Automata learning: from probabilistic to quantum

Chu, W.

Chapter 2

Probabilistic automata

This chapter introduces the basics of finite automata, a simple type of machine that can be used to recognize patterns. In particular, we discuss deterministic and non-deterministic finite automata and the corresponding probabilistic versions. We cover their definitions, how they work, their applications, and the relationship between the two models. Probabilistic finite automata are related to Hidden Markov Models, which are widely used models for probabilistic sequences. We also discuss the relationship between Hidden Markov Models (HMMs) and probabilistic automata (PAs). HMMs and PAs are particularly significant in speech recognition [7, 68], natural language processing [62], bioinformatics [8], and many other domains involving discrete time-series analysis. They allow us to predict, understand, and model sequential data, aiding in decision-making, pattern recognition, and forecasting [66, 98, 17, 1].

When working with probabilistic models, it is essential to provide a quantitative measure to assess, for example, the model performance or to select the appropriate model among many. There are several distances intended to measure how similar or dissimilar two probability distributions are. Here, we focus on two distances: the Euclidean distance L_2 and another distance based on the confusion matrix, a tabular representation commonly used in machine learning to evaluate the accuracy of a classification algorithm based on the counts of the predicted true positive, true negative, false positive, and false negative. Both distances are used in the next chapter to compare the distributions generated by probabilistic automata.

While all concepts we treat in this chapter are standard, we also present a novel algorithm to compute the Euclidean distance between two regular distributions using weighted graphs.

2.1 Finite automata

Finite automata are a fundamental concept in the field of theoretical computer science. They are used to recognize whether a given string belongs to a given language, making them an essential tool for language recognition and processing. As such, they play a crucial role in many areas, including biological sequences representation [32], aural pattern recognition [100], sequences classification [113], and image recognition [77].

In this section, we recall the two most basic automata models: deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs). We will provide their formal definitions and for the sake of completeness, we show how they are related to each other.

2.1.1 Nondeterministic and deterministic finite automata

An NFA consists of a finite set of states, a finite set of input symbols, a transition function, a set of initial states, and a set of final states. In an NFA, the transition function maps a state and an input symbol to a set of possible next states. This non-determinism allows the NFA to explore multiple paths simultaneously, branching out and potentially backtracking as it processes the input.

Definition 1. ***Nondeterministic finite automaton*** *A nondeterministic finite automaton (NFA) is a 5-tuple* $A = \langle \Sigma, Q, I, F, \delta \rangle$*, where*

- Σ *is a finite alphabet,*
- Q *is a finite set of states,*
- $I : Q \to \{0,1\}$ *maps to* 1 *all states that are initial,*
- $F : Q \to \{0,1\}$ *maps to* 1 *all states that are final,*
- $\delta : Q \times \Sigma \to \{0,1\}^Q$ *is the transition function.*

To check if a given string $x \in \Sigma^*$ is accepted by an NFA we need to calculate the set of states that can be reached by all possible paths that can be taken by following one action of x at the time. To this purpose, we use the extension δ^* of δ that takes as input a state, a string, and returns the set of states that can be reached from them. Formally, the extended transition function $\delta^*$is defined recursively, for all $q \in Q$, as follows:

- $\delta^*(q, \varepsilon)(q) = 1$
- $\delta^*(q, ax)(q'') = 1$ if there exist q' such that $\delta(q,a)(q') = 1$ and $\delta^*(q', x)(q'') = 1$.

Here ε is the empty string, $a \in \Sigma$ and $x \in \Sigma^*$. The acceptance of a string by an NFA is determined by whether there exists at least one computation path that leads to a final state when the entire input is consumed. More formally, given an NFA A and a state $q \in Q$, we say that the language $L(A,q)$ consists of all strings for which there exists a state q' such that $\delta^*(q,x)(q') = 1$, and $F(q') = 1$. The language $L(A)$ accepted by an NFA A is the union of all $L(A,q)$ for states q such that $I(q) = 1$. A language L is said to be regular if there exists an NFA A that accepts exactly the language L. A path π for an accepting string $x = a_1 \dots a_n \in L(A)$ is a sequence of states $q_0 \dots q_n$ on Q^* such that $\delta(q_i, a_{i+1})(q_{i+1}) = 1$ for all $0 \leqslant i \leqslant n-1$, starting from an initial state, i.e. $I(q_0) = 1$, and ending in a final state i.e. $F(q_n) = 1$. We use $Paths(x)$ to denote the set of all accepting paths for a given string x. Note that the set $Paths(x)$ is necessarily finite. An accepting path contains a cycle if it contains the same state twice, that is, there exists different i and j such that $q_i = q_j$.

Unlike an NFA, a deterministic finite automaton (DFA) has a unique next state for each input symbol in each state. Accordingly, we say that an NFA A is deterministic (DFA) when the following holds:

- $|\{q | I(q) = 1\}| = 1$ (one single initial state),
- $\forall q \in Q$ and $a \in \Sigma$, $|\{q' | \delta(q,a)(q')\} = 1| \leqslant 1$ (at most one next state).

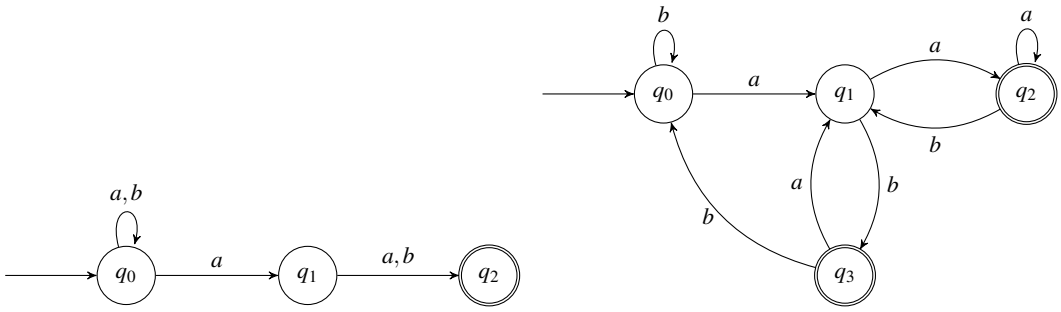
Typically, we denote by q_0 the unique initial state of a DFA. This deterministic (and complete) behavior of a DFA makes it easier to implement, at cost, however, of flexibility and conciseness when compared to equivalent NFAs. Note that, for a DFA, $|Paths(x)| = 1$ for every accepted string x.

Every DFA is an NFA. Conversely, for any NFA we can construct a DFA that accepts the same language [96]. However, NFAs are always smaller than or equal to their equivalent DFAs in terms of the number of states. The construction involves creating a DFA where each state represents a subset of the original NFA's states. The transitions in the new DFA are determined by the transitions of the original NFA, considering the set of states that can be reached by following those transitions. The final states correspond to subsets of the original NFAs containing at least one final state.

For example, consider the language L over $\Sigma = \{a, b\}$ for which the 2nd symbol before the end is an a. This language can be recognized by an NFA with 3 states, whereas a DFA needs at least 4 states. See Fig. 2.1 for the two automata. This example can be generalized to the nth symbol before the last one to notice that the DFA will need exponentially more states than an NFA.

An advantage of DFAs over NFAs is that for a DFA it is possible to effectively construct another DFA that is minimal [57], meaning that (1) it has the fewest number of states among

all DFAs that recognize the same language, and (2) each state is essential, that is, reachable from the initial state and leading to a final state.



(a) An NFA. (b) A DFA.

Fig. 2.1 Automata accepting strings over $\Sigma = \{a,b\}$ for which the 2nd symbol before the end is an a.

2.2 Probabilistic finite automata

Unlike ordinary languages that classify strings based on whether they belong or not to the language, a probabilistic language associates probabilities to strings enabling the representation of uncertain or stochastic processes. As such, a probabilistic language is a (discrete) distribution $D\colon \Sigma^* \to [0,1]$ mapping each string $x \in \Sigma^*$ a probability $P(x)$ that satisfies the following constraint:

$$\sum_{x \in \Sigma^*} P(x) = 1$$

By allowing transitions of an NFA to have probabilities on each input, and considering the choice of an initial and final state to be probabilistic, one obtains probabilistic finite automata:

Definition 2. ***Probabilistic finite automaton*** *A probabilistic finite automaton (PFA) is a* $5-$*tuple* $A = \langle \Sigma, Q, I_p, F_p, \delta_p \rangle$*, where:*

- Σ *is a finite alphabet,*
- Q *is a finite set of states,*
- $I_p : Q \to (\mathbb{Q} \cap [0,1])$ *is the initial probability,*
- $F_p : Q \to (\mathbb{Q} \cap [0,1])$ *is the final probability,*
- $\delta_p : Q \times \Sigma \to (\mathbb{Q} \cap [0,1])^Q$ *is the transition function.*

Where I_p, F_p and δ_p must satisfy the following two conditions:

$$\sum_{q \in Q} I_p(q) = 1,$$

$$\forall q \in Q, F_p(q) + \sum_{a \in \Sigma, q' \in Q} \delta_p(q,a)(q') = 1.$$

The first condition requires to have a distribution of initial states, while the second condition says that the selection of either finishing or proceeding to the next state for any possible symbol is probabilistic. Technically, these two conditions are necessary to guarantee that the language accepted by a PFA is a distribution.

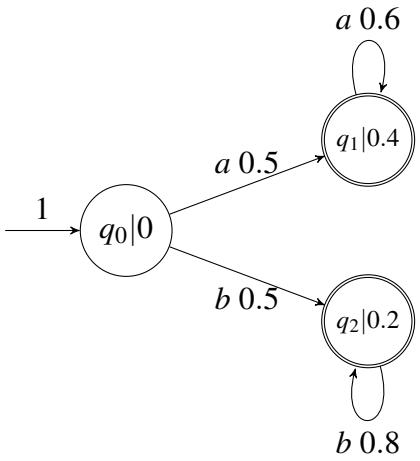


Fig. 2.2 A PFA

By adding probabilities to transitions, PFAs offer a natural way to capture and represent uncertainty. They serve as building blocks for various machine learning algorithms handling uncertain and noisy data or that make informed decisions under uncertainty.

The support of a PFA $A = \langle \Sigma, Q, I_p, F_p, \delta_p \rangle$ is the NFA $supp(A) = \langle \Sigma, Q, I, F, \delta \rangle$, where $I(q) = 1$ if and only $I_p(q) > 0$, $F(q) = 1$ if and only if $F_p(q) > 0$, and $\delta(q,x)(q') = 1$ if and only if $\delta_p(q,x)(q') > 0$. An accepting path π of A for a string $x = a_1 \dots a_n$ is a sequence of $n+1$ states $q_0 \dots q_n$ such that:

- $I_p(q_0) > 0$,
- $\delta_p(q_i, a_{i+1})(q_{i+1}) > 0$, where $0 \leqslant i < n$,
- $F_p(q_n) > 0$.

In other words, a path is accepted by a PFA if and only if it is accepted by the NFA of its support.

The probability generated by a probabilistic automaton for a string x is calculated by considering each accepting path for x as independent, thus summing up their probabilities. For a given accepting path, the probability is determined by starting with the probability of the initial state, and for each transition taken, multiplying the probability of reaching the current state by the probability of the transition until all string is consumed, and thus finally we can multiply with the probability of ending in a final state.

More precisely, given a PFA A and a path $\pi = q_0 \dots q_n$ for an accepting string $x = a_1 \dots a_n$ in $supp(A)$, we denote the probability $I_p(q_0)$ of its initial state q_0 by $i_p(\pi)$, the probability $F_p(q_n)$ of the final state q_n by $e_p(\pi)$, and the product of all probabilities of transitions along this path by $\delta_p(\pi)$. That is, inductively, if x is an empty string then π consists of one state only, say $\pi = q_0$ and then $\delta_p(\pi) = 1$. Otherwise,

$$\delta_p(\pi) = \Pi_{i=0}^{n-1} \delta_p(q_i, a_{i+1})(q_{i+1})$$

Notably, $i_p(\pi)$, $e_p(\pi)$, and $\delta_p(\pi)$ are all strictly positive and smaller than or equal to 1 for every accepting path π of $supp(A)$. The probability of an accepting path $\pi \in Paths(x)$ for string x is $i_p(\pi) \cdot \delta_p(\pi) \cdot e_p(\pi)$, while the probability of a string x is defined by:

$$P_A(x) = \sum_{\pi \in Paths(x)} i_p(\pi) \cdot \delta_p(\pi) \cdot e_p(\pi).$$

We call P_A the probability distribution on Σ^* generated by A. If a PFA A is consistent then it is easy to show [39] that P_A is indeed a distribution on Σ^*, that is $\sum_{x \in \Sigma^*} P_A(x) = 1$. Here a PFA A is said to be consistent if all its states are essential in $supp(A)$, that is, they appear in at least one accepting path. We say that a distribution on σ^* is regular if generated by a PFA A. In general, not all distributions on regular languages are regular [95].

Note that if we do not consider the final probability $e(\pi)$ when computing $P_A(x)$ for a string $x \in \Sigma^*$, we are then computing the probability of A generating an infinite string with prefix x:

$$\bar{P}_A(x) = \sum_{|\pi|=|x|+1, \pi \in Q^*} i_p(\pi) \delta_p(\pi).$$

Note that we consider all paths here of length $|x| + 1$ instead of only accepting paths. This is useful for non-terminating PFAs, i.e. PFA with $F_p(q) = 0$ for all states q. Note that for these automata the probability $P_A(x)$ accepting x is 0, but $\bar{P}_A(x)$ needs not, as x is seen as only a prefix of an infinite string.

Example 1. *Given* $\Sigma = \{a, b\}$*, the distribution* $P(a^n) = 0.4 \cdot 0.6^{n-1} \cdot 0.5, P(b^n) = 0.2 \cdot 0.8^{n-1} \cdot 0.5$ *assigning* 0 *to all other strings is regular. It can be generated by the PFA shown in Figure 2.2.*

Note that we used only rational numbers as probabilities so to work with computable algorithms. As a consequence, even distribution with finite support such as $P(a) = \frac{\pi}{4}$, $P(b) = 1 - \frac{\pi}{4}$ is not regular.

2.2.1 The forward and backward algorithms

There are several algorithms commonly used to compute the probability of a string given a PFA. Here we describe two classical ones: the forward and the backward algorithms [10].

The forward algorithm calculates the forward probabilities, representing the probability of being in a particular state at each position in the string. The final probability is obtained by summing the forward probabilities of the final states at the last position. More specifically, assuming the states of a PFA are $Q = \{q_1, \ldots, q_n\}$ then given a string $x = a_0 \cdots a_m$, the algorithm computes a table of values $F[i][j]$, which represents the probability of reaching state q_j after processing the i-th symbol of the string. The algorithm works as follows: first, initialize $F[0][j]$ to $I_p(q_j)$ for all $1 \leq j \leq n$ (lines 1-2), then for each $i \leq m$, and for each $1 \leq j \leq n$, compute the $F[i][j]$ (lines 7-13). Finally, by multiplying $F[n][j]$ by the final probability of the state q_j and summing up all alternatives, we get the actual probability of the string x (lines 15-17).

A complementary approach is taken by the backward algorithm, which computes the probabilities of being in each state at each position in the string from right to left, starting from the final position.

While the forward algorithm computes the probability of being in a particular state at a given position in a string, given the symbols observed up to that position, the backward algorithm focuses on the probability of generating the remaining symbols of the string, given a specific position. This is called smoothing and plays an important role in the Baum-Welch algorithm to learn the parameter probabilities of a PFA.

Given a string x, the algorithm computes a table of values $B[i][j]$, which represents the probability of reaching state q_j after processing the suffix $a_i + 1 \cdot a_m$ of the string x. The algorithm works as follows: first, it initializes $B[m][j]$ to $F_p(q_j)$ for all $1 \leq j \leq n$ (line 2-4). Then in reverse order for each $m \geq i \geq 0$, and for each $1 \leq j \leq n$, it computes the backward probability $B[i][j]$ as the sum of all $B[i+1][k]$ weighted by the probability of the transition from q_j to state q_k labeled by a_i (line 7-13). Finally, the sum of backward all probabilities $B[0][j]$ multiplied by the initial ones gives the actual probability of the string (line 15-17).

Algorithm 1 Forward Algorithm

Input: A PFA A = $\langle \Sigma, Q, I_p, F_p, \delta_p \rangle$ and a string $x = a_1 \dots a_n$
Output: the probability $P_A(x)$ of string x

for $1 \leqslant j \leqslant |Q|$ **do**
  $F[0][j] = I_p(q_j)$
  for $1 \leqslant i \leqslant n$ **do**
    $F[i][j] = 0$
  end for
end for
for $1 \leqslant i \leqslant n$ **do**
  for $1 \leqslant j \leqslant |Q|$ **do**
    for $1 \leqslant k \leqslant |Q|$ **do**
      $F[i][j] = F[i][j] + F[i-1][k] \cdot \delta_p(q_k, a_i)(q_j)$
    end for
  end for
end for
$P_A(x) = 0$
for $1 \leqslant j \leqslant |Q|$ **do**
  $P_A(x) = P_A(x) + F[n][j] \cdot F_p(q_j)$
end for
return the probability $P_A(x)$

Algorithm 2 Backward Algorithm

Input: A PFA A = $\langle \Sigma, Q, I_p, F_p, \delta_p \rangle$ and a string $x = a_1 \dots a_n$
Output: the probability $P_A(x)$ of string x

for $1 \leqslant j \leqslant |Q|$ **do**
  $B[n][j] = F_p(q_j)$
  for $i : 1 \leqslant i \leqslant n$ **do**
    $B[i][j] = 0$
  end for
end for
for $0 \leqslant i \leqslant n-1$ **do**
  for $1 \leqslant j \leqslant |Q|$ **do**
    for $1 \leqslant k \leqslant |Q|$ **do**
      $B[i][j] = B[i][j] + B[i+1][k] \cdot \delta(q_j, a_i)(q_k)$
    end for
  end for
end for
$P_A(x) = 0$
for $1 \leqslant j \leqslant |Q|$ **do**
  $P_A(x) = P_A(x) + B[0][j] \cdot I_p(q_j)$
end for
return the probability $P_A(x)$

The time complexity of both the forward and backward algorithms is $\mathcal{O}(|x| \cdot |\delta_p|)$, where $|\delta_p|$ is the size of the transition function. This is a significant improvement over the brute-force approach of enumerating all possible paths, which has exponential complexity $\mathcal{O}(|Q|^{|x|})$ [10].

2.2.2 Deterministic probabilistic finite automata

As a special case of PFAs, a deterministic probabilistic finite automaton (DPFA) satisfies the additional constraints that its support is a DFA:

- $|\{q|I_p(q) > 0\}| \leqslant 1$,
- $\forall q \in Q, \forall a \in \Sigma, |\{q' \mid \delta_p(q,a)(q') > 0\}| \leq 1$.

Differently from PFAs, a DPFA has at most one initial state, and for each state there is at most one transition to the next state labeled by an alphabet symbol and with nonzero probability. These constraints ensure that, given any string, there is at most one accepting path through the automaton.

The probabilistic choice of PFA introduces uncertainty as different transitions may be taken for the same symbol with different probabilities. While we have seen that this non-determinism in an NFA can be eliminated, we show next that not every regular distribution can be generated by a DPFA.

Theorem 1. *[39] The class of distributions generated by DPFAs is a proper subclass of regular distributions.*

Proof. Let A be a probabilistic automaton and define $\rho(x)$ as follows:

$$\forall x \in \Sigma^*,\ \rho(x) = \begin{cases} P_A(x)/\bar{P}_A(x) & if\ \bar{P}_A(x) > 0, \\ 0 & otherwise. \end{cases}$$

If A is a DPFA, then the set $\{\rho(x)|x \in \Sigma^*\}$ is necessarily finite as it is bounded by the number of states with the final probability strictly positive.

Consider now the PFA described in Figure 2.3. We have:

$$\begin{aligned} \rho(a^n) &= P(a^n)/\bar{P}(a^n) \\ &= \frac{0.5 \cdot 0.6^n \cdot 0.4 + 0.5 \cdot 0.8^n \cdot 0.2}{0.5 \cdot 0.6^n + 0.5 \cdot 0.8^n} \\ &= 0.2 + \frac{0.4}{(1+2^n)}, \end{aligned}$$

which is a strictly decreasing series for strictly increasing values of n. Therefore the set $\{\rho(x) \mid x \in \Sigma^*\}$ cannot be finite. □ □

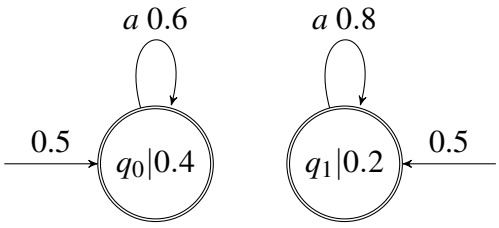


Fig. 2.3 A PFA that cannot be represented by a DPFA.

2.3 Hidden Markov Models

A Hidden Markov Model (HMM) is a statistical model alternative to PFA to model and analyze sequential data. An HMM is composed of two main components: a set of observable symbols or emissions and a set of hidden states. The hidden states cannot be directly observed, but they emit observable symbols with certain probabilities. These emissions are dependent on the current hidden state but not on the previous history of states[74]. Transitions between hidden states indicate the likelihood of moving from one state to another. HMMs are used in the field of machine learning, with applications in computational biology [8, 41], speech recognition [7, 60, 68, 97], and information extraction [107].

Definition 3. *A discrete HMM (with state emission) is a* $5-$*tuple* $M = \langle \Sigma, Q, A, B, i \rangle$*, where*

- Σ *is a finite alphabet,*
- Q *is a finite set of states,*
- $A : Q \times Q \to (\mathbb{Q} \cap [0,1])$ *is the probability of each transition such that*

$$\forall q \in Q, \sum_{q' \in Q} A(q,q') = 1,$$

- $B : Q \times \Sigma \to (\mathbb{Q} \cap [0,1])$ *is the emission probability of each letter on each state such that*

$$\forall q \in Q, \sum_{a \in \Sigma} B(q,a) = 1,$$

- $i : Q \to (\mathbb{Q} \cap [0,1])$ *is the initial probability such that*

$$\sum_{q \in Q} i(q) = 1.$$

Given an HMM $M = \langle \Sigma, Q, A, B, i \rangle$, a path π is a sequence defined on Q^*. For any path π, q_i denotes the ith state of π, and $|\pi|$ denotes the length of path. For any string $x = a_1 \cdots a_n \in \Sigma^*$ and any path $\pi \in Q^*$, the probabilities $P_M(x,\pi)$ and $P_M(x)$ are defined as follows:

$$\begin{aligned} P_M(x,\pi) &= \begin{cases} i(q_0)\prod_{i=0}^{n-2}[B(q_i,a_{i+1})A(q_i,q_{i+1})]B(q_{n-1},a_n) & if\ |x| = |\pi| > 0, \\ 1 & if\ |x| = |\pi| = 0, \\ 0 & otherwise. \end{cases} \\ P_M(x) &= \textstyle\sum_{\pi \in Q^*} P_M(x,\pi). \end{aligned}$$

Sometimes in an HMM the emission probability B of a current state q is given dependent not only on the symbol of the alphabet but also on the next state, that is, $B : Q \times \Sigma \to (\mathbb{Q} \cap [0,1])^Q$ such that

$$\forall q, q' \in Q, \sum_{a \in \Sigma} B(q,a)(q') = \begin{cases} 1 & if\ A(q,q') > 0, \\ 0 & otherwise, \end{cases}$$

These models are called HMM with transition emission (HMMT). For a HMMT, string $x = a_1 \cdots a_n \in \Sigma^*$ and any path $\pi = q_0 \cdots q_n \in Q^*$, the probability $P_M(x,\pi)$ is now defined as follows:

$$P_M(x,\pi) = \begin{cases} i(q_0)\prod_{i=0}^{|x|-1}[B(q_i,a_{i+1})(q_{i+1})A(q_i,q_{i+1})] & if\ |\pi| = |x| + 1, \\ 0 & otherwise. \end{cases}$$

No changes are needed in computing the probability $P_M(x)$ given the above $P_M(x,\pi)$.

Proposition 1. *[39] HMMs, HMMTs, and non-terminating PFAs (i.e. with no final state probabilities) are all equivalent in the sense that they recognize the same class of distributions.*

Proof. We will prove this proposition in three steps. First, we show that every HMMT can be transformed into an equivalent HMM. Second, we show that a PFA with no final probabilities is equivalent to an HMMT, and finally, we give a construction transforming an HMM to a PFA.

Let $M = \langle \Sigma, Q, A, B, i \rangle$ be an HMMT, and define $M' = \langle \Sigma, Q', A', B', i' \rangle$ to be an HMM, where:

- $Q' = \{(q,q') \in Q \times Q | A(q,q') > 0\}$.
- $A'((q,q')(q'',q''')) = \begin{cases} A(q'',q''') & q' = q'' \\ 0 & otherwise. \end{cases}$
- $B'((q,q'),a) = B(q,a)(q')$

- $i'((q,q')) = i(q) \cdot A(q,q')$.

The idea is that the states of M' represent pairs of states in Q that are connected by a strictly positive transition probability. Following the definition of A' the only interesting paths to considers in calculating $P_{M'}(x)$ are of the form $\pi' = (q_0,q_1)(q_1,q_2)\dots(q_{n-1},q_n)$ that are in one-to-on correspondence with path $\pi = q_0q_1\dots q_{n-1}q_n$ in M. Note that π' has length n while π has length $n+1$. We then have for any non-empty string $x = a_1\cdots a_n$ and path $\pi = q_0\dots q_n$ we have

$$\begin{aligned}
P_M(x,\pi) &= i(q_0)\prod_{i=0}^{|x|-1}[B(q_i,a_{i+1})(q_{i+1})A(q_i,q_{i+1})] \\
&= i(q_0)A(q_0,q_1)\prod_{i=0}^{|x|-2}[B(q_i,a_{i+1})(q_{i+1})A(q_{i+1},q_{i+2})]B(q_{n-1},a_n)(q_n) \\
&= i'((q,q_1))\prod_{i=0}^{|x|-2}[B'(q_i,q_{i+1},a_{i+1})A'(q_i,q_{i+1})(q_{i+1},q_{i+2})]B'((q_{n-1},q_n),a_n) \\
&= P_{M'}(x,\pi')
\end{aligned}$$

We thus have that every HMMT can be transformed into an equivalent HMM.

Next, we will show that for every non-terminating PFA $M = \langle \Sigma, Q, I_p, F_p, \delta_p\rangle$ we can define an equivalent HMMT $M = \langle \Sigma, Q, A, B, i\rangle$, where, for all $q,q' \in Q$ and $a \in \Sigma$

- $i(q) = I_p(q)$,
- $A(q,q') = \sum_{a\in\Sigma}\delta_p(q,a)(q')$, and
- $B(q,a)(q) = \begin{cases} \frac{\delta_p(q,a)(q')}{\sum_{a\in\Sigma}\delta(q,a)(q')} & if\ \sum_{a\in\Sigma}\delta_q(q,a)(q') > 0, \\ 0 & otherwise. \end{cases}$

It is easily shown that M' is a HMMT. Furthermore M and M' generate the same distribution because for any string non-empty string $x = a_1\cdots a_n$ and path $\pi = q_0\dots q_n$ we have

$$\begin{aligned}
\bar{P}_M(x,\pi) &= I_p(q_0)\prod_{i=0}^{n-1}\delta_p(q_i,a_{i+1}(q_{i+1}) \\
&= I_p(q_0)\prod_{i=0}^{n-1}B(q_i,a_{i+1})A(q_i,q_{i+1}) \\
&= i(q_0)\prod_{i=0}^{n-1}B(q_i,a_{i+1})A(q_i,q_{i+1}) \\
&= P_{M'}(x).
\end{aligned}$$

For the last step, given an HMM $M = \langle \Sigma, Q, A, B, i \rangle$ we can construct a non-terminating PFA $M' = \langle \Sigma, Q, I_p, F_p, \delta_p \rangle$ by setting $\forall q, q' \in Q$ and $a \in \Sigma$:

- $I_p(q) = i(q)$,
- $F_p(q) = 0$
- $\delta_p(q,a)(q') = B(q,a) \cdot A(q,q')$

It is easily shown that M' is a PFA and that M and M' generate the same distribution. □ □

Example 2. *Fig. 2.4a presents a HMMT $M = \langle \Sigma, Q, A, B, i \rangle$ defined as follows:*

- $\Sigma = \{a, b\}$,
- $Q = \{1, 2\}$,
- $A(1,1) = 0.2$, $A(1,2) = 0.8$, $A(2,1) = 0.3$, $A(2,2) = 0.7$,
- $B(1,a)(1) = 0.4$, $B(1,b)(1) = 0.6$, $B(1,a)(2) = 0.4$, $B(1,b)(1) = 0.6$, $B(2,a)(1) = 0.8$, $B(2,b)(1) = 0.2$, $B(2,a)(2) = 0.1$, $B(2,b)(2) = 0.9$,
- $i(1) = 0.7$, $i(2) = 0.3$.

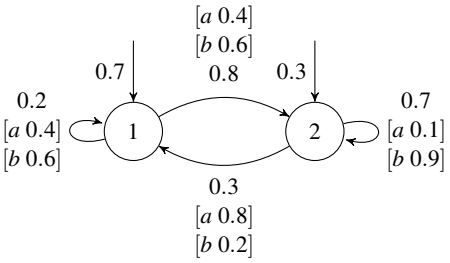
There exists an equivalent HMM $M' = \langle \Sigma', Q', A', B', i' \rangle$ shown in Fig. 2.4b, which defines as follows:

- $\Sigma' = \Sigma$,
- $Q' = \{(1,1), (1,2), (2,1), (2,2)\}$,
- $A'((1,1),(1,1)) = 0.2$, $A'((1,1),(1,2)) = 0.8$, $A'((1,2)(2,1)) = 0.3$, $A'((1,2)(2,2)) = 0.7$, $A'((2,1)(1,1)) = 0.2$, $A'((2,1)(1,2)) = 0.8$, $A'((2,2)(2,1)) = 0.3$, $A'((2,2)(2,2)) = 0.7$,
- $B'((1,1),a) = 0.4$, $B'((1,1),b) = 0.6$, $B'((1,2),a) = 0.4$, $B'((1,2),b) = 0.6$, $B'((2,1),a) = 0.8$, $B'((2,1),b) = 0.2$, $B'((2,2),a) = 0.1$, $B'((2,2),b) = 0.9$,
- $i(1,1) = 0.14$, $i(1,2) = 0.56$, $i(2,1) = 0.09$, $i(2,2) = 0.21$.

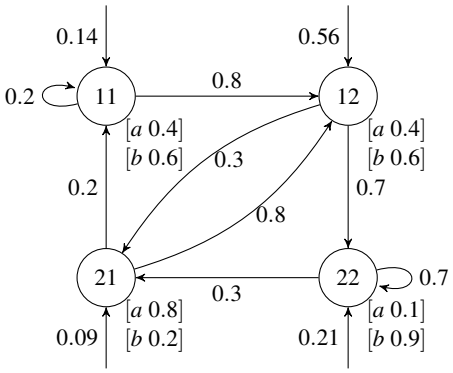
Correspondingly, there is an equivalent non-terminating PFA $M'' = \langle \Sigma'', Q'', I_p, \delta_p \rangle$ shown in Fig. 2.4c, which is defined as:

- $\Sigma'' = \Sigma$,

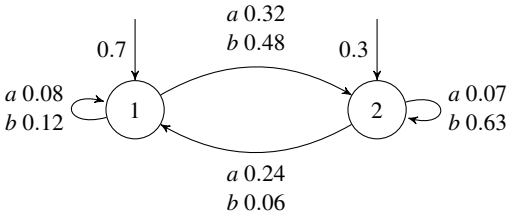
- $Q'' = Q$,
- $I_p(1) = 0.7$, $I_p(2) = 0.3$,
- $\delta_p(1,a)(1) = 0.08$, $\delta_p(1,b)(1) = 0.12$, $\delta_p(1,a)(2) = 0.32$, $\delta_p(1,b)(2) = 0.48$, $\delta_p(2,a)(1) = 0.24$, $\delta_p(2,b)(1) = 0.06$, $\delta_p(2,a)(2) = 0.07$, $\delta_p(2,b)(2) = 0.63$.



(a) An HMMT. This subfigure illustrates the graphical representation of an HMMT, showcasing its two states (labeled 1 and 2) and their associated transition probabilities.



(b) An equivalent HMM. Presented here is an equivalent HMM that captures the same probabilistic behavior as the HMMT.



(c) An equivalent PFA. The subfigure displays a PFA equivalent to the previously shown HMM and HMMT. It maintains the essential two-state structure and transition probabilities.

Fig. 2.4 An HMMT with its equivalent HMM and PFA.

2.4 Distances between two distributions

There are several ways to define the distance between two discrete probability distributions, depending on the reason why we want to know how close two distributions are. Important measures used in machine learning algorithms include the Kullback-Leibler divergence,

the L_p distance, the Hellinger distance, and the triangle distance [31, 117, 71, 30]. In this chapter, we concentrate on regular distributions and discussed two methods for computing the distances between discrete probability distributions on strings. One method relies on the presentation of the two distributions via PFAs. For this case, we present a novel variation of the algorithm presented in [30] for stochastic weighted automata to calculate the L_p distance. The other method compares a probability distribution presented by a PFA only with respect to a finite set of strings that is taken as ground truth. In this case, we use a probabilistic version of accuracy, precision, and recall.

2.4.1 Computing the Euclidean distance between regular distributions

For any integer $p > 1$, the L_p distance between two distributions D_1 and D_2 on Σ^* is defined as:

$$L_p(D_1, D_2) = (\sum_{x \in \Sigma^*} |D_1(x) - D_2(x)|^p)^{1/p}. \tag{2.1}$$

While the above distances are commonly used to compare vectors, they can also be applied to compare distributions by treating them as multidimensional vectors. Examples include the Euclidean distance L_2 and the "Manhattan" distance L_1. In general, the problem of computing L_{2p+1} given two probabilistic finite automata is known to be NP-hard even for automata without cycles [30, 71]. The same holds also for L_∞, a distance adapted from the L_1 by substituting the sum with the supremum. Therefore we restrict to the distance L_{2p} for any p and present a novel algorithm to compute the L_2 distance between two regular distributions given two probabilistic automata that generate them. Generalizing the algorithm to any other even number $2p$ is trivial. For simplicity we compute $(L_2(A_1, A_2))^2$, and then we can obtain the L_2 distance between A_1 and A_2 straightforwardly by taking square root:

$$\begin{aligned}
L_2(A_1, A_2) &= (L_2(A_1, A_2)^2)^{\frac{1}{2}} \\
&= (\sum_{x \in \Sigma^*} |P_{A_1}(x) - P_{A_2}(x)|^2)^{\frac{1}{2}} \\
&= (\sum_{x \in \Sigma^*} (P_{A_1}(x) - P_{A_2}(x))^2)^{\frac{1}{2}} \\
&= (\sum_{x \in \Sigma^*} P_{A_1}(x)^2 - 2P_{A_1}(x)P_{A_2}(x) + P_{A_2}(x))^2)^{\frac{1}{2}} \\
&= (\sum_{x \in \Sigma^*} P_{A_1}(x)^2 - 2\sum_{x \in \Sigma^*} P_{A_1}(x)P_{A_2}(x) + \sum_{x \in \Sigma^*} P_{A_2}(x)^2)^{\frac{1}{2}}.
\end{aligned} \tag{2.2}$$

In the second equality, the absolute values can be removed since they are squared. The last three summations can be computed separately via a shortest-distance algorithm for weighted graphs. In general, we consider three different situations.

First, when A_1 and A_2 are acyclic, those summations are finite and can be computed directly.

Second, when both A_1 and A_2 are deterministic probabilistic automata, we compute their intersection automaton A using the product construction. In short, each state corresponds to a pair of states, one from each original automaton. The initial and final probability on the Cartesian product of the states is the product of the respective probabilities in both automata. Similarly, the transition probabilities for a pair of states are obtained by multiplying the probabilities of the corresponding transitions in the original automata for the same symbol. The resulting state is the product of the resulting state of each transition.

To avoid computing three intersections, we keep the probability labeling each transition $\delta_p((q_1,q_2),a)(q_1',q_2')$ as a pair $(\delta_{p1}(q_1,a)(q_1'),\delta_{p2}(q_2,a)(q_2'))$, where δ_{p1} is the transition function of A_1 and δ_{p2} is the one of A_2. When calculating $P_{A_i}(x)^2$, we only need to square the $i-$th component of the pair, while we multiply the two components to calculate $P_{A_1}(x)P_{A_2}(x)$. This is possible because, for any string $x \in \Sigma^*$, there is at most one accepting path in A_1 and A_2. Finally, we use the shortest distance algorithm over the intersection automaton with weight modified as described above to compute $\sum_{x\in\Sigma^*}(P_{A_1}(x))^i(P_{A_2}(x))^{2-i}$ for $i = 0, 1$ and 2.

The third and last situation is when A_1 and A_2 are arbitrary PFAs. In this case, there may be multiple paths with the same label, which means we cannot avoid performing three different intersection automata: one of A_1 with itself, another of A_1 with A_2, and the last of A_2 with itself. As before, we use the shortest distance algorithm over the intersection automaton to compute $\sum_{x\in\Sigma^*}(P_{A_1}(x))^i(P_{A_2}(x))^{2-i}$ for $i = 0, 1$ and 2.

A shortest distance algorithm for weighted graphs

The classical shortest paths problems compute the shortest paths from one set of source vertices to all other vertices in the graph. This problem has been generalized to the weighted graph [80]: The shortest distance from a set of vertices I to a vertex F is the sum of the weights of all paths from nodes in I to nodes in F. In [80], a generic algorithm is given to compute single-source shortest distances for a directed graph with weight in a semiring. Termination of the algorithm depends on the graph being $k-$closed, a condition that unfortunately is not satisfied by probabilistic automata (or by their intersection). Therefore, we must adapt the algorithm to work with a weaker condition, namely boundness.

A weighted graph $\langle \Sigma, Q, I, F, \delta \rangle$ consists of an alphabet Σ, a finite set of states Q, an initial weight $I : Q \to \mathbb{Q}$, a final weight $F : Q \to \mathbb{Q}$, and a transition function $\delta : Q \times \Sigma \to \mathbb{Q}^Q$. It is

similar to a PFA but does not need to satisfy its restrictions. Every probabilistic automaton is a weighted graph, and the intersection of two PFAs, as sketched above, is a weighted graph (but, in general, not a PFA).

Definition 4. *A weighted graph* $\langle \Sigma, Q, I, F, \delta \rangle$ *is bounded, if for any cycle* π *there exists a* $k \in \mathbb{Q}$ *such that* $\sum_{n=1}^{\infty} \delta(\pi)^n = k$.

For example, every PFA $\langle \Sigma, Q, I_p, F_p, \delta_p \rangle$ is bounded because the probability of a path with a cycle is always strictly less than 1. It follows that $\sum_{n=1}^{\infty} \delta_p(\pi) = \frac{r}{1-r}$, where $\delta_p(\pi) = r < 1$. Also, the intersection of two PFAs is a bounded weighted graph, but not necessarily a PFA because weights are not normalized.

Next, we provide a shortest-distance algorithm for bounded weighted graphs. The pseudo-code is given in Algorithm 3. The algorithm uses a set S to maintain the states after transitions and M to store the sequence of transitions visited. S is initialized as a set of initial states. $d[q]$ is the total weight from an initial state to the current state q and $r[q]$ is the weight of the current transition from an initial state to state q.

In the while loop from line 10 to 31, each time we extract a state q from set S, then store the value of $r[q]$ in r' and set $r[q]$ to 0. Lines $13-31$ calculate the distance. First, for all transitions starting from state q, if the following state q' does not exist in $M[q]$, update $M[q']$ and the value of $d[q']$ and $r[q']$. If next state q' is not in S, add q' into S. If the next state q' exists in $M[q]$, find path π of the repetition part, then update $d[q]$. When q is the last state in set S, and there are no more transitions, the while loop ends. In the end, for each state q, $d[q]$ is multiplied by the final weight of the state.

2.4.2 Metrics based on a probabilistic confusion matrix

The above Euclidean distance assumes the availability of two probabilistic automata generating the distributions we want to measure. In many cases, however, we only have a PFA representation of one distribution to be compared against a finite distribution. This is the case, for example, when we want to learn a PFA and we want to compare its predictions with the given probabilities in a finite dataset.

In a similar binary classification situation but without probabilities, we can use the confusion matrix as a summary to evaluate the performance of a model. It organizes the evaluation into four categories [109]: true positives (TP) are the instances that are correctly predicted as positive by the model. True negatives (TN) are the instances that are correctly predicted as negative by the model. False positives (FP) are the instances incorrectly predicted as positive by the model and false negatives (FN) are the instances incorrectly predicted as negative by the model. In general, we are not interested in the specific instances, but only in

how many there are, so we let TP, TN, FP, and FN denote the number of instances in each of these sets.

The confusion matrix allows for the derivation of various performance metrics, including:

- Accuracy: The overall correctness of the model's predictions against a finite set of data, calculated as $(TP+TN)/(TP+TN+FP+FN)$;
- Precision: The ability of the model to correctly predict positive instances, calculated as $TP/(TP+FP)$;
- Recall (also called Sensitivity): The ability of the model to identify all positive instances, calculated as $TP/(TP+FN)$.

For a more balanced measure combining both precision and recall, one often uses the $F1$ score, calculated as $2 \cdot (Precision \cdot Recall)/(Precision + Recall)$.

The above confusion matrix consists of counting predicted and true instances. However, when working with PFAs, predictions provide probabilities or confidence scores for string. We, therefore, extend the confusion matrix to take into account these probabilities instead of a binary classification. Our probabilistic confusion matrix expands the concept of true positives, true negatives, false positives, and false negatives to incorporate probabilities using the L_1 distance between the probability of the true instances P_s and the one predicted by a PFA P_A automaton.

Confidence on True Positives (cTP): a measure of the global error between the probability of the correctly classified instances, calculated as $\sum_{x \in TP} 1 - |P_s(x) - P_A(x)|$;

Confidence on False Positive (cFP): a measure of the global error of all instances that are incorrectly classified as negative, calculated as $\sum_{x \in FP} P_A(x)$.

Confidence on False Negatives (cFN): a measure of the global error of all instances that are incorrectly classified as negative, calculated as $\sum_{x \in FN} P_s(x)$.

We do not have a confidence variant of true negative instances, as the probability of not belonging to language is 0. An interesting extension would be to consider probabilistic languages with a certain probability threshold [95].

The above probabilistic confusion matrix provides a more detailed and nuanced analysis of a PFA in terms of its generated distribution but takes into account its NFA support. This generalization leads to a new definition of precision, sensitivity, and specificity for PFAs:

$$Precision = \frac{cTP}{TP + cFP}, \tag{2.3}$$

$$Sensitivity = \frac{cTP}{TP + cFN}, \tag{2.4}$$

$$Specificity = \frac{TN}{TN + cFP} \tag{2.5}$$

Note the asymmetry between TP and cFP in the denominator of Precision and Sensitivity (TP does not use the confident value) because TP refers to the total number of corrected instances needed to average confidence cTP.

The closer the predicted distribution of a PFA is to that of the grounded truth test set, the closer will precision, and sensitivity be to 1. On the other hand, when the less true positive the more precision, the sensitivity will be closer to 0.

2.5 Summary

In this chapter, we introduced finite automata, probabilistic automata, and Hidden Markov Models, and studied their relationships. In particular, we have seen that probabilistic automata are strictly more general than their deterministic version and that they are very similar to the Hidden Markov Model. Our focus is on probabilistic automata, and therefore we presented two classical algorithms to compute the probability of string in a given PFA and a novel way to compute the Euclidean distance between two regular distributions presented by probabilistic automata.

Algorithm 3 A shortest distance algorithm for weighted graphs

Input: A bounded weighted graph $\langle \Sigma, Q, I, F, \delta \rangle$
Output: A rational number d, the shortest distance between I and F

Let S and M be an empty set
for $q \in Q$ **do**
  if $I_p(q) \neq 0$ **then**
    $d[q] = I_p(q)$; $r[q] = I_p(q)$; $M[q] = \{q\}$;
    add state q to S
  else
    $d[q] = 0; r[q] = 0$
  end if
end for
while $S \neq \emptyset$ **do**
  remove an element q from S and add it to P;
  $r' = r[q]$; $r[q] = 0$
  for all $a \in \Sigma, q' \in Q$ **do**
    if $\delta_p(q,a)(q') \neq 0$ **then**
      if q' is not in $M[q]$ **then**
        $M[q'] = M[q] + aq'$;
        $d[q'] = d[q'] + (r' \times \delta_p(q,a)(q'))$; $r[q'] = r[q'] + (r' \times \delta_p(q,a)(q'))$
        if $q' \notin S$ **then**
          add q' to S
        end if
      else
        find cyclic subsequence $q'xq'$ in M[q] and store it in Re;
        remove alphabet symbols from $q'xq'$ and store the resulting path in π
        if $Re \notin M[q']$ **then**
          $l = \delta_p(\pi)$; $k = \frac{l}{1-l}$
          $d[q'] = d[q'] + (r' \times k)$; $r[q'] = r[q'] + (r' \times k)$
        end if
      end if
    end if
  end for
end while
for $q \in Q$ **do**
  $d[q] = d[q] \times F_p[q]$
end for
return d