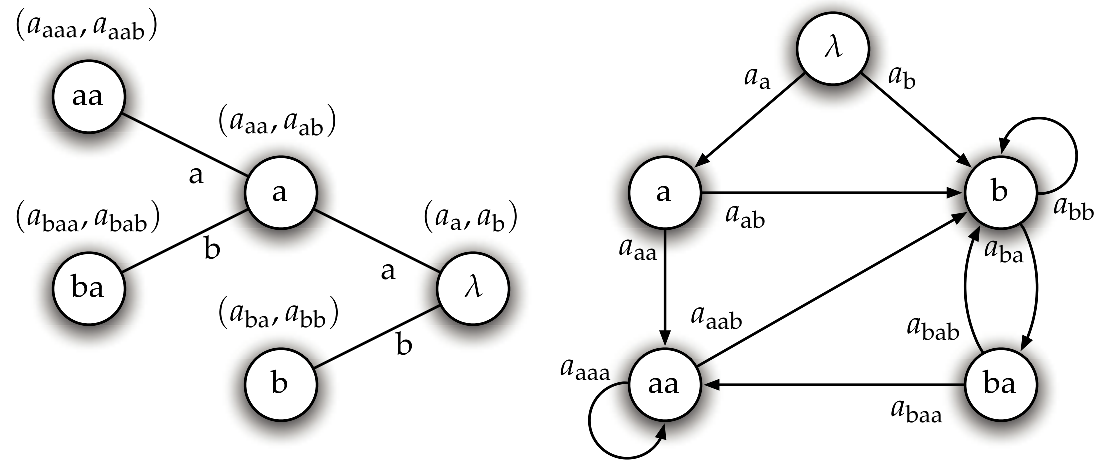


Figure 2.3: A PST on the binary alphabet $\Sigma = \{\text{a,b}\}$ (left), and its equivalent PSA (right). The next-symbol distribution associated with each context is indicated above the relevant PST node in brackets, where $a_{q \cdot s} = \eta(q, s)$. By drawing the root node of the PST on the right and the leaves on the left, the context strings grow longer to the left, as they would in the symbol sequence.

is similar to that of the PSA, but it lacks the concept of a state variable, focusing instead on the idea of contexts.

Another key feature of the PST is that it is represented by a tree instead of a general (cyclic) graph like the PSA. Each node in the tree is labelled by a unique context string $q \in \mathcal{Q}$, and has at most $|\Sigma|$ children.[12] The root node is labelled by the empty string λ. The edges of the tree are labelled by single symbols. Given an internal node with label q, each of its children has a unique symbol s on the edge connecting it to the parent, and the label of the child node is given by $s \cdot q$. The label of any node can therefore be obtained by reading off the edge symbols as the tree is traversed from the node to the root. The leaf nodes of the tree represent the longest contexts, and all node labels in a subtree of the PST share the same suffix, given by the label at the root of the subtree. An L-PST has L levels (not counting the root node as a level). An example PST is shown in Figure 2.3.

[12]In the original definition of the PST in [22], each internal node had exactly $|\Sigma|$ children, or at least all children labelled by context strings with non-zero probability of occurrence. That requirement is relaxed here, by allowing internal nodes to provide the next-symbol probability distribution for missing leaf nodes, thereby tying their distributions. This provides an equivalent but more compact representation.

2.3.3 Scoring and generating data

A PST calculates the probability of a symbol sequence s_1^T as

$$P\left(s_1^T\right) = \prod_{t=1}^{T} \eta(q_{t-1}, s_t),$$

where $q_0 = \lambda$ and q_t is the longest suffix of s_1^t found in $\mathcal{Q}$ (hence the name of the model). That is, the context q_t is determined by starting at the root node of the PST and following the edges labelled by s_t, s_{t-1}, and so forth back in time, until either s_1 is reached or no edge labelled by the specific symbol is found. The label of the node where the walk has ended up becomes q_t. For example, the PST in Figure 2.3 would score the string $s_1^5 = \texttt{abaab}$ as $P(s_1^5) = a_{\text{a}} a_{\text{ab}} a_{\text{ba}} a_{\text{baa}} a_{\text{aab}}$ (where $a_{q \cdot s} = \eta(q, s)$), with the corresponding context sequence given by $q_0^5 = \{\lambda, \texttt{a}, \texttt{b}, \texttt{ba}, \texttt{aa}, \texttt{b}\}$.

To ensure that this scoring procedure is well-defined and to cater for the start of the sequence, the context set $\mathcal{Q}$ is extended by adding all suffixes of each context string, i.e. the new context set becomes $\mathcal{Q}' = \cup_{q \in \mathcal{Q}} \mathcal{S}(q)$. This fills in any missing internal nodes in the PST, so that any leaf node can be reached from the root. The additional contexts $\mathcal{Q}' - \mathcal{Q}$ are referred to as *non-coding contexts*, as each $q \in \mathcal{Q}' - \mathcal{Q}$ does not introduce a new next-symbol distribution, but assumes the distribution of the coding context in $\mathcal{Q}$ which is the longest suffix of q. This distribution is easily obtained by traversing the tree towards the root until a coding context is found. It is therefore useful to require the root node to be a coding context (by including it in $\mathcal{Q}$), as it provides a default next-symbol distribution for any non-coding context and also serves as initial context. Alternatively, the default or *a priori* next-symbol distribution can be a separate entity $\eta_0(s)$, which comes into play whenever a suitable coding context cannot be found.

2.3.4 Equivalence of PSAs and PSTs

Any L-PSA $(\Sigma, \mathcal{Q}, \tau, \eta)$ has an equivalent L-PST $(\Sigma, \mathcal{Q}, \eta)$ that provides the same probability for the same symbol strings. The models are equivalent by construction, which simplifies the derivation of equivalence found in [22]. Similarly, each

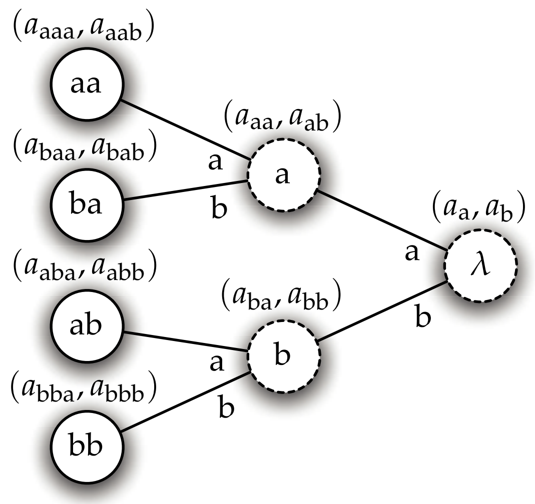


Figure 2.4: The 2-PST that is equivalent to the 2-MC of Figure 2.2. The dashed nodes are the start-up states of the MC, while the solid nodes are its persistent states.

L-PST can be converted into an equivalent L-PSA, as long as glue states are taken into account [22]. That is, the PST context set $\mathcal{Q}$ is extended to include all prefixes, becoming $\mathcal{Q}' = \cup_{q \in \mathcal{Q}} \mathcal{P}(q)$. The additional nodes $\mathcal{Q}' - \mathcal{Q}$ are non-coding contexts, deriving their next-symbol distributions from coding contexts nearer to the root, as before. The equivalent PSA has $\mathcal{Q}'$ as state space, with the same next-symbol distribution as the PST, and the standard state transition function design.

An L-PST is therefore also a variable-order Markov chain with maximum order L. An L-MC is represented by an L-PST with all its leaf nodes on level L, and all internal nodes representing start-up states. In general, if all the internal nodes of a PST associated with an ergodic PSA has all possible children, they represent the start-up states of the PSA, while the leaf nodes represent the persistent states. A typical mixed-order ergodic PST therefore has leaf nodes on different levels. Figure 2.4 shows the 2-PST associated with the second-order Markov chain of Figure 2.2. Its leaf nodes are all on level two, unlike the mixed-order PSTs shown in Figures 2.3 and 2.5.

Figure 2.5 shows a more complicated example of a 3-PST. It has a binary alphabet $\Sigma = \{a,b\}$ and five significant contexts, $\mathcal{Q} = \{\lambda, \mathtt{aa}, \mathtt{aba}, \mathtt{bba}, \mathtt{b}\}$. To complete the internal nodes of the tree, non-coding contexts `a` and `ba` are added as suffixes, which play a role during sequence start-up. These nodes do not have their own next-symbol distribution, but use the distribution at the root node instead. In addition, nodes `ab` and `bb` are added as prefixes, to make the deeper contexts `aba` and `bba` reachable in the equivalent PSA. These nodes share the next-symbol distribution of node `b`. The equivalent (ergodic) PSA is also shown in Figure 2.5. Note that the PSA has nine states and 18 links, but many links are tied to the same value. In comparison, the PST has only five significant nodes and ten next-symbol probabilities, which makes it a more compact representation.

While PSTs can be used for calculating string probabilities and generating symbol sequences, it is not an efficient process. For every symbol to be scored or generated, the search for the current context has to start at the root of the tree, due to the lack of a state machine structure. The computational cost of using an L-PST in these cases is therefore up to L times higher than that of the equivalent L-PSA. It is also difficult to obtain a long-run context distribution, in the case of ergodic models. For these purposes it is more prudent to convert the PST to a PSA, especially if large sequences are to be processed. On the other hand, the PST is a more compact representation of the significant contexts in the model, since no glue states are required and non-coding contexts are simple to incorporate. The PST is also more suited to inference from data, as the tree structure represents a natural way to discover longer contexts incrementally in a data set.

2.3.5 Learning PSTs from data

The original PST learning algorithm, here referred to as Learn-PSA [22], is motivated by the Probably Approximately Correct (PAC) framework [100]. Given a training set of K symbol sequences with lengths totalling T, the algorithm makes use of the *empirical string probability*

$$\hat{P}(q) = \frac{\#(q*)}{T-K}$$

and the *empirical next-symbol probability* $\hat{\eta}(q,s)$ of (2.2.2). It is a *top-down* approach that starts with an empty tree and adds nodes until some criterion is satisfied.

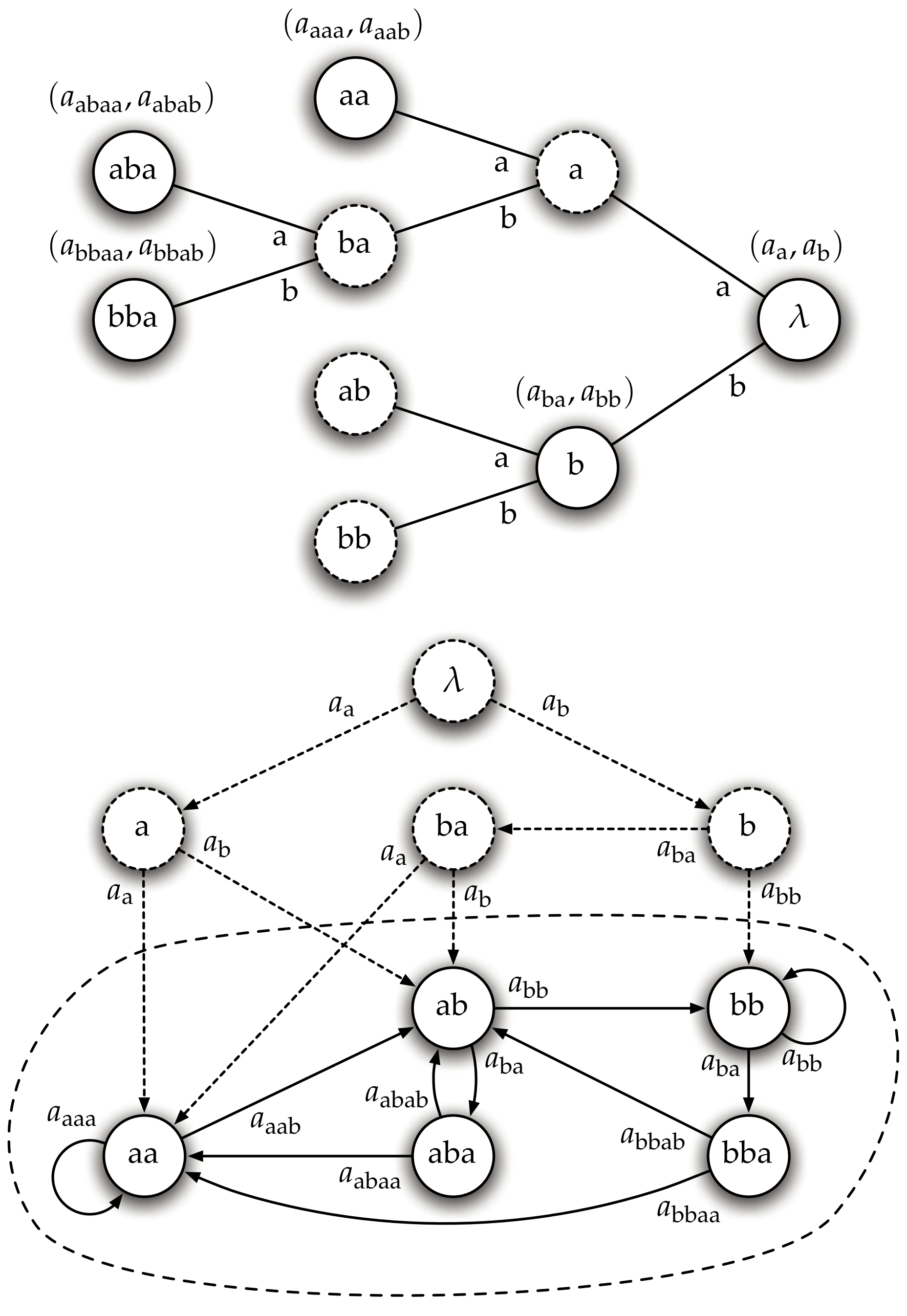


Figure 2.5: A more involved example of a PST (above) on the binary alphabet $\Sigma = \{\text{a,b}\}$ with context set $\mathcal{Q} = \{\lambda, \texttt{aa}, \texttt{aba}, \texttt{bba}, \texttt{b}\}$, and its equivalent PSA (below). The dashed PST nodes are non-coding contexts without unique next-symbol distributions, which instead use the first distribution found when walking from the node to the root. PST nodes a and ba were added as missing suffixes to serve as start-up states, while nodes ab and bb were added as missing prefixes to aid conversion to the PSA. The dashed PSA states are the start-up states, while the solid states within the circled area are the persistent states, which correspond to the leaf nodes of the PST.

```
Learn-PSA
1. Initialise: contexts $\mathcal{Q} = \{\lambda\}$, frontier $F = \{s | s \in \Sigma \text{ and } \hat{P}(s) \geq P_{\min}\}$.
2. While $F \neq \emptyset$, pick any $q \in F$ and do:
    (A) Remove $q$ from $F$;
    (B) Let $p$ be the parent of $q$ such that $q = rp$ for $r \in \Sigma$;
    (C) If there exists a symbol $s \in \Sigma$ such that
            $\hat{\eta}(q,s) \geq \alpha \quad \text{and} \quad \frac{\hat{\eta}(q,s)}{\hat{\eta}(p,s)} \geq 1 + \beta,$
        then add $q$ to $\mathcal{Q}$ and add all missing suffixes $(\mathcal{S}(q) - \mathcal{Q})$ to $F$;
    (D) If $|q| < L$, then for every $r \in \Sigma$, if $\hat{P}(r \cdot q) \geq P_{\min}$, then add $r \cdot q$ to $F$.
3. For each coding context $q \in \mathcal{Q}$ and each symbol $s \in \Sigma$, let
        $\eta(q,s) = \hat{\eta}(q,s)(1 - M\eta_{\min}) + \eta_{\min}.$
```

Figure 2.6: Pseudo-code for the Learn-PSA algorithm, adapted from [22].

Specifically, a new leaf node with context string q is added if q occurs frequently enough in the training set, and its next-symbol probability $\hat{\eta}(q,s)$ differs significantly from that of its parent node, at least for one symbol s that occurs with non-negligible probability. The set of nodes that are considered for addition is called the *frontier* F. The maximum depth of the tree is limited to L. Once the structure of the PST (represented by the context set $\mathcal{Q}$) is fixed, the next-symbol probabilities are calculated. This includes a smoothing term that limits the minimum probability to $\eta_{\min}$. The pseudo-code for Learn-PSA is adapted from [22] and reproduced in Figure 2.6.

Learn-PSA contains five user-controllable parameters: the three thresholds $P_{\min}$, α and β, the depth bound (maximum memory length) L and the probability floor $\eta_{\min}$. The threshold $P_{\min}$ eliminates contexts (especially longer ones) with very low frequency in the data, which prevents the inclusion of poorly estimated next-symbol distributions and also prevents exponential growth in node evaluation. The probability floor $\eta_{\min}$ ensures that the Kullback-Leibler divergence is well-behaved when models are compared. All these parameters have default values in terms of M, L and an approximation accuracy parameter ϵ, provided in [22]. Nevertheless, the user also has the option to fine-tune these parameters via cross-validation [20].

The next-symbol distribution of a node may be very similar to that of its parent, but very different from those of its children. This subtlety forces the algorithm to consider all context strings with lengths up to the length bound L, otherwise a useful subtree could be eliminated prematurely. It is therefore up to threshold P_{min} to curb the tree growth.

The worst-case computational complexity of the standard Learn-PSA inference algorithm is $O(LT^2)$, where L is the maximum context string length considered by the PST and T is the total length of the training data [22, 25]. This is based on a straightforward implementation that performs a complete pass over the data for each context string q added to F, in order to count occurrences of q and to collect next-symbol statistics. The worst-case complexity for scoring sequences of length T with the PST is $O(T^2)$ [25].

2.4 Hidden Markov Models

A hidden Markov model (HMM) [101, 49, 102, 2] is an extension of a Markov chain that in essence allows for ambiguous or "soft" symbols.[13] The HMM models sequences of *observations*, denoted by $x_1^T = \{x_1, x_2, ..., x_T\}$. These observations x_t are elements of an *observation space* $\mathcal{X}$, which can be discrete (e.g. $\mathcal{X} \subset \mathbb{N}$), giving rise to so-called *discrete* HMMs, or a continuous vector space[14] $\mathcal{X} = \mathbb{R}^D$ of dimension D, resulting in *continuous* or *semi-continuous* [107, 108] HMMs. A continuous HMM effectively models trajectories in observation space.

The observations themselves do not necessarily display Markovian statistics. Instead, it is assumed that there are underlying structures in the observation space that do behave in a Markovian manner. Specifically, a finite number of regions are identified in the observation space, where each region is described by a probability density function (pdf) over $\mathcal{X}$. This allows the regions to overlap,

[13]HMMs were originally referred to as *probabilistic functions of Markov processes/chains*, for example in the classic papers of Baum and his colleagues [103, 104, 105, 106].

[14]This is frequently called *feature space*, where the observations x become *feature vectors*.

as each $x \in \mathcal{X}$ belongs to a varying degree to each region, as described by the set of pdfs. The regions are referred to as *symbols*.[15] Each symbol has a label $s \in \Sigma$ taken from an alphabet Σ, and an associated *symbol probability density function*[16] $\sigma(s,x)$ for $x \in \mathcal{X}$.

The HMM assumes that the symbols show a Markovian dependence, whereby a sequence of symbols can be adequately modelled by a Markov chain of the appropriate order. This dependence is obscured, however, because the symbols are not directly observed. Instead, the actual observations can be interpreted as fuzzy or corrupted versions of the underlying symbols when the symbol pdfs overlap. This is the *hidden* aspect of the model.

A classic example of a time series that shows this kind of behaviour is a speech signal. Speech consists of a sequence of basic sounds, or phones. The recorded signal of a specific phone varies a lot, depending on the context in which it is produced, the speaker, and so forth. This complicates the process of phone recognition and frequently causes some phones to be confused with others. Phones are therefore not completely distinguishable when observed in a recorded speech signal. On the other hand, natural language constrains the sequence of phones by grouping them into words and sentences. These constraints can be crudely modelled as Markovian. It is therefore natural to attempt to model speech with an HMM, where the symbols represent phones.

An Lth-order hidden Markov model is defined as a 4-tuple $(\mathcal{X}, \Sigma, \sigma, \mathcal{M}_L)$, where $\mathcal{X}$ is the D-dimensional observation space, Σ is an alphabet of M symbol labels, $\sigma : \Sigma \times \mathcal{X} \to \mathbb{R}^+$ are the symbol probability density functions for various s, and $\mathcal{M}_L = (\Sigma, \mathcal{Q}, \tau, \eta)$ is the underlying Lth-order Markov chain with $N = |\mathcal{Q}|$ states. The symbol pdfs are normalised so that $\int_{\mathcal{X}} \sigma(s,x)dx = 1$ for all $s \in \Sigma$. An example of a continuous HMM is shown in Figure 2.7.

Recall that the symbol sequence s_1^T has a one-to-one correspondence with the state sequence q_0^T of a Markov chain, since $q_t = \tau(q_{t-1}, s_t)$ and s_t is the last sym-

[15]In the standard HMM nomenclature (e.g. in [49, 95]), symbols are only found in discrete HMMs, where they refer to the elements of the discrete observation space. By redefining symbols to be the observation densities (or regions) themselves, a closer fit is achieved to the notation of Markov chains, especially for continuous HMMs. This has advantages for higher-order HMMs, where it improves the notation by clearly separating the concepts of state and symbol.

[16]This is commonly referred to as an observation pdf [49], output pdf or emission distribution [2], where s is usually considered a state index instead of a symbol label.

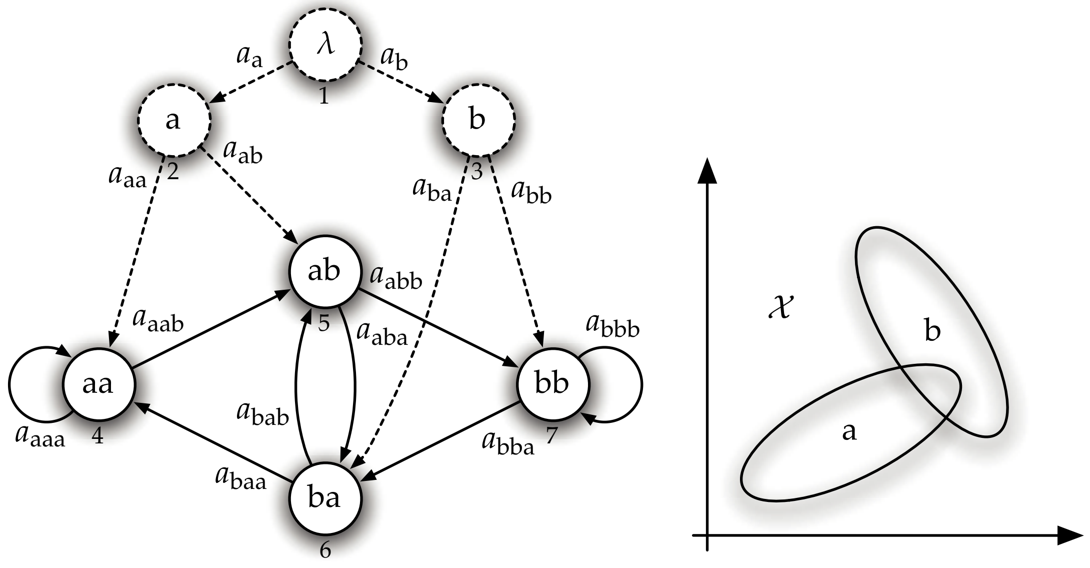


Figure 2.7: An example of a second-order continuous HMM with seven states and two symbols $\Sigma = \{a,b\}$ in observation space $\mathcal{X} = \mathbb{R}^2$. The HMM therefore describes sequences of two-dimensional feature vectors. The symbol pdfs are assumed to be Gaussian, and are represented by their 1σ contours on the plot. They show substantial overlap.

bol in the context string of q_t. HMM calculations tend to be simpler in terms of states instead of symbols, since the states always have first-order dependencies while the symbols potentially have higher-order dependencies. It is therefore useful to reinterpret the HMM as the 6-tuple $(\mathcal{X}, \Sigma, \sigma, \mathcal{Q}, s_q, a)$, where $\mathcal{X}, \Sigma, \sigma$ and $\mathcal{Q}$ have the same meanings as before, $s_q : \mathcal{Q} \to \Sigma$ is the state-to-symbol mapping where $s_q(q)$ is the last symbol of the context string of q, and $a : \mathcal{Q} \times \mathcal{Q} \to [0,1]$ is the state transition probability function (i.e. the elements of the transition matrix $\boldsymbol{A}$).

The Markov assumption of the underlying Markov chain $\mathcal{M}_L$ can be restated in the context of the HMM as

$$\begin{aligned} P(q_{t+1} | x_1^t, q_0^t) &= P(q_{t+1}|q_t) \\ &= P(\tau(q_t, s_{t+1})|q_t) \\ &= P(s_{t+1}|q_t) = \eta(q_t, s_{t+1}). \end{aligned} \tag{2.4.1}$$

This means that, given knowledge of the current state, the probability of the next state (or next symbol) is independent of the past states and past and current observations.

2.4.1 Scoring and generating data

The probability (or pdf height in the case of continuous HMMs) $P(x_1^T)$ assigned by an HMM to an observation sequence x_1^T is more difficult to compute than in the case of Markov chains, as x_1^T can arise from many underlying state sequences q_0^T. The *joint sequence probability density function* $P(x_1^T, q_0^T)$ has a simpler form, and can be factored as

$$P\left(x_1^T, q_0^T\right) = P\left(x_1^T \middle| q_0^T\right) P\left(q_0^T\right).$$

The state sequence probability $P(q_0^T)$ is calculated by the underlying Markov chain $\mathcal{M}_L$ of the HMM as a product of state transition probabilities

$$P\left(q_0^T\right) = \prod_{t=1}^{T} a(q_{t-1}, q_t).$$

The conditional sequence pdf $P(x_1^T | q_0^T)$ can be factored via the chain rule as

$$P\left(x_1^T \middle| q_0^T\right) = P\left(x_1 \middle| q_0^T\right) \prod_{t=2}^{T} P\left(x_t \middle| x_1^{t-1}, q_0^T\right).$$

In order to make this decomposition tractable, the HMM *output independence assumption* is introduced, which states that

$$P\left(x_t \middle| x_1^{t-1}, q_0^t\right) = P(x_t | q_t) = \sigma(s_q(q_t), x_t). \tag{2.4.2}$$

This means that, given knowledge of the current underlying symbol (or the current state), the probability of the current observation is independent of past observations and states.[17] The joint sequence pdf therefore simplifies to

$$P\left(x_1^T, q_0^T\right) = \prod_{t=1}^{T} a(q_{t-1}, q_t)\sigma(s_q(q_t), x_t). \tag{2.4.3}$$

The observation sequence pdf is then obtained by marginalising the joint sequence pdf over all possible state sequences, resulting in

$$P\left(x_1^T\right) = \sum_{q_0^T \in \mathcal{Q}^T} P\left(x_1^T, q_0^T\right). \tag{2.4.4}$$

This sum is computationally intractable if done directly, as the number of state sequences grows exponentially with T. Fortunately, a tractable recursive version of (2.4.4) known as the *forward* algorithm [104, 49, 2] exists, which makes effective use of the Markov assumption (2.4.1) and the output independence assumption (2.4.2) to reduce the number of calculations. The algorithm focuses on $P(x_1^t, q_t)$, the probability of observing the subsequence x_1^t while also ending up in state q_t at time t. Using (2.4.1) and (2.4.2), this can be written as

$$\begin{aligned}
P(x_1^t, q_t) &= P\left(x_t \middle| x_1^{t-1}, q_t\right) P\left(x_1^{t-1}, q_t\right) \\
&= P(x_t|q_t) \sum_{q_{t-1}} P\left(x_1^{t-1}, q_t, q_{t-1}\right) \\
&= P(x_t|q_t) \sum_{q_{t-1}} P\left(q_t \middle| x_1^{t-1}, q_{t-1}\right) P\left(x_1^{t-1}, q_{t-1}\right) \\
&= P(x_t|q_t) \sum_{q_{t-1}} P(q_t|q_{t-1})\, P\left(x_1^{t-1}, q_{t-1}\right).
\end{aligned}$$

This probability is frequently referred to as the *forward variable* [49] $\alpha(t, j) = P(x_1^t, q_t = j)$, which simplifies the calculation of $P(x_1^T)$ to the three steps shown in Figure 2.8 (where it is assumed that state λ has index $j = 1$). The computational complexity of the forward algorithm is $O(GT)$, where $G \leq MN$ represents the number of non-zero transition probabilities. The memory requirements are minimal, involving two buffers of size N that contain $\alpha(t, j)$ and $\alpha(t-1, j)$.

[17] There is still an *indirect* dependence on past values, as the current state depends on the previous state, and so forth. This assumption merely states that knowledge of the current symbol allows the best possible prediction of the current observation.

Forward	
1. Initialise:	$\alpha(0,j) = \begin{cases} 1, & j=1 \\ 0, & \text{otherwise} \end{cases}$
2. Recurse:	$\alpha(t,j) = \left[\sum_{i=1}^{N} \alpha(t-1,i) a_{ij} \right] \sigma(s_q(j), x_t)$
3. Terminate:	$P(x_1^T) = \sum_{j=1}^{N} \alpha(T,j)$

Figure 2.8: The forward algorithm for calculating $P(x_1^T)$.

The HMM is a generative model like the Markov chain, which implies that it can efficiently generate random observation sequences. The generation process has two components: the underlying MC generates a symbol sequence, and each symbol is independently transformed to an observation by sampling from the appropriate symbol pdf $\sigma(s,x)$.

2.4.2 Obtaining the optimal state sequence

In many applications of HMMs, the state (or symbol) sequence has meaning in itself. For example, when HMMs are used to model speech, the symbol sequence might reflect the underlying sequence of phones, words or other speech units. It is therefore useful to estimate the state sequence that best matches a given observation sequence. The most likely state sequence q_0^{T*} associated with observation sequence x_1^T is given by

$$q_0^{T*} = \arg\max_{q_0^T} P\left(q_0^T \middle| x_1^T\right) = \arg\max_{q_0^T} P\left(q_0^T, x_1^T\right).$$

The *Viterbi* algorithm [48, 109] is an efficient procedure for calculating q_0^{T*}, based on Bellman's dynamic programming [110]. It recursively updates the quantity

$$V(t,j) = \max_{q_0^{t-1}} P\left(x_1^t, q_0^{t-1}, q_t = j\right),$$

which can be interpreted as the best score produced by a single state sequence which accounts for the first t observations and ends up in state j at time t. At the

Viterbi	
1. Initialise:	$V(0,j) = \begin{cases} 1, & j=1 \\ 0, & \text{otherwise} \end{cases}$
2. Recurse:	$V(t,j) = \left[\max_i V(t-1,i)a_{ij}\right] \sigma(s_q(j), x_t)$
	$i^*(t,j) = \arg\max_i V(t-1,i)a_{ij}$
3. Terminate:	$V^* = \max_j V(T,j)$
	$q_T^* = \arg\max_j V(T,j)$
4. Backtrack:	$q_{t-1}^* = i^*(t, q_t^*)$

Figure 2.9: The Viterbi algorithm for calculating ${q_0^T}^*$.

same time, the best previous state $i^*(t,j)$ associated with state j at time t is also recorded. It calculates

$$\begin{aligned} V(t,j) &= P(x_t|q_t=j)\max_i \left[P(q_t=j|q_{t-1}=i)V(t-1,i)\right] \\ i^*(t,j) &= \arg\max_i \left[P(q_t=j|q_{t-1}=i)V(t-1,i)\right] \end{aligned}$$

in a forward pass from $t = 1$ to $t = T$. This produces $V^* = \max_{q_0^T} P(q_0^T, x_1^T) = \max_j V(T,j)$, which is the maximum score achieved by ${q_0^T}^*$. The sequence itself is obtained by starting at $q_T^* = \arg\max_j V(T,j)$ and following the path matrix $i^*(t,j)$ back in time as $q_{t-1}^* = i^*(t, q_t^*)$. This *backtracking* procedure is a standard feature of dynamic programming.

The Viterbi algorithm is summarised in Figure 2.9. Its recursion step is similar to that of the forward algorithm in Figure 2.8, replacing the sum over previous states with a maximisation. The computational complexity of the Viterbi algorithm is therefore also $O(GT)$, where $G \leq MN$ represents the number of non-zero transition probabilities. Its memory requirements are higher than that of the forward algorithm, though, as the path matrix $i^*(t,j)$ with NT elements has to be stored in its entirety.

2.4.3 Learning HMMs from data

The last important procedure involving an HMM is to infer it from training data. A distinction is commonly made between the *topology* or structure of the model, and the remaining HMM parameters [2]. The topology of an HMM refers to the set of state context strings $\mathcal{Q}$ (trivial in the case of first-order models), as well as the set of all state transitions with non-zero probability. This determines the graph structure and node labels of the underlying MC. Once the topology of the HMM is specified, the remaining parameters are the transition probabilities and symbol pdfs, which will be indicated by θ.

It is customary to specify the topology of the model beforehand, based on knowledge of the problem domain. For example, a popular topology in speech modelling is the *left-to-right* or Bakis configuration [111, 112], which arranges the HMM states in a sequence and only allows transitions in one direction along the sequence. An example is shown in Figure 2.10. This first-order HMM is frequently used to represent phones, where the three states correspond to the beginning, middle and end of the phone. It therefore models the phone as a trajectory that visits three consecutive regions in observation space. Another popular topology is the ergodic HMM, such as the example in Figure 2.7, where the underlying MC is ergodic. The ability to specify the structure of an HMM in a flexible way to match the problem at hand is indeed one of the strengths of this model.

Once the topology of the HMM is fixed, the transition probabilities and symbol pdfs must be estimated from the training data. The maximum likelihood estimate of θ is given by

$$\theta_{\text{ML}} = \arg\max_{\theta} P\left(x_1^T \middle| \theta\right). \tag{2.4.5}$$

This is a difficult optimisation problem for which no analytical solution exists [49]. As an alternative, there are iterative procedures that increment the likelihood step by step until a local maximum is reached. The most popular of these are based on the EM algorithm, which is discussed next.

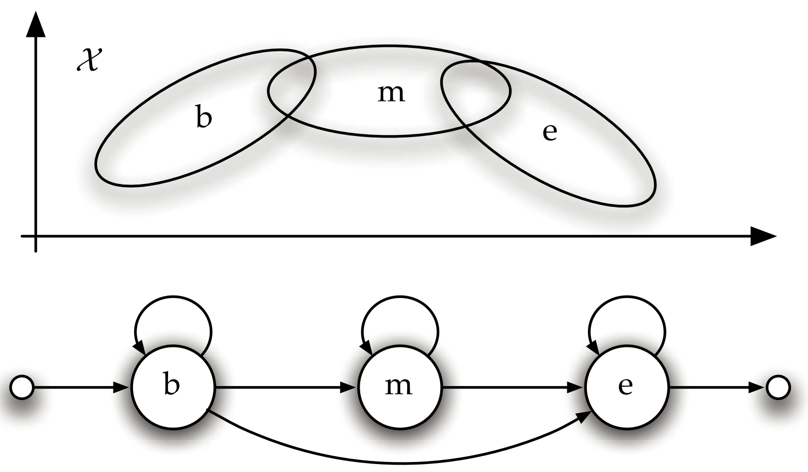


Figure 2.10: An example of a left-to-right HMM, which is a popular topology for speech phone models. The edges of the graph indicate transitions with non-zero probability. The symbols `b`, `m` and `e` correspond to the beginning, middle and end regions of the phone, respectively. Examples of these regions are indicated in observation space $\mathcal{X}$ by the contours of the associated Gaussian symbol pdfs. Note that this HMM is not ergodic.

2.4.4 The EM algorithm

The *Expectation-Maximisation* (EM) algorithm is a general optimisation procedure that maximises the likelihood of a statistical model with hidden variables. The original ideas behind the algorithm have been around since the 1950s, for example in [113] and the work of Baum and his colleagues on HMM training [103, 106], but the seminal paper of [114] illustrated the generality of the approach and coined the name. Modern textbooks on the algorithm include [115] and [116]. We follow the derivation of Neal and Hinton [117, 118, 119], which provides a deeper understanding of the algorithm and its convergence properties.

Consider a statistical model with parameters θ that models two random variables, X and Q. Suppose that we have observed the value of variable $X = x$, but not the value of variable Q. We refer to X as the observed variable and to Q as the hidden or latent variable. We wish to obtain the maximum likelihood estimate $\theta_{\text{ML}} = \arg\max_\theta P(x|\theta)$, given the observed data x. Alternatively, we can maximise the log likelihood $L(\theta) = \log P(x|\theta)$. If the joint distribution of the observed and hidden variables $P(x, q|\theta)$ is simpler in form than the marginal distribution of the observed variable $P(x|\theta) = \int P(x,q|\theta)dq$ that we wish to maximise, we can use it to make the problem tractable.[18]

The key idea behind the EM algorithm is to maximise a lower bound of the likelihood instead of the likelihood itself. This bound is obtained by an application of Jensen's inequality [65], which states, informally, that the logarithm of a weighted average (or convex linear combination) is always greater than or equal to the weighted average of the logarithms, since the logarithm is a concave function. Thus we have

$$\begin{aligned} L(\theta) = \log P(x|\theta) &= \log\left[\int P(x,q|\theta)dq\right] = \log\left[\int W(q)\frac{P(x,q|\theta)}{W(q)}dq\right] \\ &\geq \int W(q)\log\frac{P(x,q|\theta)}{W(q)}dq, \end{aligned} \tag{2.4.6}$$

for any weight function $W(q)$ on the sample space of the hidden variable Q which satisfies the properties of a pdf: $W(q) \geq 0$ and $\int W(q)dq = 1$.

The bound in (2.4.6) can be written in terms of the introduced weight function W and model parameters θ as

$$F(W,\theta) = \int W(q)\log\frac{P(x,q|\theta)}{W(q)}dq = \mathrm{E}_W\left[\log\frac{P(x,q|\theta)}{W(q)}\right],$$

where the expectation operator $\mathrm{E}_W[\cdot]$ is compact notation for a weighted average over the values of the hidden variable Q, with respect to weights W. The properties of the bound become clear if the bound is rewritten in two different ways. The first way isolates θ in a simpler term as

$$\begin{aligned} F(W,\theta) &= \mathrm{E}_W[\log P(x,q|\theta)] - \mathrm{E}_W[\log W(q)] \\ &= \mathrm{E}_W[\log P(x,q|\theta)] + H(W), \end{aligned} \tag{2.4.7}$$

where $H(W)$ is the entropy of distribution W. The second way isolates W in a

[18]The joint distribution $P(x,q|\theta)$ is also known as the *complete likelihood*, while the marginal distribution $P(x|\theta)$ is the *incomplete likelihood*.

simpler term as

$$\begin{aligned} F(W,\theta) &= \mathrm{E}_W\left[\log\frac{P(x|\theta)P(q|x,\theta)}{W(q)}\right] \\ &= \mathrm{E}_W[\log P(x|\theta)] - \mathrm{E}_W\left[\log\frac{W(q)}{P(q|x,\theta)}\right] \\ &= L(\theta) - \mathrm{D}(W\|P_\theta)\,, \end{aligned} \tag{2.4.8}$$

where $P_\theta(q)$ is shorthand for $P(q|x,\theta)$. From (2.4.8) it is immediately obvious that $F(W,\theta) \leq L(\theta)$ for any W and θ, as the Kullback-Leibler divergence is always non-negative.

Equation (2.4.7) has an analogy in statistical physics, where it defines the *variational free energy* [117], a quantity that is used to approximate the more fundamental *free energy* of a system of particles. In the analogy, $-F(W,\theta)$ represents the variational free energy, q represents the state of a physical system, $-\log P(x,q|\theta)$ represents the energy of state q, $-L(\theta)$ represents the free energy F, $P(x|\theta)$ becomes the *partition function* Z, and $P_\theta(q)$ is known as the *Boltzmann*, *Gibbs*, *canonical* or *equilibrium distribution* that minimises the variational free energy. Statistical physics has had a valuable influence on statistical inference, inspiring concepts such as mean field approximations, Ising models, Boltzmann neural networks, deterministic annealing and variational Bayes methods. The EM algorithm also belongs to this family of techniques, which is explored in more detail in [65].

Let $W = W_0 = P_{\theta_0}$ for a specific choice of $\theta = \theta_0$. The lower bound $F(W_0,\theta)$ then touches the objective function $L(\theta)$ at θ_0 without crossing it, as the divergence in (2.4.8) becomes zero. This implies that the gradients of $F(W_0,\theta)$ and $L(\theta)$ with respect to θ are equal at θ_0, assuming the gradients exist and are finite. If $F(W,\theta)$ achieves a local maximum at W^* and θ^*, we also have $W^* = P_{\theta^*}$, as this maximises the right-hand side of (2.4.8). Since θ^* is a stationary point of $F(W^*,\theta)$, it is also a stationary point of $L(\theta)$. Moreover, it is a local maximum of $L(\theta)$, due to continuity. See [117] for a proof based on the assumption that P_θ varies continuously with θ.

Maximisation of the lower bound therefore leads directly to maximisation of the log likelihood. Neal and Hinton [117] show that any technique for maximising or even just increasing $F(W,\theta)$ is acceptable, leading to many variations of the EM algorithm. Nevertheless, Minka [118] points out that generic gradient ascent on $F(W,\theta)$ is no different from gradient ascent on $L(\theta)$, in which case nothing is gained by using the EM algorithm. The standard approach to the maximisation of $F(W,\theta)$ is to use *coordinate ascent*. That is, the standard EM algorithm starts with an initial guess of the model parameters, θ^0, and then alternately maximises $F(W,\theta)$ with respect to W and θ until convergence.

Maximisation with respect to W is known as the *expectation step* or *E step*, and is simplest to do on the version of the bound in (2.4.8). This turns out to be a simple distance minimisation, with optimum

$$\textbf{[E]}\quad W^k(q) = \arg\max_W F\left(W,\theta^{k-1}\right) = P_{\theta^{k-1}}(q) = P\left(q\middle|x,\theta^{k-1}\right). \tag{2.4.9}$$

Maximisation with respect to θ is known as the *maximisation step* or *M step*, and is more natural to do on the version of the bound in (2.4.7). This is problem-dependent, with optimum

$$\textbf{[M]}\quad \theta^k = \arg\max_\theta F\left(W^k,\theta\right) = \arg\max_\theta \mathrm{E}_{W^k}[\log P(x,q|\theta)]\,. \tag{2.4.10}$$

The algorithm takes its name from these two steps. It is sensible to use EM as long as $P(q|x,\theta)$ and $P(x,q|\theta)$ are easier to compute than $P(x|\theta)$.

The effect of these two steps on the lower bound is illustrated in Figure 2.11. The E step finds the optimum bound $F(W^k,\theta)$ that touches the likelihood function at θ^{k-1}, and the M step finds the optimum θ^k that maximises this bound. This process is repeated until θ^k converges to a local maximum at θ^*. The EM algorithm can hereby be seen as a form of gradient ascent, with an automatically determined step size that generally ensures fast convergence. It also suffers from the same problems as gradient ascent, such as convergence to saddle points and possibly even to minima if the initial guess θ^0 happens to fall on a minimum (see [118] for an example).

The EM algorithm replaces the missing value of Q with a distribution $W(q)$ over the range of Q instead of a single value. If a single estimate is used instead, we obtain an approximate algorithm that is simpler to implement and typically faster than standard EM. This EM variant is referred to as *winner-take-all* in [117].

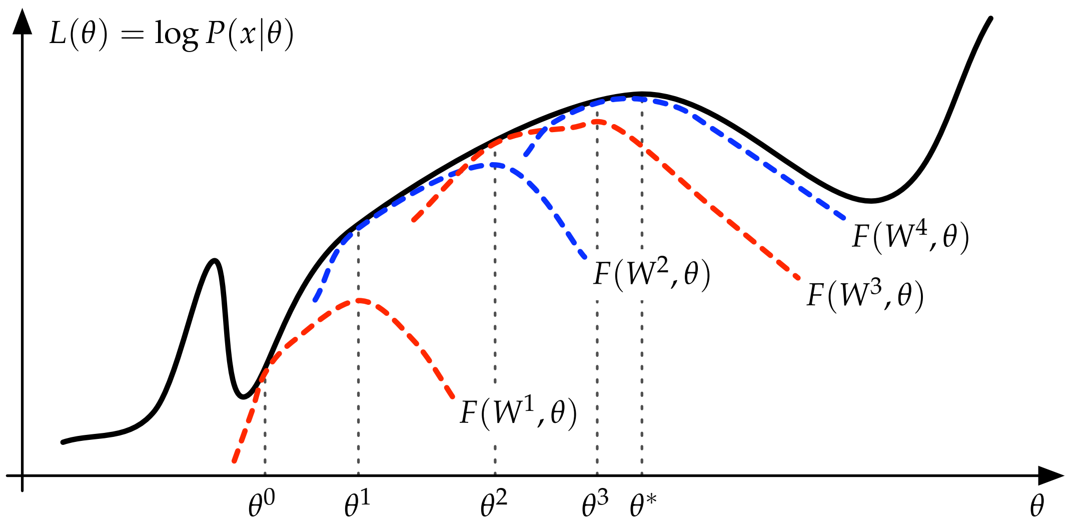


Figure 2.11: The EM algorithm as lower bound maximisation. The initial guess of the model parameters is θ^0. The E step finds the optimum bound $F(W^k, \theta)$ that touches the likelihood function at θ^{k-1}, and the M step finds the optimum θ^k that maximises this bound. This process converges to a local maximum of $L(\theta)$ at θ^*.

The approximation is good if the hidden variable posterior $P(q|x, \theta)$ is highly peaked at a single dominant value $q = q_*$. It can be interpreted as a "hard" approximation to "soft" EM.

The winner-take-all variant restricts the weight function W to be an impulse, that is

$$W(q) = \delta(q - q_*) = \begin{cases} 1, & q = q_*, \\ 0, & q \neq q_*, \end{cases} \tag{2.4.11}$$

where we assume that Q is discrete for simplicity. This weight function assigns all probability mass to a specific value q_* of q, and zero probability to the rest. The lower bound $F(W, \theta)$ can now be rewritten in terms of q_* as

$$\begin{aligned} F(q_*, \theta) &= \sum_q \delta(q - q_*) \log \frac{P(x, q|\theta)}{\delta(q - q_*)} \\ &= \log P(x, q_*|\theta). \end{aligned}$$

This EM variant therefore uses the *joint distribution* $P(x, q|\theta)$ as a lower bound for the likelihood $P(x|\theta)$. As with standard EM, the bound is maximised by co-ordinate ascent, which alternately maximises $P(x, q|\theta)$ with respect to q (E step) and θ (M step).

Maximisation of $F(q, \theta)$ is not equivalent to maximisation of $F(W, \theta)$, due to the extra constraint of (2.4.11). The winner-take-all method therefore does not guarantee a monotone increase in likelihood, and generally also does not converge to a local maximum of $L(\theta)$. Nevertheless, it increases a lower bound on the likelihood, it is guaranteed to converge [120], and it has been successfully applied in several domains. An example is the well-known K-means clustering algorithm [121, 122, 65], which is a winner-take-all variant of the standard EM algorithm for Gaussian mixture models (GMMs) [117]. Instead of weighting the contribution of each data point to each cluster like EM, the K-means algorithm assigns each point to the nearest cluster in a "hard" fashion.

2.4.5 HMM training revisited

An HMM can be regarded as a statistical model with hidden variables, where the observation sequence x_1^T is the observed variable and the state sequence q_0^T is the (discrete) hidden variable. The likelihood $P(x_1^T|\theta)$ can therefore be maximised by the EM algorithm.

The E step (2.4.9) determines a set of weights $W(q_0^T)$ defined on all possible state sequences as

$$W^k(q_0^T) = P\left(q_0^T \middle| x_1^T, \theta^{k-1}\right).$$

Instead of committing to a single state sequence, the EM algorithm therefore provides a distribution over all state sequences. These weights can be decomposed into a set of state posterior probabilities $P(q_t = j|x_1^T, \theta)$ and transition posterior probabilities $P(q_t = j, q_{t-1} = i|x_1^T, \theta)$, due to the conditional independence structure of the HMM imposed by (2.4.1) and (2.4.2) (see [2] for details). The posterior probabilities can be calculated efficiently using the forward recursion of Figure 2.8 and a corresponding backward recursion. In the M step of (2.4.10), the model parameters θ are updated to

$$\theta^k = \arg\max_{\theta} \mathrm{E}_{W^k}\left[\log P\left(x_1^T, q_0^T \middle| \theta\right)\right],$$

which has the form of a weighted maximum likelihood estimate and is also an efficient process for many standard choices of symbol pdf.

This application of the EM algorithm to HMM training is known as the *Baum-Welch* or *forward-backward re-estimation* algorithm [103, 104, 105, 106, 49], and it was one of the precursors of general EM. The algorithm is frequently explained in terms of Baum's *auxiliary function* [49, 2], which is related to the lower bound $F(W, \theta)$ by

$$Q\left(\theta^{k-1}, \theta\right) = \mathrm{E}_{P_{\theta^{k-1}}}[\log P(x, q|\theta)] = F\left(P_{\theta^{k-1}}, \theta\right) - H\left(P_{\theta^{k-1}}\right).$$

In terms of the auxiliary function, the E step determines $Q(\theta^{k-1}, \theta)$ and the M step maximises the auxiliary function to obtain the next model estimate as $\theta^k = \arg\max_\theta Q(\theta^{k-1}, \theta)$. More details on the Baum-Welch algorithm can be found in [49, 2].

In this study we focus on a related training procedure known as *Viterbi re-estimation* or *segmental k-means* [51], a winner-take-all variant of Baum-Welch re-estimation which is faster but suboptimal. It maximises the joint distribution $P(x_1^T, q_0^T|\theta)$, which serves as a lower bound for the likelihood $P(x_1^T|\theta)$, using co-ordinate ascent. The E step determines the optimal state sequence q_0^{T*} using the Viterbi algorithm of Figure 2.9, based on the training data and a previous estimate of θ. The M step then calculates the maximum likelihood estimate of θ, based on the training data and the state sequence obtained in the E step. Convergence is checked by observing the Viterbi score V^* after each iteration.

The maximum likelihood estimates of the state transition probabilities a_{ij} and symbol pdfs $\sigma(s, x)$ are calculated independently, since the joint distribution can be factored as in (2.4.3). The transition probabilities form part of the underlying Markov chain, and are estimated from the state sequence in a similar fashion to (2.2.2). The maximum likelihood estimate is

$$\hat{a}_{ij} = \frac{\#(i \cdot j)}{\#(i*)},$$

where $\#(i \cdot j)$ is the number of times that state i is followed by state j in the state sequence, and $\#(i*) = \sum_{j=1}^{N} \#(i \cdot j)$. On the symbol pdf side, each observation x_t

```
Viterbi re-estimation
1. Obtain initial model $\theta$
2. Iterate until convergence (based on Viterbi score $V^*$):
    E: Obtain optimal state sequence $q_0^{T*}$ using Viterbi
    M: Update model to $\theta_{\mathrm{ML}} = \arg\max_{\theta} P\left(x_1^T, q_0^{T*} \middle| \theta\right)$
       In terms of transition probabilities $a_{ij}$ and symbol pdf parameters $\theta_s$:
       $\hat{a}_{ij} = \frac{\#(i \cdot j)}{\#(i*)}$
       $\hat{\theta}_s = \arg\max_{\theta_s} \prod_{x \in X_s} P(x|\theta_s)$, for $X_s = \{x_t \mid s_q(q_t) = s\}$
```

Figure 2.12: The Viterbi re-estimation algorithm.

is assigned to a symbol estimation set

$$X_s = \{x_t \mid s_q(q_t) = s\}$$

associated with the symbol pdf of state q_t. The observations in each set are assumed to be independent and identically distributed, and form the training data for the corresponding symbol pdf. The maximum likelihood estimates of the parameters θ_s of symbol pdf $\sigma(s, x)$ are then obtained by

$$\hat{\theta}_s = \arg\max_{\theta_s} \prod_{x \in X_s} P(x|\theta_s),$$

where $P(x|\theta_s) = \sigma(s, x)$. The algorithm is summarised in Figure 2.12.

Viterbi re-estimation is guaranteed to converge [51, 120], but typically does not converge to a local maximum of the likelihood, as it is a winner-take-all variant of EM. Its computational complexity is dominated by the E step for standard choices of symbol pdf, and can be expressed as $O(KGT)$, where K is the average number of iterations until acceptable convergence (typically 20-30), $G \leq MN$ represents the number of non-zero transition probabilities and T is the size of the training set.

An important aspect of HMM training is proper initialisation of the model parameters. The likelihood function is highly complex in most typical problems, and both Baum-Welch and Viterbi re-estimation can get stuck on poor local maxima. This is alleviated by proper specification of the HMM structure and initialisation of the symbol pdfs, for example by unsupervised clustering.

2.5 Connections with other finite-state models

The models described in this chapter all have a natural connection with finite-state automata [123, 124, 125]. This connection inspires the graphical representation of these models as state machines. The most general model in this class is the *probabilistic finite-state automaton* (PFA or PFSA) [126, 95], which is associated with a stochastic regular grammar [127, 41].

Discrete hidden Markov models are equivalent to PFAs [128, 95]. Markov chains, on the other hand, are encountered in the theoretical computer science literature as *k-testable stochastic automata in the strict sense* (k-TSAs) [129, 95] or *finite context automata* (FCAs) [10], a subclass of PFAs. A k-TSA is equivalent to a $(k-1)$-MC, and the special case of a 2-TSA (or first-order Markov chain) is known as a *stochastic local language* [95].

Markov chains are also commonly encountered in the fields of natural language processing and speech recognition as *n-gram models* [3]. A *bigram* is a pair of symbols (typically words or phonemes), a *trigram* is a triplet of symbols, and so forth. The bigram ($n = 2$) model is equivalent to a first-order Markov chain, trigrams ($n = 3$) correspond to a 2-MC, and, in general, an n-gram model is equivalent to an $(n-1)$-MC. N-gram models for $n > 3$ are uncommon, as the large alphabet size typically found in natural language problems makes them very difficult to estimate from the available data.

Probabilistic suffix automata and prediction suffix trees are PFA subclasses as well, due to their equivalence to Markov chains. Several studies suggest that the general class of PFAs and discrete HMMs cannot be efficiently learnt from data [130, 131, 132]. However, Ron proves the following theorem in [22]: Given a bound L on the order of a source PSA, and a bound $N_{\max}$ on the number of states, the Learn-PSA algorithm will generate a model arbitrarily close to the source PSA, from a set of sequences generated by the source model, in time polynomial in L, $N_{\max}$, alphabet size M, and the desired accuracy parameters [20]. PSTs (and PSAs) therefore form an *efficient* subclass of PFAs.

An interesting finite-state model with modelling power between that of a Markov chain and a discrete HMM is the deterministic probabilistic finite automaton (DPFA) [60]. It is not a hidden-variable model like the HMM, as it maintains the one-to-one relationship between a symbol sequence and its underlying state sequence. Unlike a Markov chain, however, it can model dependencies between symbols that are arbitrarily far apart in the sequence. The examples which follow illustrate the difference between MCs, DPFAs and HMMs.

Figure 2.13 shows a DPFA with five symbols and eight states. Each state emits a single symbol with probability one, except states 1 and 8, which are the initial and final states. It *admits* or assigns a non-zero probability to symbol sequences such as

`abc`, `abbbbc`, `dbe`, `dbbbe`, ...

It is always possible to deduce the state sequence from the symbol sequence, as the initial or final symbol uniquely identifies which of the two paths in the model graph was followed. This DPFA cannot be reduced to a Markov chain, as the identity of the final symbol depends on the symbol which preceded the intervening `b`'s, of which there may be arbitrarily many. The problem is that the DPFA in Figure 2.13 contains two states that emit symbol `b`, and while both states resemble first-order contexts, their next-symbol distributions differ.

The closest Markov chain to this DPFA is shown in Figure 2.14. It merges the two problematic states of the DPFA to create a unique first-order context for `b`. This destroys the long-range dependence between the initial and final symbols, and expands the set of admitted symbol sequences to

`abc`, `abbbbc`, `abe`, `abbbe`, `dbc`, `dbbbbbc`, `dbe`, `dbbbe`, ...

All strings admitted by the DPFA are therefore also admitted by the MC, but the MC assigns them half the probability that the DPFA does. It preserves the relative probabilities of these strings, though. The other half of the probability mass is assigned to extra strings not admitted by the DPFA.

Figure 2.15 shows an example of a discrete HMM that cannot be reduced to

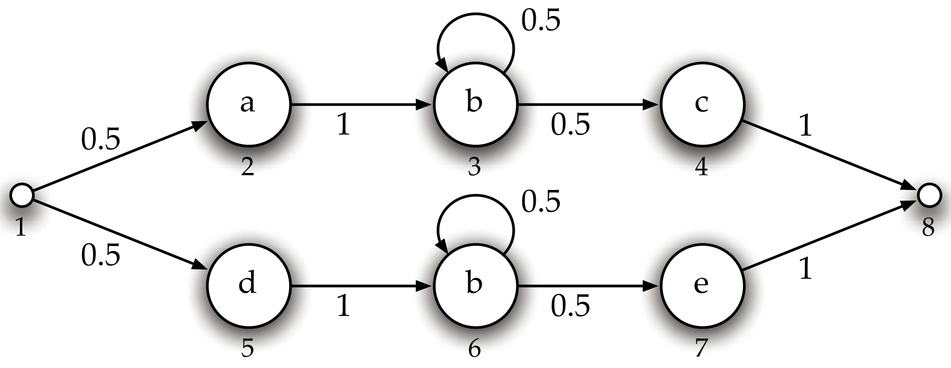


Figure 2.13: Example of a deterministic probabilistic finite automaton (DPFA) that cannot be reduced to a Markov chain. Each state emits the symbol printed inside its circle with probability one, while the state index is indicated below the circle. State 1 is the initial state and state 8 is the final state, which do not emit symbols.

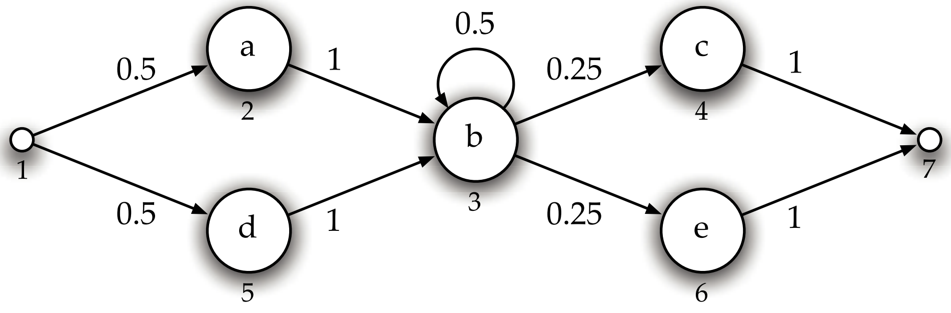


Figure 2.14: The Markov chain that is the closest match to the DPFA in Figure 2.13.

a DPFA. Its graph also contains two paths like that of the DPFA in Figure 2.13, but the paths of the HMM contain the same symbols in the same order. The only distinction between the paths is a different expected number of occurrences of the symbol `b` in the symbol sequence. The HMM therefore admits sequences of the form

$$\texttt{abc}, \texttt{abbc}, \texttt{abbbbbbc}, \texttt{abbbbbbbbc}, \ldots$$

where the number of `b`'s in the sequence is a random variable with a mixture distribution. The symbol sequence cannot unambiguously identify which of the two paths in the HMM graph was followed, which prevents the model from being reduced to a DPFA.

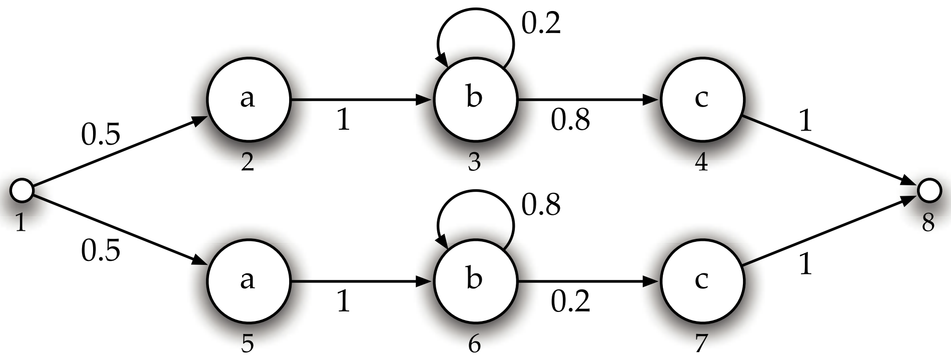


Figure 2.15: Example of a discrete HMM that cannot be reduced to a DPFA.

The surveys of [128, 133, 95] explore the connection between Markov chains, discrete HMMs, DPFAs and PFAs in detail. They point out that there are frequently confusion in the literature over the definitions and capabilities of these models.[19] Figure 2.16 illustrates the relationship between the various stochastic sequential models described in [133, 95, 22], which highlights that Markov chains and PSTs are equivalent, but discrete HMMs and PFAs are more powerful models.

2.6 Summary

This chapter provides a brief introduction to information theory and the concept of minimum description length. It defines a Markov chain as a mixed-order model from the onset, and carefully distinguishes between symbols and states, which are strings of symbols serving as contexts. While the concepts of symbols and states coincide in first-order models, which have context strings of length one, their distinction simplifies the notation for higher-order Markov models.

Prediction suffix trees are defined next, and are shown to be equivalent to mixed-order Markov chains. The basic Learn-PSA inference algorithm is also introduced.

The hidden Markov model is defined as a mixed-order Markov chain combined with a set of observation densities, which simplifies the discussions of mixed-order models. Section 2.4 reviews the basic HMM algorithms, such as

[19]For example, the PFSAs in [10] and [24, 7] are in actual fact DPFAs.

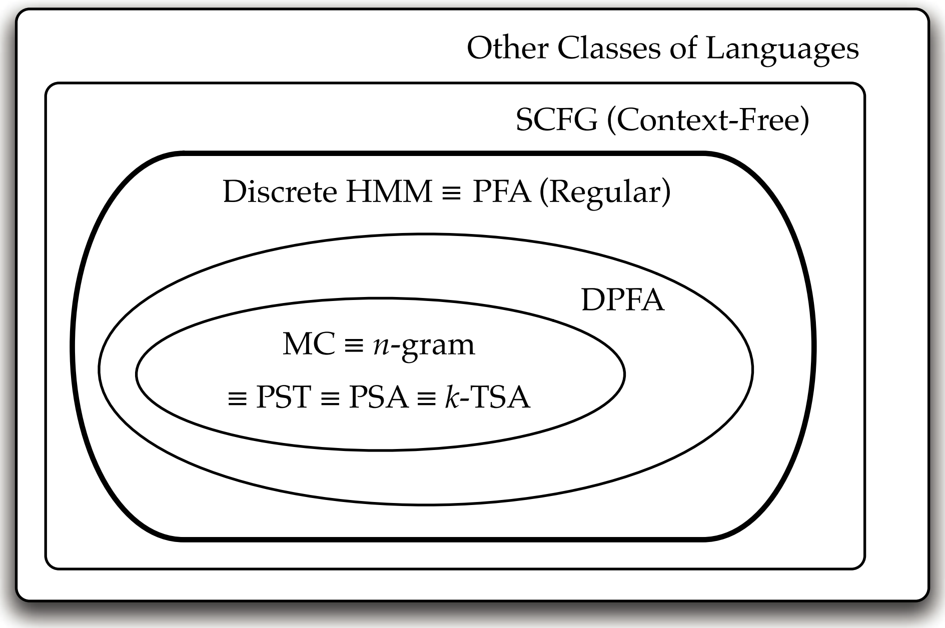


Figure 2.16: The relationship between various stochastic language models, including stochastic context-free grammars (SCFGs), stochastic regular grammars, discrete hidden Markov models, probabilistic finite-state automata (PFAs), deterministic probabilistic finite automata (DPFAs), Markov chains (MCs), *n*-gram models, prediction suffix trees (PSTs), probabilistic suffix automata (PSAs), and *k*-testable stochastic automata in the strict sense (*k*-TSAs). See [133, 95, 22] for more details.

the forward algorithm, Viterbi algorithm, forward-backward re-estimation and Viterbi re-estimation, along with an introduction to the Expectation-Maximisation approach.

Finally, HMMs and Markov chains are shown to be part of a family of finite-state statistical models. An interesting intermediate model is the deterministic probabilistic finite automaton (DPFA), which does not have the hidden nature of an HMM, but is effectively an infinite-order Markov chain.

Chapter 3

Literature Study

This chapter describes a selection of methods for the training of prediction suffix trees (PSTs) and the inference of hidden Markov model (HMM) topology from data. Some of these methods serve as a backdrop to the development of the smallest encoded context tree (SECT) and hidden SECT algorithms, while the rest illustrate the current state of the art in these problem domains.

3.1 PST algorithms

3.1.1 Learn-PSA

The original PST training algorithm is Learn-PSA, which is introduced by Ron, Singer and Tishby in [23, 98, 22] and is described in more detail in Section 2.3.5. The algorithm has five user-specified parameters. Two of these parameters, the tree depth bound L and the empirical probability threshold $P_{\min}$, control the expansion of the PST. The maximum memory length of the PST is therefore user-specified, or determined via a validation experiment. The algorithm is a batch or two-pass procedure. This requires the training data to be available for the computation of next-symbol counts while the PST is grown or, alternatively, the relevant counts are collected for contexts up to the desired depth bound L before the PST inference starts. The worst-case computational complexity of Learn-PSA is $O(LT^2)$, where T is the total length of the training data. In terms of memory requirements, Learn-PSA either stores the training set of length T or a set of next-symbol counts of maximum size $O(M^{L+1})$, where M is the number of symbols in the alphabet.

3.1.2 Guyon and Pereira's method

Guyon and Pereira use a slightly different approach to learn PSTs in [24, 7]. They first build a prefix tree (similar to a trie [134]) of depth $L+1$ by sliding a window of length $L+1$ over the training sequence. Each prefix string within the window is added as a node to the tree if it is not there already, otherwise its node count is incremented. This operation has a relatively low computational complexity of $O(LT)$, but a large storage requirement of up to $O(M^{L+2})$. When the algorithm runs out of memory, it prunes contexts with a probability of occurrence below a threshold ϵ_{p}.

The prefix tree is converted to a PST by a procedure similar to Learn-PSA, which recursively visits the nodes of the prefix tree up to depth L, and adds a context to the PST if the Kullback-Leibler divergence[1] between its next-symbol distribution and the next-symbol distribution of its suffix exceeds a threshold ϵ_{s}. This procedure has a worst-case computational complexity of $O(M^{L+1})$. The algorithm therefore has three user-specifiable parameters: the tree depth L and thresholds ϵ_{p} and ϵ_{s}. The threshold ϵ_{s} controls the capacity of the model in a more fine-grained manner than the order L, and is optimised on a validation set to provide a form of structural risk minimisation. The algorithm is also a batch procedure like Learn-PSA.

The final step of the method converts the PST into a PSA. Instead of smoothing the next-symbol probabilities, this PSA uses a back-off strategy [97] during sequence scoring. Whenever the current state assigns zero probability to the next symbol in the sequence, the state is reset to the initial state, thereby backing off to the zeroth-order empty context. The algorithm is tested by training a PSA on the AP news corpus of English text. The resulting PSA achieves a similar cross-entropy on the Brown corpus than a state-of-the-art word trigram model, for a fraction of the model size and training set size.

[1]The original Learn-PSA algorithm [22] measures the difference between next-symbol distributions by the maximum ratio between corresponding probabilities instead.

3.1.3 Bejerano's variants

Bejerano describes several alternative PST learning algorithms in his Ph.D. dissertation [20]. The "optimised" variant, also described in [25], addresses the computational complexity of Learn-PSA. It uses the efficient data structure of a classical suffix tree [135, 136] to achieve a computational complexity of $O(MT)$, as well as memory requirements that are linear in T. A compact suffix tree is constructed for the data set, and the nodes of this tree are visited from the bottom up. For each node, the next-symbol counts are calculated and the tests of Learn-PSA are carried out to determine whether the context should be included in the PST. If a context is included, all suffixes of the context are also included. Any excluded nodes are finally pruned from the tree. This is also a batch algorithm, due to the construction of the suffix tree. The optimised PST algorithm still has the same five user-specified parameters of Learn-PSA, which also control the maximum memory length of the model.

The "non-parametric" variant, also described in [26, 137], uses the minimum description length (MDL) principle to remove the need for user-specified parameters and to obtain a *self-bounded* tree. This means that the maximum tree depth is automatically adjusted to the amount of training data. A two-part message is constructed, which describes the PST and the training data in terms of the PST. The description length of a node q in the PST is given by

$$L(q) = M + M \log_2 \sqrt{\#(q*)}, \tag{3.1.1}$$

where the first term represents M one-bit flags that indicate the existence of the corresponding child nodes of q, and the second term codes the M next-symbol counts $\#(qs), s \in \Sigma$, to within an accuracy of the square root of their total $\#(q*)$, based on [47]. The description length of the PST is the sum of the description lengths of all the nodes in the tree. The data description length is the data set size, T, times the entropy of the PST.

The algorithm consists of two steps. The first step starts with an empty tree, and grows the PST by recursively inserting all potentially useful nodes. These nodes have a chance of coding the data assigned to them better than their parent nodes can, even after including their model overhead in the description length.

A node with context string $q = rp, r \in \Sigma$, and parent node p is *potentially useful* if it has a potentially useful parent and its node description length satisfies

$$L(q) < \#(q*)H(p),$$

where $H(p)$ is the entropy of the next-symbol distribution of the parent node. The second step in the algorithm recursively prunes the tree from the bottom up, by removing subtrees that did not deliver on their promise to improve the description length. That is, if the total description length of the subtree rooted at q exceeds the code length $\#(q*)H(p)$ assigned by the parent node p to the next-symbols of q, the subtree is removed.

This algorithm is a batch procedure, like the previous algorithms. The maximum tree depth is controlled by the data set size T, and up to $O(T)$ nodes are inserted in the first step of the algorithm, while the maximum depth L can be $O(T)$ in the worst case. Depending on how the next-symbol counts are obtained, the algorithm has a computational complexity of $O(T^2)$, which can potentially be improved to $O(T)$ using ideas from the optimised variant described above.

3.1.4 Online version of Dekel et al.

Dekel et al. describe an online learning algorithm in [27], which learns a decision-theoretic version of the PST. The algorithm can accommodate an additional optional sequence of real-valued input vectors $\boldsymbol{x}_1^t$, which is ignored here. Its formulation is particularly simple in the case of a binary alphabet $\Sigma = \{+1, -1\}$, but it can be extended to larger alphabets. The algorithm predicts the next symbol s_t in a sequence, based on the past symbols s_1^{t-1}. The hypothesis for s_t takes the form

$$h(s_1^{t-1}) = \sum_{i=1}^{t-1} 2^{-i/2} g(s_{t-i}^{t-1}),$$

where $g : \Sigma^* \to \mathbb{R}$ is a *context function* that assigns weights to strings. The sign of h is the prediction $\hat{s}_t$ of the next symbol, while the magnitude of h indicates the confidence in this prediction.

The learning algorithm starts with an initial hypothesis $h_1 = 0$ with context function $g_1(q) = 0, q \in \Sigma^*$, and then adapts the hypothesis after each symbol is observed. The symbol s_t at time t is therefore predicted by hypothesis h_t, with corresponding context function g_t. The hypothesis update rule is designed to increase the *margin* $s_t h_t(\boldsymbol{x}_t, s_1^{t-1})$, or, alternatively, to reduce the *hinge loss*

$$\ell_t = \max\{0, 1 - s_t h_t(\boldsymbol{x}_t, s_1^{t-1})\},$$

which is zero for correct predictions with sufficient confidence and increases linearly otherwise. The context function is therefore updated to

$$g_{t+1}(q) = g_t(q) + \frac{1}{3} s_t \ell_t 2^{-|q|/2}, \quad \forall q \in \mathcal{S}(s_1^{t-1}),$$

for all strings in the suffix set of the symbol history. The strings with non-zero values in the context function resemble the context set of a traditional probabilistic PST.

The main advantage of this algorithm is its online nature. It only requires a single pass through the training set, which therefore does not have to be stored and makes very large data sets viable. However, in the basic version of the algorithm, the size of the PST grows unbounded with the data set size T, which potentially poses an even bigger storage problem. Instead of resorting to a user-specified bound, [27] introduces a self-bounded version of the algorithm which remains competitive with larger fixed-size PSTs. The computational complexity of the unbounded version is $O(T^2)$, as all $(t-1)$ suffixes of the symbol history is added to the context function at time t. The self-bounded version has a complexity of $O(LT)$, where L is the automatically determined bound on the tree depth which typically grows as $O(\log T)$.

3.2 HMM topology inference

The topology of a hidden Markov model refers to its graphical structure, which includes the number of states, the symbols associated with each state, and the state transitions with non-zero probabilities. The standard HMM training procedure [49, 2] assumes that the HMM topology is specified beforehand, usually based on domain knowledge. While this is a powerful way to include expert knowledge in the HMM, it can also be problematic if the problem domain provides no guidance on the topology design. In this instance, it makes more sense to infer the HMM topology from the training data as well.

This section describes various algorithms that have been proposed for HMM topology inference. Standard HMM algorithms use parameter estimation to modify the HMM structure, by removing any zero-probability links after training. Brand's method [61] also falls in this category. The rest of the algorithms can be broadly divided into *state merging* methods [138, 41, 60], which start with a maximal-size model and shrink it, and *state splitting* methods [42, 39, 43], which start with a minimal-size model and grow it. Algorithms that explicitly incorporate a PST [13, 63] can also be labelled as state splitting methods, but are discussed in a separate section instead, because of their close relationship to hidden SECT.

3.2.1 Entropic prior

Brand proposes a maximum *a posteriori* (MAP) estimation procedure in [61] that uses a special *entropic prior* to bias the HMM towards a sparse structure. This prior has the form $P_\mathrm{e}(\boldsymbol{p}) \propto \exp[-H(\boldsymbol{p})]$, with $H(\boldsymbol{p})$ the entropy of distribution $\boldsymbol{p}$. For a multinomial distribution $\boldsymbol{p} = \{p_1, p_2, ..., p_M\}$ it has the elegant form of $P_\mathrm{e}(\boldsymbol{p}) \propto \prod_i p_i^{p_i}$. Unlike the usual Dirichlet prior, the entropic prior favours highly skewed, low-entropy distributions above uniform ones. The entropic MAP estimator for multinomials is calculated in a fast iterative procedure which solves transcendental equations involving the Lambert W function. This estimator has a minimum description length interpretation, and selects the strongest model hypothesis compatible with the data, rather than the fairest one. This drives irrelevant parameters toward extinction, leading to a sparse model structure.

The algorithm can be applied to continuous and discrete HMMs alike, by incorporating the entropic MAP estimator in the M step of the standard EM algorithm for HMMs. The estimator replaces the maximum likelihood estimator for the transition probabilities of each state in the HMM, and also estimates the symbol distributions if the HMM is discrete. The training is accelerated by a trimming procedure that removes low-probability transitions that would otherwise slowly decay to zero, and pinches off states that do not contribute to the most likely paths through the model. Since the algorithm is a simple modifi-

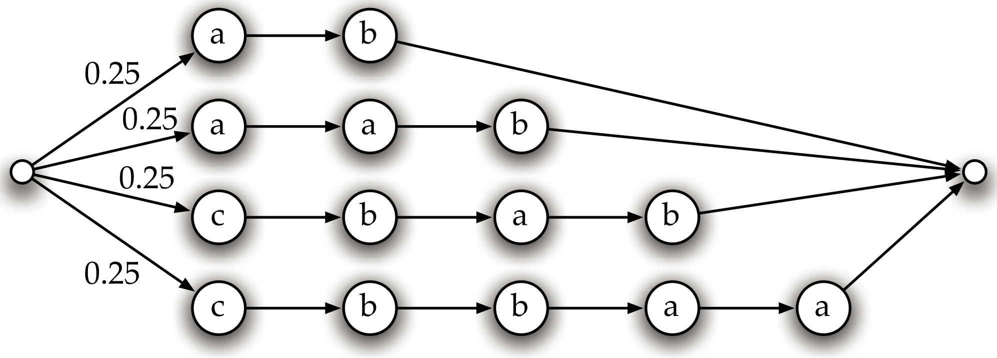


Figure 3.1: An example of the initial HMM in Stolcke's state merging algorithm, based on the data set $\{\texttt{ab}, \texttt{aab}, \texttt{cbab}, \texttt{cbbaa}\}$. Each state emits the observation symbol in its circle with probability one, and all transitions without labels have a probability of one.

cation of EM, its computational complexity is similar to that of standard HMM training, which is linear, $O(T)$, in the data set size T.

3.2.2 State merging methods

3.2.2.1 Stolcke's method

Stolcke and Omohundro introduce a Bayesian model merging technique [40, 138, 139] that learns the structure of discrete HMMs from sequences of discrete observations. It is an incremental greedy search for the MAP HMM. The algorithm starts with an initial HMM that includes each observation sequence in the training set as a path of corresponding HMM states. The initial state of this HMM has a transition to the start of each state path, and the rest of the states emit a single observation and make a single transition to the next state in the path (or the final state). This HMM is the most specialised model that assigns the highest probability to the training set, and represents a rote memorisation of the data. Figure 3.1 shows an example of such an HMM.

The algorithm uses a model prior that penalises large HMMs, based on the MDL principle [46, 45] for the HMM structure and Dirichlet priors [140] for the HMM parameters. It then iteratively merges HMM states in a greedy fashion, until the posterior probability of the model reaches a local maximum. This process generalises the HMM, which allows it to model data outside the training set.

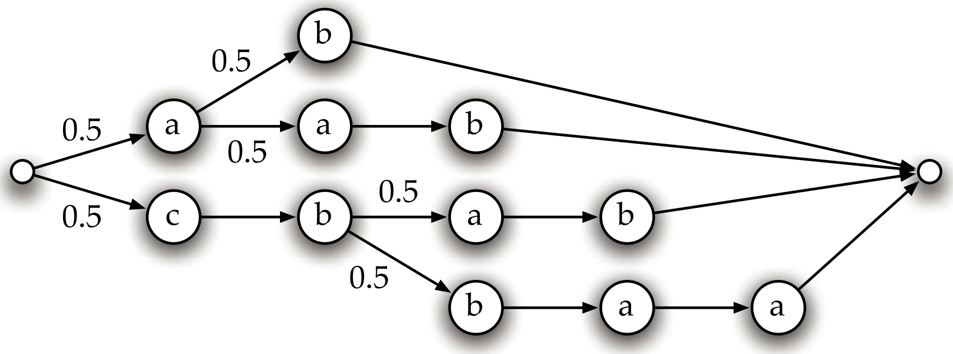


Figure 3.2: An example of the initial PPTA in ALERGIA, for the data set {`ab`, `aab`, `cbab`, cbbaa}. Each state emits the observation symbol in its circle with probability one, and all transitions without labels have a probability of one.

The algorithm can be adapted to train continuous HMMs, as long as the symbol pdfs have suitable priors and can be merged efficiently during the search process [139]. The computational complexity of the method is $O(T^3)$, where T is the number of observations in the training set [60], which makes it intractable for large data sets.

3.2.2.2 ALERGIA

Carrasco and Oncina describe a similar algorithm, known as ALERGIA, in [41]. It determines the structure of deterministic probabilistic finite automata (DPFAs), a subclass of discrete HMMs, from sequences of observations. The algorithm proceeds in a similar fashion to Stolcke's method, by first forming a large DPFA that best fits the training set and then iteratively merging states to create a smaller and more generalised model. The initial model is a *probabilistic prefix tree automaton* (PPTA) [41, 133] that has the prefix tree of the training data as state structure. It is equivalent to Stolcke's initial HMM, but more compact, as can be seen in Figure 3.2.

ALERGIA uses the Mealy form for its DPFAs, with observations emitted on the transitions instead of by the states, which rolls the transition and symbol distributions into one. Two states are considered similar if their corresponding empirical transition probabilities are all within confidence intervals based on the Hoeffding bound [141] (similar to Chernov bounds). If two states and all their corresponding successor states are similar, the two states are merged. The merging process visits the merge candidates in a breadth-first order, and also performs extra merges to keep the model deterministic. The typical computational complexity of ALERGIA is $O(T)$ [41], and the algorithm has been proven to converge in polynomial time with probability one to the correct DPFA [142].

3.2.2.3 MDI

Thollard et al. combine ALERGIA and ideas from Stolcke's method into the minimal divergence inference (MDI) algorithm [60] for training DPFAs. Similar to ALERGIA, the algorithm starts with the PPTA of the training data and attempts to merge similar states in a specified order. Instead of measuring the similarity between two states locally, based on the difference between their transition probabilities, MDI evaluates the proposed structural change globally, based on the Kullback-Leibler divergence between the DPFAs. The PPTA is considered to be the reference model, as it fits the data best. Any state merge will tend to increase the divergence between the final and reference models, but will also decrease the size of the model. If the ratio of the divergence increase to the size decrease is less than a specified threshold, the merge is accepted. MDI outperforms ALERGIA in terms of test set perplexity on a language modelling task, at the cost of a computational complexity of $O(T^2)$.

3.2.3 State splitting methods

3.2.3.1 ML-SSS

Ostendorf and Singer propose the maximum likelihood successive state splitting (ML-SSS) algorithm in [42], which is an improvement on the original SSS algorithm in [143], and use it to train context-dependent triphone models. The algorithm starts with a simple initial HMM, and runs the Baum-Welch algorithm to calculate state and transition posteriors for the training data. Each state in the HMM is now split in two ways, independent of the rest of the states. These splits are illustrated in Figure 3.3. The temporal split replaces the state with two states in series, initialised to have the same symbol pdf, and self-loop probabilities that match the duration of the state pair to that of the original state. The parameters

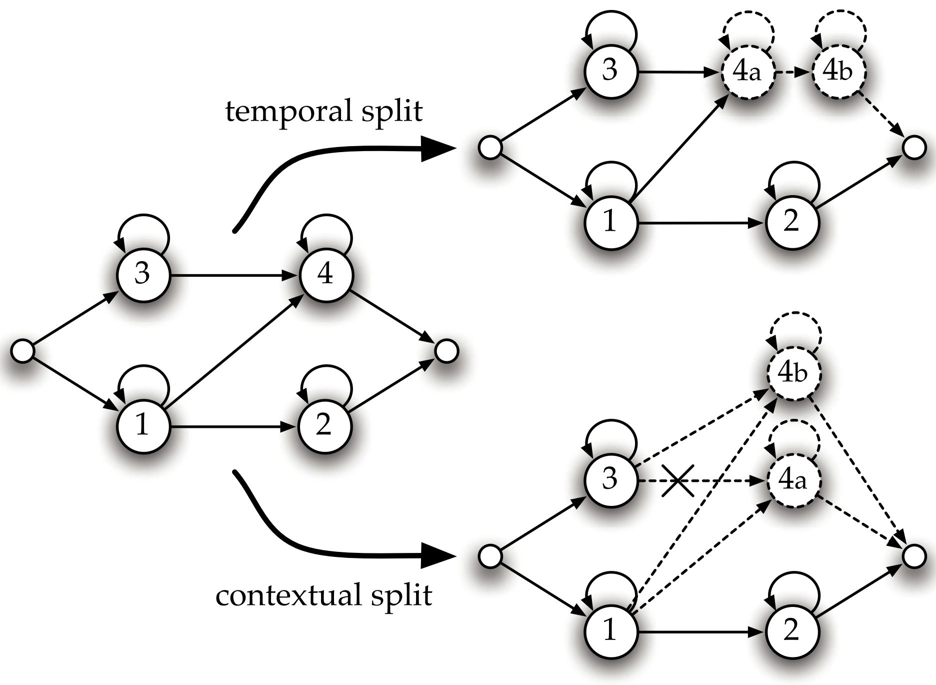


Figure 3.3: The two types of split in the successive state splitting (SSS) algorithm. If state 4 in the original model on the left is to be split, it can either undergo a temporal or a contextual split, resulting in one of the models on the right. In the process, the symbol pdf of 4 is split into two new pdfs for states 4a and 4b. Note that the contextual split may invalidate some transitions that are not compatible with the phonetic contexts of the new states, which are indicated by a crossed-out link.

of the two new states are then retrained on the subset of data associated with the original state, using a few iterations of the Baum-Welch algorithm. The contextual split partitions the data associated with the original state into two subsets, using a decision tree design technique known as Chou's iterative partitioning algorithm, from which the parameters of two new states connected in parallel are derived.

The ML-SSS algorithm applies the single split which leads to the largest increase in the lower bound of the EM algorithm (see Section 2.4.4), which can be calculated efficiently for each state and split type. This step of ML-SSS is an incremental variant of the EM algorithm [117], which ensures that the overall model likelihood does not decrease because of the split. The Baum-Welch algorithm is rerun for all states affected by the split in order to update their state and transition posteriors, and the splitting process is repeated on the new model. The state splitting stops when the HMM reaches a pre-specified size. This is a weakness of ML-SSS, as the appropriate model size has to be determined by a model validation experiment. The computational complexity appears to be $O(T)$, as the algorithm effectively consists of multiple iterations of the Baum-Welch algorithm. The hidden constant factor in this complexity is large, however.

3.2.3.2 FIT

Another approach to topology inference focuses on the Markov order of the HMM. An Lth-order Markov chain describes Lth-order dependencies between the symbols in a symbol sequence. However, by introducing states associated with symbol context strings of length L, the model becomes first-order in terms of the state indices in the corresponding state sequence, as described in Section 2.2. This equivalent first-order model has many states tied to the same symbol, and a highly constrained sparse topology. The same idea can be applied to hidden Markov models. An HMM can encode higher-order dependencies between the soft symbols (or observation distribution indices), but it will be first-order in terms of state indices if the states are identified with symbol context strings of the appropriate length. This equivalent first-order model can be trained with standard HMM methods, as long as the state tying and constrained topology imposed by the higher-order symbol dependencies are respected.

This concept is explored in [39, 144, 145], where Du Preez introduces two algorithms for training high-order[2] HMMs. The order reducing (ORED) algorithm reduces any high-order HMM to an equivalent first-order HMM (first-order in terms of states), which can then be trained by standard algorithms such as Baum-Welch and Viterbi re-estimation. This equivalent first-order model is frequently very large, which results in excessive computational complexity and overfitting on the training data. The situation is improved by the fast incremental training (FIT) algorithm. The algorithm is initialised with a standard first-order HMM (first-order in terms of both symbols and states) with a topology dictated by the application of the HMM. This initial model is trained with a standard algorithm,

[2]In the description that follows, all higher-order models describe higher-order *symbol* dependencies. These models always have an equivalent model that is first-order in terms of *state indices*.

and the trained first-order HMM is then expanded to form an initial second-order HMM. These steps are illustrated in Figure 3.4.

The order expansion is effectively a state splitting procedure, as each state in the first-order HMM is split into multiple states, one for each link entering the state. Each transition with non-zero probability in the first-order model therefore becomes a state in the second-order model. These states share the symbol pdf and transition probability values of the state in the first-order model from which they were derived, but have different transition destinations to satisfy the constrained topology of the second-order model. This ensures that the first-order and initial second-order HMMs are equivalent, providing identical scores for the same observation sequence. The second-order HMM is now trained in the same way as the first-order model, by considering it to be a first-order HMM in terms of its state indices. The symbol pdfs are updated but remain tied among the same states, while the transition probabilities are free to change. This changes the HMM from a degenerate first-order model into a fully-fledged second-order model. The steps of order expansion and training are alternately repeated until a specified model order is reached.

The advantage of FIT is that any transition that disappears during the training of a lower-order model is not expanded any further, which reduces the number of parameters in the final higher-order model. Another useful property is that the likelihood cannot decrease during training if Baum-Welch re-estimation is used, because the order expansion step increases the model complexity without changing its likelihood. Each subsequent phase of Baum-Welch training can therefore increase the model likelihood even further, while searching for an optimal HMM in a model space of ever-increasing size. The computational complexity of the order expansion step only depends on the model size and is negligible compared to that of the subsequent (standard) training step. The overall complexity of FIT is therefore $O(T)$, although the large model sizes associated with higher-order HMMs can make it comparatively slow.

3.2.3.3 tSnob

Edgoose and Allison describe tSnob, an extension to the Snob classification program [82] that is capable of hidden Markov modelling, in [43, 146]. The program

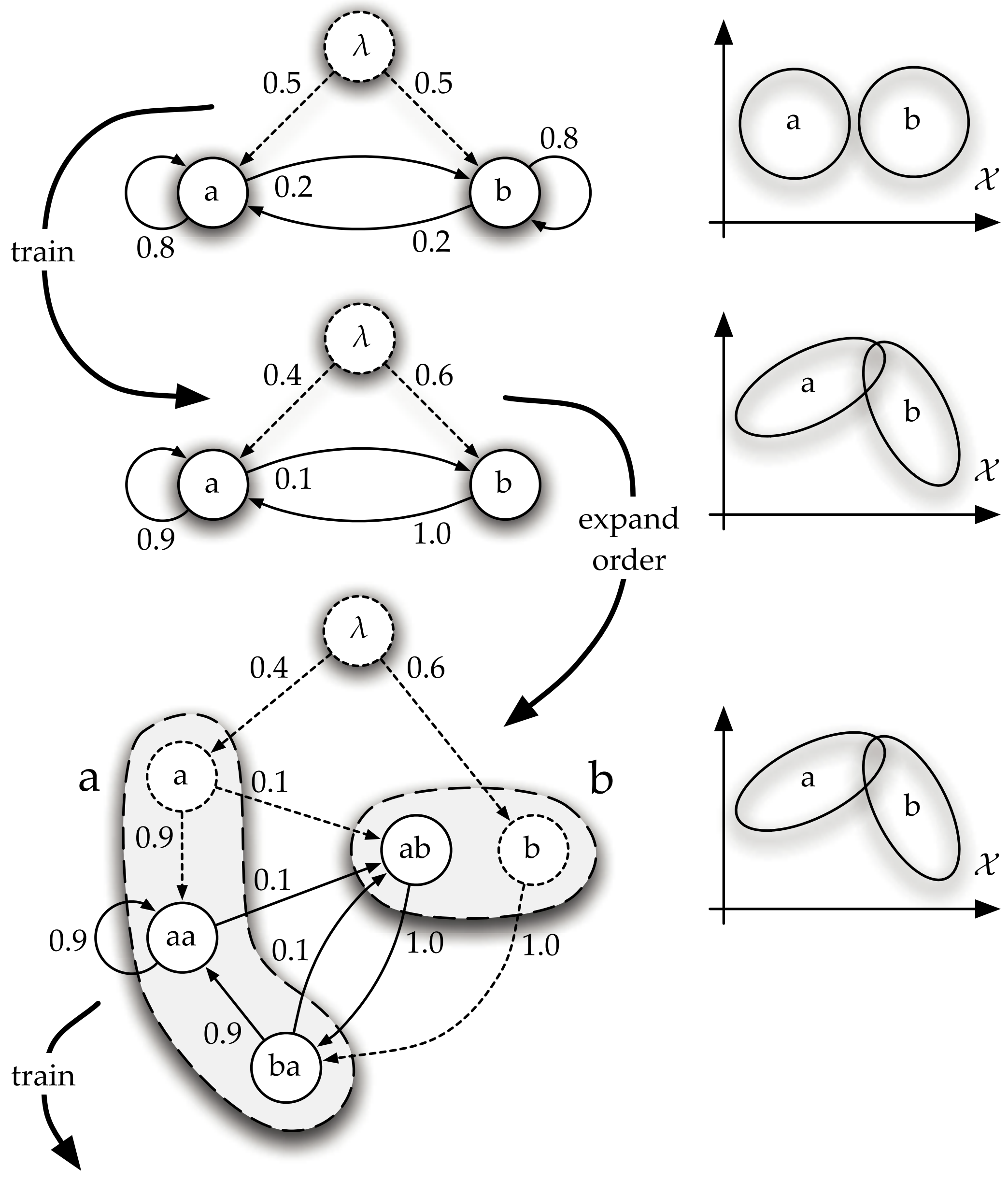


Figure 3.4: The two basic steps of the FIT algorithm. The example initial model at the top is a continuous first-order HMM with two symbols, a and b, in a two-dimensional observation space $\mathcal{X}$. The Gaussian symbol pdfs are indicated by their 2σ contours in the graphs on the right-hand side. The dashed nodes and links indicate start-up states. The first-order HMM in the middle is obtained after training, during which transition $a_{\mathtt{bb}}$ was removed. The corresponding initial second-order HMM is shown at the bottom. The states within each encircled area are second-order extensions of the same first-order state, with tied transition probability values and symbol pdfs, but different transition destinations. The training and order expansion steps are alternately repeated.

uses the minimum message length (MML) [45, 86] paradigm to calculate the length of a two-part message describing a first-order HMM and the observation data in terms of the HMM. The symbol pdfs of the HMM can be discrete, diagonal Gaussian or von Mises distributions, in univariate or multivariate form. The model part of the message serves as model prior, and encodes the number of symbols, the symbol pdfs and the next-symbol probability distributions. Each observation is assigned a symbol and encoded relative to the corresponding symbol pdf. The contributions of all possible symbol assignments are summed to obtain the data part of the message, in a step similar to the calculation of HMM likelihoods with the forward algorithm.

tSnob is also an unsupervised clustering algorithm. It has an efficient search strategy to discover the relevant number of symbols underlying the observation data, and hence the number of states in the HMM. Starting with a hypothesis of M symbols, it attempts to merge and split symbols to reduce the overall message length. It considers the $\binom{M}{2}$ possible ways in which two symbols can be merged into one symbol, and also chooses a randomised split of each symbol into two new symbols. The merge and split candidates resulting in the shortest message length are selected as challengers for the current M-symbol model. The model parameters of both challengers are improved using an EM algorithm closely related to the standard forward-backward algorithm, and if a challenger produces a lower message length than the M-symbol model, it becomes the new incumbent model. If the M-symbol model prevails, a new pair of candidates are chosen as challengers.

Note that tSnob has the capability to merge states, although it is grouped with the state splitting methods. The algorithm has a computational complexity that is linear in the data size T, since all interactions with the data set are through forward and forward-backward algorithms of $O(T)$ complexity. The optimal number of symbols, M^*, may depend on the data set size, which would make the algorithm super-linear in T, unless the search for the number of symbols is initialised close to M^*. The search procedure has a computational complexity of $O(M^4T)$ if the current model has M symbols, which is dominated by the large number of merge candidates (about M^2) combined with the $O(M^2T)$ complexity of the forward algorithm, which is required to calculate the data parts of their message lengths. This makes the algorithm intractable for large values of M.

3.2.4 Algorithms based on PSTs

3.2.4.1 Galata et al.

Galata et al. apply PSTs to real-valued data in [147, 13]. While they do not infer HMMs, their approach is one of the first attempts (together with this study) to marry PSTs with a continuous feature space, and it provides the background for the HMM inference method in the following section. They model trajectories in D-dimensional feature space, where each feature vector represents a set of points describing the shape of an object, augmented by the first derivatives of these points. The feature space is discretised by robust vector quantisation, which determines M centroids or *prototype* vectors. Each feature vector sequence is therefore turned into a symbol sequence by assigning each feature vector to the nearest prototype.

Consecutive repetitions of the same prototype symbol are replaced by a single occurrence of that prototype. Without this step, large context memories are potentially required to "see past" long periods where the object remains stationary. This also alleviates the effects of sampling the feature trajectory at a rate that is much higher than the rate of the actual events to be modelled. The technique is similar to the context-emphasised mixed-order HMM introduced in [39]. Modelling the pruned symbol sequence with a Markov chain effectively models the original symbol sequence with a DPFA, which is a more powerful model.[3] As described in [39], the downside of this step is that the duration of a specific prototype is poorly modelled.

A PST is inferred from the symbol sequence using the method described in [7], and converted into an equivalent PSA. Once trained, the PSA can be used to generate or predict object trajectories. This algorithm is combined with a particle filter for hand tracking in [14], where it is used to learn repetitive hand motions.

3.2.4.2 Sage et al.

Sage et al. [62, 63] extend the previous method by replacing the prototypes with Gaussian pdfs, which provide soft clustering. They train a standard first-order HMM with M Gaussian symbol pdfs $\sigma(m, x)$ on a training sequence x_1^T, and

[3]The DPFA example in Figure 2.13 illustrates this. The distinct symbol which follows a arbitrarily long run of b's depends on the distinct symbol which precedes the run.

identify the symbol pdf indices as an alphabet $\Sigma = [1, M]$. The trained transition probabilities are replaced by uniform probabilities of value $1/M$, which turns the HMM into a Gaussian mixture model (GMM) with equal mixture weights. The probability to have a symbol m at time t then simply becomes the *responsibility* of the mth Gaussian component for the observation x_t, given by

$$p(m,t) = P(s_t = m | x_t) = \frac{\sigma(m, x_t)}{\sum_{m=1}^{M} \sigma(m, x_t)}.$$

Instead of estimating a symbol sequence s_1^T from the observation sequence x_1^T, the symbol probabilities $p(m,t)$ are used to derive soft counts for any desired symbol string r_1^l with length $l \leq L$, given by

$$\#'(r_1^l) = \sum_{t=l}^{T} \prod_{i=1}^{l} p(r_i, t + i - l).$$

A PST is now trained on these soft counts, using a modified version of Learn-PSA based on Kullback-Leibler divergence. This PST inference algorithm has two free parameters: the maximum tree depth L and the divergence threshold ϵ.

The PST is converted into a PSA with N states, which is used to derive an $N \times T$ forward evaluation trellis with elements $\alpha(t, j)$ for a test sequence x_1^T (see Section 2.4.1 for details). This trellis is similar to the one produced by the HMM that has the PSA as underlying Markov chain, combined with the same M Gaussian symbol pdfs as before, which are shared between the N states. The pdf sharing is now disabled, after which the symbol pdfs are refined by performing one iteration of Baum-Welch re-estimation. This results in N independent symbol pdfs, one per state. Refinement of the pdfs improves the ability to generate trajectories similar to the training sequence. The final model is applied to a hand tracking experiment, where it replaces missing observations with stochastically generated ones.

This method trains continuous HMMs of mixed order. The computational complexity is dominated by the calculation of soft counts, which is $O(LM^LT)$ in the worst case.

3.3 Summary

Many PST inference algorithms have been proposed since the original Learn-PSA algorithm. Some variants reduce the computational complexity from quadratic to linear in the training sequence length, while others remove the need for tunable parameters, which otherwise require expensive cross-validation steps. This includes not having to specify a bound on the PST depth, which is determined automatically from the training set. Dekel's version also allows online operation, which does not require the storage of the training sequence.

The various inference algorithms have different measures of the discrepancy between parent and child node distributions, which range from probability ratios to the Kullback-Leibler divergence. They also differ in the way they smooth their probability estimates, which range from no smoothing and fixed offsets to Krichevsky-Trofimov estimators and back-off strategies.

While no HMM training algorithms exist that explicitly infer generic mixed-order HMMs, there are several algorithms that infer the topology of an HMM from data. Most of these algorithms either start with a minimal-size model and split states, or start with a maximal-size model and merge states. This is typically done in a greedy iterative fashion, by continually picking the split or merge that maximises the increase in a model fitness function. This fitness function ideally combines the usual likelihood score with some measure of model complexity, which allows a trade-off between model accuracy and complexity in a way similar to minimum description length methods. In some of the cases the model complexity is specified by a prior distribution over HMMs, or by the number of states in the HMM.

HMM training algorithms of particular relevance to this study are the FIT algorithm, which infers generic fixed-order HMMs, and Sage's algorithm, which combines prediction suffix trees with HMMs.

Chapter 4

The SECT Algorithm

The *smallest encoded context tree* (SECT) algorithm infers prediction suffix trees (PSTs) from training data. It improves on the standard Learn-PSA algorithm [22] by not having any explicit thresholds and user-controllable parameters to tune, and by allowing the tree to expand without an explicit bound on its depth.

While Learn-PSA follows the PAC learning framework [100], SECT is based on minimum description length (MDL) or Bayesian maximum a posteriori (MAP) arguments [84]. It selects the PST that results in the smallest combined description length for the training data and the PST itself. That is, given training data D and the set of PSTs or hypotheses $\mathcal{H}$, SECT chooses $H^* \in \mathcal{H}$ as

$$H^* = \arg\min_{H\in\mathcal{H}} \left[L(H) + L(D|H)\right],$$

where $L(H)$ is the description length of the PST model, and $L(D|H)$ is the description length of the training data, as encoded by the model H, in bits. In Bayesian MAP notation, this becomes

$$H^* = \arg\max_{H\in\mathcal{H}} P(H)P(D|H) = \arg\max_{H\in\mathcal{H}} P(H|D).$$

Using SECT, simple (low-order) PSTs have short description lengths $L(H)$, but do not fit the data as well, resulting in larger $L(D|H)$. Similarly, highly complex PSTs may fit the training set well (i.e. have low $L(D|H)$), but are penalised by higher $L(H)$. The complexity of the inferred PST also depends on the amount of available training data. With small samples, simpler models are preferred, while larger data sets can support more complex PSTs if they are warranted. In this way, the SECT algorithm curbs overfitting and balances modelling power with model complexity.

The SECT algorithm follows a two-part MDL approach which divides the total description length into a model part, $L(H)$, and a data part, $L(D|H)$. The model part is based on a constructive code that attempts to store the PST structure and parameters as efficiently as possible, while remaining uniquely decodable, while the data part is coded according to the predictive distribution provided by the PST. This approach is referred to as "crude" MDL in [84].

4.1 Basic routines

Three basic routines are commonly encountered in the SECT algorithm. The first involves the coding of a data set of independent (exchangeable) symbols according to a prescribed probability distribution. The data set is based on an alphabet of M symbols, $\Sigma = \{s_1, s_2, ..., s_M\}$, and contains n_i occurrences of symbol s_i. The data set is therefore represented by the set of symbol counts $\boldsymbol{n} = \{n_1, n_2, ..., n_M\}$. The probability distribution is specified as a set of symbol probabilities $\boldsymbol{p} = \{p_1, p_2, ..., p_M\}$. The optimal code length of symbol s_i is $l_i = -\log_2 p_i$, according to (2.1.2), and the code length of data set n according to distribution p is defined as

$$D(\boldsymbol{n}|\boldsymbol{p}) = \sum_{i=1}^{M} n_i \log_2 \frac{1}{p_i}. \tag{4.1.1}$$

The second routine involves the efficient storage of a table of M counts, $\boldsymbol{n} = \{n_1, n_2, ..., n_M\}$. It is assumed that the sum of counts $N = \sum_i n_i$ is known before coding begins. The counts are coded in sequence, and each successive count requires fewer bits, as the number of possible values of the count decreases. The first count, n_1, can take any value in the range $[0, N]$. If these values are considered equiprobable, storing n_1 requires $\log_2(N+1)$ bits. The next count, n_2, has a range of $[0, N - n_1]$, and, by the same token, requires $\log_2(N - n_1 + 1)$

bits. In general, count n_i requires

$$l_i = \log_2 \left(1 + \sum_{m=i}^{M} n_m \right)$$

bits of storage, and the entire table has a code length of

$$L_{\mathrm{T}}(\boldsymbol{n}) = \sum_{i=1}^{M-1} l_i = \sum_{i=1}^{M-1} \log_2 \left(1 + \sum_{m=i}^{M} n_m \right). \tag{4.1.2}$$

The last count, n_M, does not have to be coded, as its value can be inferred from the first $M-1$ counts as $n_M = N - \sum_{i=1}^{M-1} n_i$.

The third routine involves the storage of a list of N symbols, $S \subset \Sigma$, which forms a subset of the M-symbol alphabet $\Sigma = \{s_1, s_2, ..., s_M\}$. The alphabet is assumed to be known before coding begins, and the list contains no duplicate symbols. Using the same reasoning as in the previous routine, the first symbol of S is one of M possibilities, and therefore requires $\log_2 M$ bits of storage (again assuming equiprobable values). The second symbol is one of $M-1$ possibilities, and requires a code length of $\log_2(M-1)$, and so forth. The code length of the subset S becomes

$$L_{\mathrm{L}}(S) = \sum_{i=0}^{N-1} \log_2 (M-i). \tag{4.1.3}$$

4.2 The philosophy of SECT

The input to the SECT algorithm is an alphabet Σ of M symbols, and a sequential data set D defined on this alphabet. The data set typically contains a single symbol sequence s_1^T of length T, but the algorithm can also operate on multiple symbol sequences. The purpose of SECT is to produce a prediction suffix tree (PST) $\mathcal{T}$ that captures the relevant Markovian statistics of the data set.

The approach of the SECT algorithm is based on information theory. Suppose a transmitter, Alice, wants to transmit the data set to a receiver, Bob. The message she selects to send depends on the state of knowledge of the receiver. If Bob knows the alphabet Σ and expects a sequence of symbols from Σ, Alice can encode each symbol with a *raw* code that assigns $\log_2 M$ bits to each symbol in the alphabet.[1] This assumes that the symbols are uniformly distributed in the data set. If their actual distribution is significantly non-uniform, this information can be used to transmit a shorter message. The specific distribution used to construct the improved code has to be communicated to Bob as well, otherwise he will not be able to decode the message. For example, Alice can send the counts of each symbol in the data set as a table to Bob, which he can then use to reconstruct the code. If the benefit of the improved data coding outweighs the extra overhead of the table, this is a useful step to take.

In mathematical terms, a data set with symbol counts $\boldsymbol{n} = \{n_1, n_2, ..., n_M\}$ is to be coded according to a (possibly sub-optimal) default distribution $\boldsymbol{p}_0 = \{p_1, p_2, ..., p_M\}$ (with $p_i = 1/M$ in the raw case). This results in a code length for the data set of $D(\boldsymbol{n}|\boldsymbol{p}_0)$ bits. The minimum code length for this data set, based solely on the counts $\boldsymbol{n}$, is produced by the *maximum likelihood* distribution $\boldsymbol{p}^* = \boldsymbol{n}/T = \{n_1/T, n_2/T, ..., n_M/T\}$, where $T = \sum_{i=1}^{M} n_i$. The optimal distribution $\boldsymbol{p}^*$ is preferred to the default distribution $\boldsymbol{p}_0$ if

$$D(\boldsymbol{n}|\boldsymbol{p}^*) + L(T) + L_{\mathrm{T}}(\boldsymbol{n}) < D(\boldsymbol{n}|\boldsymbol{p}_0)\,. \tag{4.2.1}$$

The term $L(T) + L_{\mathrm{T}}(\boldsymbol{n})$ represents the table overhead, which stores the total count T as a code of length $L(T)$ (depending on the expected range for T), and the table itself as a code of length $L_{\mathrm{T}}(\boldsymbol{n})$, constructed according to (4.1.2).

Another way to express the model acceptance test (4.2.1) is as

$$\mathrm{D}(\boldsymbol{p}^* \| \boldsymbol{p}_0) > \frac{L(T) + L_{\mathrm{T}}(\boldsymbol{n})}{T}, \tag{4.2.2}$$

where $\mathrm{D}(\boldsymbol{p}^* \| \boldsymbol{p}_0)$ is the Kullback-Leibler divergence between the optimal and default coding distributions. The table overhead grows as $O(\log T)$ for large T, which means that the right-hand side of (4.2.2) approaches zero as T increases. This indicates that $\boldsymbol{p}^*$ will be preferred to the default even if it is very close to $\boldsymbol{p}_0$, as long as the data set is large enough to support the choice.

[1]In the majority of practical communication systems, even less knowledge of the data set is assumed. The exact size M of the alphabet is also considered unknown, and a superset of Σ of size $M' > M$ is used as alphabet instead. This results in a code length of $\log_2 M'$ per symbol. Examples of this include the representation of arbitrary data as 8-bit ASCII characters or 32-bit integers.

If the model $\boldsymbol{p}^*$ is accepted, the data symbols are modelled as independent and identically distributed according to $\boldsymbol{p}^*$. This amounts to a zeroth-order Markov model. First-order Markovian dependencies are captured by first partitioning the data set according to first-order context. That is, all symbols which follow symbol s_1 are collected together and counted; the same for all symbols following s_2, and so forth. This results in M sets of symbol counts, which add up[2] to the original zeroth-order counts $\boldsymbol{n}$. The SECT algorithm repeats the basic model selection process of (4.2.1) on each of these subsets, where the zeroth-order distribution $\boldsymbol{p}^*$, if accepted, becomes the new default distribution to beat. In addition to the table overhead, the context symbol associated with each subset also has to be transmitted.

A similar process is repeated to detect higher-order Markovian dependencies. The higher-order empirical distributions have to differ significantly from the lower-order ones to justify their inclusion in the model. The model overhead for each higher-order context includes the specification of its context string and the table of symbol counts. The core of the SECT algorithm is an efficient recursive procedure that tests as many contexts as the data set will allow, and collects the useful ones in a PST structure.

4.3 The algorithm

The inputs to the SECT algorithm include an M-symbol alphabet Σ and a symbol sequence s_1^T of length T, with symbols taken from Σ. Alternatively, the data set may have multiple sequences of total length T—this only slightly complicates the symbol counting. Also required is a default distribution $\boldsymbol{\eta}_0$, which will be used to code the data in the absence of any further knowledge of the data, and which the PST will have to improve upon to justify its inference. A sensible choice for $\boldsymbol{\eta}_0$ is the uniform distribution, with $\eta_0(s) = 1/M$. The last input is the maximum expected size $T_{\max}$ of the data set, which will be used as constraint

[2]There will actually be one less symbol in the accumulated first-order counts, as no symbol follows the last symbol of the data set, s_T. If the data set contains K symbol sequences, the first-order counts will be K less than the zeroth-order counts. This discrepancy is negligible for $T \gg K$.

to encode the actual size T of the data set. Its value is not critical and may be chosen as 10^9, for example.

SECT starts with an empty tree, and grows the PST in a top-down fashion. This is because the usefulness of higher-order contexts largely depends on the lower-order contexts, and not the other way around. The PST node structure is represented by a set of context strings $\mathcal{Q}$, which contains the useful or *coding* nodes of the PST. Each context $q \in \mathcal{Q}$ has an associated set of *next-symbol counts*

$$\boldsymbol{\#}_q = \{\#(qs) \mid s \in \Sigma\} \tag{4.3.1}$$

derived from the training data set, where $\#(qs)$ is the number of times that string $qs = q \cdot s$ occurs in the data. These counts are used to estimate the associated *next-symbol probability distribution*

$$\boldsymbol{\eta}_q = \{\eta(q,s) \mid s \in \Sigma\}$$

of PST node q in a separate procedure, usually as the maximum-likelihood estimate given by

$$\eta_{\mathrm{ML}}(q,s) = \frac{\#(qs)}{\#(q*)} = \frac{\#(qs)}{\sum_s \#(qs)}.$$

The core SECT algorithm focuses on determining $\mathcal{Q}$, which starts out empty.

The main routine in SECT is `CodeSubtree`. This routine codes the symbols which follow the context string q in the data set, and attempts to improve on the code of the default distribution for context q. In the process, it recursively maps out the useful contexts in the entire subtree rooted at q. It outputs a set of *coded-symbol* counts, $\boldsymbol{n}_q$, which represents the symbols which were successfully coded within subtree q, and a score, $L(q)$, which is the combined model overhead and data code length for these symbols. It also updates the PST context set $\mathcal{Q}$ with any useful contexts it found. The top-level routine of SECT calls `CodeSubtree` with the empty context string λ as argument, which corresponds to the root node of the PST, thereby constructing the entire tree.

The end result of the SECT algorithm is the set $\mathcal{Q}$ of useful coding contexts, and a score, $L(\mathcal{T})$, which represents the total code length of the data set, including the model overhead of the PST. Any symbols which have not been coded

```
SECT
Inputs:   alphabet $\Sigma$, data $s_1^T$, default distribution $\boldsymbol{\eta}_0$, maximum length $T_{\max}$
Outputs:  PST context set $\mathcal{Q}$, score $L(\mathcal{T})$

  1. Start with empty tree: $\mathcal{Q} = \varnothing$
  2. Code tree and update $\mathcal{Q}$: $[\boldsymbol{n}_\lambda, L(\lambda)] = \textbf{CodeSubtree}(\lambda, \boldsymbol{\eta}_0, T_{\max})$
  3. Code any uncoded symbols: $L(\mathcal{T}) = L(\lambda) + D(\#_\lambda - \boldsymbol{n}_\lambda | \boldsymbol{\eta}_0)$
```

Figure 4.1: Pseudo-code for the top level of the SECT algorithm. The highlighted `CodeSubtree` routine is described elsewhere in this chapter.

by the PST are coded by the default distribution $\boldsymbol{\eta}_0$ instead, which adds to the final score. Any non-coding internal nodes can be added to the PST as described in Section 2.3.3, and the next-symbol distribution of each coding node q can be estimated in various ways from the corresponding next-symbol counts $\#_q$. The top-level routine of the SECT algorithm is summarised in Figure 4.1.

4.3.1 Calculating model overheads

In the `CodeSubtree` routine it will be necessary to quantify the model overhead of each potential node in the prediction suffix tree. A PST node with context string q has to keep track of the presence or absence of its children nodes, as well as the state of its next-symbol probability distribution $\boldsymbol{\eta}_q$. Coding nodes have to store their own next-symbol distributions, while non-coding nodes use the distributions of their parents (or a default distribution $\boldsymbol{\eta}_0$ if it is the root node). The model overhead of node q is therefore constructed as

$$M(*q, \boldsymbol{\eta}_q) = M_{\mathrm{P}}(*q) + M_{\mathrm{N}}(\boldsymbol{\eta}_q),$$

where $M_{\mathrm{P}}(*q)$ is the overhead associated with the children nodes, and $M_{\mathrm{N}}(\boldsymbol{\eta}_q)$ is the overhead associated with the next-symbol distribution $\boldsymbol{\eta}_q$.

The children nodes of node q are identified in a *previous-symbol list* $P_q \subset \Sigma$. If node mq is a child of node q, then $m \in P_q$, while an empty list indicates that node q is a leaf node. A typical PST has roughly the same number of internal and leaf nodes, especially if it has a fixed order L. This motivates the choice to indicate the presence or absence of a previous-symbol list with a one-bit flag. A leaf node therefore has overhead

$$M_{\mathrm{P}}(*q) = M_{\mathrm{PE}}(-) = 1,$$

where the dash indicates the absence of children. If the previous-symbol list exists, a straightforward encoding of the list is to indicate the presence or absence of child node mq with a one-bit flag. With one flag for each $m \in \Sigma$, this *full list* requires M bits of storage. It assumes that the connections in the tree are fairly random.[3] If the tree is known to be sparsely connected, the previous-symbol list may be more compactly stored as an explicit *sparse list* of symbols, especially for large M. The number of elements in the list is stored using $\log_2 M$ bits (as the list is expected to be non-empty with maximum size M), and each symbol is stored with decreasing bit width, as specified by (4.1.3). In SECT, both list codes are attempted, and the smallest one is kept, with an extra one-bit flag indicating the choice. The full previous-symbol list option therefore has overhead

$$M_{\mathrm{P}}(*q) = M_{\mathrm{PF}}(P_q) = 1 + 1 + M,$$

while the sparse list has overhead

$$M_{\mathrm{P}}(*q) = M_{\mathrm{PS}}(P_q) = 1 + 1 + \log_2 M + L_{\mathrm{L}}\left(P_q\right).$$

The next-symbol distribution $\boldsymbol{\eta}_q$ of a coding node q also has to be communicated to the receiver. The calculation of the description length of an arbitrary probability distribution is complicated by the infinite information content of irrational-valued probabilities, as indicated by an infinite non-repeating decimal expansion. One option would be to quantise the probability values to some accuracy, thereby making the space of distributions countable. SECT codes the *sufficient statistics* of the distribution instead. The sufficient statistics of a multinomial distribution such as $\boldsymbol{\eta}_q$ is the set of symbol counts $\#_q$ observed in the data set. The advantage of this approach is that the counts are already discrete, and

[3]Note that the previous-symbol list itself is unnecessary if the tree is of full degree M and each internal node therefore has exactly M children. In this case, the leaf flags are sufficient.

also allow perfect reconstruction of η_q, once the choice of estimator is known.

A coding node q therefore has to store its next-symbol counts table $\#_q$. As with the previous-symbol list, the presence or absence of the table is indicated by a one-bit flag. This assumes that roughly half of the PST nodes are coding nodes. Next, the total of the next-symbol counts, $\#(q*)$, is stored. This is assumed to take values in the range $[1, \#(p*)]$, where $\#(p*)$ is the total of the next-symbol counts of the parent node p, and $q = np$ for $n \in \Sigma$. It requires $\log_2 \#(p*)$ bits of storage. The total count is useful to coding nodes, as it constrains the next-symbol counts and thereby reduces their description lengths.

It is also useful to non-coding nodes, as it constrains the total counts of children nodes further down the tree. An example will illustrate this advantage. Suppose the PST has an alphabet size M and fixed order L, and the data set has T symbols which are evenly distributed among the M^L leaf nodes of the PST. Suppose also that all leaf nodes are coding and all internal nodes are non-coding. The first scenario only stores the total data size T, leaving the internal non-coding nodes blank. This requires each count total in a leaf node to be coded with $\log_2 T$ bits, as this is the only known constraint on these counts, leading to a combined length of $M^L \log_2 T$ for the counts in this scenario.

The second scenario stores a network of total counts, one for each PST node. The root node contains the total data size T, as before. The first tree level has M nodes, and each of their total counts requires $\log_2 T$ bits. The second tree level has M^2 nodes, and each of their totals requires $\log_2(T/M)$ bits, as it uses the knowledge of the total counts of their parents as constraint. This process repeats until the leaf nodes are reached, which require $\log_2(T/M^{L-1})$ bits per total count, leading to a length of $M \log_2 T + M^2 \log_2(T/M) + ... + M^L \log_2(T/M^{L-1})$ in this scenario.

The second scenario is better than the first if

$$\begin{aligned} M \log_2 T + M^2 \log_2(T/M) + ... + M^L \log_2(T/M^{L-1}) &< M^L \log_2 T \\ (M + ... + M^L) \log_2 T - (M^2 \log_2 M + ... + M^L \log_2 M^{L-1}) &< M^L \log_2 T, \end{aligned}$$

or if

$$\begin{aligned}(M + M^2 + ... + M^{L-1}) \log_2 T &< (M^2 + 2M^3 + .. + (L-1)M^L) \log_2 M \\ \frac{\log_2 T}{\log_2 M} &< \frac{M^2(1 + 2M + ... + (L-1)M^{L-2})}{M(1 + M + ... + M^{L-2})},\end{aligned}$$

where the right-hand side is approximately equal to $(L-1)M$ for large M and L. Therefore, if the data size approximately satisfies (for $L \geq 2$)

$$\log_2 T < (L-1)M \log_2 M, \quad \text{or} \quad T < M^{(L-1)M},$$

it is useful to save the total counts at each node. This is always true for large enough M (e.g. $M \geq 10$) and L, and also becomes more likely if the PST has a mixed order with relatively few internal nodes, or if the next-symbols are non-uniformly distributed among the leaf nodes. A non-coding node therefore has overhead

$$M_{\text{N}}(\boldsymbol{\eta}_q) = M_{\text{NE}}(-) = 1 + \log_2 \#(p*),$$

where the dash indicates the absence of a next-symbol table, and the parent total count $\#(p*)$ is always available during decoding.

The next-symbol counts table $\boldsymbol{\#}_q$ also has to be stored for coding nodes. As with the previous-symbol list, there are two options. The *full table* stores the counts in the order in which their corresponding symbols appear in the alphabet, using the construction of (4.1.2). This has the advantage that no symbol information is required, but it can be suboptimal if the largest counts appear near the end of the table. The bit widths of each count will show a maximal decrease if the table is sorted in descending order first, which is the approach of the *sparse table* encoding. The new symbol order has to be specified too, by storing the list of next-symbols with non-zero counts, $N_q = \{s \in \Sigma \mid \#(qs) > 0\}$, according to (4.1.3). The number of elements in the list is in the range $[1, M]$, and requires $\log_2 M$ bits. The choice of full or sparse table is indicated by a one-bit flag. The full table option therefore has overhead

$$M_{\text{N}}(\boldsymbol{\eta}_q) = M_{\text{NF}}(\boldsymbol{\#}_q) = 1 + 1 + \log_2 \#(p*) + L_{\text{T}}(\boldsymbol{\#}_q),$$

Table 4.1: Model overhead for various node configurations.

Node configuration	Model overhead
Internal, coding	$M(*q, \boldsymbol{\eta}_q^*) = \min\left[M_{\mathrm{PF}}(P_q), M_{\mathrm{PS}}(P_q)\right] + \min\left[M_{\mathrm{NF}}(\#_q), M_{\mathrm{NS}}(\#_q)\right]$
Internal, non-coding	$M(*q, -) = \min\left[M_{\mathrm{PF}}(P_q), M_{\mathrm{PS}}(P_q)\right] + M_{\mathrm{NE}}(-)$
Leaf, coding	$M(-, \boldsymbol{\eta}_q^*) = M_{\mathrm{PE}}(-) + \min\left[M_{\mathrm{NF}}(\#_q), M_{\mathrm{NS}}(\#_q)\right]$
Leaf, non-coding	$M(-, -) = 0$

while the sparse table has overhead

$$M_{\mathrm{N}}(\boldsymbol{\eta}_q) = M_{\mathrm{NS}}(\#_q) = 1 + 1 + \log_2 M + L_{\mathrm{L}}(N'_q) + \log_2 \#(p*) + L_{\mathrm{T}}(\#'_q),$$

where $\#'_q$ is the sorted next-symbol count table, and N'_q is the corresponding set of next-symbols.

The previous-symbol terms $M_{\mathrm{P}}(*q)$ and next-symbol terms $M_{\mathrm{N}}(\boldsymbol{\eta}_q)$ can be combined in four different configurations, as shown in Table 4.1. The only one that is not found in the `CodeSubtree` routine is the term $M(-,-)$. This is because a non-coding leaf node has no modelling value and is therefore eliminated by the SECT algorithm. Figure 4.2 shows an example of the various model overheads described in this section.

4.3.2 The `CodeSubtree` routine

The `CodeSubtree` routine is described in more detail in Figure 4.3. The first important step in the routine is the counting of the next-symbols, which are the symbols that follow occurrences of the context string q in the data set. This produces a table of next-symbol counts $\#_q$ as in (4.3.1). There are various ways to obtain these counts, and they have a major impact on the computational complexity of the algorithm, which will be discussed in Section 4.5.

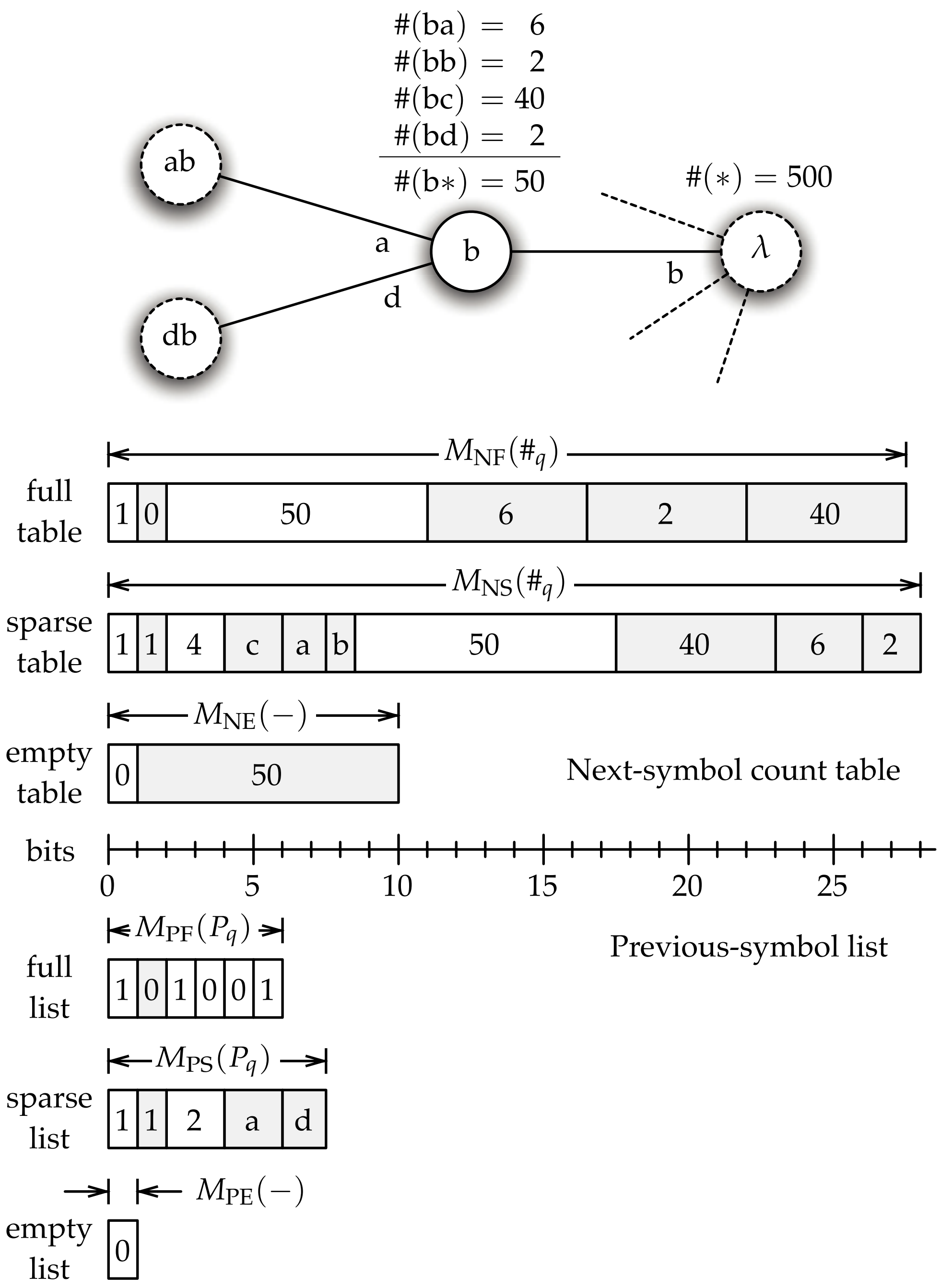


Figure 4.2: An example of the model overheads involved in the SECT algorithm. The various options for the next-symbol count table and previous-symbol list of node `b` are shown. The width of each box containing a value indicates the number of bits required to store that value. The box colours alternate to highlight the various sections of each table and list.

```
CodeSubtree
Inputs:   root context $q$, default distribution $\boldsymbol{\eta}^0_q$, parent next-symbol total $\#(p*)$
Outputs:  coded-symbol counts $\boldsymbol{n}_q$, score $L(q)$

1. Initialise $n_q(s) = 0$ for all $s \in \Sigma$, and let children score $L(*q) = 0$.
2. Count next-symbols to obtain table $\boldsymbol{\#}_q$ and optimal distribution $\boldsymbol{\eta}^*_q$.
3. **Calculate model overhead** $M(-, \boldsymbol{\eta}^*_q)$.
4. Table useful if $M(-, \boldsymbol{\eta}^*_q) + D\left(\boldsymbol{\#}_q \middle| \boldsymbol{\eta}^*_q\right) < D\left(\boldsymbol{\#}_q \middle| \boldsymbol{\eta}^0_q\right)$.
5. If **worthwhile to go deeper**:
    a) If table useful, child default distribution $\boldsymbol{\eta}^0_{mq} = \boldsymbol{\eta}^*_q$, else $\boldsymbol{\eta}^0_{mq} = \boldsymbol{\eta}^0_q$.
    b) For each previous-symbol $m$ in $\{m \in \Sigma \mid \#(mq) > 0\}$, do recursion
           $[\boldsymbol{n}_{mq}, L(mq)] = \text{CodeSubtree}(mq, \boldsymbol{\eta}^0_{mq}, \#(q*))$
       and accumulate $\boldsymbol{n}_q = \boldsymbol{n}_q + \boldsymbol{n}_{mq}$ and $L(*q) = L(*q) + L(mq)$.
    c) Useful children form previous-symbol list $P_q = \{m \in \Sigma \mid L(mq) > 0\}$.
6. **Calculate model overheads** $M(*q, \boldsymbol{\eta}^*_q)$ and $M(*q, -)$.
7. Table still useful if it codes uncoded symbols better than default; i.e. if
       $M(*q, \boldsymbol{\eta}^*_q) + D\left(\boldsymbol{\#}_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}^*_q\right) < M(*q, -) + D\left(\boldsymbol{\#}_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}^0_q\right)$.
8. If table useful, subtree rooted at $q$ takes care of all counts $\boldsymbol{\#}_q$, so that
       $L(q) = L(*q) + M(*q, \boldsymbol{\eta}^*_q) + D\left(\boldsymbol{\#}_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}^*_q\right)$   and   $\boldsymbol{n}_q = \boldsymbol{\#}_q$;
   otherwise, only $\boldsymbol{n}_q$ symbols are coded so far, and
       $L(q) = L(*q) + M(*q, -)$.
9. If subtree $q$ cannot code $\boldsymbol{n}_q$ better than default, i.e. if $L(q) \geq D(\boldsymbol{n}_q | \boldsymbol{\eta}^0_q)$,
   delete subtree $q$ (remove all nodes with suffix $q$ from $Q$) and reset $n_q(s) = 0$
   and $L(q) = 0$; otherwise, add node $q$ to context set $\mathcal{Q}$ if the table is useful.
```

Figure 4.3: Pseudo-code for the `CodeSubtree` routine in the SECT algorithm. The steps in bold are described elsewhere in this chapter.

The first of three tests in `CodeSubtree` now checks whether the optimal coding (maximum-likelihood) distribution

$$\boldsymbol{\eta}_q^* = \frac{\#_q}{\#(q*)} = \left\{ \frac{\#(qs)}{\#(q*)} \middle| s \in \Sigma \right\} \tag{4.3.2}$$

based on the next-symbol counts can improve on the default distribution $\boldsymbol{\eta}_q^0$ when coding the next-symbols, while also including the overhead for specifying $\boldsymbol{\eta}_q^*$. This occurs when

$$M(-, \boldsymbol{\eta}_q^*) + D\left(\#_q \middle| \boldsymbol{\eta}_q^*\right) < D\left(\#_q \middle| \boldsymbol{\eta}_q^0\right), \tag{4.3.3}$$

where the model overhead $M(-, \boldsymbol{\eta}_q^*)$ is that of a coding leaf node. If this test succeeds, the table $\#_q$ is marked as provisionally useful. This indicates that even if the rest of the subtree fails to improve on the coding of the next-symbols, node q will be kept as a coding leaf node. On the other hand, the children of q might compress the next-symbols even further, in which case table $\#_q$ will be dropped and q will remain as a non-coding internal node.

The next step determines whether it is worthwhile to continue expanding the tree. The expansion will stop if any of the following conditions are met:

- The length of context string q (the tree level or order) is greater than or equal to an optional user-specified limit $L_{\max}$. This allows the user to control the maximum order of the inferred PST, if so desired.

- The next-symbol table $\#_q$ contains a single symbol with non-zero count. This indicates that node q is a *zero entropy* node which perfectly predicts the next symbol based on the context q. It is therefore impossible to improve on its coding of the next-symbols (which receive zero bits each), and further expansion of the tree is pointless.

- The next-symbol table $\#_q$ is not useful, and the most optimistic subtree scenario is useless as well. This scenario has one zero-entropy child node for every symbol with non-zero count in $\#_q$. In this scenario, the next-symbols of q can be perfectly predicted, with zero bits allocated for each symbol, if the context length is increased by one more symbol. The code length for the subtree, given by

$$L'(q) = M(*q, -) + |N_q| \cdot (1 + 1 + \log \#(q*) + 1 + 2\log_2 M),$$

only contains model overhead terms, where the current node q becomes an internal non-coding node with overhead $M(*q, -)$, $|N_q|$ is the number of non-zero counts in $\boldsymbol{\#}_q$ and also the number of children nodes, and each child node is a coding leaf node with a sparse table containing a single symbol. In a sense, this subtree scenario is the most radical departure from the distribution of $\boldsymbol{\eta}_q$, which have already failed to beat the default distribution. If the new attempt also fails to improve on the default coding, by having $L'(q) \geq D(\boldsymbol{\#}_q|\boldsymbol{\eta}_q^0)$, the tree expansion stops and the current subtree is discarded.

If expansion of the tree is accepted, all unique symbols which precede context string q in the data set is collected in a *previous-symbol list* $P_q = \{m \in \Sigma \,|\, \#(mq) > 0\}$. For each symbol m in this list, the child context string mq is formed. If the current next-symbol table was found to be useful, $\boldsymbol{\eta}_q^*$ becomes the new default distribution for the children nodes to beat. The `CodeSubtree` routine is then recursively called on the child context, also passing on the appropriate default distribution and the total next-symbol counts $\#(q*)$ of the current node. The routine returns with a list of coded symbols $\boldsymbol{n}_{mq}$ and an associated score $L(mq)$, which are accumulated for all child nodes. The end result is a list of symbols, $\boldsymbol{n}_q$, that have been successfully coded by some offspring of q, and the combined description length $L(*q)$ for these symbols. The previous-symbol list P_q is also pruned to contain only children nodes who managed to code some of the next-symbols better than q did.

The second test in `CodeSubtree` checks whether the next-symbol table $\boldsymbol{\#}_q$ is still useful to store. If all next-symbols have been successfully coded by the children of q, the table can be safely discarded. On the other hand, any uncoded symbols will be sent back up the tree until some node is able to code it. If $\boldsymbol{\eta}_q^*$ compresses the uncoded symbols $\boldsymbol{\#}_q - \boldsymbol{n}_q$ better than the default distribution does, it is still worthwhile to keep the next-symbol table.[4] This succeeds if

$$M(*q, \boldsymbol{\eta}_q^*) + D\left(\#_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}_q^*\right) < M(*q, -) + D\left(\#_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}_q^0\right), \tag{4.3.4}$$

where the model overheads $M(*q, \boldsymbol{\eta}_q^*)$ and $M(*q, -)$ describe q as a coding and non-coding node, respectively. These overheads make use of the pruned previous-symbol list P_q. This list could be empty, in which case q is a leaf node and the second test becomes identical to the first test in (4.3.3).

If the table $\#_q$ is still found to be useful, all uncoded counts are coded by $\boldsymbol{\eta}_q^*$, and the score for the subtree becomes

$$L(q) = L(*q) + M(*q, \boldsymbol{\eta}_q^*) + D\left(\#_q - \boldsymbol{n}_q \middle| \boldsymbol{\eta}_q^*\right).$$

This combines the scores of all children nodes with the model overhead of having q as a coding node and the data code length of the previously uncoded counts. If the previous-symbol list P_q is empty, q becomes a coding leaf node and the score simplifies to

$$L(q) = M(-, \boldsymbol{\eta}_q^*) + D\left(\#_q \middle| \boldsymbol{\eta}_q^*\right).$$

All symbols in $\#_q$ are also marked as coded, by setting $\boldsymbol{n}_q = \#_q$. On the other hand, if the table is rejected in test (4.3.4), the subtree score only contains the children scores and the overhead of setting up q as a non-coding node, i.e. $L(q) = L(*q) + M(*q, -)$, and the coded counts $\boldsymbol{n}_q$ remain unchanged. Without children, the node q is doomed in this case, as no symbols have been coded yet.

The final test in `CodeSubtree` checks whether all the effort of constructing the PST subtree was worthwhile, by comparing the final subtree score with the default code length of the symbols $\boldsymbol{n}_q$ that have been coded by the subtree. If

$$L(q) < D(\boldsymbol{n}_q | \boldsymbol{\eta}_q^0), \tag{4.3.5}$$

[4]Note that this is not ideal, as the optimal code for the uncoded counts is the maximum likelihood distribution based on these counts, which generally differs from $\boldsymbol{\eta}_q^*$. This situation is referred to as wild-card nodes in [20]. The table $\#_q$ is not replaced by an arbitrary uncoded-symbol count table, however, to preserve the consistency of the PST encoding. The next-symbol table of each node is currently a subset of the table of its parent, which allows further refinement of the constraints on the table and more efficient coding. This advantage would be lost if tables are replaced by arbitrary smaller ones.

the subtree improves on the default coding, and node q is added to the coding context set $\mathcal{Q}$ if it is a coding node. This will be the case if test (4.3.4) found the next-symbol table $\#_q$ to be useful. If q is non-coding, nothing needs to be done. Any coding children nodes will have been added to $\mathcal{Q}$ already, and the non-coding nodes can be added to the PST afterwards, by following the procedure in Section 2.3.3. Note that if q is a coding leaf node, test (4.3.5) is again identical to tests (4.3.3) and (4.3.4).

If the subtree fails test (4.3.5), it is discarded. All children nodes of q that were added to the coding context set $\mathcal{Q}$ are removed again in this case. These nodes are identified by having q as a suffix of their context string. Note that it is quite possible for a deeper context q to be added to the list of useful contexts, only to be removed again later when the overhead of non-coding internal nodes required to connect node q with the root node became too much. If the subtree fails test (4.3.5), the coded-symbol counts $\boldsymbol{n}_q$ and score $L(q)$ are also reset to zero.

The `CodeSubtree` routine finally returns the values of $\boldsymbol{n}_q$ and $L(q)$.

4.3.3 Estimation of next-symbol probabilities

The core SECT algorithm discovers the useful coding nodes in a PST that allows a compact description of the data set. The output of the algorithm is the context set $\mathcal{Q}$, which represents the node structure of the PST. The full specification of a PST is $\{\Sigma, \mathcal{Q}, \eta\}$, however, where the alphabet Σ is assumed to be known, but the next-symbol distribution η is still unspecified. The SECT algorithm is flexible in this regard, and allows the use of any estimator for the next-symbol distribution function $\eta(q,s)$ that is based on the next-symbol counts $\#_q = \{\#(qs) \mid s \in \Sigma\}$, with total count $\#(q*) = \sum_s \#(qs)$. Some popular estimators include:

$$\text{Maximum likelihood:} \quad \eta_{\text{ML}}(q,s) = \frac{\#(qs)}{\#(q*)}$$

$$\text{Krichevsky-Trofimov:} \quad \eta_{\text{KT}}(q,s) = \frac{\#(qs) + \frac{1}{2}}{\#(q*) + \frac{M}{2}}$$

$$\text{Laplace rule:} \quad \eta_{\text{LR}}(q,s) = \frac{\#(qs) + 1}{\#(q*) + M}$$

Dirichlet prior: $$\eta_{\mathrm{DP}}(q,s) = \frac{\#(qs)+\alpha}{\#(q*)+M\alpha}, \quad 0 < \alpha \leq 1$$

Learn-PSA: $$\eta_{\mathrm{PSA}}(q,s) = \frac{\#(qs)}{\#(q*)}(1-\eta_{\min}M)+\eta_{\min}, \quad 0 < \eta_{\min} < \frac{1}{M}$$

The maximum likelihood estimate $\eta_{\mathrm{ML}}(q,s)$ assigns the maximum probability to the training data, but $\eta_{\mathrm{ML}}(q,z) = 0$ for any symbol z which did not follow the context string q in the training data. If the data set is small and symbol z rarely follows context q, it is highly likely that $\#(qz) = 0$. Any test string containing the substring qz will be awarded zero probability by the maximum likelihood PST if node q is the relevant coding context for qz. This is referred to as the *zero frequency problem* [148, 149], and it is especially an issue in the fields of language modelling [150, 3, 151] and text compression [152, 153].

The other estimators mentioned above all include some form of smoothing, which prevents the next-symbol probabilities from becoming zero. The Krichevsky-Trofimov estimator [154] is applied to PSTs in [20], and Laplace's rule of succession [155, 156] is commonly used for smoothing. Both of these estimators are special cases of the use of Dirichlet priors [140]. In contrast, the Learn-PSA algorithm provides a lower bound $\eta_{\min}$ for $\eta(q,s)$, which has the disadvantage that the resulting bias in the estimate does not diminish with training set size. A good overview of smoothing techniques for language modelling can be found in [151, 157].

4.3.4 Refinement of SECT

The SECT algorithm constructs a description of a PST, and a description of the data in terms of the PST. The data description is straightforward to optimise, by using the optimal code of (2.1.2) associated with the predictive distribution of the PST. On the other hand, the PST description is more difficult to optimise, as it is constructed heuristically to be a compact representation that maintains decodability. The performance of SECT can be improved by shrinking the PST description length as much as possible, without losing decodability. This allows the correct model to be inferred using less training data, as implicated by (4.2.2).

The description length of the PST depends largely on the description length of the next-symbol count tables, as these take up most of the space in the representation. The table descriptions can be shrunk if tighter constraints on the values of the next-symbol counts in them are discovered and exploited.

The current incarnation of SECT, as described in Section 4.3.1, stores the next-symbol table $\#_q$ of node q as the total count $\#(q*)$ followed by the individual counts $\#(qs)$, where $s \in \Sigma$. The total count of q is constrained to be less than or equal to that of its parent node p, which allows it to be stored in $\log_2 \#(p*)$ bits. Since all nodes store their total counts, decodability of $\#(q*)$ is guaranteed. Each individual count in the table is constrained to be less than or equal to the total count. Furthermore, as each successive count is decoded, the sum of the remaining counts can be calculated, which serves as a tighter constraint on the value of the next count. The last count is superfluous and need not be stored, as it can be determined by subtracting the rest of the counts from the total count. These constraints are captured in the table length formula of (4.1.2).

Additional constraints can be obtained if the parent is a coding node. Suppose node $q = n \cdot p$ has parent p, with $n \in \Sigma$. The next-symbol counts of q are constrained to be less than or equal to the corresponding next-symbol counts of p; that is, $\#(nps) \leq \#(ps)$ for any $s \in \Sigma$. For example, this constraint comes into effect if the first count in $\#_p$ is less than $\#(q*)$, which allows the first count in $\#_q$ to be stored in fewer bits. The symbols with non-zero counts in $\#_p$, indicated by the set $N_p = \{s \in \Sigma \mid \#(ps) > 0\}$, can also replace the full alphabet Σ in calculations involving the next-symbol table $\#_q$, as $\#(ps) = 0$ implies $\#(nps) = 0$. The full table option only has to code $|N_p|$ counts, while the symbol list in the sparse table option are drawn from N_p instead of Σ.

The constraint can be refined even more by subtracting any accumulated coded-symbol counts $\boldsymbol{n}_p$ from the parent table $\#_p$ before presenting it to the child node q. The subtracted symbol counts represent next-symbols that were successfully coded by sibling subtrees of q that have already been encoded by the SECT algorithm. As a concrete example, consider a PST on the binary alphabet $\{\mathtt{a}, \mathtt{b}\}$, with a parent node $p = \mathrm{bb}$ having a next-symbol table $\#_p = \{\#(\mathtt{bba}) : 120, \#(\mathtt{bbb}) : 50\}$. Node p has two children, $r = \mathtt{abb}$ and $q = \mathtt{bbb}$. Suppose node r is encoded first, and its subtree manages to code 50 a's and 30 b's of the next-symbols of p, resulting in accumulated coded-symbol counts of

$\boldsymbol{n}_p = \{\texttt{a} : 50, \texttt{b} : 30\}$. Instead of using $\#_p$ as upper bound for $\#_q$, a tighter bound is given by $\#_p - \boldsymbol{n}_p = \{\texttt{a} : 70, \texttt{b} : 20\}$. These *uncoded-symbol counts* perform the same role as the sum of remaining symbols in the original constraints of (4.1.2).

It is important to check unique decodability when these constraints are employed. Although the SECT algorithm adds coding children nodes to $\mathcal{Q}$ before their parents in `CodeSubtree`, a practical compression scheme will write out the tree in a top-down left-to-right fashion. This is because the children nodes were encoded based on constraints derived from their parents. Using this scheme, the next-symbol table of a parent node will therefore be decoded and available before it is required by a child node. Similarly, the coded-symbol counts associated with a subtree will be available for the calculation of uncoded-symbol count constraints for the next child node, as long as the children nodes are decoded in the same order as they were encoded.

The only complication is that the decision to keep the next-symbol table of a node, and thereby mark it as coding, is made *after* all its children have been encoded. If the parent table is discarded, it cannot be used as constraint for the children tables anymore, and the children nodes would have to be re-encoded, based on the original constraints of (4.1.2). To prevent two passes through the subtree in this case, the subtree score $L(q)$ is replaced by two scores. The "best" score assumes that the parent p of the subtree root q is a coding node, while the "safe" score assumes that it is not. Both these scores are returned to the parent node, which incorporates the appropriate one in its own scores after the fate of its table has been decided. Any other tests in `CodeSubtree` are based on the "safe" score, which preserves decodability at the expense of code length.

An implementation of these refined constraints modifies the `CodeSubtree` routine to receive the uncoded-symbol counts $\#_p - \boldsymbol{n}_p$ of the parent node as extra argument, and to return both the "best" and "safe" subtree scores.

4.4 Conversion to Markov chain

While the prediction suffix tree has an efficient structure for inference, it is less efficient when it comes to scoring and generating data sequences. A PST with maximum depth L requires $O(LT)$ computations to score or generate a sequence of length T, while the equivalent Markov chain has $O(T)$ complexity. This problem can be overcome by converting the PST into its equivalent Markov chain, also known as a probabilistic suffix automaton (PSA).

The equivalent PSA starts off by having all the coding nodes $\mathcal{Q}$ of the PST as states. If $\mathcal{Q}$ does not contain the zero-order context λ, it is added to $\mathcal{Q}$ with the default distribution $\boldsymbol{\eta}_0$ as next-symbol distribution. The conversion process then adds all required glue states to the PSA, as described in Section 2.3.1. That is, the state space $\mathcal{Q}$ is extended to $\mathcal{Q}' = \cup_{q \in \mathcal{Q}} \mathcal{P}(q) \cup \Sigma$, where $\mathcal{P}(q)$ is the set of all prefixes of the coding context string q. All first-order contexts Σ are explicitly added to $\mathcal{Q}'$, as the PST may be empty or missing some of these contexts.[5] The extra added states $\mathcal{Q}' - \mathcal{Q}$ are non-coding and do not have their own next-symbol distributions. Instead, glue state $q \in \mathcal{Q}' - \mathcal{Q}$ uses the distribution of the longest suffix of q in $\mathcal{Q}$. The extended PSA contains at most $L|\mathcal{Q}| + M$ states, and typically considerably fewer states. For example, a fixed-order PST with all leaf nodes as coding nodes has an equivalent PSA with less than $\frac{M}{M-1}|Q|$ states for $M > 1$.

Each state q in $\mathcal{Q}'$ now receives a unique numerical index $k(q) \in [1, N]$, where $N = |\mathcal{Q}'|$ is the number of states in the PSA. The standard formulation of a Markov chain describes it in terms of a transition probability matrix $\boldsymbol{A}$ of dimensions $N \times N$. If context q is followed by symbol s with probability $\eta(q, s)$, while making a transition to context $r = \tau(q, s)$, the transition matrix $\boldsymbol{A}$ contains the corresponding element $a_{ij} = \eta(q, s)$, with $i = k(q)$ and $j = k(r)$. The construction of matrix $\boldsymbol{A}$ starts by initialising it with zeroes. For each source state $q \in \mathcal{Q}'$ and each symbol $s \in \Sigma$, the destination state $r = \tau(q, s)$ of the corresponding state transition is determined by finding the longest suffix r of string qs in $\mathcal{Q}'$. Both q and r are expressed by their numerical indices $k(q)$ and $k(r)$, and the corresponding element of $\boldsymbol{A}$ becomes $a_{k(q)k(r)} = \eta(q, s)$. The symbol associated with state $k(q)$ is the last symbol of the context string q, and the initial state of the Markov chain has index $k(\lambda)$. This completes the specification of the Markov chain equivalent to the PST.

[5]This assumes that the PSA is implemented in a *Moore* form, where symbols are emitted by the states. Each symbol therefore requires its own state, and the PSA cannot have less than M states. If the PSA is implemented in a *Mealy* form, with symbol emissions on the state transitions, this is unnecessary.

4.5 Computational complexity

The two most time-consuming steps in Learn-PSA, SECT, and several other PST inference algorithms [22, 25, 19, 20] are the calculation of the next-symbol counts $\{\#(qs) \mid s \in \Sigma\}$ and previous-symbol candidates $\{m \in \Sigma \mid \#(mq) > 0\}$ of node q. In the algorithms mentioned, these steps also represent the only interaction of the algorithm with the data set. The two steps can easily be combined into one step, by searching for the context string q in the data and simultaneously observing the symbols which precede and follow it.

The original Learn-PSA algorithm [22] adds context strings from a data set to a list, and scans the entire data set to count the symbols following each of these strings. The scanning process requires $O(T)$ time. If L is the maximum depth of the tree (and maximum length of the context strings) and T is the data length, the list potentially contains $T - l$ different context strings for each $1 \leq l \leq L$, leading to a worst-case computational complexity of $O(LT^2)$ [22, 20].

The time complexity of Learn-PSA can be improved from quadratic to linear in the data length T, by modifying the counting process as mentioned in [22]. Each node q in the PST receives a list of pointers to the occurrences of the string q in the data set. To initialise the process, the root node λ receives a list of T pointers, one for each symbol in the data set. When the next-symbols of q are counted, only the pointer list of q needs to be scanned, instead of the whole data set. The pointer list of q is then partitioned to create the pointer lists of the children nodes. Both scanning and partitioning require time that is linear in the number of pointers. There are $T - l$ pointers in total on level $l \in [0, L]$ of the tree, and with $L + 1$ levels in the tree this results in a computational complexity of $O(LT)$. The drawback of this method is a corresponding increase in storage requirements, as $O(LT)$ pointers have to be stored.

The SECT algorithm uses a similar approach, but maintains a single list containing T pointers for the entire tree, instead of a separate list for each node. Each pointer in the list points to a unique symbol in the data set. Each node q is associated with a contiguous section of this list, which represents the next-symbols following context q in the data. The section associated with the root node is the full list, as all symbols follow the empty context λ. The next-symbols of node q are counted by scanning the pointers in the list section associated with

q, and counting the symbols to which they point. In the partitioning step, the list section of q is sorted according to the previous-symbols which precede each occurrence of the string q in the data. This subdivides the section of q into contiguous subsections based on the deeper contexts, which are then assigned to the children nodes of q.

For an example of this process, consider the data sequence abcbabcac on the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. The pointer list is initialised to $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, where the pointers are implemented as symbol indices. The next-symbol counts for the root node are $\#_\lambda = \{\texttt{a} : 3,\ \texttt{b} : 3,\ \texttt{c} : 3\}$. The pointer list is now sorted according to the symbol preceding each symbol which is pointed to, resulting in an updated list $\{2, 6, 9, 3, 5, 7, 4, 8, 1\}$. The first three elements in the new list point to symbols following a, the next three elements point to symbols following b and the next two elements point to symbols following c. The last pointer in the new list points to the first symbol in the data sequence, which has no preceding symbol and is therefore ignored. Node a is associated with the first three elements in the list, from which $\#_\texttt{a} = \{\texttt{b} : 2,\ \texttt{c} : 1\}$. In the next round, the section of node a is sorted according to the symbol two positions behind each symbol which is pointed to, resulting in the new list $\{6, 9, 2, 3, 5, 7, 4, 8, 1\}$. Symbol 6 is the next-symbol of context ba, symbol 9 is the next-symbol of context ca, and symbol 2 is once again discarded, as it is not preceded by a second-order context.

While the scanning of the pointer list in SECT still requires time linear in the number of pointers, the partitioning step is now more expensive, as it is based on sorting. A list of T elements is sorted in $O(T \log T)$ time. On level 0 of the tree, the entire pointer list of T elements is sorted. If the symbols are roughly equiprobable, M lists of T/M elements are sorted on level 1 in the tree, M^2 lists of T/M^2 elements on level 2, and so forth. At the other extreme, a single symbol might dominate the data set, which results in roughly $T \log T$ operations on each level of the tree. Both scenarios indicate a computational complexity of $O(LT \log T)$ for the SECT algorithm, where L is the maximum tree depth explored during inference. While SECT is asymptotically slower than the improved Learn-PSA algorithm, it requires less storage. Excluding the tree structure itself, the algorithm only needs to store the data sequence and pointer list, both of length T. SECT therefore represents a useful compromise between time complexity and space complexity.

The Learn-PSA algorithm requires a user-specified bound L on the maximum depth of the tree. The SECT algorithm is self-bounded, on the other hand, with a maximum tree depth that is effectively a function of the data size T and underlying model complexity. If the symbols are roughly equiprobable, SECT will expand the tree up to a maximum depth of $L = \log_M T$, which results in an overall computational complexity of $O(T(\log T)^2)$. In the worst case, the maximum depth $L = O(T)$ (although this would require a pathological data set), resulting in a computational complexity of $O(T^2 \log T)$.

The optimised PST inference algorithm described in [25, 20] has time and space requirements that are both linear in the data length T. It is based on the efficient constructs of the trie [134], the classical suffix tree [135, 136] and the multiple pattern matching machine (MPMM) [158]. Its main ideas can be adapted to benefit the SECT algorithm as well. SECT is designed to be flexible: it can incorporate any of the symbol counting procedures mentioned in this section. The specific implementation, based on a single sorted pointer list, was chosen for its simplicity and trade-off between speed and storage.

The conversion algorithm in Section 4.4, which converts a PST to an equivalent Markov chain, has a computational complexity of $O(LG)$, where L is the maximum depth of the PST and $G \leq MN$ is the number of non-zero transition probabilities in the Markov chain. This results from the last step in the algorithm, which traverses the tree to find the destination state of each allowed state transition. There are G transitions with non-zero probability, and up to L nodes have to be followed from the root of the PST to reach the appropriate destination state. In contrast, the original conversion routine in [22] is claimed to have a worst-case complexity of $O(LT^2)$ in [20], which is similar to that of Learn-PSA.

4.6 Relation to Bejerano's MDL algorithm

The non-parametric PST training algorithm described in [26, 20] and summarised in Section 3.1.3 is the algorithm that bears the closest relation to SECT. Like SECT, it is also based on the minimum description length principle, which allows it to be self-bounded and removes the need for user-specifiable parameters.

The basic test that determines whether a specific node is an improvement on its parent node is similar for both algorithms. The version of SECT is more accurate, though. Let $\boldsymbol{\eta}_q$ be the next-symbol distribution derived from the next-symbol counts of node q, $L(q)$ the description length of node q, and $\#(q*)$ the number of occurrences of the context q in the data. As described in Section 4.2, the SECT version considers a node q to be useful when the Kullback-Leibler divergence between its next-symbol distribution and that of its parent p exceeds a threshold; that is, if

$$\mathrm{D}(\boldsymbol{\eta}_q \| \boldsymbol{\eta}_p) > \frac{L(q)}{\#(q*)}.$$

Based on the description in [20, Section 5.2], Bejerano's algorithm tests whether

$$H(\boldsymbol{\eta}_p) - H(\boldsymbol{\eta}_q) > \frac{L(q)}{\#(q*)}$$

instead, where $H(\boldsymbol{\eta}_q)$ is the entropy[6] of the next-symbol distribution $\boldsymbol{\eta}_q$. While the entropy difference is similar to divergence in some regards (for example, it is also zero when the two distributions are the same), this test will only allow nodes with a lower entropy than their parents in the final PST. Although it caters for a common scenario, this form of the test nevertheless restricts the type of PSTs that can be learnt by Bejerano's algorithm. The SECT algorithm does not have this restriction.

From a computational perspective, SECT is an improvement. Bejerano's algorithm shares the quadratic complexity of Learn-PSA, which means that SECT has a lower computational complexity. Furthermore, Bejerano's algorithm first fully expands the PST before pruning it. The SECT algorithm rolls these two steps into one recursive tree-building phase, whereby useless subtrees are discarded before new ones are explored. This requires less memory.

[6]In Bejerano's algorithm, the parent node p assigns a code length $\#(q*)H(\boldsymbol{\eta}_p)$ to the next-symbols of its child node q, which can be expanded to

$$\sum_{s\in\Sigma} \#(q*)\eta(p,s)\log_2 \frac{1}{\eta(p,s)}.$$

In comparison, SECT assigns a (more correct) code length of

$$D(\#_q|\boldsymbol{\eta}_p) = \sum_{s\in\Sigma} \#(qs)\log_2 \frac{1}{\eta(p,s)}.$$

On the other hand, the node description length of Bejerano's algorithm, as specified by (3.1.1), is potentially more efficient than the conservative heuristic approach of SECT. The encoding of its next-symbol counts is related to the approaches in [45, 47], which have stronger theoretical underpinnings than SECT. This could allow Bejerano's algorithm to learn the correct PST structure from less data.

4.7 Summary

This chapter describes the SECT algorithm in detail. In summary, the algorithm starts with an empty context tree and recursively adds nodes that can potentially improve on the coding of their parent nodes, while also considering the overhead of specifying these nodes. The tree expansion stops if an optional bound on the tree depth is reached, or the children nodes of the current node cannot improve on its coding, even in the most optimistic case. As the recursive process moves back up the tree, it discards subtrees that do not deliver on their promises.

The coding difference between a node and its parent is measured by the Kullback-Leibler divergence between the next-symbol distributions of the two nodes, while the node overhead consists of specifying its next-symbol table and previous-symbol list. The algorithm uses sorting to count the next-symbols faster without incurring a large memory overhead.

The computational complexity of SECT is typically $O(T(\log T)^2)$, where T is the training sequence length. The resulting PST can be efficiently converted into an equivalent Markov chain. The conversion process has a computational complexity of $O(LG)$, where L is the maximum order of the model and G is the number of links with non-zero transition probability in the Markov chain.

Chapter 5

The Hidden SECT Algorithm

The *hidden smallest encoded context tree* (hidden SECT) algorithm applies the SECT algorithm for prediction suffix trees to the inference of hidden Markov models. The key idea behind this algorithm is to replace the underlying Markov chain in a hidden Markov model with an equivalent prediction suffix automaton. This provides a straightforward way to infer the HMM topology from data, by focusing on symbols instead of states.

5.1 Rationale (symbols versus states)

It is instructive to view an HMM from the perspective of *symbols*. Recall from Section 2.2.2 that an Lth-order Markov chain $\mathcal{M}_L = (\Sigma, \mathcal{Q}, \tau, \eta)$ models symbol sequences $s_1^T \in \Sigma^T$, where Σ is the alphabet of M symbols, $\mathcal{Q}$ is a set of N states, τ is the state transition function and η is the next-symbol distribution. Each state $q \in \mathcal{Q}$ is a string of up to L symbols, representing a symbol context. There is a one-to-one correspondence between the symbol sequence s_1^T and its associated state sequence q_0^T. To obtain the symbol sequence from the state sequence, write down the last symbol in the context string of each state. To obtain the state sequence from the symbol sequence, start with $q_0 = \lambda$ and iteratively apply the state transition function $q_t = \tau(q_{t-1}, s_t)$. The distinction between symbol and state becomes very useful when $L > 1$, since the dependencies between states remain first-order by design, while the dependencies between symbols become higher-order.

The Markov chain is turned into an HMM by introducing an observation space $\mathcal{X}$ and identifying each symbol $s \in \Sigma$ with a probability density function $\sigma(s, x)$ on this space. The HMM models observation sequences $x_1^T \in \mathcal{X}^T$ instead of symbol sequences. If the symbol pdfs overlap, the symbol sequence cannot be deduced from the observation sequence and effectively becomes hidden.

A symbol refers to an observation density label (output pdf index) in this notation. In a higher-order HMM, many states are tied to the same observation density or symbol, which makes symbols more fundamental than states, in a sense. This is especially true for continuous HMMs, where a symbol is bound to a specific region in observation space, while states function as symbol contexts on a higher level.

This use of the term "symbol" can also be confusing. In a discrete HMM, the observation space $\mathcal{X}$ is also an alphabet of (different) symbols, and the observation sequence is usually called the "symbol sequence" in the discrete HMM literature. This views a discrete HMM as a more powerful MC instead of as a two-level model. It is possible to merge the two alphabets into one Cartesian product alphabet and redesign the HMM accordingly, but this tends to obscure the Markovian relationships in the model. As we are more interested in higher-order continuous HMMs, we will tolerate this potential confusion, and always refer to *symbol* and *observation* to distinguish between the two cases.

Given a symbol sequence s_1^T, there are several ways to infer an Lth-order Markov chain from it. The PST learning algorithms are particularly useful, as they provide variable-order models. Most of these methods are based on the counts of various context strings in s_1^T and the symbols which follow them. In an HMM, the symbol sequence is hidden, however, and has to be uncovered in some way during training to estimate the underlying Markov chain in the HMM. Baum-Welch re-estimation uncovers s_1^T probabilistically, while Viterbi re-estimation estimates it explicitly.

Consider a first-order HMM with parameters θ. The Baum-Welch algorithm provides an efficient way to calculate the probability $P(q_t = i, q_{t+1} = j | x_1^T, \theta)$, based on the forward and backward variables. Since states and symbols are equivalent in a first-order HMM (ignoring the initial state λ), this can also be written as $P\big(s_t = i, s_{t+1} = j \big| x_1^T, \theta\big)$. The quantity

$$\#'(ij) = \sum_{t=1}^{T-1} P\Big(s_t = i, s_{t+1} = j \Big| x_1^T, \theta\Big)$$

is the expected number of occurrences of the symbol string ij in the hidden symbol sequence. This "soft" count is normally used in Baum-Welch re-estimation to estimate the transition probability a_{ij} as

$$\hat{a}_{ij} = \frac{\#'(ij)}{\#'(i*)} = \frac{\#'(ij)}{\sum_j \#'(ij)},$$

but it can also be used as the basis for any other MC inference algorithm. The idea can be extended to longer context strings: e.g., the count $\#'(ijk)$ can be obtained as the sum over time of the probability $P\big(s_t = i, s_{t+1} = j, s_{t+2} = k \big| x_1^T, \theta\big)$.

The calculation of higher-order soft symbol counts can become complicated if the HMM is higher-order to start with, as the Baum-Welch algorithm operates on states instead of symbols, and the mapping between states and symbols is not so straightforward anymore. Alternatively, in order to obtain soft counts of all symbol strings up to length L, the HMM can be extended to order L before applying the forward-backward procedure. The posterior state probability $P\big(q_t = i \big| x_1^T, \theta\big)$ simultaneously serves as the probability that the context string of i occurs in s_1^T with its last symbol at time t, and summing the state probability over time produces the soft count of its context string. This is in effect what the FIT algorithm [39] does. This symbol counting procedure can be costly in terms of memory and computation time, though.

These costs can be avoided by using the Viterbi algorithm instead. This produces a single optimal state sequence q_0^{T*}, which is converted into its corresponding symbol sequence s_1^{T*}. While a single sequence does not capture all the variability of the model, it greatly simplifies the symbol counting procedure. In fact, any MC inference algorithm can be applied to s_1^{T*}. In order to control the size of the resulting Markov chain, it is useful to incorporate a model prior during training. This leads to the topic of maximum a posteriori (MAP) estimation of HMMs.

5.2 MAP estimation of HMMs

The Expectation-Maximisation (EM) algorithm [114] is normally used to maximise the likelihood $P(x|\theta)$ of the parameters θ of a model, but it can maximise the posterior density $P(\theta|x)$ or the joint density $P(x, \theta)$ as well (for an example, see the derivation in [118]). This is because Jensen's inequality can be used to obtain a lower bound for any of these densities, which can then be maximised.

Let $\boldsymbol{x} \equiv x_1^T$ be compact notation for the observation sequence, while $\boldsymbol{s} \equiv s_1^T$ is the symbol sequence and $\boldsymbol{q} \equiv q_0^T$ is the state sequence. The maximum a posteriori (MAP) estimate of the HMM parameters θ is given by

$$\theta_{\mathrm{MAP}} = \arg\max_{\theta} P(\theta|\boldsymbol{x}) = \arg\max_{\theta} P(\boldsymbol{x}, \theta) .$$

The EM algorithm maximises a lower bound for the joint density $P(\boldsymbol{x}, \theta)$. If we choose the lower bound to be

$$P(\boldsymbol{x}, \boldsymbol{s}, \theta) \leq P(\boldsymbol{x}, \theta) ,$$

we obtain a *winner-take-all* variant of EM [117], which is a MAP version of Viterbi re-estimation for HMMs. Although it does not converge to a local maximum of the posterior density and is therefore an approximation to MAP, Viterbi re-estimation greatly simplifies the MC inference step we want to add. The bound $P(\boldsymbol{x}, \boldsymbol{s}, \theta)$ is maximised using coordinate ascent. The E step optimises the bound with respect to $\boldsymbol{s}$, resulting in

$$\boldsymbol{s}^* = \arg\max_{\boldsymbol{s}} P(\boldsymbol{x}, \boldsymbol{s}, \theta) = \arg\max_{\boldsymbol{s}} P(\boldsymbol{x}, \boldsymbol{s}|\theta) ,$$

where the optimal state sequence $\boldsymbol{q}^*$ is first found using the familiar Viterbi algorithm and then converted to the corresponding optimal symbol sequence $\boldsymbol{s}^*$. This is possible because of the one-to-one correspondence between $\boldsymbol{q}$ and $\boldsymbol{s}$.

The M step of the algorithm maximises the bound with respect to θ, while setting the symbol sequence to its optimal value, to obtain

$$\theta^* = \arg\max_{\theta} P(\boldsymbol{x}, \boldsymbol{s}^*, \theta) .$$

The HMM parameters $\theta = (\theta_x, \theta_s)$ are now divided into two sets, where θ_x is the parameter set of the symbol pdfs, and θ_s is the parameter set of the underlying Markov chain. The set θ_x therefore represents the spatial structure of the HMM, while θ_s represents its temporal structure. The significance of this is that θ_s is estimated solely from the symbol sequence $\boldsymbol{s}$, while the estimation of θ_x requires both $\boldsymbol{x}$ and $\boldsymbol{s}$. Furthermore, the parameter sets are conditionally independent, given knowledge of the symbol sequence. The lower bound is factorised as

$$\begin{aligned} P(\boldsymbol{x}, \boldsymbol{s}^*, \theta_x, \theta_s) &= P(\boldsymbol{x}, \boldsymbol{s}^*, \theta_x)\, P(\theta_s | \boldsymbol{x}, \boldsymbol{s}^*, \theta_x) \\ &= P(\boldsymbol{x}, \boldsymbol{s}^* | \theta_x)\, P(\theta_x)\, P(\theta_s | \boldsymbol{s}^*) \,. \end{aligned}$$

It is possible to impose a prior $P(\theta_x)$ on the symbol pdfs as well, which allows control over the model complexity in observation space. As the focus is currently on the underlying MC, we ignore this prior by setting $P(\theta_x) = 1$. The M step now determines the optimal model $\theta^* = (\theta_x^*, \theta_s^*)$ as

$$\begin{aligned} \theta_x^* &= \arg\max_{\theta_x} P(\boldsymbol{x}, \boldsymbol{s}^* | \theta_x) \\ \theta_s^* &= \arg\max_{\theta_s} P(\theta_s | \boldsymbol{s}^*) \,. \end{aligned}$$

The parameters θ_x of the symbol pdfs are therefore updated to their maximum likelihood estimates, as in standard Viterbi re-estimation. All observations in $\boldsymbol{x}$ with the same symbol label according to $\boldsymbol{s}^*$ are grouped together as a set of independent samples and passed to the corresponding symbol pdf for re-estimation. The Markov chain parameters θ_s undergo MAP estimation, however, which incorporates a model prior $P(\theta_s)$ to control the complexity of the MC.

This approximate MAP training algorithm for HMMs is guaranteed to converge, as the winner-take-all variant of EM is guaranteed to converge [51, 120].

5.3 The algorithm

The hidden SECT algorithm replaces the underlying Lth-order Markov chain of the HMM with an Lth-order prediction suffix automaton (PSA). The L-PSA has identical modelling capabilities to the L-MC, but is potentially much more compact because of its variable-order nature. Furthermore, it can be efficiently learnt from the symbol sequence by first casting it as a prediction suffix tree (PST). Since the PSA also has a state machine structure, it is really a drop-in replacement for the Markov chain.

The approximate MAP HMM training algorithm introduced in the previous section allows MAP estimation of the Markov chain. The SECT algorithm is useful in this regard, as it is a MAP estimation procedure for PSTs. Once the PST is inferred from the symbol sequence, it can be converted to an equivalent PSA state machine. This provides an efficient way to control the size of the underlying Markov chain, and hence the order and topology of the HMM.

The code construction process of SECT insists on unique decodability, which tends to make the practical code length of a PST θ_s slightly larger than the optimal value of $-\log_2 P(\theta_s)$. In turn, this means that the PST prior $P(\theta_s)$ is underestimated by SECT. This mismatch in the prior cannot increase the lower bound

$$P(\boldsymbol{x}, \boldsymbol{s}, \theta_x, \theta_s) = P(\boldsymbol{x}, \boldsymbol{s} | \theta_x, \theta_s)\, P(\theta_s)$$

of the EM algorithm, which preserves its properties as bound.

The hidden SECT algorithm is summarised in Figure 5.1. The algorithm starts with an initial HMM, which is usually a standard first-order model with a topology that matches the application domain of the HMM. The symbol pdfs are also appropriately initialised, for instance by unsupervised clustering. The E step of the EM variant obtains the optimal symbol sequence, using the standard Viterbi algorithm. The M step is split into two parts, which separately handles the spatial and temporal sections of the HMM. The temporal structure is re-estimated by inferring a PST from the symbol sequence with SECT and converting it to an equivalent PSA, which replaces the existing Markov chain structure in the HMM. The spatial structure is re-estimated using standard maximum likelihood estimation. The E and M steps are alternately repeated until the Viterbi score settles[1] or a specified maximum number of iterations are reached.

[1]The Viterbi score is given by

$$V^* = \max_{\boldsymbol{s}} P(\boldsymbol{x}, \boldsymbol{s} | \theta)\,,$$

which tracks changes in the likelihood instead of the actual objective function, $P(\boldsymbol{x}, \boldsymbol{s}, \theta)$. It is simpler to calculate, however, and works well in practice.

```
Hidden SECT
1. Obtain initial HMM $\theta^0 = (\theta_x^0, \theta_s^0)$ for iteration $k = 1$
2. Iterate until convergence (based on Viterbi score $V^*$):
    a) (E) Obtain optimal state sequence $q_0^{T*}$ and $V^*$ from Viterbi($x_1^T, \theta^{k-1}$)
    b) Convert state sequence $q_0^{T*}$ to corresponding symbol sequence $s_1^{T*}$
    c) (M) Infer PST from $s_1^{T*}$ using SECT
    d) Convert PST to PSA, which becomes new underlying MC, $\theta_s^k$
    e) (M) Obtain ML estimate $\theta_x^k$ of symbol pdfs, based on $x_1^T$ and $s_1^{T*}$
    f) Update HMM to $\theta^k = (\theta_x^k, \theta_s^k)$, and increment iteration $k$
```

Figure 5.1: The hidden SECT algorithm.

5.4 Training schedules

The basic hidden SECT algorithm in Figure 5.1 expands the PST as far as the training data will allow it on each iteration. This may result in an HMM with a large number of states at the start of the next iteration, which complicates the estimation of the optimal state sequence and increases the chances of getting stuck at a poor local optimum. To aggravate the problem, the symbol pdfs do not start at their optimal positions in observation space. Informally stated, the symbol pdfs should settle down first before higher-order contexts can be reliably estimated from them.

A useful remedy for complex optimisation problems of this kind is to optimise a simpler version of the objective function first. The complexity of the objective function is then gradually increased to its full level as the optimisation progresses. This steers the solution towards optima with broader basins of attraction, which tend to be more robust choices. This form of complexity control is reminiscent of deterministic annealing [52], and is one of the main virtues of the FIT algorithm [39]. It is also proposed for PST learning in [20].

The hidden SECT algorithm can also employ this trick, by limiting the depth to which the SECT algorithm may expand the PST. The implementation of this depth limit, $L_{\max}$, is described in Section 4.3.2. Note that the PST is not forced to attain a depth of L_{max}—it merely serves as the maximum possible order of the HMM. The depth limit can be gradually increased as training progresses, to allow more complex HMMs in a controlled fashion. The procedure by which the limit is increased is referred to as a *training schedule*, in analogy to the annealing schedule or cooling-off schedule of annealing methods [52].

Three training schedules are considered for the hidden SECT algorithm:

- Unlimited: The PST can expand without limit at all times, and training stops when the EM algorithm converges.
- Incremental: The maximum allowed depth of the PST at iteration k is $L_{\text{max}} = k$, and training stops when the EM algorithm converges.
- Rounds: The training process is divided into *rounds*, where each round is a full run of the EM algorithm. The round stops when the EM algorithm converges, after which the PST depth limit is increased and training continues with the next round. The depth limit for round k is $L_{\text{max}} = k$, and training stops after a fixed number of rounds, or when the Viterbi score at the end of each round does not change appreciably. This is similar to the approach of FIT [39].

5.5 Computational complexity

Consider an HMM with M symbols, N states and $G \leq MN$ non-zero transition probabilities. The typical computational complexity of the SECT algorithm is $O(T(\log T)^2)$ for a data set size of T, while the Viterbi algorithm has a complexity of $O(GT)$. It would appear that the hidden SECT algorithm is much slower than standard Viterbi re-estimation for large data sets. This is not the case, however, as the $(\log T)^2$ term in SECT's complexity can be easily surpassed by the corresponding G term of Viterbi, even for medium-sized HMMs. This makes the computational complexity of hidden SECT comparable to that of Viterbi re-estimation, as long as the HMM is large enough.

The complexity of SECT is also independent of the observation space dimension D, which has a major impact in typical HMM applications such as speech recognition. Consider an HMM with K-component full-covariance Gaussian mixture models as symbol pdfs. The symbol pdf evaluations in Viterbi would require $O(MKD^2T)$ calculations in this case, typically making it more expensive than the main dynamic programming complexity of $O(GT)$.

A more relevant source of computational complexity in hidden SECT is the large sizes of higher-order HMMs. Mixed-order HMMs are more compact than fixed-order models, but typically still have hundreds or thousands of states. This affects the performance of the Viterbi algorithm in the E step of hidden SECT, which requires more memory and computation time.

5.6 Summary

The hidden SECT algorithm considers an HMM to be a Markov chain coupled with a set of unique symbol pdfs. The Markov chain describes sequences of symbols, while the symbol pdfs map these symbols into observations. Instead of inferring the HMM topology on the state level, hidden SECT does it in terms of symbols, which it considers to be more basic than states. The topology of the underlying Markov chain (and hence the topology of the HMM) is inferred from the optimal symbol sequence provided by the Viterbi algorithm, using the SECT algorithm. This topology inference step can be slotted into the standard Viterbi re-estimation procedure to train the complete HMM. Hidden SECT is guaranteed to converge, as it is a winner-take-all MAP variant of the EM algorithm.

As hidden SECT is an iterative algorithm, it is useful to impose various training schedules on it. The order can be expanded incrementally during training in various ways. This controls the complexity of the cost function to be optimised and typically allows the optimisation process to reach better optima.

The computational complexity of hidden SECT is dominated in practice by the E step of Viterbi re-estimation, which has a complexity of $O(GT)$, where G is the number of links in the model with non-zero transition probabilities, and T is the size of the training set. It can be reduced by limiting the order to which the HMM is expanded.

Chapter 6

Experiments and Results

The experiments in this study are divided into three main sets. The first set examines the inference of prediction suffix trees from synthetic data, using the smallest encoded context tree (SECT) algorithm. The second set repeats this framework, by inferring hidden Markov models from synthetic data, using the hidden SECT algorithm. The final experiment compares the performance of mixed-order and fixed-order HMMs on an automatic language recognition task.

6.1 Synthetic experiments with SECT

The experiments in this section verify the performance and properties of the SECT algorithm on synthetic data. The estimated model can be directly compared with the known true model in this case. Each experiment starts off by creating a true prediction suffix tree from scratch. This PST is converted into an equivalent Markov chain (PSA), which efficiently generates a (random) symbol sequence of length T. The generated symbol sequence can optionally be corrupted, by randomly relabelling some of the symbols. The SECT algorithm then infers a PST from this training sequence. This model is assumed to be ergodic, as it can be reliably inferred from a single sequence. SECT can also infer non-ergodic models, but this requires multiple training sequences, which introduces an extra level of complexity. The outcome of the experiment is a comparison of the inferred PST with the true PST on various criteria. In order to form an idea of the variance of the results, each experiment is repeated twenty times.

6.1.1 PST setup

The true PST is characterised by three parameters: alphabet size M, order O, and perplexity P. The model can have a fixed order of one, two or three, or a mixed order with first-order and second-order contexts, indicated by $O = 1.5$. These options are illustrated in Figure 6.1, for a binary alphabet.

Each PST potentially has a large number of transition probabilities as parameters. One way to control this complexity is to randomise the parameter values while maintaining some properties of the model. This typically requires extra experimental trials to offset the increased variance of the results. We focus on simple constrained models instead. While these models might not be representative of ones used in actual applications, they allow fine-grained control over their properties, which can then be studied in isolation.

To this end, all leaf nodes of the PST are coding nodes, with next-symbol distributions constrained to have the same entropy, H, specified by the *perplexity* $P = 2^H$. To simplify the setup even further, each leaf distribution contains a single large probability, $p_{\max}$, while the rest of its probabilities are equal. The relationship between $p_{\max}$ and P is given by

$$p_{\max} \log_2 \frac{1}{p_{\max}} + (1 - p_{\max}) \log_2 \frac{M-1}{1-p_{\max}} = \log_2 P. \tag{6.1.1}$$

An iterative binary search for $p_{\max}$ in terms of P is effective, since the relationship in (6.1.1) is one-to-one and monotonic. The remaining internal nodes of the PST are non-coding, and share a uniform default next-symbol distribution which typically only plays a role during sequence start-up.

The only degree of freedom left in each next-symbol distribution is the identity of the most likely symbol. A simple assignment rule is chosen that ensures that each model exhibits its intended order. For the first-order, second-order and third-order models, the next-symbol distribution of each leaf node assigns the maximum probability $p_{\max}$ to the first symbol of the node's context string. The mixed-order PST starts out like the first-order model, but one of its first-order contexts is expanded to second order. The resulting second-order contexts all have the same next-symbol distribution as their first-order parent node, except one, which assigns the maximum probability to a different symbol.

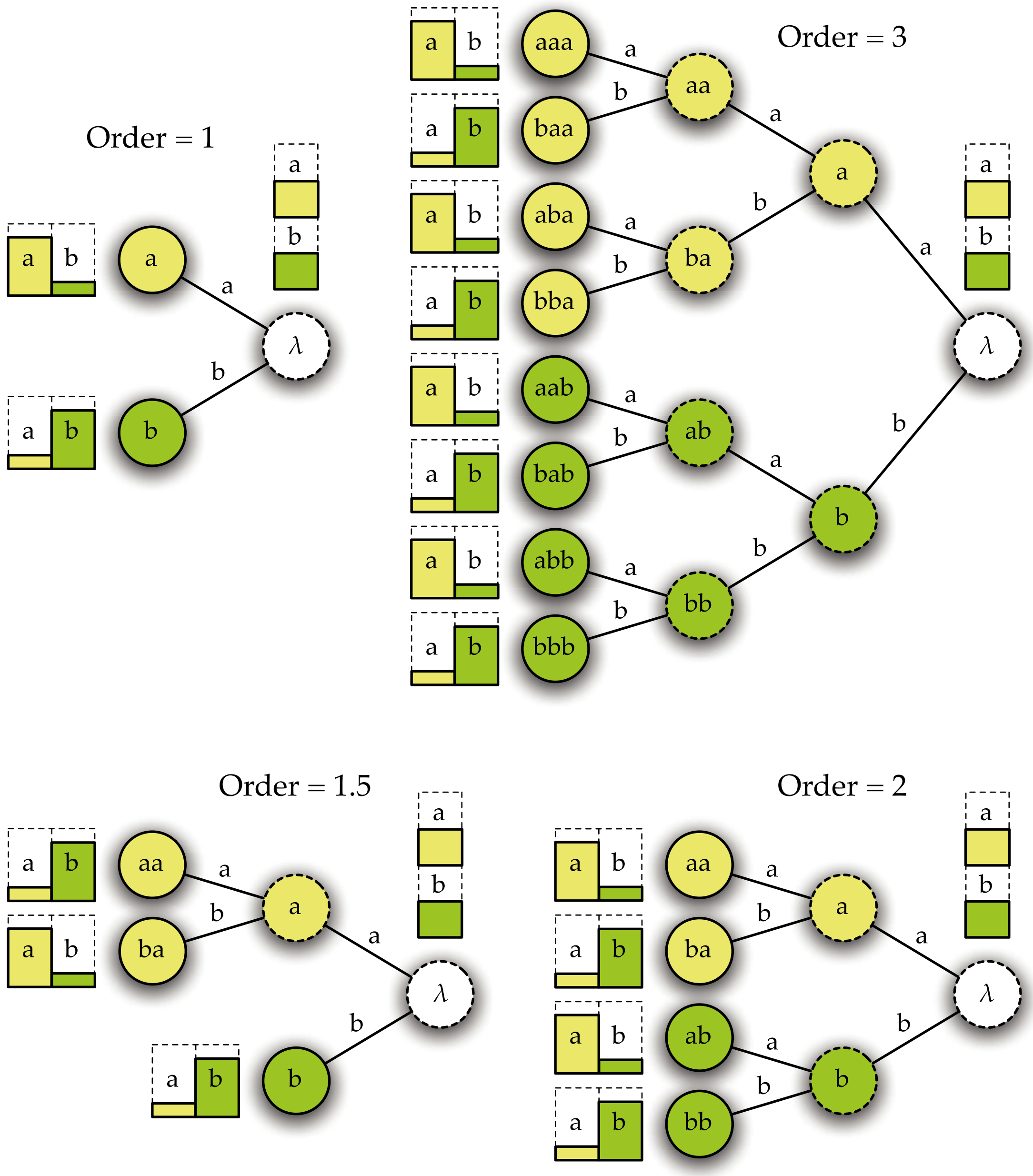


Figure 6.1: An example of the PST structures used in the synthetic experiments, illustrated for a binary ($M = 2$) alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and a perplexity of $P = 1.65$. The miniature bar graph next to each solid node indicates the next-symbol distribution of that context, while the dashed nodes are non-coding. The bar graph above each root context is the uniform default distribution. The yellow bars and nodes are associated with symbol a, while the green ones are associated with symbol b. The first-order, second-order and third-order PSTs are fixed-order models, while the PST with $O = 1.5$ is a mixed-order model.

Table 6.1: Example symbol sequences generated by medium-perplexity PSTs of various orders on the binary alphabet $\Sigma = \{\texttt{.},\texttt{0}\}$. The alphabet is chosen to make the observed symbol patterns stand out more.

PST order	Example symbol sequence
1	`...00....000000.....0000..0......00000.000...00...000`
1.5	`...00.......00...00........000.......0......00....00.`
2	`.....0.0.0.0000000.0.0.0.0....0.....0.0.0.0.0...00000`
3	`.00.00.00.000000000..0..0..0..0.00.00..0..0..0....0..`

Table 6.1 illustrates the typical symbol sequences associated with each PST order. The first-order PST is an ergodic Markov model with large self-loops on each state, which is associated with sequences containing long runs of each symbol. The next symbol in the sequence is most likely the same as the symbol directly preceding it. Similarly, the second-order PST predicts the next symbol in the sequence as the symbol two positions in the past, which leads to runs of symbol pairs. The third-order PST predicts the next symbol based on the symbol three positions in the past, which leads to runs of symbol triples. The sequences associated with the mixed-order PST are similar to those of the first-order model, except that one of the symbols tend to have runs of length two before switching to another symbol.

When the PST is constructed in this way, each of its parameters plays a distinct role. The alphabet size, M, is an important overall setting. The order, O, determines the memory length of the model, as well as its topology. Together, these two parameters dictate the model size. The perplexity, P, determines the predictability of the symbol sequences associated with the PST, which influences the learnability of the PST. A perplexity close to its minimum value of one is associated with largely deterministic and highly structured symbol sequences, while a perplexity close to its maximum value of M is associated with highly random, largely unstructured sequences.

6.1.2 Comparison of PSTs

The simplest comparison between the true and inferred PSTs is based on the number of nodes in the PST, which coincides with the number of states N in its equivalent Markov chain. A fixed-order true PST of order O has $N = \sum_{i=0}^{O} M^i$ states, while the mixed-order PST with $O = 1.5$ has $N = 2M + 1$ states (consider for example the models in Figure 6.1). These counts include both transient and persistent states. Each true PST also has $G = NM$ non-zero transition probabilities, or links. While two PSTs are not guaranteed to have the same structure if they have the same number of nodes, the state count is nevertheless a useful indicator which highlights many properties of the inference process.

The structure of the inferred PST can also be directly verified, as the true PST is available. The inferred PST is considered to have the *correct* structure if it has the same context set as the true PST. This is checked by recursing down both trees and comparing the labels of the children of each node, an operation which would be much more difficult in the PSA domain.

While its structure may be correct, the inferred PST can still differ significantly from the true PST if their corresponding next-symbol distributions differ. The two PSTs are only equivalent if they assign the same probabilities to the same symbol strings. A distance measure that takes this into account is the Kullback-Leibler divergence of (2.1.3). In the case of two Markov models $\mathcal{M}_1$ and $\mathcal{M}_2$, the divergence is measured between the respective probability distributions $P_1(s_1^t)$ and $P_2(s_1^t)$ induced by the models on symbol sequences of length t. In order to obtain an invariant quantity independent of t, we turn to the *Kullback-Leibler divergence rate*

$$\mathrm{D}(\mathcal{M}_1 \| \mathcal{M}_2) = \lim_{t\to\infty} \frac{1}{t} \mathrm{D}\big[P_1(s_1^t) \big\| P_2(s_1^t)\big] \,,$$

which is the average divergence per symbol. Since the Markov models in this experiment are homogeneous and ergodic, the divergence rate is easily calculated as the divergence between corresponding next-symbol distributions, averaged with respect to the long run distribution of the first model [53, 159].

There are a few practical issues with this calculation. First, the context sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$ of the two models should both be extended to $\mathcal{Q}' = \mathcal{Q}_1 \cup \mathcal{Q}_2$, to ensure that each state in the one model has a corresponding state in the other. The extra contexts added to each model use the next-symbol distributions of their parent nodes, which effectively makes them non-coding nodes.

Second, in order to obtain the long run distribution of the first PST, it is much easier to convert the PST into an equivalent PSA first. The long run distribution is then found by applying the *power method* [160] to the transition matrix $\boldsymbol{A}$ of the PSA. This is an iterative procedure that finds the largest eigenvalue and the corresponding dominant eigenvector of a square matrix. In the case of a transition matrix of an ergodic Markov model, the maximum eigenvalue is one and the (normalised) dominant eigenvector is the long run distribution. The power method is well suited to large sparse matrices, and forms a prominent part of Google's PageRank algorithm [161].

The last issue arises when the inferred PST assigns zero probability to transitions that may occur according to the true PST, which results in infinite divergence values. This is a common situation when the training set is small, and another manifestation of the zero frequency problem [149]. The standard solution is to smooth the next-symbol probabilities of the inferred PST, so that no probability is exactly zero [23, 151, 20]. We use Laplace smoothing (see Section 4.3.3), which adds one to each next-symbol count of the inferred PST before calculating the divergence.

The formula for the divergence between two PSTs is therefore given by

$$\mathrm{D}(\mathcal{M}_1 \| \mathcal{M}_2) = \sum_{q \in \mathcal{Q}'} \pi_1(q)\, \mathrm{D}[\eta_1(q,s) \| \eta_2(q,s)]\,,$$

where $\mathcal{Q}'$ is the combined state space of the two models, $\pi_1(q)$ is the long run probability of state q in the first PST, $\eta_1(q,s)$ is the (unsmoothed) next-symbol distribution of state q in the first PST, and $\eta_2(q,s)$ is the corresponding smoothed next-symbol distribution of state q in the second PST.

6.1.3 PST base experiment

The first PST experiment verifies that the SECT algorithm is indeed learning the correct structure of the model, and also evaluates comparison criteria to be used in the rest of the experiments.

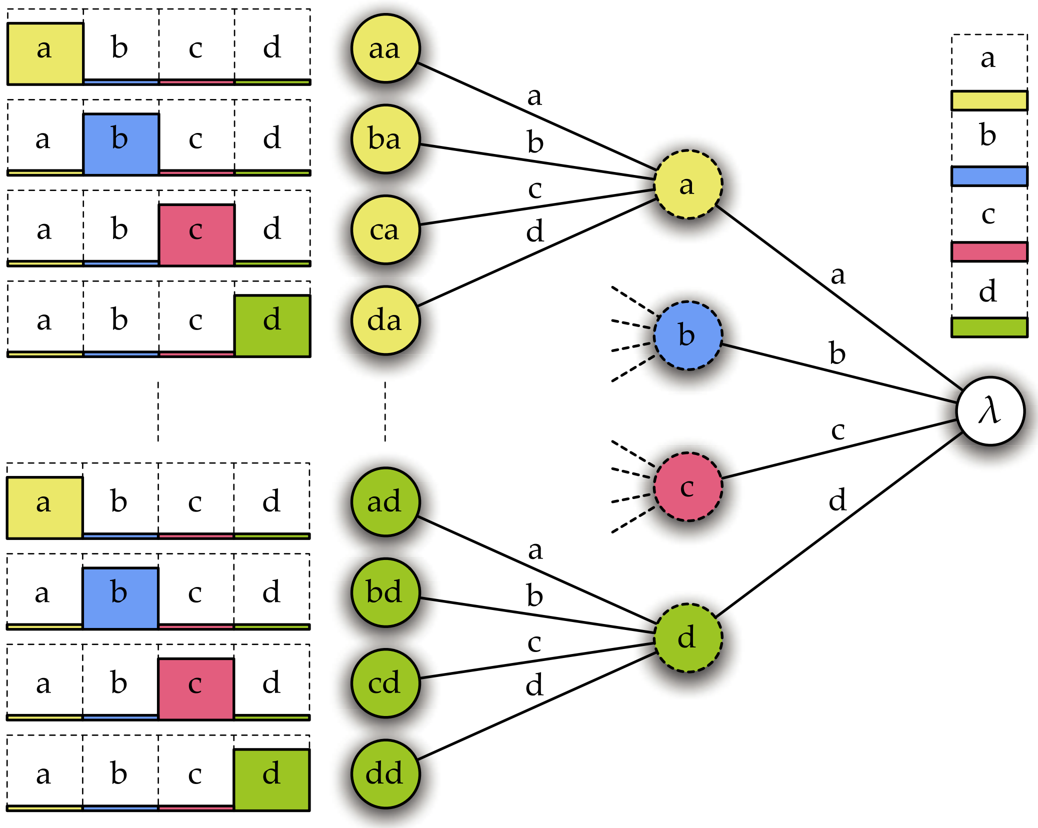


Figure 6.2: The true PST used in the PST base experiment, with parameters $M = 4$, $O = 2$ and $P = 2$. The miniature bar graph next to each solid node is the next-symbol distribution associated with that coding context, while the dashed nodes are non-coding. The eight second-order contexts ending with symbols `b` and `c` are omitted to save space.

The true PST is selected to be a second-order model with $M = 4$ symbols and a perplexity of $P = 2$, which is shown in Figure 6.2. The model has 21 states and 84 transition links, and the perplexity implies a maximum probability of $p_{\max} = 0.81071$. The training sequence length T is varied from 100 to 1 000 000. The length T is incremented by a hundred until it reaches 1000, after which it is incremented by a thousand until it reaches 10 000, and so forth. Twenty trials are run for each setting of T.

Figure 6.3a shows the number of inferred PSTs that have the correct structure for each setting of T, expressed as a percentage. The graph can be divided into three regions. If the sequence length is 500 or less, the data is insufficient to infer

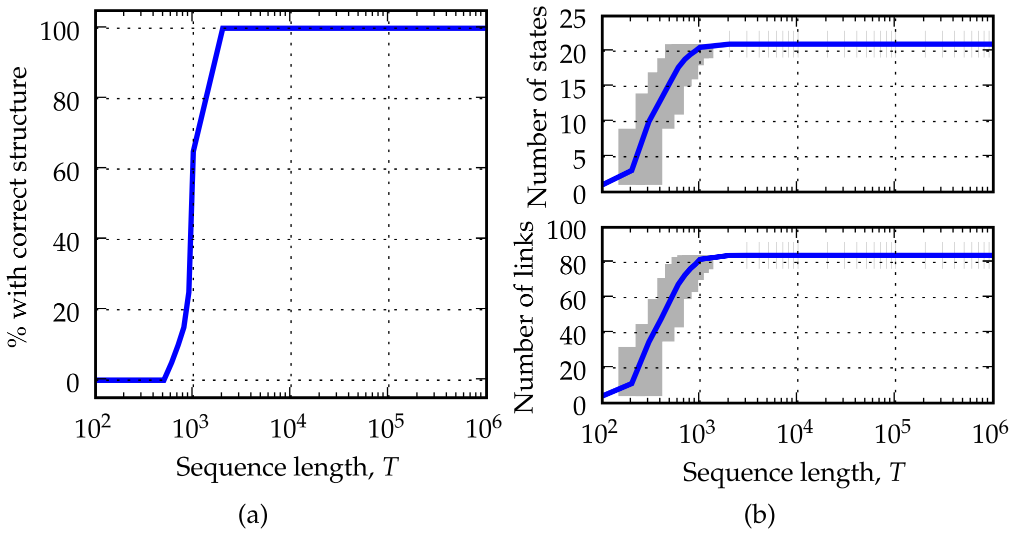


Figure 6.3: Results of the PST base experiment. (a) The percentage of the twenty PSTs inferred for each setting of T that have the correct structure. (b) The average number of states and links in the inferred PSTs. The grey bars indicate the range of values (from minimum to maximum) observed in the twenty trials at each setting of T. The true model has 21 states and 84 links.

the full structure of the PST. For sequence lengths between 500 and 2000, the SECT algorithm sometimes finds the correct structure. Once the sequence length is 2000 or more, SECT reliably finds the correct PST structure. This structural correctness check is therefore useful to determine the minimum training set size for the reliable learning of a specified PST.

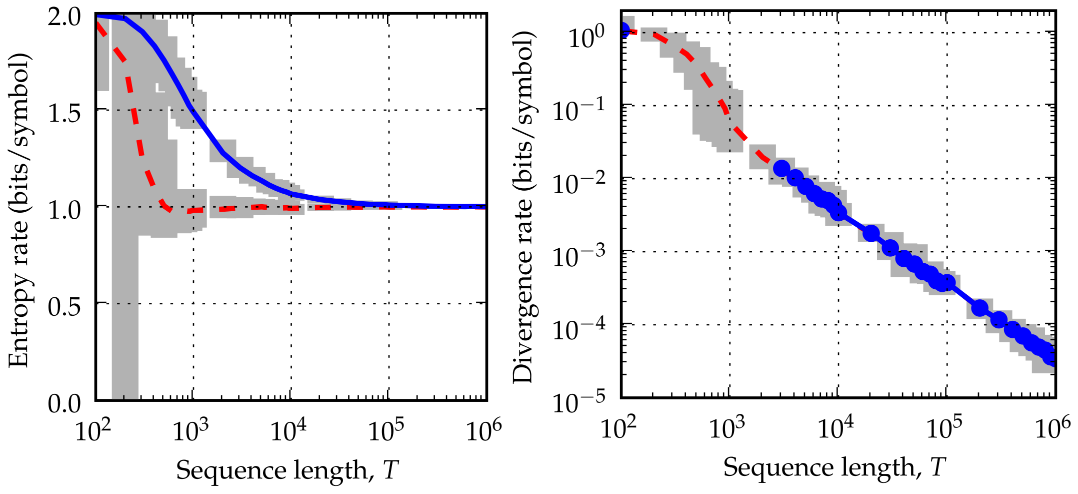
The structural correctness is corroborated by the number of states and links in the inferred PST, shown in Figure 6.3b. The plots include the average, minimum and maximum values observed over twenty trials. They show that the SECT algorithm already finds partial PST structures at data lengths down to 200. Since the number of links show a very similar pattern to the number of states, it is not considered in the rest of this study. The usefulness of the state count is that it reveals PST learning at an earlier stage than the correctness measure, and is more widely applicable.

Recall that the SECT score is the total code length of the training data set obtained during SECT encoding, which includes the model overhead of the PST. On the other hand, the entropy of the PST only represents the data portion of the code length. We calculate an *entropy rate* or average entropy per symbol for the PST, based on the same principles that apply to the divergence rate [53]. That is, the entropy rate is obtained as the average entropy of each next-symbol distribution, with the average taken with respect to the long run distribution of the PST. The true PST has an entropy rate of $\log_2 P$ regardless of its long run distribution, since all next-symbol distributions are constrained to have the same entropy. This is the justification for choosing the next-symbol distributions in this way. The SECT score is normalised with respect to the training sequence length, to allow it to be compared with the entropy rate. Both quantities therefore have a unit of bits per symbol.

Figure 6.4a shows that both the SECT score and entropy rate start at the raw code length of $\log_2 M$ when the data set is too small to infer the correct PST. As the data size increases, the entropy rate quickly drops to the true rate of $\log_2 P$. The difference between the SECT score and the entropy rate is the model overhead. It is very small when no model can be found for small data sets, increases for medium-sized data sets that support inference of the PST structure, and finally converges to zero for large data sets.

The divergence rate between the true and inferred PSTs is shown in Figure 6.4b, for various settings of T. The plot includes the average, minimum and maximum values observed over twenty trials, similar to many of the other plots in this chapter. Two divergence rates are shown: the one is based on smoothed next-symbol distributions for the inferred PST, and the other uses unsmoothed distributions. For sequence lengths below 3000, some trials produce an unsmoothed divergence of infinity, which causes the average divergence rate to blow up. In contrast, the smoothed divergence rate is well-defined for all sequence lengths, and also closely approximates the unsmoothed version. For this reason, all divergence rates will hereinafter be smoothed.

For small data sets, the divergence rate is effectively calculated between the true PST and a zeroth-order model with a uniform symbol distribution (the "raw" model). The divergence rate remains at this level, until there is sufficient data to uncover the PST structure. The graph displays a knee at this point, after which the divergence rate converges to zero at a rate of the order $1/T$. In this regime, the model structure is correct and only the estimates of the transition probabilities remain to be improved.



(a) (b)

Figure 6.4: Results of the PST base experiment. The grey bars indicate the range of values observed at each setting of T. (a) The SECT score (solid blue line) represents the total code length (model + data), while the PST entropy rate (dashed red line) represents the average data code length. Both quantities are normalised by the sequence length to have a unit of bits per symbol. (b) The divergence rate between the true and inferred PSTs, averaged over twenty trials. The solid blue line calculates the divergence based on unsmoothed next-symbol distributions, while the dashed red line results from smoothing.

Based on this preliminary experiment, the rest of the experiments will use the structural correctness check, the state count and the divergence rate as the main criteria for comparing PSTs.

6.1.4 PST perplexity experiment

The next experiment examines the effect of perplexity on the learning of PSTs. The true PST is still a second-order model with $M = 4$ symbols, but its perplexity P is now varied between its minimum value of one and its maximum value of $M = 4$. Each perplexity value P is converted into a corresponding maximum next-symbol probability p_{max}, which is used in the construction of the PST. The relationship between these two quantities is shown in Figure 6.5.

The percentage of inferred PSTs with the correct structure is shown in Figure 6.6, for various values of the perplexity P and different settings of T. The cor-

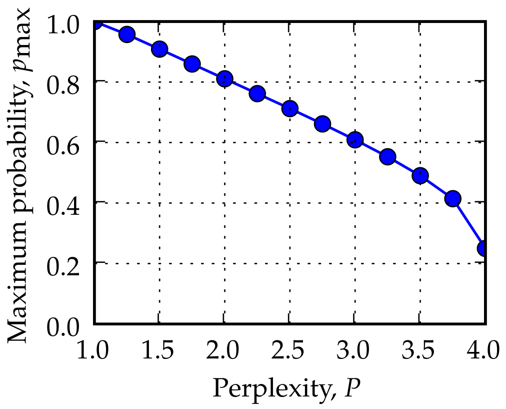


Figure 6.5: The maximum next-symbol probability, $p_{\max}$, as a function of perplexity, P, for an alphabet size of $M = 4$.

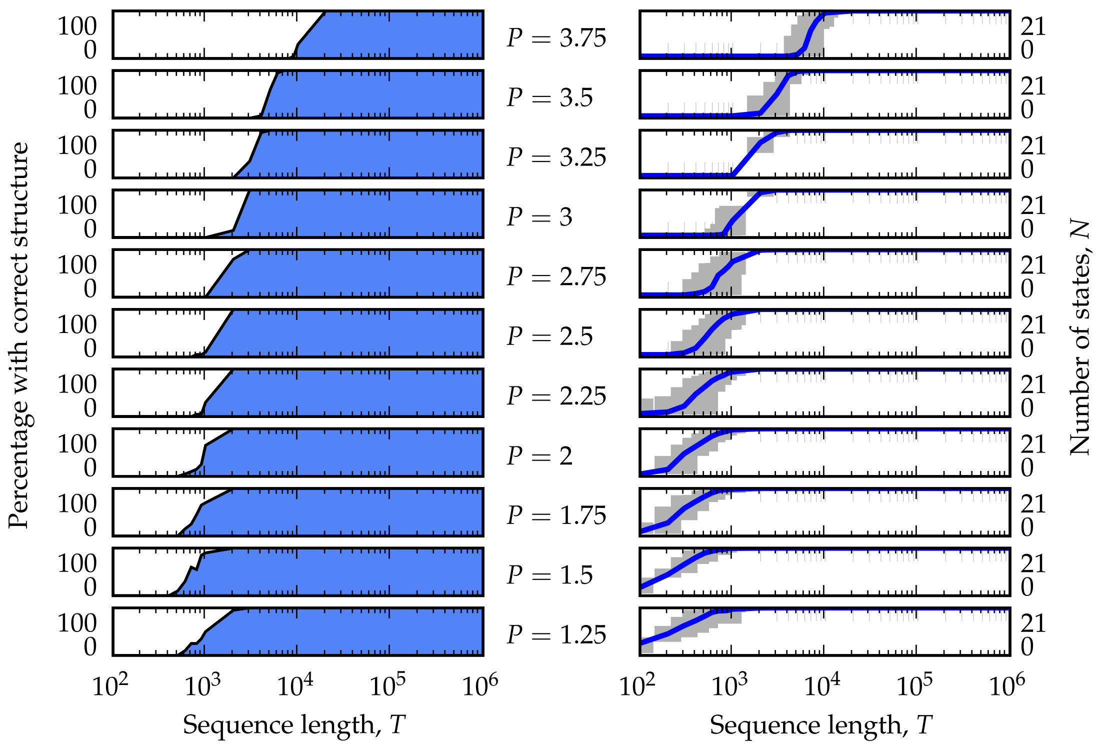


Figure 6.6: The left-hand graphs show the percentage of inferred PSTs with the correct structure, for various values of the perplexity P. The right-hand graphs show the corresponding average number of states in each inferred PST, where the grey bars indicate the observed range of values.

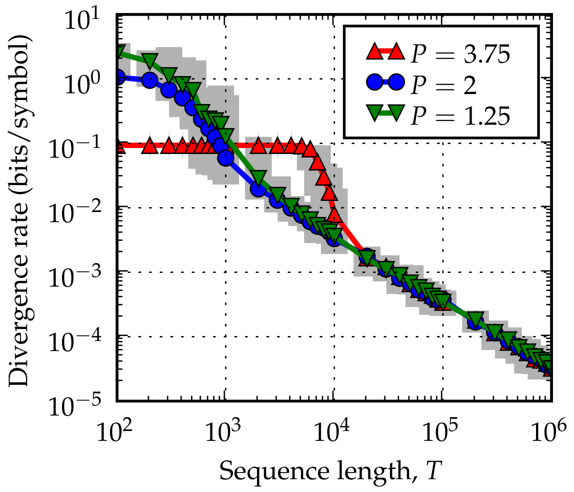


Figure 6.7: The divergence rate between the true and inferred PSTs for various values of the perplexity P, averaged over twenty trials. The grey bars indicate the range of values observed in each set of trials.

responding number of states in the inferred PST is shown alongside, providing a similar picture of how learning progresses with increasing training set size. As expected, models with higher perplexity require more data to learn, because they are less distinguishable from the default raw model, which is maximally random. A more surprising result is that models with very low perplexities also require more data to learn. This is because the sequences associated with these models typically contain long runs of the dominant contexts of the model, and a short sequence may not contain all these contexts if the runs are long enough.

Figure 6.7 illustrates the effect of perplexity on the divergence rate between the true and inferred PSTs. The plot contrasts three models, with a high ($P = 3.75$), medium ($P = 2$), and low ($P = 1.25$) perplexity, respectively. The high-perplexity model starts off with the lowest divergence rate, as it is the closest to the default raw model. It requires more data than the other models to uncover its structure, however. While it has a tenth of the divergence of the medium-perplexity model on small data sets, it requires ten times as much data to learn its structure. The low-perplexity model starts with the highest divergence rate, but also requires more data than the medium-perplexity model. The SECT algorithm successfully learns the structure of all three models on sequences with more than 30 000 symbols, and their divergence rates become very similar.

The following experiments will focus on the model order and alphabet size, and it is desired to fix the perplexity to a sensible value. Models with medium perplexity require less data to learn, and the perplexity will henceforth be fixed at $P = M/2$. This has the added advantage that the divergence rate between the true PST and the default raw model is fixed to one bit per symbol, which makes it easier to compare models with different alphabet sizes. One disadvantage of this choice of P is that models with a binary alphabet is excluded from the experiments, as these models would end up with the degenerate perplexity of one. Choosing a different perplexity for models with $M = 2$ does not help, as this prevents them from being directly comparable to the other models.

6.1.5 PST alphabet size experiment

With the perplexity fixed at half the alphabet size, both the order and alphabet size are varied in the next experiment, to determine their effect on PST inference.

Figure 6.8 shows the percentage of inferred PSTs with the correct structure, and the number of states in the inferred PSTs, for various orders and alphabet sizes. Because they provide very similar information, these measures are hereinafter combined on the same plot, by scaling the structural correctness percentage so that 100% coincides with the correct number of states. This makes it easier to see when the SECT algorithm has produced the correct model size.

Both the order and alphabet size greatly influence the amount of training data required to learn the PST structure successfully. The data requirements increase with increasing order and alphabet size. This suggests that the required training set size really depends on the number of parameters in the PST, which is the number of transition probabilities, G.

The mixed-order PST inference proceeds in phases, whereby the first-order structure is discovered before the second-order structure, as the training set size increases. The mixed-order model seems to require an excessive amount of training data—even more than the full second-order model. This is an artifact of the way in which the correct structure is defined. The mixed-order PST in this experiment effectively only has a single second-order context, chosen to differ significantly from its first-order parent context. This context will be referred to as the *major* second-order context, an example of which is context aa in the mixed-

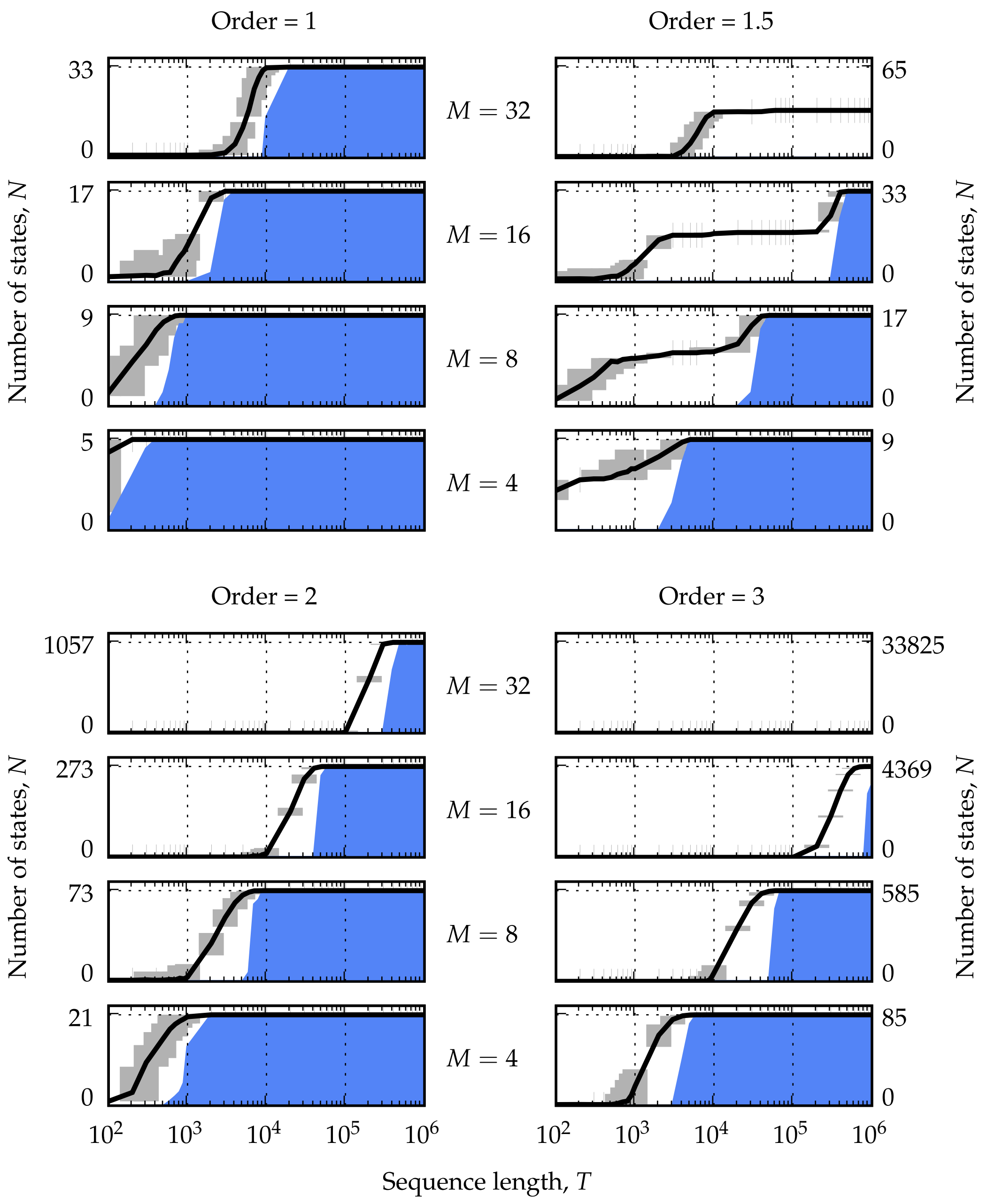
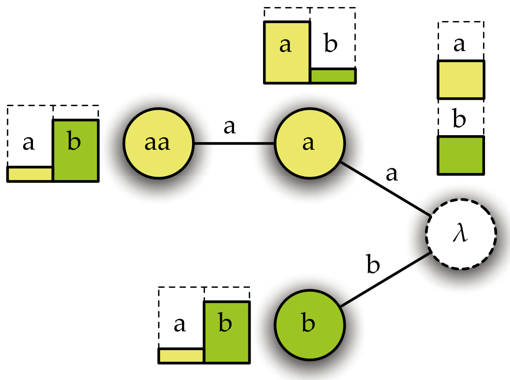
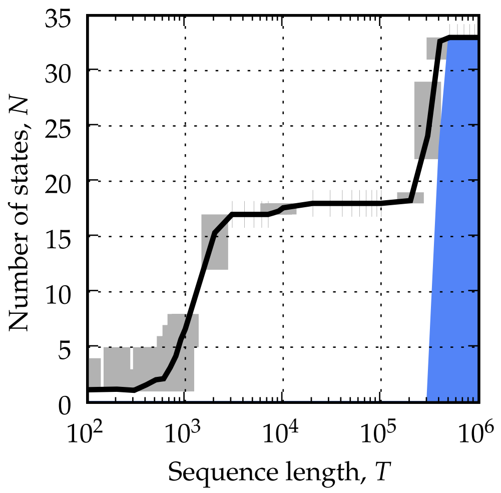


Figure 6.8: Results of the PST alphabet size experiment. The black line in each graph is the number of states in each inferred PST for various values of the true order O and alphabet size M, averaged over twenty trials. The grey bars indicate the minimum and maximum number observed in each set of trials. The height of the blue region represents the percentage of inferred PSTs with the correct structure, scaled so that 100% coincides with the correct number of states, which is indicated on the axis.



(a)



(b)

Figure 6.9: A closer look at mixed-order PST inference. (a) The most compact PST that is equivalent to the mixed-order PST in Figure 6.1. (b) The result of the inference of a PST with $O = 1.5$, $M = 16$ and $P = 8$. The black line is the average number of states in the PSTs inferred in twenty trials, and the height of the blue region represents the percentage of these PSTs that have the correct structure.

order PST in Figure 6.1. Its second-order siblings share the same next-symbol distribution, which happens to be the original distribution of the parent node. These *minor* second-order contexts can be collapsed back into the parent node, resulting in a more compact but equivalent PST, of which an example can be seen in Figure 6.9a.

The SECT algorithm cannot learn such compact PSTs, though, because the next-symbol distribution assigned to a node is forced to coincide with its distribution observed in the training data. The parent of the second-order contexts will have an actual next-symbol distribution that is a mixture of those of its children, and therefore differs from all of them. Given enough data, SECT will distinguish all these second-order contexts from their parent, and add them to the PST structure instead.

The major second-order context will typically be discovered in much smaller data sets than its siblings. This is apparent in Figure 6.9b, which shows the number of states and structural correctness of an inferred mixed-order PST with $M = 16$ symbols. The full mixed-order structure of 33 states is only reliably

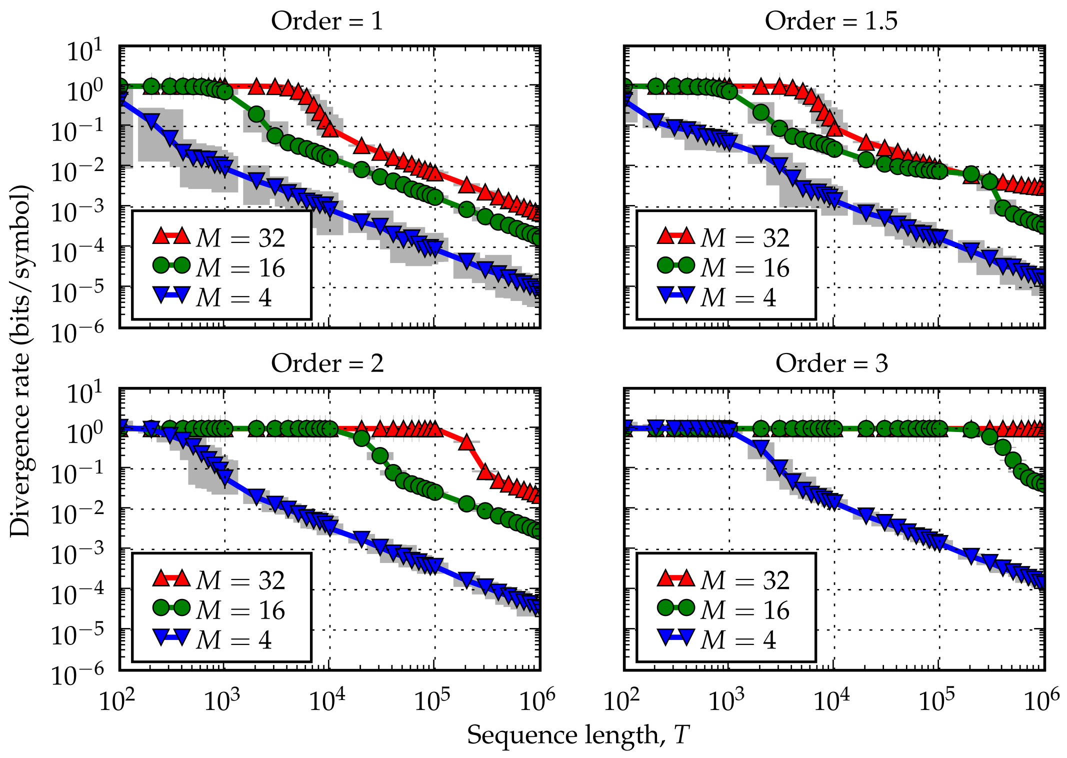


Figure 6.10: The divergence rate between the true and inferred PSTs for various values of the true order O and alphabet size M, averaged over twenty trials. The grey bars indicate the range of values observed in each set of trials.

inferred from sequences containing 500 000 symbols or more, while the major second-order context is already discovered in sequences of 20 000 symbols, leading to a model with 18 states. While this compact model is not considered to have the correct structure, it nevertheless represents the essence of the mixed-order model, and has data set requirements between those of the first-order and second-order models.

The divergence rate between the true and inferred PSTs is illustrated in Figure 6.10, for various orders and alphabet sizes. All divergence rates start at one bit per symbol, due to the choice of perplexity. The graphs confirm that the data requirements for learning increase with order and alphabet size. The mixed-order graphs have a knee in the same position as their corresponding first-order graphs, and display an additional knee (sometimes merged with the first) that represents learning of the second-order structure.

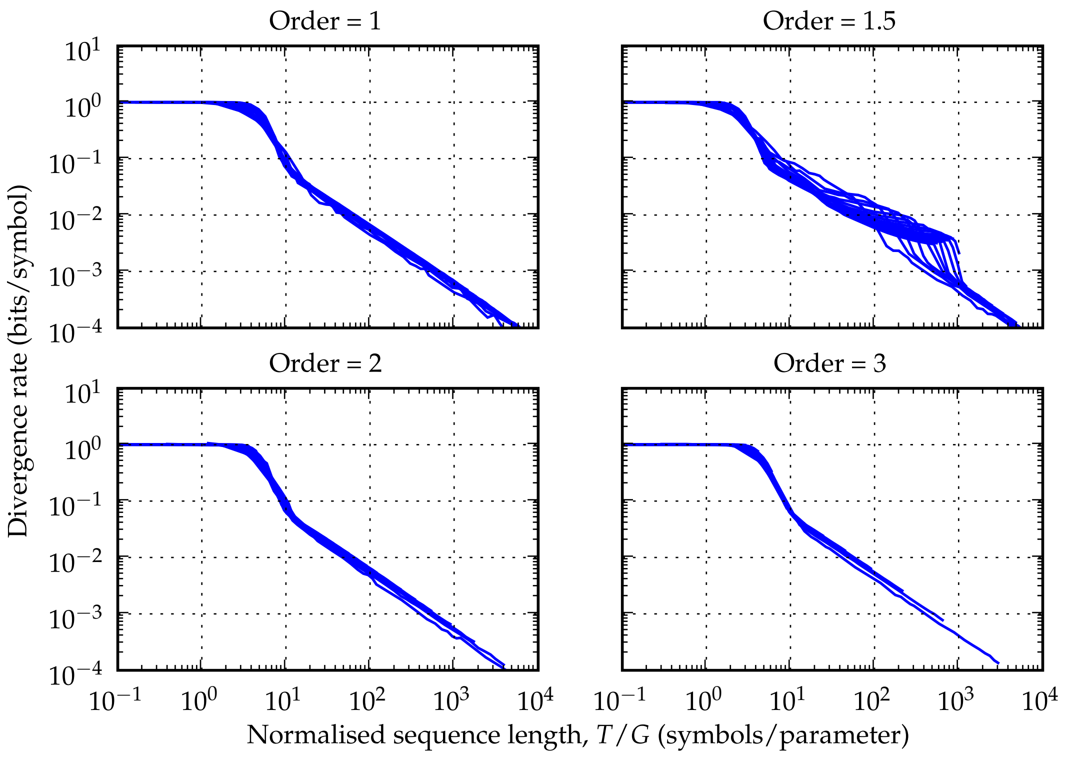


Figure 6.11: The divergence rate between the true and inferred PSTs, for various values of the true order O, averaged over twenty trials. The difference from Figure 6.10 is that the divergence rate is plotted against a normalised sequence length T/G, which is the number of symbols per transition probability (or model parameter). Each plot is the superposition of the divergence rates for several values of the alphabet size M, which ranges from four to 32 in steps of two.

It is instructive to plot the divergence rate against a normalised sequence length, which is obtained by dividing T by the number of model parameters or transition probabilities, G. Figure 6.11 shows this for various orders, and for alphabet sizes ranging from four to 32 in steps of two. The fixed-order graphs are remarkably similar, and indicate that the fixed-order structure is successfully inferred once the normalised sequence length exceeds 10-20 symbols per parameter.[1] The mixed-order graphs do not line up as well, because the data requirements of the first-order and second-order contexts differ in their depen-

[1]This is coincidentally a general rule of thumb in the pattern recognition community, used to estimate an adequate data set size for the training of a model of given size.

dence on M. This makes the parameter count less useful as a tool to predict an adequate training set size. A more accurate measure has to take the different long run probabilities of each context into account.

Figure 6.12 shows the time taken to infer a PST with SECT, for various orders, alphabet sizes and training sequence lengths, as measured on a 2 GHz Intel Core Duo processor. The measurements do not include the time taken to convert the PST to a Markov chain, which is considerably less than the PST inference time. The clock resolution is 10 ms, which causes training times for sequence lengths $T < 10^4$ to be unreliable. The relationship between the observed training times and the sequence length T fits the expected computational complexity of $O(T(\log T)^2)$. Furthermore, the training time is effectively independent of the true order and alphabet size, because of the self-bounded nature of SECT. As the alphabet size increases, there are more possible symbol contexts of a given length, but also fewer observations of each context in a data set of fixed size T. In this case, the SECT algorithm will stop its exploration of the data set at shorter context lengths, which counters the increase in computational complexity because of the extra PST nodes. On the other hand, smaller alphabets allow SECT to explore longer contexts for the same data set size.

6.1.6 Corrupted symbol experiment

The last SECT experiment trains PSTs on noisy or corrupted symbol sequences, which serves as a precursor to the hidden SECT experiments. This is similar to a study by Angluin and Csűrös [162].

The true PST is selected to have an alphabet size of $M = 4$ symbols and a perplexity of $P = 2$. After the true PST generates a symbol sequence in an experimental trial, the sequence is corrupted. Each symbol in the sequence is independently corrupted with a probability of $1 - A$, where A is a new experimental parameter known as the *symbol accuracy*. A symbol is corrupted by replacing it by a different symbol selected with a uniform probability. The corruption hides the random symbol sequence behind another stochastic process, and the corrupted sequence is therefore properly modelled by a hidden Markov model. Nevertheless, it is instructive to model it by a PST, which is the purpose of this experiment.

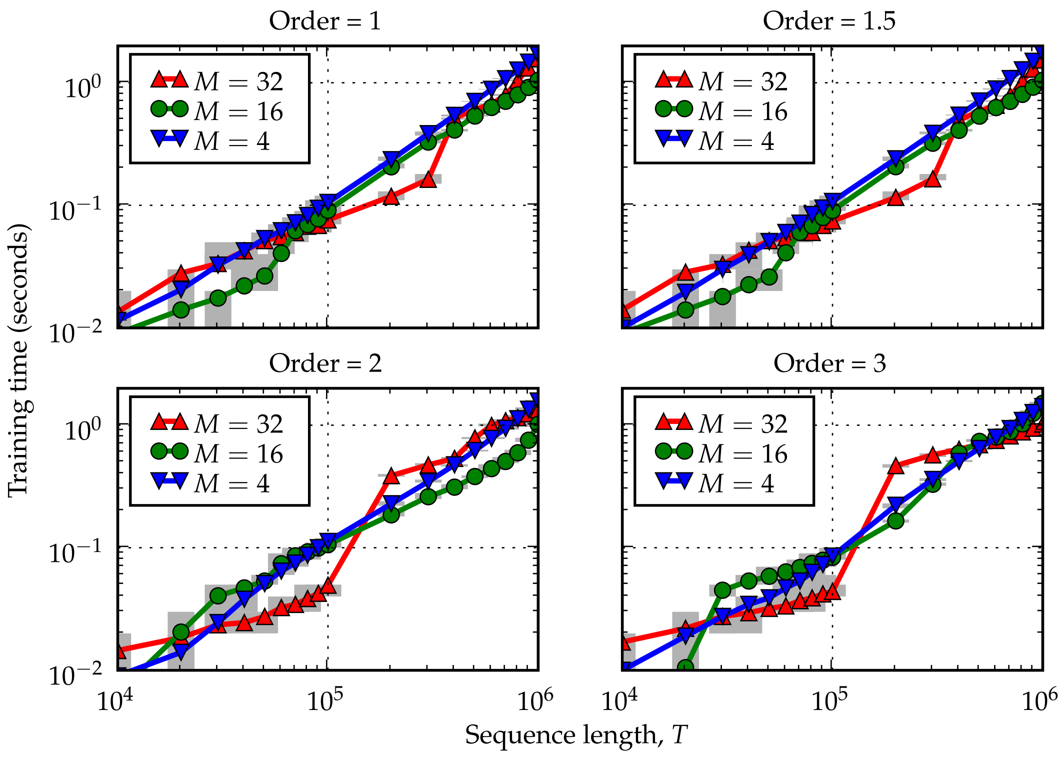


Figure 6.12: The time taken by SECT to infer a PST for various values of the true order O and alphabet size M, averaged over twenty trials. The grey bars indicate the range of values observed in each set of trials. The measurements were done on a 2 GHz Intel Core Duo processor.

Figure 6.13a shows the structural correctness and number of states of PSTs inferred from noisy sequences of various lengths, where the underlying true PST is first-order and the symbol accuracy is $A = 80\%$. The most important observation is that the number of states do not stabilise on the correct value of five as the training sequence length increases, but shows a step-wise increase with T. This is a sign that the PST is attempting to model the noise process as well as the underlying sequence statistics.

This behaviour can be explained by means of an example. Consider a first-order true PST with a binary alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$ and relatively low perplexity. This model might generate sequences of the form

$$\texttt{aaaaaabbbbbaaaaaaabbabbbbbbbaaabbb}\ldots$$

After corruption with a symbol accuracy of 80%, the sequence becomes

$$\texttt{aa\underline{b}aaabbb\underline{a}ba\underline{b}aa\underline{b}aabbabb\underline{a}bbb\underline{a}aaab\underline{a}b}\ldots,$$

where the corrupted symbols are underlined.

The corruption disturbs the sequence statistics. For example, the context `b` is usually followed by another `b` in the uncorrupted sequence. In the corrupted sequence, however, the context `ab` is frequently associated with a single corrupted `a` in a run of `a`'s, which increases the probability that it is followed by an `a`. Since the next-symbol distributions of `b` and `ab` differ, the PST will be expanded to second order, given enough data. This process repeats itself for higher orders. For instance, a context of `bab` is more likely to be associated with a single erroneous `a` in a run of `b`'s, which increases the probability that the next symbol is a `b`. The contexts `bab` and `ab` now have different next-symbol distributions, which allows the PST to be expanded up to third order, and so forth. This explains the stepwise increase in the number of PST states in Figure 6.13a, as the SECT algorithm finds contexts of increasing length with slightly different statistics.

Figure 6.13b shows the effect of a noisy symbol sequence on the divergence rate between the true and inferred PSTs. On uncorrupted sequences, the divergence rate decays to zero as the sequence length increases, which indicates that the inferred PST converges to the true PST. On corrupted sequences, however, the divergence rate initially decreases and then remains stuck at a non-zero level. The inferred PST cannot converge any further, as it contains elements of the noise model which are not found in the true PST.

Figure 6.14 shows the structural correctness and number of states of PSTs inferred from corrupted symbol sequences, for various sequence lengths, symbol accuracies and true model orders. The effect of symbol corruption is twofold. The structure of the inferred PST grows past its true size as the training data increases, and learning the PST requires progressively more data as the symbol accuracy decreases. To put the symbol accuracy values into perspective, consider that the worst symbol accuracy for an alphabet size of $M = 4$ is $A = 1/M = 25\%$, which represents total obliteration of the underlying sequence statistics. The SECT algorithm can still learn the PST structure at the relatively high noise levels of $A = 40\%$, although much more data is required to combat

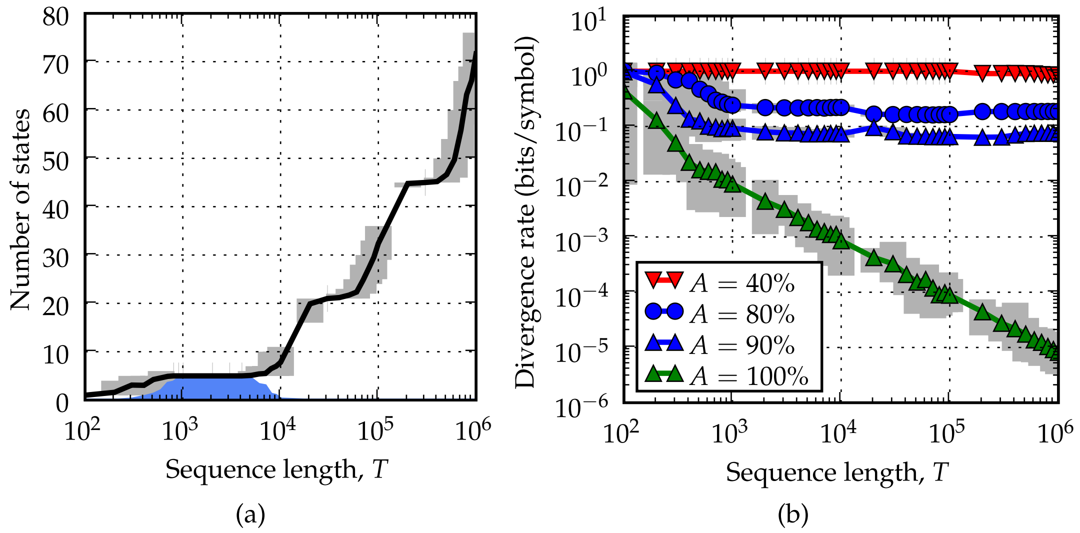


Figure 6.13: Results of the PST corrupted symbol experiment. For both these plots, the true PST that generated the uncorrupted sequences is first-order, with $M = 4$ and $P = 2$. As before, the grey bars indicate the range of values observed in each set of trials. (a) The black line is the number of states in the PSTs inferred from sequences with a symbol accuracy of $A = 80\%$, averaged over twenty trials. The height of the blue region represents the percentage of these PSTs that have the correct structure. (b) The divergence rate between the true and inferred PSTs for various values of the symbol accuracy A, averaged over twenty trials.

the noise. The mixed-order learning results should again be interpreted in light of the discussion in the previous section.

Angluin and Csűrös study the learning of PSTs from noisy symbol sequences in [162]. They modify the Learn-PSA algorithm [22] to include an explicit noise model, which changes the way in which the next-symbol probabilities are estimated from the noisy data. They suggest that a PST inferred from corrupted data will have comparable performance to a PST inferred from uncorrupted data if the corrupted training sequence is longer than the uncorrupted sequence by a factor of $(1+\theta)^{2(L+2)}$, where

$$\theta = \frac{\nu}{1 - \frac{M}{M-1}\nu},$$

and $\nu = 1 - A$ is the symbol error rate, M is the alphabet size and L is the maximum depth of the PST during Learn-PSA training.

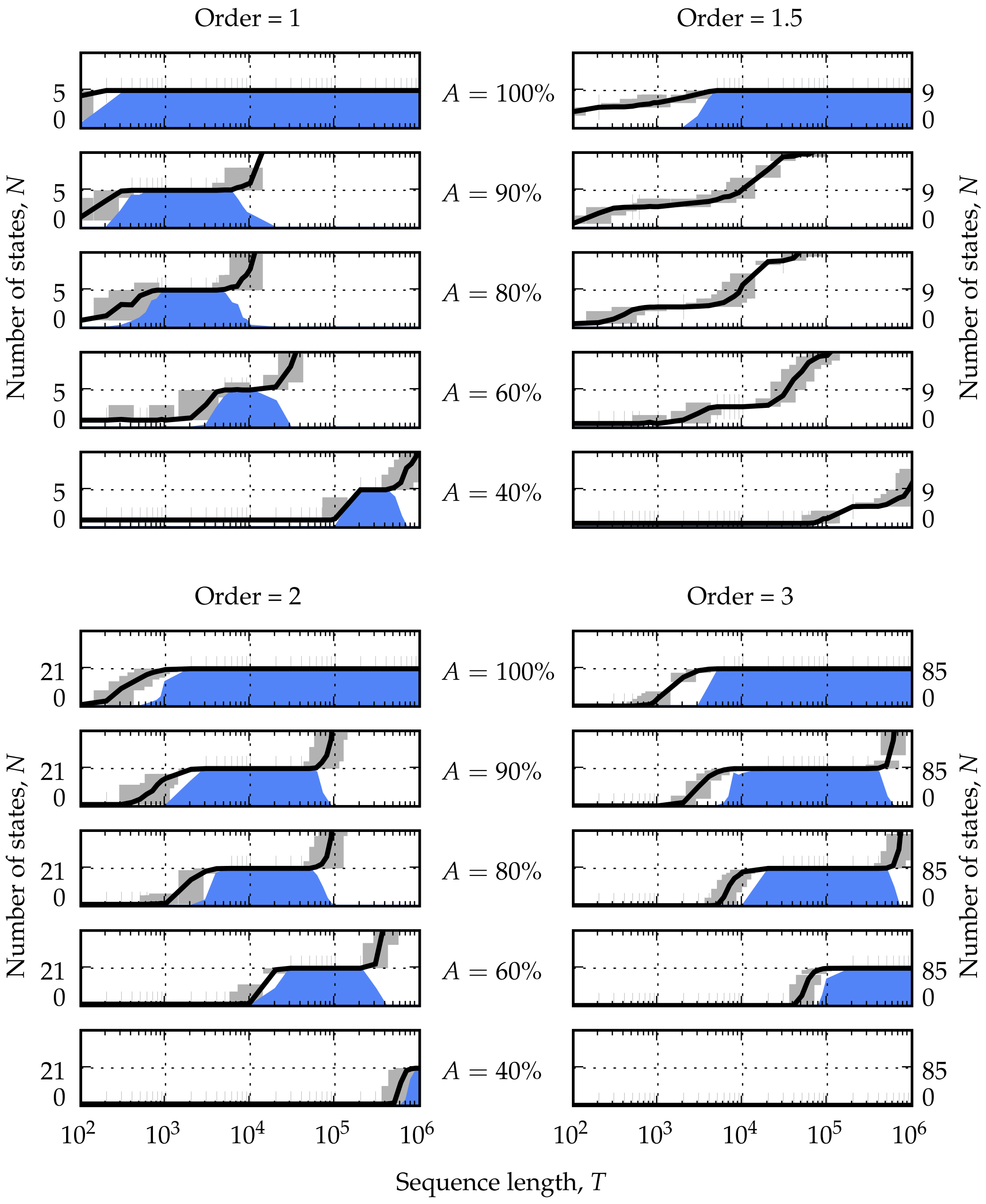


Figure 6.14: Results of the PST corrupted symbol experiment. The true PST has $M = 4$ symbols and a perplexity of $P = 2$. The black line in each graph is the number of states in each inferred PST for various values of the true order O and symbol accuracy A, averaged over twenty trials. The grey bars indicate the minimum and maximum number observed in each set of trials. The height of the blue region represents the percentage of inferred PSTs with the correct structure, scaled so that 100% coincides with the correct number of states.

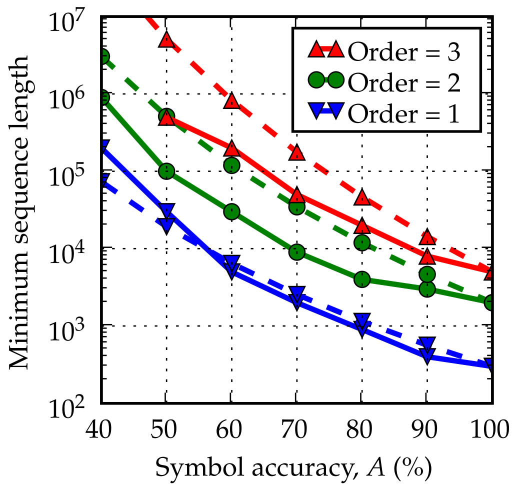


Figure 6.15: The solid lines show the minimum sequence lengths measured for various symbol accuracies and fixed model orders, while the dashed lines are the lengths predicted by the Angluin-Csűrös factor.

We now check if this factor applies to models inferred by the SECT algorithm. In order to identify PSTs with comparable performance, we focus on the *minimum sequence length*, defined as the smallest length for which more than 75% of inferred PSTs have the correct structure. The minimum lengths are clearly visible on the plots in Figure 6.14 as the points where the state counts and structural correctness graphs first rise to their correct values. The effect of symbol corruption is to shift these points to larger sequence lengths, and the idea is to compare the relative size of the shift with the factor predicted by Angluin et al. The tree depth bound L is taken to be the order O of both the true and correctly inferred PST (although SECT is technically a self-bounded algorithm). The Angluin-Csűrös factor is applied to the minimum uncorrupted sequence length to obtain predictions of the minimum lengths of the corrupted sequences.

The results are shown in Figure 6.15. The minimum sequence length increases significantly slower with decreasing symbol accuracy than predicted by the Angluin-Csűrös factor, except for first-order models with low symbol accuracies. It should be kept in mind, however, that the Angluin-Csűrös PST inference method includes an explicit error model, which is absent from the SECT method used in this study. This error model can also be added to SECT, which should ensure a fairer comparison.

6.2 Synthetic experiments with hidden SECT

The hidden SECT algorithm is first evaluated on synthetic data, which allows a better understanding of its properties. Each experiment in this section starts off by creating a true HMM with a specified set of properties. The HMM can be discrete or continuous. The true HMM generates an observation sequence x_1^T of length T in two phases. The underlying Markov chain of the true HMM first generates a symbol sequence s_1^T, which is then converted to observations by sampling from the HMM symbol pdfs. An initial HMM is created and trained on the observation sequence, using the hidden SECT algorithm. Finally, the trained HMM is compared with the true model on various criteria. Each experiment is repeated twenty times, to incorporate variance information in the results.

6.2.1 HMM setup

The true HMM is characterised by four parameters: alphabet size M, order O, perplexity P, and symbol accuracy A. The first three parameters specify a PST as defined in the previous section. This PST is converted into an equivalent Markov chain, which forms the underlying MC of the true HMM. This allows direct comparison of the hidden SECT results with the results in the previous section. The symbol accuracy, A, sums up the effect of the HMM symbol pdfs in a single number. It is the probability that a symbol pdf assigns the highest value of all the symbol pdfs to an observation generated by itself, averaged over all the symbol pdfs. A small symbol accuracy implies that the symbol pdfs have a large overlap, which makes the HMM more "hidden".

If the HMM is discrete, the M symbol pdfs are discrete distributions on $\mathcal{X} = \Sigma$. The observation symbol alphabet is therefore the same as the underlying symbol alphabet. Each symbol distribution assigns a probability of A to a unique symbol in Σ, and a probability of $(1 - A)/(M - 1)$ to the rest. The symbol distributions only differ in their choice of the symbol with the maximum probability. This discrete HMM therefore exactly models the corrupted symbol sequences studied in Section 6.1.6.

For a continuous HMM, the observation space is $\mathcal{X} = \mathbb{R}$ and the M symbol pdfs are one-dimensional Gaussian densities, with unique means at the integers

$[0, M-1]$ and a common variance. With this setup, the decision boundary between adjacent symbol pdfs is always halfway between their means, regardless of the value of the variance. The variance is chosen so that each symbol pdf, on average, has a probability weight of A within the decision boundaries with its neighbours.[2]. Its calculation is complicated by the fact that the "outside" symbols 0 and $(M-1)$ behave differently from the "inside" ones, and it also depends on the long run symbol distribution of the underlying Markov chain, which may be non-uniform.

The continuous symbol pdfs have different overlaps with each other, and a specific symbol is more easily confused with adjacent symbols than those further away. In order to reduce any interactions this might have with the Markov dependencies of the symbols (which is a relevant concern in the mixed-order case), the mapping between symbols and means is randomised in each experimental trial involving continuous HMMs. An example of the symbol pdfs in an alphabet of size $M = 4$ is shown in Figure 6.16, for various values of the symbol accuracy, A. Note that $A = 100\%$ results in a degenerate Gaussian pdf with zero variance, which is avoided by introducing a very small symbol error.

While $\mathbb{R}$ is a particularly simple continuous observation space, there are other attractive choices for $\mathcal{X}$ in controlled synthetic experiments like the ones in this section. For example, the symbol pdf means can be placed at the vertices of a regular $(M-1)$-simplex. Coupled with a common variance, this highly symmetrical setup ensures that each symbol pdf pair has identical overlap. This is the closest analog to the discrete case, but requires a observation space dimension of $D = M - 1$, which complicates the parameter count and increases data requirements for proper estimation of the symbol pdfs. Another option for $\mathcal{X}$ is the unit circle[3] in the complex plane, e^{jx}. This gets rid of special "outside" symbols, but complicates the form of the symbol density, which typically has to change from Gaussian to von Mises to account for the wrap-around nature of the observation space.

[2]Calculating the Gaussian variance based on A is the inverse process of determining the probability of error for a M-ary pulse amplitude modulation (PAM) scheme transmitting over an additive white Gaussian noise channel [163, p. 408] In this communication system analog, the probability of error becomes $(1 - A)$, the PAM symbol amplitudes have unity spacing, and the desired variance is the noise power $N_0/2$.

[3]This setup is analogous to M-ary phase-shift keying (PSK).

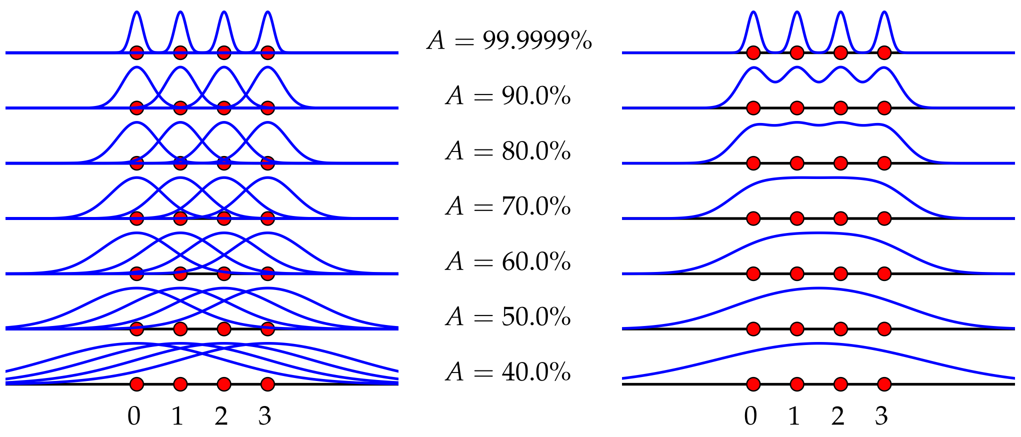


Figure 6.16: The symbol pdfs of the continuous HMMs in this section, for an alphabet size of $M = 4$ and various values of the symbol accuracy, A. The left-hand plot shows the individual densities, while the right-hand plot shows their combined mixture density, to emphasise the severe symbol overlap at low symbol accuracies. The pdf heights are not drawn to scale.

The hidden SECT training algorithm is an iterative scheme, which requires an initial HMM. The underlying Markov chain of this HMM is constructed as a first-order model ($O = 1$) with the same number of symbols M as the true HMM, and a perplexity of $P = M$. Since the perplexity is maximal, all transition probabilities in the model are equal to $1/M$, and the HMM is actually a zeroth-order model. The symbol pdfs of the discrete HMM are difficult to initialise due to their flexibility, and are therefore initialised (somewhat optimistically) to their true values. The continuous symbol pdfs are initialised by performing unsupervised clustering on the observation sequence. The M symbol pdf means are obtained by a binary split algorithm [164] followed by k-means clustering [121, 65], a procedure similar to Linde-Buzo-Gray vector codebook design [122]. The symbol pdf variances are initialised to one. The initial continuous HMM is therefore a Gaussian mixture model with M one-dimensional components and equal mixture weights.

6.2.2 Comparison of HMMs

The simplest comparison between the true and inferred HMMs is based on the number of states in each model. Unfortunately, the structural correctness check of Section 6.1.2 is not available for HMMs, because the symbol pdfs of the inferred HMM may not correspond to those of the true HMM, causing their alphabets to differ. The inferred HMM structure is verified by observing the number of states and doing manual checks of the structure for some parameter settings.

It is also very difficult to compute the exact Kullback-Leibler divergence rate between two HMMs. The alternatives include fast upper bounds [165, 166, 167] and Monte-Carlo numerical approximations [166]. The latter casts the divergence as the average log-likelihood ratio of two distributions, P_1 and P_2, given by

$$\mathrm{D}[P_1(x) \| P_2(x)] = \mathrm{E}_{P_1}\left[\log_2 \frac{P_1(x)}{P_2(x)}\right].$$

The divergence can therefore be approximated by generating a large but finite number of sequences from distribution P_1, calculating the log-likelihood ratio for each sequence, and averaging these values over all the sequences. Since the HMMs in this section are ergodic, a single long sequence will suffice.

We therefore approximate the divergence rate between the true HMM θ_1 and inferred HMM θ_2 by using the true model to generate a test observation sequence $x_1^{T_{\mathrm{KL}}}$ with a length of $T_{\mathrm{KL}} = 100\,000$, and calculating

$$\mathrm{D}(\theta_1 \| \theta_2) \approx \frac{1}{T_{\mathrm{KL}}}\left[\log_2 P\left(x_1^{T_{\mathrm{KL}}} \middle| \theta_1\right) - \log_2 P\left(x_1^{T_{\mathrm{KL}}} \middle| \theta_2\right)\right].$$

This measure is referred to as the *average log-likelihood ratio per symbol*. It is important to use the forward algorithm to calculate the HMM likelihoods instead of approximating them by Viterbi scores, otherwise the divergence rate may become negative. It is also important to calculate the divergence rate on an independently generated sequence instead of the original training sequence, otherwise the estimate may be overly optimistic.

Another useful criterion examines the estimation of the symbol sequence underlying an observation sequence x_1^T, based on an HMM that models x_1^T. The estimated symbol sequence $\hat{s}_1^T$ will typically differ from the true underlying symbol sequence s_1^T, due to the obscuring effect of the symbol pdfs during generation of x_1^T. We are interested in the accuracy of $\hat{s}_1^T$, which is obtained by comparing it to the true sequence and expressing the number of correct symbols as a percentage of T.

If the provided HMM is not the true HMM that generated x_1^T, but a model inferred from training data, it will have a different set of symbol pdfs. This makes it difficult to compare the estimated and true symbol sequences directly. One recourse is to map the inferred symbols to the closest true symbols. For discrete symbol pdfs, the symbol with the highest probability is chosen as the associated true symbol. In the continuous case, the true and inferred symbol pdf sets are aligned with dynamic programming [110]. This assigns a single true pdf to each inferred pdf, while ensuring that the resultant mapping between the pdf means are monotonic. The dynamic programming procedure operates on a matrix of distances between each true and inferred pdf, for which a suitable distance measure is the Kullback-Leibler divergence, which has a simple analytical expression for one-dimensional Gaussian pdfs.

Given a set of symbol pdfs, $\{\sigma(s,x)|s \in \Sigma\}$, a simplistic symbol estimation procedure assigns to each observation x_t the symbol $\hat{s}_t$ whose pdf has the highest value at x_t; that is, $\hat{s}_t = \arg\max_s \sigma(s, x_t)$. The symbols estimated in this way have a maximum accuracy of A, as dictated by the true symbol pdf overlap.

More accurate estimation is possible by exploiting the Markov dependencies of the underlying symbols. We use the Viterbi algorithm to obtain the optimal symbol sequence $\hat{s}_1^T$, based on the training observation sequence x_1^T and the inferred HMM. If the inferred HMM successfully captures the symbol statistics, we expect the estimated symbol accuracy to equal or exceed A. This problem can be cast in communication system terms, by viewing the symbol pdfs of the true generating HMM as noise models, and x_1^T as a noisy version of s_1^T. The Viterbi algorithm can then be seen to correct[4] some of the errors introduced by the generation of x_1^T, and the resulting measure is therefore called the *corrected symbol accuracy*.

[4]The Viterbi algorithm was originally introduced in the domain of error correction [48].

6.2.3 Hidden SECT and discrete HMMs

Preliminary experiments revealed that the current incarnation of the hidden SECT procedure is not suitable for discrete HMMs, because of the way in which it combines Viterbi re-estimation with a zeroth-order initial model. The first E step of Viterbi re-estimation will select the observation sequence itself as the optimal symbol sequence, because the transition probabilities have no influence yet. During the following M step, each symbol pdf is updated to a degenerate distribution with all its probability mass assigned to a single symbol. This effectively disables the symbol pdfs, which lose their "soft" nature. Further iterations of Viterbi re-estimation have no effect, and the trained HMM is identical to a PST inferred by the SECT algorithm from a corrupted symbol sequence, as described in Section 6.1.6.

This difficulty with training discrete HMMs stems from the flexibility of the model. The discrete symbol pdfs in this study are generic discrete distributions without any additional priors or smoothing, and are prone to overfitting during training. Furthermore, there is some redundancy between the observation symbol distributions and next-symbol distributions, so that several different HMM topologies may result in equivalent models. This corroborates the findings of studies that indicate that generic discrete HMM inference is hard [130, 131]. In contrast, continuous symbol pdfs such as Gaussian distributions have comparatively few degrees of freedom, so that continuous HMMs suffer less from these issues.

There are several ways to prevent the degenerate behaviour of the discrete symbol pdfs. The simplest way changes the zeroth-order initial HMM to a proper first-order model (for example, by adding large self-loop probabilities), but this presupposes extra knowledge of the underlying true HMM. Viterbi re-estimation can be replaced by Baum-Welch re-estimation, which allows data points to update more than one symbol pdf in the M step, but this complicates the estimation of next-symbol counts. Finally, the discrete symbol pdfs can be chosen from a different family of densities or smoothed by any of the methods described in Section 4.3.3, which prevents them from becoming degenerate.

These avenues are not explored in this study, as the inference of continuous HMMs is considered to be more fruitful. Continuous HMMs are widely used in speech recognition and other fields, but there are relatively few structural inference algorithms available for them. Discrete HMM inference, on the other hand, has been researched more extensively in the computer science literature. The rest of the experiments in this chapter will therefore focus on continuous HMMs.

6.2.4 HMM training schedule experiment

The first HMM experiment chooses an appropriate training schedule to use in the rest of the experiments. The true HMM is chosen to be second-order, with an alphabet of $M = 4$ symbols (as seen in Figure 6.16), and perplexity $P = 2$. The training sequences have lengths T that range from 100 to 500 000. The upper limit of T had to be reduced to accommodate the large memory requirements of the path matrix formed during Viterbi re-estimation. The symbol accuracies range from 40% to 99.9999%, where the upper limit ensures virtually error-free symbol sequences while avoiding degenerate symbol pdfs with zero variance. All three training schedules introduced in Section 5.4 are investigated.

Figure 6.17 shows the number of states of the inferred HMMs for a symbol accuracy of 60%. Similar to the results in Section 6.1.6, the HMM achieves the correct number of 21 states on medium-sized training sets (for values of T around 30 000 in this case). The HMM structure was manually verified to be correct at these settings. For larger data sets, the inferred HMM continues to expand, as it incorporates some of the statistics of the symbol corruption process. This is caused by discrepancies between the Viterbi-estimated and true symbol sequences.

The training schedules show similar overall behaviour, but the unlimited schedule produces larger HMMs on large data sets. The complexity control applied by the other schedules therefore has merit. The average log-likelihood ratio measure shows very little difference between the training schedules, and is therefore not included here. The rounds schedule does show a higher variance in this measure, however, and is computationally the most expensive schedule. For these reasons, the incremental schedule is used in the rest of the experiments.

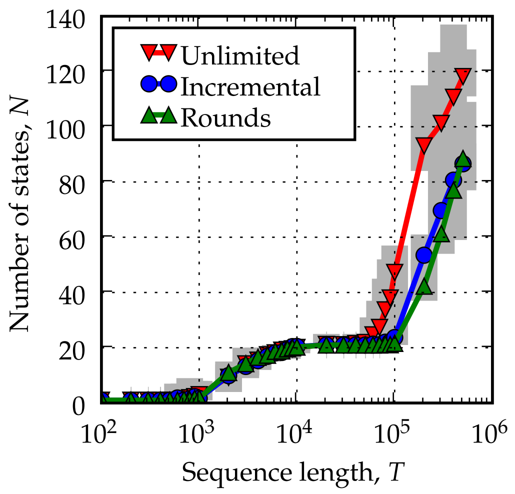


Figure 6.17: A comparison of the HMM training schedules, based on the number of states in the inferred HMM for various settings of the sequence length T. The true HMM is second-order and continuous, with $M = 4$, $P = 2$ and $A = 60\%$.

6.2.5 HMM symbol overlap experiment

The next experiment examines the effect of the true underlying order and the overlap between symbol pdfs on the inference of HMM structure. The true HMM again has an alphabet of $M = 4$ symbols, and a perplexity of $P = 2$. The sequence length, T, varies between 100 and 500 000, and the symbol accuracy, A, varies between 40% to 99.9999%.

The number of states of the inferred HMMs are shown in Figure 6.18, for various values of the sequence length, symbol accuracy and true order. The hidden SECT algorithm requires more data as the order increases and the symbol accuracy decreases, as expected. It is instructive to compare Figure 6.18 with Figure 6.14. The HMMs in this experiment require less data to uncover its structure, compared with a PST with the same experimental settings. The expansion of the model structure on large data sets is also less pronounced in the HMMs. This is due to the presence of an error model in the HMM (in the form of the symbol pdfs), and the non-uniform symbol pdf overlap in the HMM, which reduces the impact of symbol corruption on the sequence statistics.

Figure 6.19 illustrates the average log-likelihood ratio per symbol between the true and inferred HMMs. It is similar to the divergence rate for PSTs trained on corrupted symbol sequences, as seen in Figure 6.13b, since it only converges

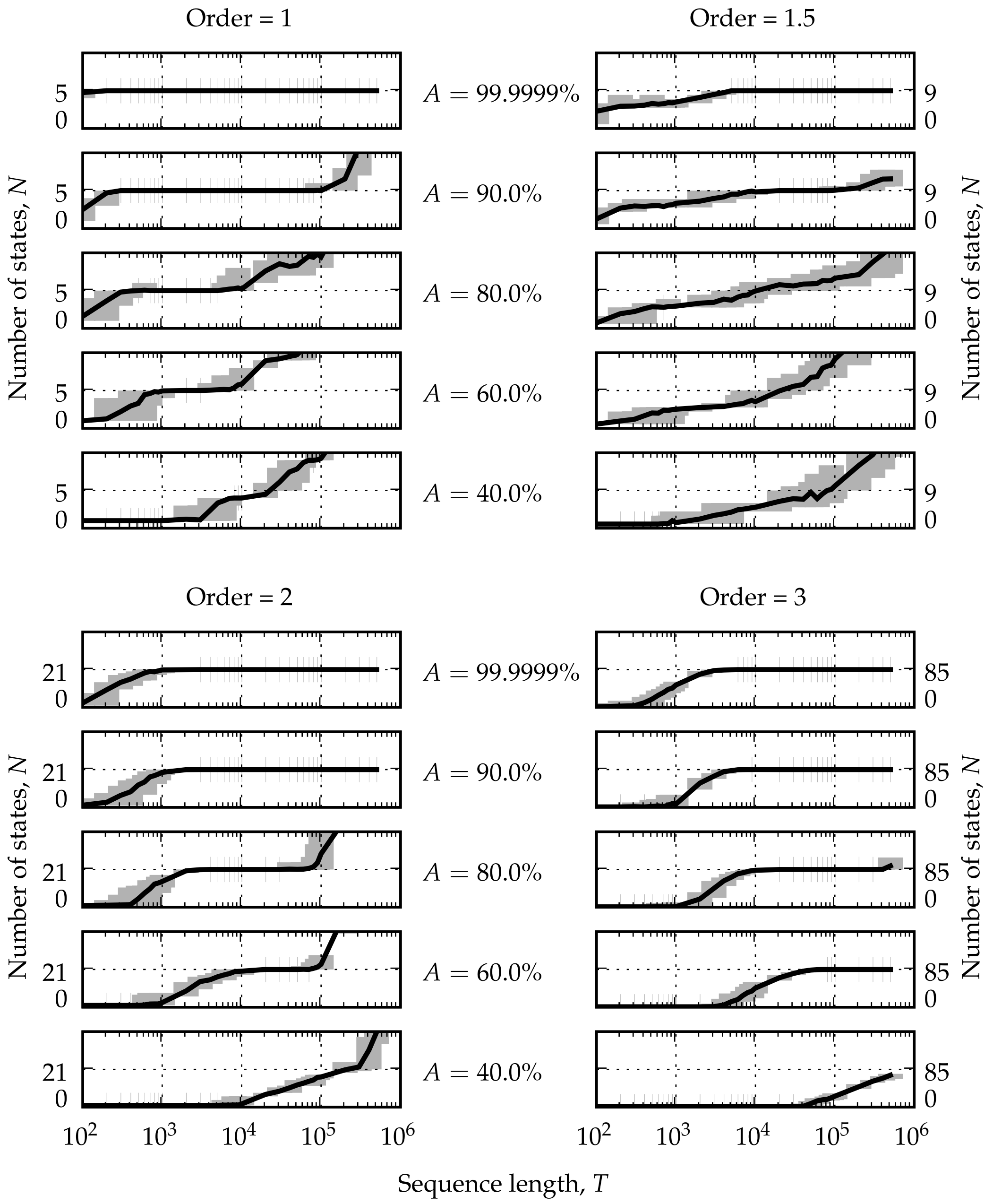


Figure 6.18: The results of the HMM symbol overlap experiment. The black lines are the number of states in the trained HMMs for various values of the sequence length T, true order O and symbol accuracy A, averaged over twenty trials. The grey bars indicate the observed range of state counts at each setting. The true HMM has $M = 4$ symbols and a perplexity of $P = 2$. It is instructive to compare this figure with Figure 6.14.

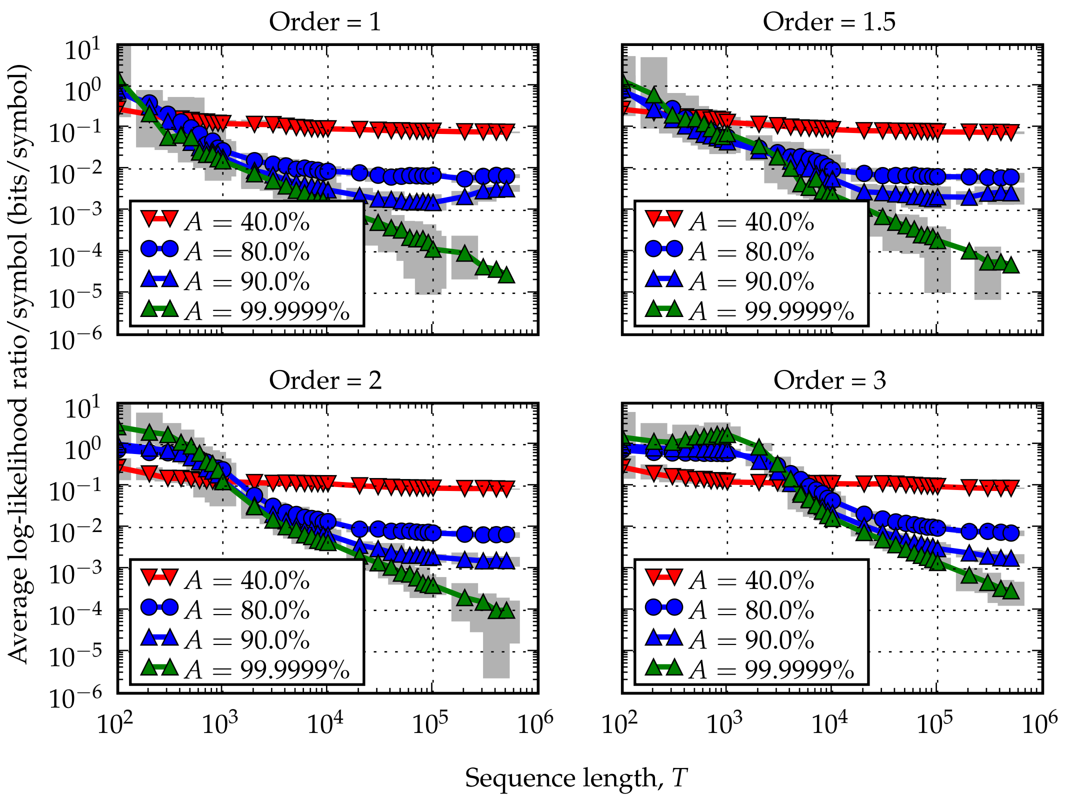


Figure 6.19: The results of the HMM symbol overlap experiment. The average log-likelihood ratio per symbol between the true and inferred HMMs, for various values of the sequence length T, true order O and symbol accuracy A, averaged over twenty trials. The grey bars indicate the observed range of values. The true HMM has $M = 4$ symbols and a perplexity of $P = 2$.

to zero with increasing training set size if there is no symbol overlap. While the PST divergence is an average divergence between next-symbol distributions, the HMM divergence measure contains an additional component, however, representing the average divergence between the symbol pdfs. Symbol pdfs with less overlap tend to have larger divergences, which increases the log-likelihood ratio, especially at small values of T.

The corrected symbol accuracy is shown in Figure 6.20. For raw symbol accuracies $A > 50\%$ and fixed-order true HMMs, the Viterbi algorithm is able to improve the accuracy of the symbol sequence, based on an HMM inferred from a sufficiently large training set. The Viterbi algorithm does even better on mixed-order sequences, presumably because of the non-uniform long run distribution

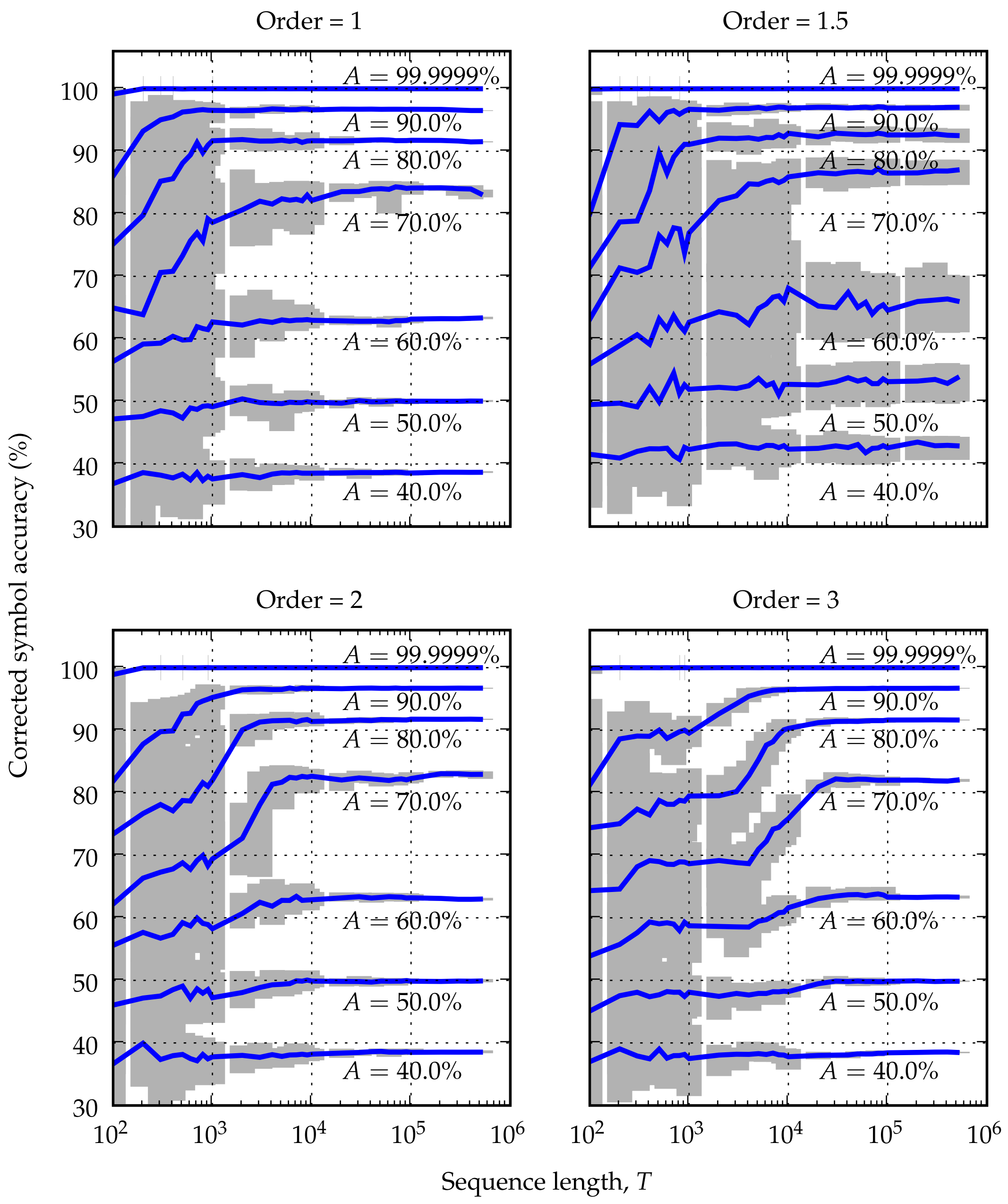


Figure 6.20: The accuracy of the symbol sequence after correction by the trained HMM, for various values of the sequence length T, true order O and raw symbol accuracy A, averaged over twenty trials. The grey bars indicate the observed range of values. The true HMM has $M = 4$ symbols and a perplexity of $P = 2$.

of the mixed-order HMM, which improves the estimation of the dominant context statistics. On small training sets, however, the Viterbi algorithm decreases the accuracy on average, because the trained HMM does not sufficiently resemble the true HMM. As the structure of the HMM is discovered, the corrected accuracy increases from below to above A. The sequence length where this change occurs is another indication of the data requirements for learning the HMM.

The expansion of the HMM structure on large training sets is clearly evident in the number of states of the HMM, but has a limited effect on the log-likelihood ratio, and does not show up at all in the corrected symbol accuracy. It might only be a problem if the HMM structure itself is the end goal of an experiment.

6.2.6 HMM alphabet size experiment

The last synthetic HMM experiment investigates the effect of alphabet size on HMM structure inference. The raw symbol accuracy is fixed to $A = 80\%$, while the alphabet size M ranges from four to 32, with a corresponding perplexity of $P = M/2$. The maximum sequence length and alphabet size had to be reduced for second-order and third-order models, as the memory requirements of Viterbi re-estimation became excessive otherwise.

Figure 6.21 shows the average log-likelihood ratio and corrected symbol accuracy for a first-order true model. The results for other orders are similar. The log-likelihood ratio increases with the alphabet size, and the knee in its graph shifts to larger sequence lengths. It is instructive to compare the log-likelihood ratios to the PST divergence rates for $O = 1$ in Figure 6.10. The corrected symbol accuracies increase from below to above 80% as the HMM structure is learnt, and confirms the increase in data requirements with an increase in alphabet size.

The number of states in the inferred HMM is displayed in Figure 6.22, for various settings of the sequence length, alphabet size and true order. It resembles the corresponding number of PST states in Figure 6.8, the main difference being the expansion of the HMM structure on large data sets. The performance of the hidden SECT algorithm on sequences with a symbol accuracy of 80% therefore approaches that of the SECT algorithm on uncorrupted symbol sequences. This is noteworthy, considering the significant symbol overlap implied by an accuracy of 80%, as shown in Figure 6.16.

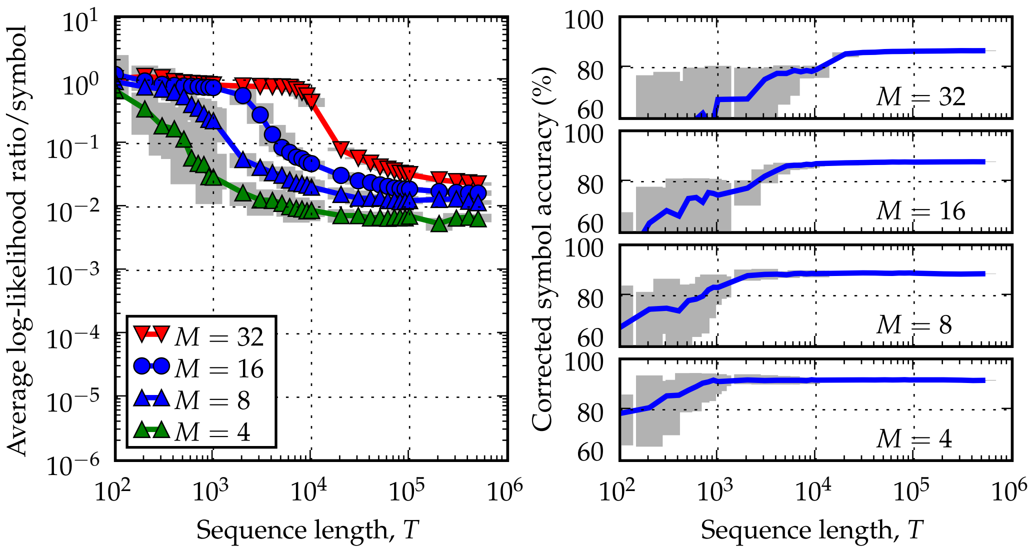


Figure 6.21: Results of the HMM alphabet size experiment. The left-hand figure shows the average log-likelihood ratio per symbol between the true and inferred HMMs, for various settings of the alphabet size M and sequence length T. The right-hand figure shows the accuracy of the corrected symbol sequence for the same settings. In both cases, the underlying true HMM is first-order, and the symbol accuracy is 80%.

6.3 Language recognition experiments

The state-of-the-art automatic language recognition (ALR) systems are based on phone recognisers and n-gram language models, which require the use of transcribed speech databases for training [168, 57]. An alternate solution to the ALR problem directly applies mixed-order hidden Markov models (HMMs) to untranscribed speech. The advantage of this approach is its ease of implementation, its applicability to a wider set of languages, and a greater abundance of training data. Its main disadvantage is poor performance compared with the more advanced phone-based methods. Most modern language recognition systems fuse the outputs of several independent ALR systems, however, to create a larger system that is better than any of its components [57, 56, 169]. The mixed-order HMM ALR system can provide a positive contribution to such a system.

The purpose of the following ALR experiments is to highlight the advantages of mixed-order models over fixed-order models in a practical application.

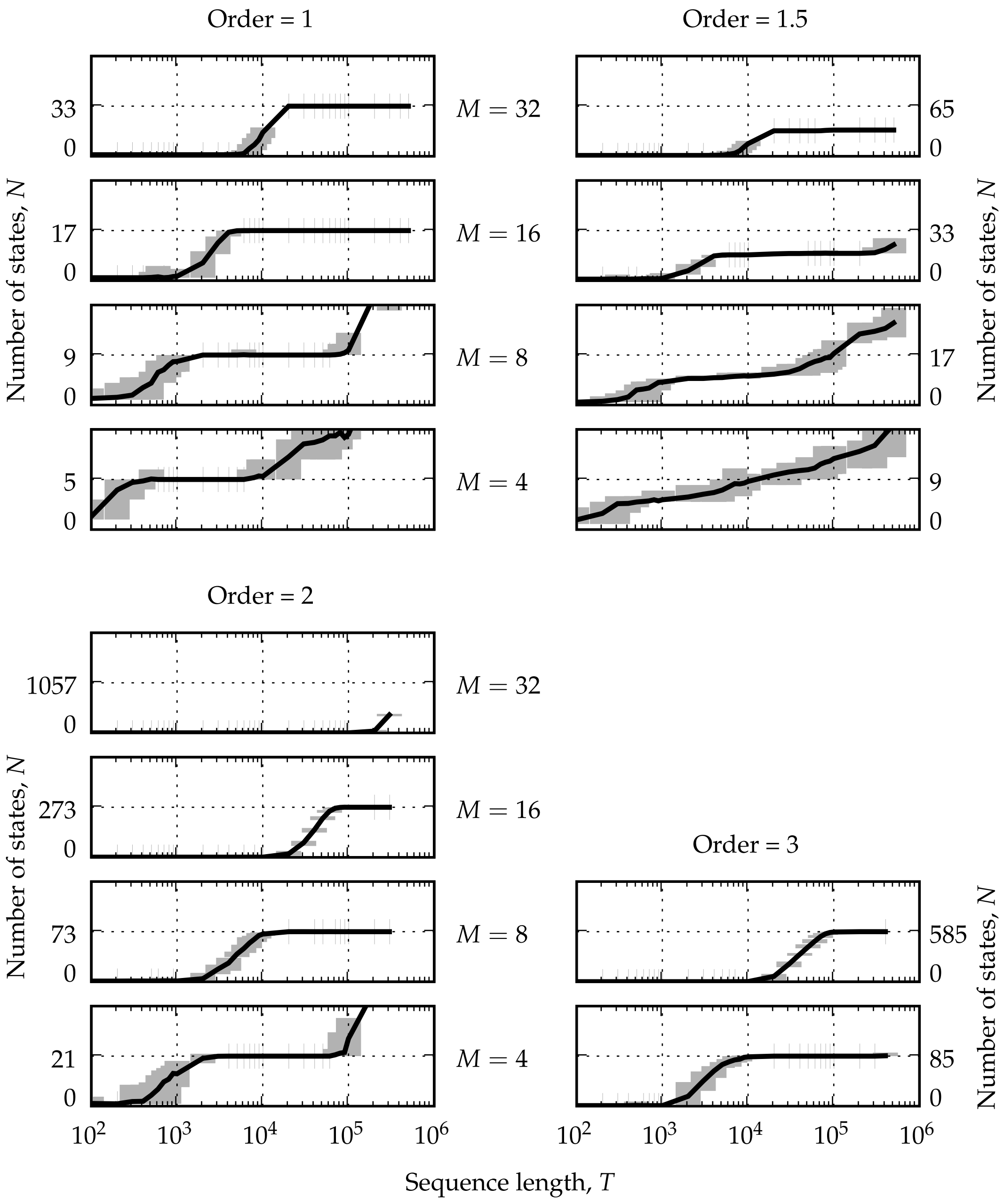


Figure 6.22: The results of the HMM alphabet size experiment. The black lines are the number of states in the trained HMMs for various values of the sequence length T, true order O and alphabet size M, averaged over twenty trials. The grey bars indicate the observed range of state counts at each setting. The state count indicated in the middle of each axis is the number of states in the true model. The true HMM has a perplexity of $P = M/2$, and the symbol accuracy is 80%. It is instructive to compare this figure with Figure 6.8.

It is therefore not intended to be competitive with state-of-the-art systems. The experimental setup mirrors that of a similar experiment involving fixed-order models in [39].

6.3.1 Experimental setup and signal processing

The ALR experiments are performed on the CALLFRIEND speech corpus [170], a large untranscribed database of conversational telephone speech. The corpus contains twelve languages, of which three (English, Mandarin and Spanish) have two dialects each, and is divided into a training, development and evaluation set. A language model is trained for each of the fifteen dialects on the full training and development set. This results in approximately fifteen hours of training data per model, after silent sections and crosstalk are removed.

After pre-emphasis and power normalisation, the speech signal is broken up into 32 ms frames, spaced at 16 ms intervals. We apply a Hamming window to each frame, and calculate tenth-order linear prediction coefficient (LPC) cepstra [171, 172] from each windowed frame. This speech feature is a well-known and closely related alternative to the popular mel-frequency cepstral coefficient (MFCC) feature [173]. Ten delta cepstral coefficients are also calculated per frame, which model correlations between adjacent feature vectors. Cepstral mean subtraction [171] is used to alleviate some of the adverse telephone channel effects, and the final twenty-dimensional feature vectors are fed directly to HMM recognisers for language recognition.

Evaluation is done on the National Institute of Standards and Technology (NIST) 1996 Language Recognition Evaluation corpus. This data set consists of telephone speech segments of various durations (3 s, 10 s and 30 s), spoken in one of twelve target languages. The vast majority of these segments are derived from the CALLFRIEND evaluation set, which are augmented with extra English conversations from the Switchboard-1 [174] and KING [175] corpora. These extra segments were ignored during testing, in line with existing published results [176]. The evaluation set represents approximately one hour of test data per dialect.

The three languages with two dialects each are treated as single languages during evaluation. This is achieved by taking the maximum of the two related dialect scores as the score for the language they represent. The final system therefore considers twelve alternatives during classification. Additional tests evaluate pairwise classification, whereby every language is compared with every other language and the error rates are averaged. In all cases the classifier is forced to choose a specific language from a closed set, thereby avoiding verification issues.

6.3.2 Language models and algorithms

The fifteen HMMs, one for each dialect, share a common acoustic alphabet of $M = 32$ diagonal Gaussian symbol pdfs, which is initialised on the training set by binary split unsupervised clustering [164] and trained along with the HMMs. This language-independent codebook makes the system more robust against adverse channel effects, a major concern when using telephone speech. The language models therefore only differ in their transition probabilities, resulting in a purely phonotactic approach to language recognition. The symbol pdfs are kept simple in form, in order to allocate more degrees of freedom to high-order transition statistics.

The choice of an acoustic alphabet size of $M = 32$ is in agreement with the similar experimental setup found in [39]. It is also intended to be comparable with the number of phonemes in a typical language. Furthermore, the fifteen hours of training data per model, coupled with a feature vector spacing of 16 ms, imply a symbol sequence length of approximately $T = 3 \times 10^6$ symbols. From the results of the synthetic experiments shown in Figures 6.11, 6.14 and 6.22, it seems reasonable that such a training set size could support models with an average order of $O = 2$ for a symbol overlap in the order of $A = 60\%$.

All HMMs are trained with Viterbi re-estimation incorporating a beam [173, Chap. 12] that discards highly unlikely state sequences at each time instant. This cuts computation times by between a factor two and four, with very little effect on the final likelihood scores. The test set scores are also calculated with a Viterbi beam similarity measure.

The baseline system is a standard ergodic first-order HMM with 32 states (termed F1 in the following). Although not as powerful as the high-order models, it will serve as a useful reference and also as initialisation for the incremental training of second-order models. The fixed second-order model, F2, and third-order model, F3, are trained with the FIT algorithm [39], which initialises the training of model F2 with the trained model F1, and similarly initialises the training of model F3 with the trained model F2.

The mixed-order approach also allows for FIT training, by incrementally lifting a constraint on the maximum order of the model. This is achieved by the rounds training schedule, which has the closest correspondence to the fixed-order FIT process. The mixed-order MR2 model is initialised from F1 and trained with hidden SECT until convergence, while limiting its maximum order to two. The order limit is increased to three, and training again continues until convergence, this time producing the MR3 model. The mixed-order MUx model uses the unlimited training schedule, and is trained directly from the initial codebook, without any limit on its context lengths.

6.3.3 Results

The mixed-order training selectively expands the HMM where the training data supports it. This allows the modelling of sparse high-order structures, which has the ability to lock onto common sound patterns in the language, at the sub-phone level. Fixed-order training becomes infeasible above about fifth order, while the large training set supports up to eighth-order contexts with unlimited mixed-order models (MUx). As can be seen in Figure 6.23, the vast majority of contexts has a length of two or three, confirming the sparse mixed-order behaviour. The average order of the MUx model can be estimated as the logarithm of its number of states to the base of the number of symbols, resulting in $\log_M(N) = \log_{32}(1123) = 2.03$. Also interesting to note is the mismatch of standard first-order modelling to this task.

Table 6.2 and Figure 6.24 summarise the results obtained on the 30 s CALL-FRIEND segments of the NIST 1996 test set. As expected, performance increases with increasing model order. The mixed-order models, MR2 and MR3, perform slightly worse than their fixed-order counterparts, F2 and F3, while training faster and ending up smaller. Their weaker performance is attributed to the conservative nature of the SECT algorithm, which underestimates the model structure and thereby misses out on useful longer contexts. Bejerano [20] came

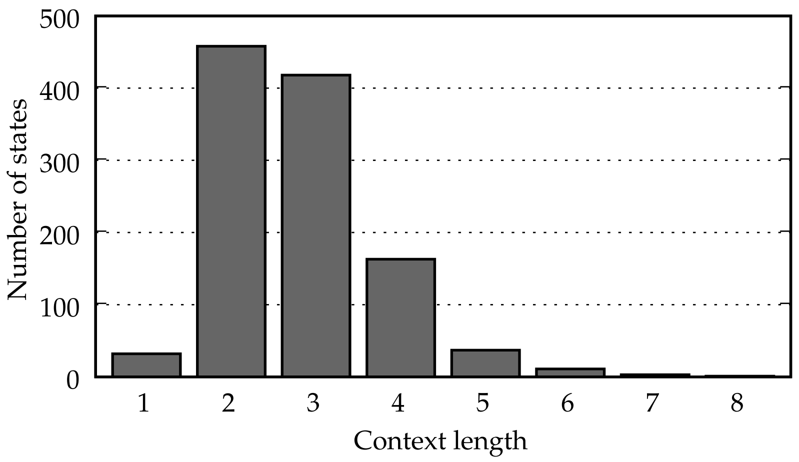


Figure 6.23: Histogram of states in mixed-order model MUx according to associated order (context length). The decline with order indicates a sparse mixed-order model.

Table 6.2: ALR results obtained on the 30 s CALLFRIEND segments of the NIST 1996 test set. F1 is a standard first-order model. The fixed-order models, F2 and F3, are trained incrementally via FIT. The mixed-order models, MR2 and MR3, are trained with hidden SECT and the rounds training schedule, while model MUx is trained with the unlimited schedule. The training times are relative to a Celeron 400 MHz processor, and the model size is the average number of links in the dialect HMMs.

Model	Training time	Model size	Test set error (%)	
	(hr/dialect)	(links)	12-way	pairwise
F1	0.5	585	48.2	13.4
F2	1.6	2640	37.8	10.2
MR2	1.4	2313	38.5	10.3
F3	5.8	9074	33.9	9.1
MR3	2.0	4313	35.0	9.1
MUx	3.2	11280	38.1	8.7

to a similar conclusion in his study of MDL-trained PSTs. The advantage of incremental (FIT) training can be seen from the fact that MR3 outperformed the directly trained MUx model on the twelve-way classification problem. Interestingly, the MUx model outperformed all other models on pairwise classification.

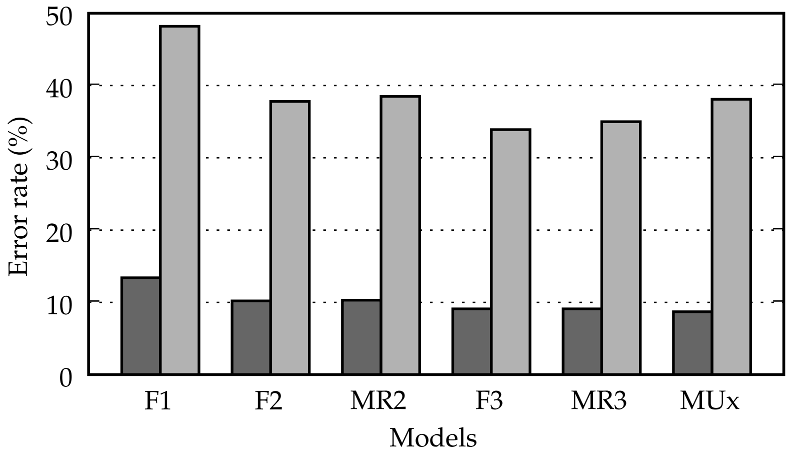


Figure 6.24: Average error rates on the 30s CALLFRIEND segments of the NIST 1996 evaluation set. The left-hand bars represent pairwise classification, while the right-hand bars represent twelve-way classification.

6.4 Summary

A set of experiments on synthetic data verifies that the SECT algorithm successfully learns PSTs. It also examines the effects of training set size, alphabet size, order and perplexity (which serves as an effective alphabet size) on the performance of the algorithm. SECT is applied to corrupted symbol sequences as well, which typically leads to overexpanded PSTs that attempt to model the symbol errors.

The synthetic data experiments are repeated for hidden SECT, while adding symbol accuracy as a parameter. The algorithm successfully learns HMMs, although it tends to overexpand them on large training sets. This indicates that the Viterbi symbol sequence still contains errors that can skew the underlying symbol statistics. The idea of gradually incrementing the HMM order during training was found to be useful.

Finally, an automatic language recognition experiment based on the CALLFRIEND corpus and ergodic HMMs shows that mixed-order HMMs train faster and are smaller than fixed-order models, for similar classification accuracies.

Chapter 7

Conclusions and Recommendations

This study confirms that the prediction suffix tree (PST) is a useful representation of Markov chains, especially for inference and storage. The smallest encoded context tree (SECT) algorithm is introduced as a promising new inference method for PSTs. The underlying Markov chain in a hidden Markov model can also be replaced by a PST, which leads to the hidden SECT algorithm for inferring HMM topology from data. The algorithms were validated on synthetic data and a language identification task.

The SECT algorithm has several advantages over the standard PST algorithm, Learn-PSA [22]. It is faster ($O(T(\log T)^2)$ versus $O(T^2)$ for a data sequence length of T), and has no user-specifiable parameters, which removes the need for expensive cross-validation steps. It is self-bounded, and will expand the model order as far as the data set warrants it. The disadvantages of SECT is that it is slower than the latest PST variants [25, 20], and it is a batch algorithm with $O(2T)$ memory requirements. Its reliance on MDL and lack of tunable parameters make it more conservative as well, and it underestimates rather than overestimates the model structures.

The hidden SECT algorithm has many advantages for HMM topology inference. It slots into the standard EM training algorithms, with a small overhead in computational complexity. It does not rely on costly state merging and splitting operations. It is guaranteed to converge. It infers variable-order instead of fixed-order models. Its main disadvantage is an overexpansion of HMM structure on large data sets, due to its dependence on Viterbi re-estimation, which slows down training and testing of the resulting HMM. It also underestimates the model structure on small data sets, due to the nature of SECT.

Two useful concepts in this study can be summed up as "compression is modelling" and "focus on symbols instead of states". The first concept is the tenet of the minimum description length approach to modelling. SECT can be used as a compression algorithm as well, as described in [44]. Similarly, many ideas from text compression are finding use in statistical modelling. While some of them are controversial (see [177] and the comments in [178, 179, 180]), studies such as [181] show the usefulness of this connection, especially for Markov modelling. The second concept advocates a different perspective on HMM structure, which simplifies the specification of high-order HMMs and opens new avenues for structural inference, such as the hidden SECT algorithm.

The remainder of this chapter examines specific improvements and recommendations for the SECT and hidden SECT algorithms.

7.1 Recommendations for SECT

Many improvements to the SECT algorithm are possible. Its computational complexity can be reduced to $O(T)$, based on the ideas expressed in [25, 20]. The MDL framework of SECT can be complemented by other Bayesian techniques. For instance, Wolpert describes a Bayesian test in [182] which determines whether two sets of symbol counts were generated by the same underlying multinomial distribution. This could be useful to check whether a child node differs sufficiently from its parent.

The smoothing of the next-symbol distributions have not been investigated in full. In particular, [183] shows that the simplistic smoothing of Learn-PSA hampers the performance of a PST on a protein domain detection task, where it is outperformed by a standard Markov chain with more advanced Kneser-Ney back-off smoothing [184]. An interesting alternative to smoothing is to include Brand's entropic prior [61], which would force the model structure to become more definite.

More careful coding of the model overheads would improve the sensitivity of SECT, allowing it to detect structure in shorter symbol sequences. The next-symbol counts table can be coded according to the approach in [45] instead. This takes the precision of the resulting next-symbol distribution into account, which enables the use of a coarser set of counts if it reduces the overall code length.

More experiments are required to characterise the performance of SECT on mixed-order models. This could result in an explicit formula for the minimum data set size required to infer the structure of a given PST, which would depend on the perplexity, the long run context distribution and the number of parameters in the model. It would also be useful to compare the SECT algorithm to other variable-order Markov methods on the same data sets. A good starting point could be the experiments of [181], which compare six prominent variable-order Markov models on tasks of general text compression, musical file compression and protein classification.

7.2 Recommendations for hidden SECT

The most immediate improvement to the hidden SECT algorithm would be to base it on Baum-Welch re-estimation instead of Viterbi re-estimation. The optimal symbol sequence produced by the Viterbi algorithm still contains errors when compared to the true symbol sequence. With sufficiently large data sets, hidden SECT attempts to model these errors in the HMM, which then over-expands its structure. Baum-Welch re-estimation allows "soft" symbol counts, which would lend extra support to the correct contexts that are ignored by Viterbi in many cases.

The implementation of a version of hidden SECT based on Baum-Welch re-estimation faces some obstacles, though. While it is simple to count the number of occurrences of any context string such as abc in a symbol sequence, Baum-Welch requires the calculation of probabilities such as $P(s_{t-2} = \mathtt{a}, s_{t-1} = \mathtt{b}, s_t = \mathtt{c})$, based on manipulation of the forward and backward variables ("alphas" and "betas"). These probabilities are summed over all values of t to obtain the corresponding soft counts. A symbol sequence of length T with alphabet size M potentially supports $M^L(T-L)$ such probabilities for contexts of length L, which indicates that the maximum context length should in practice be bounded to reduce the computational complexity.

A simpler and less accurate option is to focus on state probabilities such as $P(q_t = i)$ instead, which can be readily obtained from the forward-backward recursion. The probability of a specific symbol at time t then becomes the sum of the probabilities of the states associated with that symbol. The desired probability $P(s_{t-2} = \mathtt{a}, s_{t-1} = \mathtt{b}, s_t = \mathtt{c})$ can then be estimated as $P(s_{t-2} = \mathtt{a})P(s_{t-1} = \mathtt{b})P(s_t = \mathtt{c})$, before summing it over the length of the sequence to obtain the soft count. The problem with this approach is that it does not respect the constraints of the model structure, which might prohibit a sequence abc, even though the individual symbols are allowed to occur. A more practical alternative to Baum-Welch re-estimation is to obtain the N best symbol sequences via Viterbi [185], which would allow soft counts without the corresponding explosion in computational complexity.

A second problem with soft counts is that they lack the natural discretisation of hard counts, which complicates the coding of next-symbol count tables in SECT. A crude solution is to round off the counts to the nearest integer, while the approach in [45] is more correct and also promises a more compact next-symbol count table.

The symbol structure and consequent HMM state tying inferred by the hidden SECT algorithm are constrained to be Markovian. More general symbol structures can be inferred by replacing the SECT algorithm in hidden SECT with a procedure such as ALERGIA [41] or MDI [60]. This allows the underlying symbol structure to become a DPFA (like the example shown in Figure 2.13), which provides more general HMM state tying capabilities. While the resulting HMM would not be a more powerful model, it can be much more compact. Such a "hidden ALERGIA" algorithm might be more sensitive to symbol estimation errors and computationally more complex, though.

Another extension to the hidden SECT algorithm estimates the symbol pdfs within the minimum description length framework as well, as is done in tSnob [43]. This extension automatically determines the number of symbols in the alphabet, and favours continuous HMMs. It requires coding of the symbol pdf parameters, for which MML [86] is attractive. An important issue is the search strategy for the optimal alphabet, especially when combined with the PST inference step. A symbol splitting procedure would be computationally less expensive than one that merges symbols, as there are more symbol pairs to merge than symbols to split. This would also complement the top-down approach of the SECT algorithm, by starting with small alphabets and gradually increasing the model complexity.

The language identification experiment in this study is preliminary in nature. Its main purpose is to show the advantage of mixed-order models over fixed-order models. A major improvement to the experiment would be to replace the codebook of Gaussian pdfs with a bank of language-independent phone HMMs. These HMMs can be regarded as the symbol pdfs of a higher-level HMM. The hidden SECT algorithm would be used to train the transitions and structure of the top-level HMM, while the phone HMMs are left unchanged. A suitable framework for this process is the hierarchical HMM [186]. The end result is again an HMM language model. This allows hidden SECT to focus directly on the phonotactic constraints of the languages, instead of modelling subphone patterns of lengths typically shorter than 250 ms. Further improvements can be gained by including a Gaussian backend classifier [57], and fusing with other LID subsystems.

Mixed-order Markov models are successfully applied in many problem domains. Some of the more promising research areas are motion tracking from video and human gesture recognition [13, 14], high-frequency financial time series modelling [16, 17] and bioinformatics problems such as protein classification [19, 20]. The hidden SECT algorithm might still have its biggest success in one of these fields.

List of References

[1] D. L. Isaacson and R. W. Madsen, *Markov Chains: Theory and Applications*. Wiley, 1976.

[2] Y. Bengio, "Markovian models for sequential data," *Neural Computing Surveys*, Vol. 2, pp. 129–162, 1999.

[3] F. Jelinek, *Statistical Methods for Speech Recognition*. MIT Press, 1998.

[4] P. Bühlmann and A. J. Wyner, "Variable length Markov chains," *Annals of Statistics*, Vol. 27, pp. 480–513, 1999.

[5] G. V. Cormack and R. N. S. Horspool, "Data compression using dynamic Markov modelling," *Computer Journal*, Vol. 30, pp. 541–550, 1987.

[6] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context tree weighting method: basic properties," *IEEE Transactions on Information Theory*, Vol. 41, pp. 653–664, May 1995.

[7] I. Guyon and F. Pereira, "Design of a linguistic postprocessor using variable memory length Markov models," tech. rep., AT&T Bell Laboratories, 1995. Unpublished technical report, available at `http://www.clopinet.com/isabelle/Papers/vlmm1.ps.Z`.

[8] T. R. Niesler and P. C. Woodland, "A variable-length category-based n-gram language model," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 164–167, 1996.

[9] R. Kneser, "Statistical language modeling using a variable context length," in *Proceedings of the International Conference on Spoken Language Processing*, Vol. 1, pp. 494–497, 1996.

[10] J. Hu, W. Turin, and M. K. Brown, "Language modeling using stochastic automata with variable length contexts," *Computer Speech and Language*, Vol. 11, No. 1, pp. 1–17, 1997.

[11] S. Mori and G. Kurata, "Class-based variable memory length Markov model," in *Proceedings of the European Conference on Speech Communication and Technology (INTERSPEECH)*, pp. 13–16, 2005.

[12] A. McCallum, "Instance-based utile distinctions for reinforcement learning with hidden state," in *Proceedings of the International Conference on Machine Learning*, pp. 387–395, 1995.

[13] A. Galata, N. Johnson, and D. Hogg, "Learning variable length Markov models of behaviour," *Computer Vision and Image Understanding (CVIU) Journal*, Vol. 81, pp. 398–413, Mar. 2001.

[14] N. Stefanov, A. Galata, and R. Hubbold, "Real-time hand tracking with variable-length Markov models of behaviour," in *Proceedings of the IEEE Workshop on Vision for Human-Computer Interaction (V4HCI)*, 2005. Available at `http://www.cs.man.ac.uk/~agalata/publications/SGH-V4HCI-05.pdf`.

[15] J. Borges and M. Levene, "Evaluating variable length Markov chain models for analysis of user web navigation sessions," 2006. E-print available at `arXiv:cs.AI/0606115`.

[16] C. P. Papageorgiou, "High frequency time series analysis and prediction using Markov models," in *Proceedings of the Conference on Computational Intelligence for Financial Engineering*, pp. 182–185, 1997.

[17] C. P. Papageorgiou, "Mixed memory Markov models for time series analysis," in *Proceedings of the Conference on Computational Intelligence for Financial Engineering*, pp. 165–170, 1998.

[18] P. Tiňo, C. Schittenkopf, and G. Dorffner, "Volatility trading via temporal pattern recognition in quantised financial time series," *Pattern Analysis and Applications*, Vol. 4, pp. 283–299, 2001.

[19] G. Bejerano and G. Yona, "Variations on probabilistic suffix trees—a new tool for statistical modeling and prediction of protein families," *Bioinformatics*, Vol. 17, No. 1, pp. 23–43, 2001.

[20] G. Bejerano, *Automata learning and stochastic modeling for biosequence analysis*. Ph.D. dissertation, School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel, 2003.

[21] X. Zhao, H. Huang, and T. Speed, "Finding short DNA motifs using permuted Markov models," in *Proceedings of the International Conference on Research in Computational Molecular Biology (RECOMB)*, pp. 68–75, 2004.

[22] D. Ron, Y. Singer, and N. Tishby, "The power of amnesia: learning probabilistic automata with variable memory length," *Machine Learning*, Vol. 25, No. 2/3, pp. 117–149, 1996.

[23] D. Ron, Y. Singer, and N. Tishby, "The power of amnesia," in *Advances in Neural Information Processing Systems* (J. D. Cowan, G. Tesauro, and J. Alspector, eds.), no. 6, pp. 176–183, Morgan Kaufmann, 1994.

[24] I. Guyon and F. Pereira, "Design of a linguistic postprocessor using variable memory length Markov models," in *Proceedings of the International Conference on Document Analysis and Recognition*, pp. 454–457, IEEE Computer Society Press, 1995.

[25] A. Apostolico and G. Bejerano, "Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space," *Journal of Computational Biology*, Vol. 7, No. 3/4, pp. 381–393, 2000.

[26] Y. Seldin, G. Bejerano, and N. Tishby, "Unsupervised sequence segmentation by a mixture of switching variable memory Markov sources," in *Proceedings of the International Conference on Machine Learning*, Vol. 18, pp. 513–520, Morgan Kaufmann, 2001.

[27] O. Dekel, S. Shalev-Shwartz, and Y. Singer, "The power of selective memory: self-bounded learning of prediction suffix trees," in *Advances in Neural Information Processing Systems*, no. 17, pp. 345–352, MIT Press, 2004.

[28] A. E. Raftery, "A model for high-order Markov chains," *Journal of the Royal Statistical Society, series B*, Vol. 47, pp. 528–539, 1985.

[29] L. K. Saul and M. I. Jordan, "Mixed memory Markov models: decomposing complex stochastic processes as mixtures of simpler ones," *Machine Learning*, Vol. 37, pp. 75–87, Oct. 1999.

[30] A. Berchtold and A. Raftery, "The mixture transition distribution model for high-order Markov chains and non-Gaussian time series," *Statistical Science*, Vol. 17, No. 3, pp. 328–356, 2002. E-print available at `http://projecteuclid.org/getRecord?id=euclid.ss/1042727943`.

[31] L. K. Saul and F. Pereira, "Aggregate and mixed-order Markov models for statistical language processing," in *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP-1997)*, pp. 81–89, 1997. E-print available at `arXiv:cmp-lg/9706007`.

[32] M. Mächler and P. Bühlmann, "Variable length Markov chains: methodology, computing, and software," *Journal of Computational and Graphical Statistics*, Vol. 13, pp. 435–455, Jun. 2004.

[33] P. A. J. Volf and F. M. J. Willems, "Context maximizing: finding MDL decision trees," in *Proceedings of the 15th Symposium on Information Theory in the Benelux*, pp. 192–199, May 1994.

[34] S. Deligne and F. Bimbot, "Language modeling by variable length sequences: theoretical formulation and evaluation of multigrams," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 169–172, 1995.

[35] S. Deligne and F. Bimbot, "Inference of variable-length linguistic and acoustic units by multigrams," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1731–1734, 1997.

[36] A. K. McCallum, *Reinforcement learning with selective perception and hidden state*. Ph.D. dissertation, Department of Computer Science, University of Rochester, Rochester, NY, 1995.

[37] K. R. Pfleger, *On-line learning of predictive compositional hierarchies*. Ph.D. dissertation, Stanford University, 2002.

[38] P. Tiňo and G. Dorffner, "Predicting the future of discrete sequences from fractal representations of the past," *Machine Learning*, Vol. 45, pp. 187–217, Nov. 2001.

[39] J. A. du Preez, *Efficient high-order hidden Markov modelling*. Ph.D. dissertation, Stellenbosch University, Stellenbosch, South Africa, 1998.

[40] A. Stolcke and S. Omohundro, "Hidden Markov model induction by Bayesian model merging," in *Advances in Neural Information Processing Systems* (S. J. Hanson, J. D. Cowan, and C. L. Giles, eds.), no. 5, pp. 11–18, Morgan Kaufmann, 1992.

[41] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Proceedings of the Second International Colloquium on Grammatical Inference and Applications (ICGI-94)*, Vol. 862 of *Lecture Notes In Computer Science*, pp. 139–152, Springer-Verlag, 1994.

[42] M. Ostendorf and H. Singer, "HMM topology design using maximum likelihood successive state splitting," *Computer Speech and Language*, Vol. 11, No. 1, pp. 17–41, 1997.

[43] T. Edgoose and L. Allison, "Minimum message length hidden Markov modelling," in *Proceedings of the IEEE Data Compression Conference*, pp. 169–178, Mar. 1998.

[44] L. C. Schwardt and J. A. du Preez, "Modelling temporal structure with prediction suffix trees," in *Proceedings of the Tenth Annual Symposium of the Pattern Recognition Association of South Africa (PRASA)*, 1999.

[45] C. S. Wallace and D. M. Boulton, "An information measure for classification," *Computer Journal*, Vol. 11, pp. 185–194, 1968.

[46] J. Rissanen, "Modelling by shortest data description," *Automatica*, Vol. 14, pp. 465–471, 1978.

[47] A. Barron, J. Rissanen, and B. Yu, "The minimum description length principle in coding and modeling," *IEEE Transactions on Information Theory*, Vol. 44, No. 6, pp. 2743–2760, 1998.

[48] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimal decoding algorithm," *IEEE Transactions on Information Theory*, Vol. IT-13, pp. 260–269, Apr. 1967.

[49] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, Vol. 77, pp. 257–286, Feb. 1989.

[50] L. C. Schwardt and J. A. du Preez, "Efficient mixed-order hidden Markov model inference," in *Proceedings of the International Conference on Spoken Language Processing*, Vol. 2, pp. 238–241, Oct. 2000.

[51] B. H. Juang and L. R. Rabiner, "The segmental *k*-means algorithm for estimating parameters of hidden Markov models," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 38, pp. 1639–1641, Sep. 1990.

[52] K. Rose, "Deterministic annealing for clustering, compression, classification, regression, and related optimization problems," *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2210–2239, 1998.

[53] Z. Rached, F. Alajaji, and L. L. Campbell, "The Kullback-Leibler divergence rate between Markov sources," *IEEE Transactions on Information Theory*, Vol. 50, pp. 917–921, May 2004.

[54] A. F. Martin and M. A. Przybocki, "NIST 2003 language recognition evaluation," in *Proceedings of the European Conference on Speech Communication and Technology (EUROSPEECH)*, pp. 1341–1344, Sep. 2003.

[55] A. F. Martin and A. N. Le, "The current state of language recognition: NIST 2005 evaluation results," in *Proceedings of the IEEE Odyssey 2006 Speaker and Language Recognition Workshop*, Jun. 2006.

[56] P. Matějka, L. Burget, P. Schwarz, and J. Černocký, "Brno University of Technology system for NIST 2005 Language Recognition Evaluation," in *Proceedings of the IEEE Odyssey 2006 Speaker and Language Recognition Workshop*, Jun. 2006.

[57] W. Campbell, T. Gleason, J. Navratil, D. Reynolds, W. Shen, E. Singer, and P. Torres-Carrasquillo, "Advanced language recognition using cepstra and phonotactics: MITLL system performance on the NIST 2005 Language Recognition Evaluation," in *Proceedings of the IEEE Odyssey 2006 Speaker and Language Recognition Workshop*, Jun. 2006.

[58] M. A. Zissman, "Comparison of four approaches to automatic language identification of telephone speech," *IEEE Transactions on Speech and Audio Processing*, Vol. 4, pp. 31–44, Jan. 1996.

[59] Y. K. Muthusamy, E. Barnard, and R. A. Cole, "Reviewing automatic language identification," *IEEE Signal Processing Magazine*, Vol. 11, No. 4, pp. 33–41, 1994.

[60] F. Thollard, P. Dupont, and C. de la Higuera, "Probabilistic DFA inference using Kullback-Leibler divergence and minimality," in *Proceedings of the International Conference on Machine Learning*, pp. 975–982, Morgan Kaufmann, 2000.

[61] M. Brand, "Structure learning in conditional probability models via an entropic prior and parameter extinction," *Neural Computation*, Vol. 11, pp. 1155–1182, July 1999.

[62] K. Sage and H. Buxton, "Joint spatial and temporal structure learning for task based control," in *Proceedings of the International Conference on Pattern Recognition*, pp. 48–51, 2004.

[63] K. Sage, A. J. Howell, H. Buxton, and A. Argyros, "Learning temporal structure for task based control," *Image and Vision Computing*, 2006. Article in Press available at `doi:10.1016/j.imavis.2005.08.010`.

[64] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, Vol. 27, No. 3, pp. 379–423, 623–656, 1948.

[65] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. Available at `http://www.inference.phy.cam.ac.uk/mackay/itila`.

[66] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley, second ed., 2006.

[67] L. G. Kraft, Jr., "A device for quantizing, grouping, and coding amplitude-modulated pulses," Master's thesis, MIT, 1949. Available at `http://hdl.handle.net/1721.1/12390`.

[68] B. McMillan, "Two inequalities implied by unique decipherability," *IRE Transactions on Information Theory*, Vol. 2, pp. 115–116, 1956.

[69] S. Kullback and R. A. Leibler, "On information and sufficiency," *Annals of Mathematical Statistics*, Vol. 22, pp. 79–86, 1951.

[70] R. J. Solomonoff, "A formal theory of inductive inference I, II," *Inform. Control*, Vol. 7, pp. 1–22, 224–254, 1964.

[71] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Prob. Inform. Transmission*, Vol. 1, pp. 4–7, 1965.

[72] T. M. Cover, P. Gacs, and R. M. Gray, "Kolmogorov's contributions to information theory and algorithmic complexity," *Annals of Probability*, Vol. 17, pp. 840–865, July 1989.

[73] M. Li and P. M. B. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, second ed., 1997.

[74] G. J. Chaitin, "On the lengths of programs for computing binary sequences," *Journal of the Association for Computing Machinery*, Vol. 13, pp. 547–569, 1966.

[75] A. Gammerman and V. Vovk, eds., *Computer Journal (special issue on Kolmogorov complexity)*, Vol. 42, 1999.

[76] J. Rissanen, "A universal data compression system," *IEEE Transactions on Information Theory*, Vol. 29, No. 5, pp. 656–664, 1983.

[77] J. Rissanen, "Universal coding, information, prediction, and estimation," *IEEE Transactions on Information Theory*, Vol. 30, pp. 629–636, July 1984.

[78] J. Rissanen, "Lectures on statistical modeling theory." Unpublished note, available at `http://www.mdl-research.org/pub/lectures.pdf`, Aug. 2005.

[79] K. P. Burnham and D. R. Anderson, *Model Selection and Multimodel Inference*. Springer-Verlag, 2002.

[80] P. Cheeseman and J. Stutz, "Bayesian classification (AutoClass): theory and results," in *Advances in Knowledge Discovery and Data Mining* (U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds.), pp. 153–180, MIT Press, 1996.

[81] C. S. Wallace and D. L. Dowe, "MML clustering of multi-state, Poisson, von Mises circular and Gaussian distributions," *Statistics and Computing*, Vol. 10, pp. 73–83, Jan. 2000.

[82] C. S. Wallace and D. L. Dowe, "Intrinsic classification by MML—the Snob program," in *Proceedings of the Seventh Australian Joint Conference on Artificial Intelligence*, (Singapore), pp. 37–44, World Scientific, Nov. 1994. Available at `http://www.csse.monash.edu.au/~dld/Publications/1994/Wallace+Dowe1994_IntrinsicClassification_MML_AI94.Snob.pdf`.

[83] M. H. Hansen and B. Yu, "Model selection and the principle of minimum description length," *Journal of the American Statistical Association*, Vol. 96, pp. 746–774, 2001.

[84] P. Grünwald, "A tutorial introduction to the minimum description length principle," in *Advances in Minimum Description Length: Theory and Applications* (P. Grünwald, I. J. Myung, and M. Pitt, eds.), pp. 3–80, MIT Press, 2005.

[85] P. Grünwald, I. J. Myung, and M. Pitt, eds., *Advances in Minimum Description Length: Theory and Applications*. MIT Press, 2005.

[86] C. S. Wallace, *Statistical and Inductive Inference by Minimum Message Length*. Information Science and Statistics, Springer-Verlag, 2005.

[87] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*. Singapore: World Scientific, 1989.

[88] J. J. Oliver and D. Hand, "Introduction to minimum encoding inference," Tech. Rep. TR4-94, Statistics Department, Open University, Sept. 1994. Available at `http://citeseer.ist.psu.edu/oliver94introduction.html`.

[89] J. W. Comley and D. L. Dowe, "Minimum message length and generalized Bayesian nets with asymmetric languages," in *Advances in Minimum Description Length: Theory and Applications* (P. Grünwald, I. J. Myung, and M. Pitt, eds.), ch. 11, pp. 265–294, MIT Press, 2005.

[90] J. J. Oliver and R. A. Baxter, "MML and Bayesianism: similarities and differences," Tech. Rep. TR206, Computer Science Department, Monash University, 1994. Available at `http://citeseer.ist.psu.edu/oliver94mml.html`.

[91] S. C. Tornay, *Ockham: Studies and Selections*. La Salle, Illinois: Open Court, 1938.

[92] J. R. Norris, *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 1998.

[93] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*. Springer-Verlag, 1993. Available at `http://probability.ca/MT/`.

[94] P. Z. Peebles, *Probability, Random Variables, and Random Signal Principles*. McGraw-Hill, fourth ed., 2001.

[95] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines—Part II," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 27, pp. 1026–1039, July 2005.

[96] R. M. Neal, "Probabilistic inference using Markov chain Monte Carlo methods," Tech. Rep. CRG-TR-93-1, Dept. of Computer Science, University of Toronto, 1993. Available at `http://www.cs.toronto.edu/~radford/ftp/review.pdf`.

[97] S. M. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 35, pp. 400–401, Mar. 1987.

[98] D. Ron, Y. Singer, and N. Tishby, "Learning probabilistic automata with variable memory length," in *Computational Learning Theory*, Vol. 7, pp. 35–46, 1994.

[99] D. Ron, *Automata learning and its applications*. Ph.D. dissertation, School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel, 1995.

[100] L. G. Valiant, "A theory of the learnable," *Comm. ACM*, Vol. 27, pp. 1134–1142, 1984.

[101] A. B. Poritz, "Hidden Markov models: a guided tour," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 1, pp. 7–13, Apr. 1988.

[102] B. H. Juang and L. R. Rabiner, "Hidden Markov models for speech recognition," *Technometrics*, Vol. 33, pp. 251–272, Aug. 1991.

[103] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state Markov chains," *Annals of Mathematical Statistics*, Vol. 37, pp. 1554–1563, 1966.

[104] L. E. Baum and J. Eagon, "An inequality with applications to statistical estimation for probabilistic functions of a Markov process and to a model for ecology," *Bulletin of the American Meteorological Society*, Vol. 48, pp. 360–363, 1967.

[105] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains," *Annals of Mathematical Statistics*, Vol. 41, pp. 164–171, 1970.

[106] L. E. Baum, "An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process," *Inequalities*, Vol. 3, pp. 1–8, 1972.

[107] X. Huang, K.-F. Lee, and H.-W. Hon, "On semi-continuous hidden Markov modeling," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 689–692, 1990.

[108] X. Huang, Y. Ariki, and M. Jack, *Hidden Markov models for Speech Recognition*. Edinburgh University Press, 1990.

[109] G. D. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, Vol. 61, pp. 268–278, Mar. 1973.

[110] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[111] R. Bakis, "Continuous speech word recognition via centisecond acoustic states," in *Proc. ASA Meeting*, (Washington DC), Apr. 1976.

[112] F. Jelinek, "Continuous speech recognition by statistical methods," *Proceedings of the IEEE*, Vol. 64, pp. 532–536, Apr. 1976.

[113] H. O. Hartley, "Maximum likelihood estimation from incomplete data," *Biometrics*, Vol. 14, pp. 174–194, 1958.

[114] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society, series B*, Vol. 39, No. 1, pp. 1–38, 1977.

[115] G. J. MacLachlan and T. Krishnan, *The EM Algorithm and Extensions*. Wiley, 1997.

[116] M. Tanner, *Tools for Statistical Inference*. Springer-Verlag, third ed., 1996.

[117] R. M. Neal and G. E. Hinton, "A view of the EM algorithm that justifies incremental, sparse, and other variants," in *Learning in Graphical Models* (M. I. Jordan, ed.), pp. 355–368, MIT Press, 1998. Originally released as technical report in 1993.

[118] T. Minka, "Expectation-Maximization as lower bound maximization." Unpublished note, available at `http://research.microsoft.com/~minka/papers/em.html`.

[119] F. Dellaert, "The Expectation Maximization algorithm," Tech. Rep. GIT-GVU-02-20, College of Computing, Georgia Institute of Technology, Feb. 2002. Available at `http://www.cc.gatech.edu/~dellaert/em-paper.pdf`.

[120] W. Byrne, "Information geometry and maximum likelihood criteria," in *Proceedings of the Conference on Information Sciences and Systems*, (Princeton, USA), Princeton University, 1996. Available at `http://mi.eng.cam.ac.uk/~wjb31/ppubs/igmlc.ciss96.pdf`.

[121] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1, pp. 281–297, University of California Press, 1967.

[122] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Transactions on Communications*, Vol. COM-28, pp. 84–95, Jan. 1980.

[123] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*. Prentice Hall, 1997.

[124] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second ed., 2000.

[125] M. Sipser, *Introduction to the Theory of Computation*. Boston, Massachusetts: Thomson Course Technology, second ed., 2005.

[126] A. Paz, *Introduction to Probabilistic Automata*. Academic Press, 1971.

[127] K.-S. Fu and T. L. Booth, "Grammatical inference: introduction and survey—Part II," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-5, pp. 409–423, July 1975.

[128] P. Dupont, F. Denis, and Y. Esposito, "Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms," *Pattern Recognition*, Vol. 38, pp. 1349–1371, Sep. 2005.

[129] P. García and E. Vidal, "Inference of k-testable languages in the strict sense and application to syntactic pattern recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, pp. 920–925, Sept. 1990.

[130] N. Abe and M. Warmuth, "On the computational complexity of approximating distributions by probabilistic automata," *Machine Learning*, Vol. 9, pp. 205–260, 1992.

[131] M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie, "On the learnability of discrete distributions," in *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*, (New York, NY, USA), pp. 273–282, ACM Press, 1994.

[132] D. Gillman and M. Sipser, "Inference and minimization of hidden Markov chains," in *Computational Learning Theory*, Vol. 7, pp. 147–158, 1994.

[133] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines—Part I," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 27, pp. 1013–1025, July 2005.

[134] E. Fredkin, "Trie memory," *Communications of the Association for Computing Machinery*, Vol. 3, pp. 490–499, Sep. 1960.

[135] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the Association for Computing Machinery*, Vol. 23, No. 2, pp. 262–272, 1976.

[136] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, Vol. 14, No. 3, pp. 249–260, 1995.

[137] G. Bejerano, Y. Seldin, H. Margalit, and N. Tishby, "Markovian domain fingerprinting: statistical segmentation of protein sequences," *Bioinformatics*, Vol. 17, No. 10, pp. 927–934, 2001.

[138] A. Stolcke and S. Omohundro, "Best-first model merging for hidden Markov model induction," Tech. Rep. TR-94-003, ICSI, 1947 Center St., Berkeley, CA, 94704, 1994. Available at `ftp://ftp.icsi.berkeley.edu/pub/techreports/1994/tr-94-003.ps.gz`.

[139] A. Stolcke, *Bayesian learning of probabilistic language models*. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1994.

[140] D. J. C. MacKay and L. C. B. Peto, "A hierarchical Dirichlet language model," *Natural Language Engineering*, Vol. 1, No. 3, pp. 1–19, 1995.

[141] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, Vol. 58, pp. 13–30, 1963.

[142] C. de la Higuera and F. Thollard, "Identification in the limit with probability one of stochastic deterministic finite automata," in *Proceedings of the Fifth International Colloquium on Grammatical Inference*, Vol. 1891 of *Lecture Notes in Computer Science*, pp. 141–156, Springer-Verlag, 2000.

[143] J. Takami and S. Sagayama, "A successive state splitting algorithm for efficient allophone modeling," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 1, pp. 573–576, 1992.

[144] J. A. du Preez and D. M. Weber, "Automatic language recognition using high-order HMMs," in *Proceedings of the International Conference on Spoken Language Processing*, Vol. 2, pp. 117–120, 1998.

[145] J. A. du Preez and D. M. Weber, "Efficient high-order hidden Markov modelling," in *Proceedings of the International Conference on Spoken Language Processing*, Vol. 7, pp. 2911–2914, 1998.

[146] T. Edgoose and L. Allison, "MML Markov classification of sequential data," *Statistics and Computing*, Vol. 9, pp. 269–278, 1999.

[147] A. Galata, N. Johnson, and D. Hogg, "Learning structured behaviour models using variable length Markov models," in *Proceedings of the IEEE International Workshop on Modelling People*, pp. 95–102, Sept. 1999.

[148] M. G. Roberts, *Local order estimating Markovian analysis for noiseless source coding and authorship identification*. Ph.D. dissertation, Stanford University, 1982.

[149] I. H. Witten and T. C. Bell, "The zero frequency problem: estimating the probabilities of novel events in adaptive text compression," *IEEE Transactions on Information Theory*, Vol. 37, No. 4, pp. 1085–1094, 1991.

[150] P. F. Brown, J. Cocke, S. A. Della Pietra, V. J. Della Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin, "A statistical approach to machine translation," *Computational Linguistics*, Vol. 16, pp. 79–85, Jun. 1990.

[151] S. F. Chen and J. T. Goodman, "An empirical study of smoothing techniques for language modeling," Tech. Rep. TR-10-98, Harvard University, Cambridge, Mass., Aug. 1998.

[152] T. C. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys*, Vol. 21, pp. 557–591, Dec 1989.

[153] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Prentice Hall, 1990.

[154] R. E. Krichevsky and V. K. Trofimov, "The performance of universal encoding," *IEEE Transactions on Information Theory*, Vol. 27, pp. 199–207, Mar. 1981.

[155] E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[156] W. A. Gale and K. W. Church, "What's wrong with adding one?," in *Corpus-Based Research into Language: In honour of Jan Aarts* (N. Oostdijk and P. de Haan, eds.), (Amsterdam), pp. 189–200, Rodopi, 1994.

[157] S. F. Chen and J. T. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech and Language*, Vol. 13, No. 4, pp. 359–393, 1999.

[158] A. V. Aho and M. E. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the Association for Computing Machinery*, Vol. 18, pp. 333–340, 1975.

[159] R. M. Gray, *Entropy and Information Theory*. Springer-Verlag, 1990. Available at `http://ee.stanford.edu/~gray/it.html`.

[160] G. H. Golub and C. F. van Loan, *Matrix Computations*. Baltimore, USA: The Johns Hopkins University Press, third ed., 1996.

[161] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the Web," Tech. Rep. 1999-66, Stanford University InfoLab, 1999. Available at `http://dbpubs.stanford.edu/pub/1999-66`.

[162] D. Angluin and M. Csűrös, "Learning Markov chains with variable length memory from noisy output," in *Computational Learning Theory*, Vol. 10, pp. 298–308, 1997.

[163] J. G. Proakis and M. Salehi, *Communication Systems Engineering*. Prentice Hall, second ed., 2002.

[164] L. R. Rabiner and B. H. Juang, *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[165] Y. Singer and M. K. Warmuth, "Training algorithms for hidden Markov models using entropy based distance functions," in *Advances in Neural Information Processing Systems* (M. Moser, M. I. Jordan, and T. Petsche, eds.), no. 9, pp. 641–647, MIT Press, 1996.

[166] M. N. Do and M. Vetterli, "Fast approximation of Kullback-Leibler distance for dependence trees and hidden Markov models," *IEEE Signal Processing Letters*, Vol. 10, pp. 115–118, Apr. 2003.

[167] J. Silva and S. Narayanan, "Upper bound Kullback-Leibler divergence for hidden Markov models with application as discrimination measure for speech recognition," in *Proceedings of the IEEE International Symposium on Information Theory*, pp. 2299–2303, Jul. 2006.

[168] M. A. Zissman, "Language identification using phoneme recognition and phonotactic language modelling," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, (Detroit, USA), pp. 3503–3506, 1995.

[169] D. A. van Leeuwen and N. Brümmer, "Channel-dependent GMM and multi-class logistic regression models for language recognition," in *Proceedings of the IEEE Odyssey 2006 Speaker and Language Recognition Workshop*, Jun. 2006.

[170] A. Canavan and G. Zipperlen, "The CALLFRIEND corpus," 1996. Available from the Linguistic Data Consortium, Catalogue No. LDC96S46 to LDC96S60.

[171] B. S. Atal, "Effectiveness of linear prediction characteristics of the speech wave for automatic speaker identification and verification," *Journal of the Acoustical Society of America*, Vol. 55, pp. 1304–1312, 1974.

[172] J. D. Markel and A. H. Gray, *Linear Prediction of Speech*. Springer-Verlag, 1976.

[173] X. Huang, A. Acero, and H. W. Hon, *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice Hall, 2001.

[174] J. J. Godfrey and E. Holliman, "The Switchboard-1 Release 2 corpus," 1997. Available from the Linguistic Data Consortium, Catalogue No. LDC97S62.

[175] A. Higgins and D. Vermilyea, "The KING Speaker Verification corpus," 1995. Available from the Linguistic Data Consortium, Catalogue No. LDC95S22.

[176] M. A. Zissman, "Predicting, diagnosing and improving automatic language identification performance," in *Proceedings of the European Conference on Speech Communication and Technology (EUROSPEECH)*, Vol. 1, (Rhodes, Greece), pp. 51–54, Sept. 1997.

[177] D. Benedetto, E. Caglioti, and V. Loreto, "Language trees and zipping," *Physical Review Letters*, Vol. 88, Jan 2002. E-print available at `arXiv:cond-mat/0108530`.

[178] J. Goodman, "Extended comment on *Language trees and zipping*," 2002. E-print available at `arXiv:cond-mat/0202383`.

[179] D. V. Khmelev and W. J. Teahan, "On an application of relative entropy," 2003. E-print available at `arXiv:cond-mat/0205521`.

[180] D. Benedetto, E. Caglioti, and V. Loreto, "On J. Goodman's comment to *Language trees and zipping*," 2002. E-print available at `arXiv:cond-mat/0203275`.

[181] R. Begleiter, R. El-Yaniv, and G. Yona, "On prediction using variable order Markov models," *Journal of Artificial Intelligence Research*, Vol. 22, pp. 385–421, 2004.

[182] D. H. Wolpert, "Determining whether two data sets are from the same distribution," in *Proceedings of the International Workshop on Maximum Entropy and Bayesian Methods* (K. M. Hanson and R. N. Silver, eds.), no. 15, (Santa Fe, USA), Springer-Verlag, Aug. 1995.

[183] C. Kermorvant and P. Dupont, "Improved smoothing for probabilistic suffix trees seen as variable order Markov chains," in *Proceedings of the 13th European Conference on Machine Learning (ECML 2002)*, Vol. 2430 of *Lecture Notes In Computer Science*, pp. 185–194, Springer-Verlag, 2002.

[184] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 181–184, May 1995.

[185] Y.-L. Chow and R. Schwartz, "The N-Best algorithm: an efficient procedure for finding top N sentence hypotheses," in *HLT '89: Proceedings of the DARPA Speech and Natural Language Workshop*, (Morristown, NJ, USA), pp. 199–202, Association for Computational Linguistics, 1989. E-print available at `http://www.aclweb.org/anthology/H89-2027`.

[186] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden Markov model: analysis and applications," *Machine Learning*, Vol. 32, No. 1, pp. 41–62, 1998.