Incremental Computation of Infix Probabilities for Probabilistic Finite Automata

Marco Cognetta, Yo-Sub Han, and **Soon Chan Kwon**
Department of Computer Science
Yonsei University, Seoul, Republic of Korea
{mcognetta, emmous, soon--chan}@yonsei.ac.kr

Abstract

In natural language processing, a common task is to compute the probability of a given phrase appearing or to calculate the probability of all phrases matching a given pattern. For instance, one computes affix (prefix, suffix, infix, etc.) probabilities of a string or a set of strings with respect to a probability distribution of patterns.

The problem of computing infix probabilities of strings when the pattern distribution is given by a probabilistic context-free grammar or by a probabilistic finite automaton is already solved, yet it was open to compute the infix probabilities in an incremental manner. The incremental computation is crucial when a new query is built from a previous query. We tackle this problem and suggest a method that computes infix probabilities incrementally for probabilistic finite automata by representing all the probabilities of matching strings as a series of transition matrix calculations. We show that the proposed approach is theoretically faster than the previous method and, using real world data, demonstrate that our approach has vastly better performance in practice.

1 Introduction

Probabilistic grammars and finite automata are commonly used to model distributions in natural language processing. Among language models, probabilistic finite automata (PFAs) provide a simple, yet powerful and well-understood representation of many probabilistic language phenomena. Numerous speech processing tasks rely on PFAs in practice (Gyawali et al., 2013; Mohri et al., 2002; Wilson and Raaijmakers, 2008; Ng et al., 2000).

An important problem regarding PFAs is to calculate the probability of some affix (prefix, suffix, infix, etc.) of a string with respect to a given distribution. That is, given a PFA $\mathcal{P}$ and a string w, one might ask the probability of w appearing as a prefix, suffix, or infix in the distribution modeled by $\mathcal{P}$—in other words, the sum of the probabilities of all strings in the form of wx, xw, or xwy with respect to $\mathcal{P}$, for some strings x and y. A more general problem is to compute the sum of the probabilities of all strings in a regular language with respect to a PFA. Computing affix probabilities in probabilistic models is an important problem in natural language processing. For probabilistic context-free grammars (PCFGs) and PFAs, the problem of calculating the prefix or suffix probability of a string can be efficiently solved (Fred, 2000; Corazza et al., 1991). However, calculating the infix probability of a string or the weight of a regular language is not as straightforward (Corazza et al., 1991). Additionally, computing affix probabilities for more general probabilistic language models has proven to be quite difficult. Nevertheless, there are some approaches for computing the exact affix probabilities over a variety of models. Lattice posterior probabilities for n-grams, which are a restricted form of PFA (Vidal et al., 2005b), have a variety of uses in speech processing and can be computed efficiently (de Gispert et al., 2013; Can and Narayanan, 2015). For several affixes, Corazza et al. (1991) described algorithms to determine the probability of a string appearing as that affix in a PCFG. They made an important note that, unlike computing prefix or suffix probabilities, infix probability calculations are prone to double counting in the event of the infix appearing multiple times in a string. Then, they provided an algorithm for computing the infix probability of a string in restricted cases. Stolcke (1995) described a series of recurrences that can be used to compute affix probabilities and variations of the most probable parse of a string for PCFGs. For the general class of linear context-free rewriting systems, a method

Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 2732–2741
Brussels, Belgium, October 31 - November 4, 2018. ©2018 Association for Computational Linguistics

to compute prefix probabilities is known (Nederhof and Satta, 2011b). Fred (2000) also considered these problems and described a method to compute the infix probabilities under the condition that the infix appears at most once in any non-zero probability string when the language model is a stochastic regular grammar, which is equivalent in power to a PFA. These assumptions are rather strict and led researchers to consider a more general problem. Nederhof and Satta (2011a) solved the general infix probability problem for PCFGs. In fact, their method can be used to compute the weight of any regular language with respect to a PCFG. They also proposed an open problem of *incrementally* computing the infix probability of a string—using the numerical result of one infix computation to speed up the evaluation of another.

We design a new method for solving the infix probability problem for PFAs incrementally. Unlike the previous methods involving recurrence calculations or intersection constructions, our method is based on evaluating a series of matrices formed from regular expressions. Additionally, our method has no constraints on the input string. We show that our method is both theoretically and practically more performant than the previous algorithms. Our experimental results show a greater than 80% performance improvement. In Section 2, we review PFAs and other necessary formalisms. We recall how to obtain unambiguous regular expressions in Section 3, and introduce a matrix representation for computing the weight of a regular language in Section 4. In Section 5, we propose an algorithm to incrementally compute the infix probability of a given string. We validate the practical performance of the incremental method using a test set of PFAs obtained from real life data in Section 6 and conclude with a discussion and some open problems in Section 7.

2 Preliminaries

2.1 Finite Automata and PFAs

Following standard notation in automata theory, we let Σ be a set of characters and Σ^* be the set of all strings. For a string $w = w_1w_2 \ldots w_n \in \Sigma^*$, we write $|w| = n$ as its length. The empty string is written as λ.

A deterministic finite automaton (DFA) is a 5-tuple $\mathcal{D} = (Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $s \in Q$ is the initial state, and $F \subset Q$ is the set of final states. A language is a set of strings, and $\mathcal{D}$ recognizes a regular language denoted by $L(\mathcal{D})$. For a summary of automata theory (including regular expressions and formal language theory), we direct the reader to Hopcroft and Ullman (1979).

A PFA is a weighted finite automaton that computes a function $\mathcal{P} : \Sigma^* \to [0, 1]$. We abuse function notation and write $\mathcal{P}(w)$ or $\mathcal{P}(\pi)$ to denote the weight of a word or a path, respectively. These values are defined later. A PFA $\mathcal{P}$ is specified by a 5-tuple $\mathcal{P} = (Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \times Q \to [0, 1]$ is the transition function, $I : Q \to [0, 1]$ and $F : Q \to [0, 1]$ are the initial and final functions, respectively. The transition function is assumed to have a default value of 0, in other words, if a transition does not exist it can be considered as having weight 0. A PFA has three additional requirements:

1. $\sum_{q \in Q} I(q) = 1$
2. $\forall q \in Q,\ F(q) + \sum_{q' \in Q, c \in \Sigma} \delta(q, c, q') = 1$
3. All states are both accessible and co-accessible[1].

If these conditions hold, then for all strings w, $0 \leq \mathcal{P}(w) \leq 1$ and $\sum_{w \in \Sigma^*} \mathcal{P}(w) = 1$. A PFA can be represented in the form of transition matrices, which simplifies several computations. We denote the matrix formulation of a PFA by $\mathcal{P} = (Q, \Sigma, \{\mathbb{M}(c)\}_{c \in \Sigma}, \mathbb{I}, \mathbb{F})$ where $\{\mathbb{M}(c)\}_{c \in \Sigma}$ is a set of $|Q| \times |Q|$ transition matrices with $\mathbb{M}(c)_{i,j} = \delta(q_i, c, q_j)$. Likewise, $\mathbb{I}$ and $\mathbb{F}$ are $1 \times |Q|$ and $|Q| \times 1$ vectors with $\mathbb{I}_i = I(q_i)$ and $\mathbb{F}_j = F(q_j)$.

Consider a string $w = w_1w_2 \cdots w_n \in \Sigma^*$ and a corresponding labeled path

$$\pi = (q_0, w_1, q_1), (q_1, w_2, q_2), \ldots, (q_{n-1}, w_n, q_n)$$

in $\mathcal{P}$. Then the probability of a path π in $\mathcal{P}$ is

$$\mathcal{P}(\pi) = I(q_0) \left(\prod_{i=1}^{n} \delta(q_{i-1}, w_i, q_i) \right) F(q_n).$$

Let Φ_w be the set of all labeled paths corresponding to w. The probability of w is now

[1]Accessible states are states reachable from a state with non-zero initial weight and co-accessible states are those that can reach a state with non-zero final weight.

$\sum_{\pi \in \Phi_w} \mathcal{P}(\pi)$. There exist two equivalent dynamic programming methods—the forwards and backwards algorithms—to compute the probability of a given string (Vidal et al., 2005a). Using the matrix formulation, the probability of a string is given succinctly as

$$\mathbb{I} \prod_{i=1}^{|w|} \mathbb{M}(w_i) \mathbb{F}.$$

For brevity, we write $\mathbb{M}(\Sigma) = \sum_{c \in \Sigma} \mathbb{M}(c)$ and $\mathbf{0}$ and $\mathbf{1}$ for the zero and identity matrices when the dimensions are clear. Further, we compute

$$\sum_{i=0}^{\infty} \mathbb{M}(\Sigma)^i = (\mathbf{1} - \mathbb{M}(\Sigma))^{-1},$$

which we denote $\mathbb{M}(\Sigma^*)$. We can compute the prefix and suffix probabilities of a string w as

$$\mathcal{P}(w\Sigma^*) = \mathbb{I} \left(\prod_{i=1}^{|w|} \mathbb{M}(w_i) \right) \mathbb{M}(\Sigma^*) \mathbb{F}$$

and

$$\mathcal{P}(\Sigma^* w) = \mathbb{I}\mathbb{M}(\Sigma^*) \left(\prod_{i=1}^{|w|} \mathbb{M}(w_i) \right) \mathbb{F},$$

respectively. If an automaton $\mathcal{M}$ satisfies all of the requirements for a PFA except that $\sum_{q \in Q} I(q) \leq 1$ or $\forall\ q \in Q$, $F(q) + \sum_{q' \in Q, c \in \Sigma} \delta(q, c, q') \leq 1$, then we call $\mathcal{M}$ a sub-PFA. These machines have the property that for any string w over Σ, $0 \leq \mathcal{M}(w) \leq 1$ and $\sum_{w \in \Sigma^*} \mathcal{M}(w) \leq 1$. Since a sub-PFA $\mathcal{M}$ may not describe a probability distribution over strings, we call $\mathcal{M}(w)$ the *weight* of w instead of probability. A stochastic language over an alphabet Σ is a set $\mathcal{S} \subseteq \Sigma^*$ where each string in $\mathcal{S}$ has an associated probability, $0 \leq Pr_{\mathcal{S}}(w) \leq 1$, such that $\sum_{w \in \Sigma^*} Pr_{\mathcal{S}}(w) = 1$. Given a stochastic language $\mathcal{S}$, if there exists a PFA $\mathcal{P}$ such that $\forall w \in \Sigma^*$, $\mathcal{P}(w) = Pr_{\mathcal{S}}(w)$, we call $\mathcal{S}$ a *regular stochastic language*.

2.2 Unambiguous Regular Expressions

Regular expressions are a common representation of regular languages. A string that can be matched with a regular expression is said to be in the language of the regular expression. A match occurs when there is a valid assignment of symbols in the regular expression to the queried string. We denote the set of all strings that match a regular expression as $L(R)$. An *unambiguous* regular expression has only one valid assignment for any string in the language. For example, consider the regular expression $(a \cup b)^* aa (a \cup b)^*$, which accepts the set of strings over $\Sigma = \{a, b\}$ containing aa as an infix. We assign different subscripts to symbols that appear in more than one position to form $(a_1 \cup b_1)^* a_2 a_3 (a_4 \cup b_2)^*$. Given the string $baaa$, we find that $b_1 a_1 a_2 a_3$ and $b_1 a_2 a_3 a_4$ are both valid assignments, hence $baaa$ is in the language. However, since there are two valid assignments, we say that this regular expression is ambiguous (Book et al., 1971). An unambiguous expression for the same language is $b^* a (bb^* a)^* a (a \cup b)^*$.

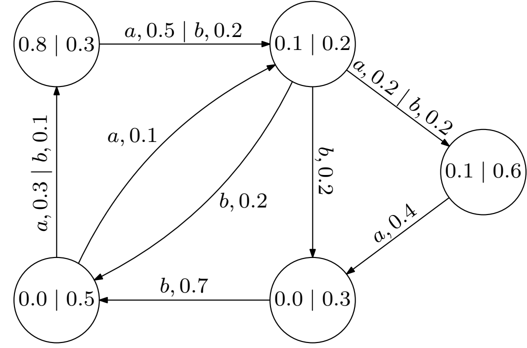

Figure 1: An example PFA over $\Sigma = \{a, b\}$. Each state has an initial and final probability, separated by a bar. The edges hold a character and the probability of the corresponding transition.

2.3 Intersecting a DFA and a PFA

The standard method to compute the weight of a regular language with respect to a PFA is to intersect its DFA representation with the PFA. This construction was already discussed in (Vidal et al., 2005a). Nederhof and Satta (2011a) considered a similar construction for DFAs and PCFGs and solved the infix problem for that class of machine. The construction creates a new sub-PFA $[\mathcal{D} \cap \mathcal{P}]$ such that:

$$[\mathcal{D} \cap \mathcal{P}](w) = \begin{cases} \mathcal{P}(w), & w \in L(\mathcal{D}); \\ 0, & \text{otherwise.} \end{cases}$$

It follows that

$$\sum_{w \in \Sigma^*} [\mathcal{D} \cap \mathcal{P}](w) = \sum_{w \in L(\mathcal{D})} \mathcal{P}(w),$$

as desired.

The intersection algorithm is similar to that of the cross product of two automata from classical automata theory. Given a DFA $\mathcal{D}$ and a PFA $\mathcal{P}$, we construct a new machine $\mathcal{W}$ with states $Q_{\mathcal{W}} = Q_{\mathcal{D}} \times Q_{\mathcal{P}}$. For two states (x, y), (x', y') and a character $c \in \Sigma$, $\delta_{\mathcal{W}}((x, y), c, (x', y')) = \delta_{\mathcal{P}}(y, c, y')$ if $\delta_{\mathcal{D}}(x, c) = x'$ and 0 otherwise. Likewise, $I_{\mathcal{W}}((p, q)) = I_{\mathcal{P}}(q)$ if p is the initial state of $\mathcal{D}$ and $F_{\mathcal{W}}((p', q')) = F_{\mathcal{P}}(q')$ if p' is a final state of $\mathcal{D}$ and are 0 otherwise.

Using Algorithm 1, we can now efficiently compute the sum of the weight of all strings in $\mathcal{L}(\mathcal{D})$ by evaluating

$$\mathbb{I}_{[\mathcal{D} \cap \mathcal{P}]}(\mathbf{1} - \mathbb{M}_{[\mathcal{D} \cap \mathcal{P}]}(\Sigma))^{-1}\mathbb{F}_{[\mathcal{D} \cap \mathcal{P}]}.$$

Algorithm 1 DFA/PFA intersection

```
1:  procedure INTERSECT(DFA D, PFA P)
2:      Q' = Q_D × Q_P
3:      for (d, p) ∈ Q' do
4:          if d is q_0 then
5:              I'((d, p)) = I(p)
6:          else
7:              I'((d, p)) = 0
8:          end if
9:          if d ∈ F_D then
10:             F'((d, p)) = F(p)
11:         else
12:             F'((d, p)) = 0
13:         end if
14:         for c ∈ Σ, (d', p') ∈ Q' do
15:             if δ_D(d, c) = d' then
16:                 δ'((d, p), c, (d', p')) = δ_P(p, c, p')
17:             else
18:                 δ'((d, p), c, (d', p')) = 0
19:             end if
20:         end for
21:     end for
22:     return [D ∩ P] = (Q', Σ, δ', I', F')
23: end procedure
```

To find the infix probability of a given string $w = w_1 w_2 \cdots w_n$, we take $\mathcal{D}$ to be a DFA that recognizes the language of all strings containing w as an infix. The Knuth-Morris-Pratt (KMP) algorithm produces such a DFA (with $O(n)$ states) in $O(n)$ time (Knuth et al., 1977). Thus, computing the infix probabilities of each of w's prefixes takes $O(n(n|Q_{\mathcal{P}}|)^m)$ time, where m is the matrix multiplication constant[2], as at each step one needs to rebuild the entire intersection automaton for the current prefix and compute an inverse on its transition matrices.

Nederhof and Satta (2011a) demonstrated that a similar method can be used to compute the infix probability of a given string in PCFGs.

3 Generating Unambiguous Regular Expressions

Book et al. (1971) showed that, given a regular language (in the form of an automaton or regular expression), one can find an equivalent unambiguous regular expression by constructing an equivalent DFA and using state elimination on the resulting DFA.

Let $\mathcal{D} = (Q, \Sigma, \delta, q_1, F)$ be a DFA with $|Q| = n$ and the states being ordered from 1 to n. We add two new states, q_0 and q_{n+1} such that q_0 is the new start state and q_{n+1} is the only final state, and add the following transitions: $\delta(q_0, \lambda) = q_1$ and $\forall q \in F,\ \delta(q, \lambda) = q_{n+1}$. We then dynamically eliminate states using the following recurrence:

$$\alpha_{i,j}^{k} = \alpha_{i,j}^{k-1} + \alpha_{i,k}^{k-1}(\alpha_{k,k}^{k-1})^{*}\alpha_{k,j}^{k-1}$$

with the base cases:

$$\alpha_{i,j}^{0} = \begin{cases} \lambda, & i = 0, j = 1; \\ \lambda, & q_i \in F \wedge j = n+1; \\ \{c \mid \delta(q_i, c) = q_j\}, & \text{otherwise.} \end{cases}$$

The equations follow the general concatenation, union, and Kleene star rules for regular expressions. In addition, we have:

- $\emptyset + c = c + \emptyset = c$, for $c \in \Sigma$
- $\emptyset c = c\emptyset = \emptyset$, for $c \in \Sigma$
- $\lambda c = c\lambda = c$, for $c \in \Sigma$
- $\emptyset^{*} = \lambda$.

The term $\alpha_{i,j}^{k-1}$ corresponds to the set of strings for which, starting from state q_i, describe a path to q_j where all intermediate states are of the form q_ℓ, where $\ell < k$ (the terminal state in the path has no such restriction). Similarly, $\alpha_{i,k}^{k-1}(\alpha_{k,k}^{k-1})^{*}\alpha_{k,j}^{k-1}$ corresponds to all of the strings which, beginning

[2]The matrix multiplication constant, m, is the order of the polynomial for the runtime of multiplying two $n \times n$ matrices together, i.e. a function in $O(n^m)$. In practice, Strassen's algorithm is often used, yielding $m \approx 2.81$ (Strassen, 1969).

at state q_i, end at q_j going through q_k, where all intermediate states have label at most k (McNaughton and Yamada, 1960).

We extract $\alpha^n_{0,n+1}$, which is an unambiguous regular expression for the language recognized by $\mathcal{D}$ (Book et al., 1971). Note that the requirement of an input automaton being deterministic is not strict. In fact, state elimination generates unambiguous regular expressions from any unambiguous automaton.

From now on, we only consider unambiguous regular expressions for any regular languages.

4 The Weight of a Regular Language

In Section 2.3, we reviewed the classical way of computing the weight of a regular language with respect to a PFA using an intersection construction. Here, we present a new method based on unambiguous regular expressions. We describe a simple transformation to convert regular expressions into operations on transition matrices. Consider the following mapping from regular expressions to matrices:

- $\emptyset \to \mathbf{0}$
- $\lambda \to \mathbf{1}$
- $c \in \Sigma \to \mathbb{M}(c)$.

Now, let R and S be regular expressions with $\mathbb{M}(R)$ and $\mathbb{M}(S)$ being their corresponding matrices:

- $R \cup S \to \mathbb{M}(R) + \mathbb{M}(S)$
- $RS \to \mathbb{M}(R)\mathbb{M}(S)$
- $R^* \to (\mathbf{1} - \mathbb{M}(R))^{-1}$.

Using these definitions, we can build a matrix calculation out of a given regular expression. We then obtain the weight of a regular expression R with respect to some PFA $\mathcal{P}$ by evaluating $\mathbb{I}_{\mathcal{P}}\mathbb{M}_{\mathcal{P}}(R)\mathbb{F}_{\mathcal{P}}$. However, the straightforward application of this method is prone to overcounting when there are many ways for a string to be matched to the expression.

We present a simple example where an ambiguous expression overcounts whereas an unambiguous expression returns the correct result when transformed into matrix calculations:

Let $R = b^*a(bb^*a)^*a(a \cup b)^*$ and $S = (a \cup b)^*aa(a \cup b)^*$, which are both regular expressions for all strings containing aa as an infix. Using the PFA in Figure 1, we have:

$$\mathbb{I}\mathbb{M}(b^*a(bb^*a)^*a(a \cup b)^*)\mathbb{F} \approx 0.153,$$

$$\mathbb{I}\mathbb{M}((a \cup b)^*aa(a \cup b)^*)\mathbb{F} \approx 0.198.$$

This gap can be made arbitrarily large by adding ambiguity to the regular expression without changing the described language.

We now show that, given a PFA $\mathcal{P}$ and a DFA $\mathcal{D}$, we can compute the weight of $L(\mathcal{D})$ with respect to $\mathcal{P}$.

Lemma 1. *Let $\mathcal{P}$ be a PFA, $\mathcal{D}$ be a DFA and R be an unambiguous regular expression for $L(\mathcal{D})$ generated by state elimination. Then $\mathbb{I}_{\mathcal{P}}\mathbb{M}_{\mathcal{P}}(R)\mathbb{F}_{\mathcal{P}} = \sum_{w \in L(\mathcal{D})} \mathcal{P}(w)$.*

Proof. Let $\mathcal{D}$ have n states, labeled q_1 to q_n. We proceed as in (Book et al., 1971) by adding two new states, q_0 and q_{n+1}. We fill out the base case table α^0. We then construct a new table, β where $\beta^k_{i,j} = \mathbb{M}_{\mathcal{P}}(\alpha^k_{i,j})$. In other words, α holds the regular expressions generated during state elimination while β holds the corresponding matrices. Since α^0 contains only unambiguous regular expressions, $\mathbb{M}_{\mathcal{P}}(\alpha^0_{i,j})$ is the matrix corresponding to the sum of $\mathbb{M}_P(w)$ for all w that, starting from state q_i travel to state q_j without passing through any states with label greater than 0. We continue the elimination process until we reach α^n and β^n. The regular expression in $\alpha^n_{0,n+1}$ corresponds to the unambiguous regular expression containing all strings accepted by $\mathcal{D}$. Thus, the matrix stored in $\beta^n_{0,n+1}$ is the matrix such that

$$\mathbb{I}_{\mathcal{P}}\mathbb{M}_{\mathcal{P}}(\alpha^n_{0,n+1})\mathbb{F}_{\mathcal{P}} = \mathbb{I}_{\mathcal{P}}\beta^n_{0,n+1}\mathbb{F}_{\mathcal{P}} = \sum_{w \in L(\mathcal{D})} \mathcal{P}(w).$$

□

5 Incremental Infix Calculation

We now tackle the open problem of incrementally computing the infix probability of a string with respect to a PFA. Suppose we have computed the infix probability of a string w. We want to use the result of that computation to compute the infix probability of wa without simply starting the computation again from scratch. Such a calculation is relatively easy for other affixes. For the prefix probability of a string represented by the unambiguous regular expression $w\Sigma^*$, one can simply compute $\mathbb{I}\mathbb{M}(w)\mathbb{M}(\Sigma^*)\mathbb{F}$ and save the vector $\mathbb{I}\mathbb{M}(w)$. When

the prefix is extended to wa, the saved vector can be multiplied by $\mathbb{M}(a)\mathbb{M}(\Sigma^*)\mathbb{F}$ and obtain the result (we can save the vector $\mathbb{I}\mathbb{M}(w)\mathbb{M}(a)$ to use in future incremental calculations). A similar process works for the incremental suffix probability of a string ($\Sigma^* w$ to $\Sigma^* aw$). These incremental approaches are due to the inherent unambiguity of regular expressions for strings appearing as a prefix or suffix.

Unfortunately, the analogous method for infix probabilities is not as straightforward. We cannot simply append (or prepend) the desired character to a precomputed regular expression because the resulting expression may not be unambiguous or may represent the a different language. To tackle this problem, we first define the language $\mathcal{F}(w)$ to be the set of strings that end in the first occurrence of w. In other words,

$$\mathcal{F}(w) = \{x \mid w \text{ appears only as a suffix of } x\}.$$

It follows that $\mathcal{F}(w) \cdot \Sigma^*$ is exactly the set of strings containing w as an infix. Thus, given an unambiguous regular expression for $\mathcal{F}(w)$, we can build an unambiguous regular expression for the infix of w by concatenating with Σ^*.

Next, we find a regular language $\mathcal{L}$ such that $\mathcal{F}(wa) = \mathcal{F}(w) \cdot \mathcal{L}$ for a character $a \in \Sigma$, which gives rise to an incremental computation using the previous result $\mathcal{F}(w)$. Given two languages R and S, we define the *left quotient* $R\backslash S$ to be:

$$R\backslash S = \{y \mid \exists x \in R \text{ such that } xy \in S\}.$$

It is known that regular languages are closed under the left quotient operation (Hopcroft and Ullman, 1979).

Corollary 2 follows from the definition of the left quotient and describes the desired $\mathcal{L}$.

Corollary 2. *Given $\mathcal{F}(w)$ and $\mathcal{F}(w)\backslash\mathcal{F}(wa)$, $\mathcal{F}(wa) = \mathcal{F}(w) \cdot \mathcal{F}(w)\backslash\mathcal{F}(wa)$.*

We use this characteristic and compute $\mathcal{F}(w)\backslash\mathcal{F}(wa)$ without explicitly computing $\mathcal{F}(wa)$ based on state elimination—the procedure is detailed later in this section. Figure 2 is an annotated example of unambiguous regular expressions for $\mathcal{F}$ of a, aa, and aab.

Let $\mathcal{D}$ be the DFA for $\mathcal{F}(w_1 w_2 \cdots w_n)$ generated by the KMP algorithm (Knuth et al., 1977). Two examples are depicted in Figure 3. $\mathcal{D}$ has $n+1$ states, with state q_{n+1} being final and having no outgoing transitions. Furthermore, each state q_i with $i < n+1$ has exactly one outgoing transition to state q_{i+1} and all other transitions are to states with labels at most i. Thus, removing state q_{n+1} and all incoming and outgoing transitions and making q_n the only final state results in the DFA for $\mathcal{F}(w_1 w_2 \cdots w_{n-1})$. This process can be repeated until the empty string is reached. This leads to the observation that, when constructing an unambiguous expression for $\mathcal{F}(w_1 w_2 \cdots w_n)$, we can recover the expression for $\mathcal{F}(w_1 w_2 \cdots w_{n-1})$ extracting the regular expression at $\alpha_{0,n}^{n-1}$. Similarly, the expression for $\mathcal{F}(w_1 w_2 \cdots w_k)$ is stored at $\alpha_{0,k+1}^{k}$.

$$\mathcal{F}(a) = b^* a$$

$$\mathcal{F}(aa) = \underbrace{b^* a}_{\mathcal{F}(a)} \cdot \underbrace{(bb^* a)^* a}_{\mathcal{F}(a)\backslash\mathcal{F}(aa)}$$

$$\mathcal{F}(aab) = \underbrace{b^* a (bb^* a)^* a}_{\mathcal{F}(aa)} \cdot \underbrace{a^* b}_{\mathcal{F}(aa)\backslash\mathcal{F}(aab)}$$

Figure 2: An example of $\mathcal{F}$ for a, aa, aab. We annotate them to show how $\mathcal{F}$ can be built up incrementally using previously computed regular expressions.

At stage k of the state elimination procedure on $\mathcal{D}$, state q_0 is only connected to states up to label $k-1$, thus $\alpha_{0,k+1}^{k-1} = \emptyset$. Since

$$\alpha_{0,k+1}^{k} = \alpha_{0,k+1}^{k-1} + \alpha_{0,k}^{k-1} (\alpha_{k,k}^{k-1})^* \alpha_{k,k+1}^{k-1},$$

we can simplify the expression as

$$\alpha_{0,k+1}^{k} = \alpha_{0,k}^{k-1} (\alpha_{k,k}^{k-1})^* \alpha_{k,k+1}^{k-1}.$$

In addition, we recall

$$\alpha_{0,k}^{k-1} = \mathcal{F}(w_1 w_2 \cdots w_{k-1}).$$

Therefore,

$$\mathcal{F}(w_1 \cdots w_k) = \mathcal{F}(w_1 \cdots w_{k-1}) (\alpha_{k,k}^{k-1})^* \alpha_{k,k+1}^{k-1}$$

and

$$(\alpha_{k,k}^{k-1})^* \alpha_{k,k+1}^{k-1} = \mathcal{F}(w_1 \cdots w_{k-1}) \backslash \mathcal{F}(w_1 \cdots w_k).$$

Algorithm 2 is for the offline setting—we know the entire string ahead of time and compute the infix probabilities of each of its prefixes incrementally. In Algorithm 2, following Section 3 and the relationship between the tables α and β in the proof of Lemma 1, we run the matrix evaluations of each regular expression instead of the

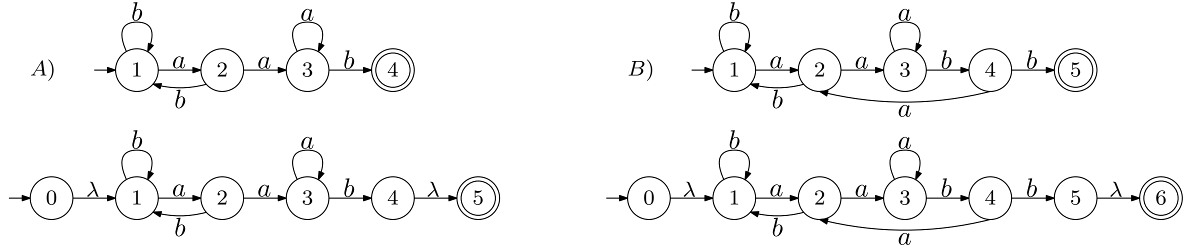

Figure 3: DFAs generated by the KMP algorithm. DFA A) accepts $\mathcal{F}(aab)$ and B) accepts $\mathcal{F}(aabb)$. The automata below are the new finite automata after adding the new initial and final state for the state elimination procedure, as described in Section 3.

regular expressions directly. At any step k, we only consider tables α^k and α^{k-1} and, thus, we can employ a standard sliding window technique for dynamic programming to reduce the required space complexity. In this scheme, instead of holding all tables up to α^k, we simply hold the two most previous ones, reducing the space complexity required by a factor of $O(k)$. We call our two tables T and T' and, to simplify the pseudocode, make elements of these two tables behave as both matrices and regular expressions. For example, $(T_{i,j})^*$ simultaneously corresponds to the regular expression $(\alpha_{i,j})^*$ and the matrix $\mathbb{M}((\alpha_{i,j})^*) = (\mathbf{1} - \mathbb{M}(\alpha_{i,j}))^{-1}$.

Algorithm 2 Offline Incremental Infix

1: **procedure** INFIX($w = w_1 w_2 \cdots w_n \in \Sigma^*$)
2: $\quad\mathcal{D} \leftarrow$ DFA accepting $\mathcal{F}(w)$
3: $\quad T \leftarrow (n+3) \times (n+3)$ table
4: $\quad T_{0,1} \leftarrow \mathbf{1}$
5: $\quad T_{n+1,n+2} \leftarrow \mathbf{1}$
6: $\quad$**for** $i \in [1, n+2]; j \in [1, n+2]; c \in \Sigma$ **do**
7: $\qquad$**if** $\delta(q_i, c) = q_j$ **then**
8: $\qquad\quad T_{i,j} \leftarrow T_{i,j} + \mathbb{M}(c)$
9: $\qquad$**end if**
10: $\quad$**end for**
11: $\quad\mathbb{V} \leftarrow \mathbb{I}$
12: $\quad$**for** $k \in [0, n+1]$ **do**
13: $\qquad\mathbb{V} \leftarrow \mathbb{V}(T_{k,k})^* T_{k,k+1}$
14: $\qquad$**yield** $\mathbb{VM}(\Sigma^*)\mathbb{F}$
15: $\qquad T' \leftarrow (n+3) \times (n+3)$ table
16: $\qquad$**for** $i \in [0, n+2]; j \in [0, n+2]$ **do**
17: $\qquad\quad T'_{i,j} \leftarrow T_{i,j} + T_{i,k}(T_{k,k})^* T_{k,j}$
18: $\qquad$**end for**
19: $\qquad T \leftarrow T'$
20: $\quad$**end for**
21: **end procedure**

Algorithm 2 begins by constructing DFA for $\mathcal{F}(w)$, which takes linear time in the length of w (Knuth et al., 1977) on Line 2. This DFA is modified to contain the new start and end state as described in Section 3 for a total of $n + 3$ states. We then begin to step through the state elimination algorithm. After constructing the initial table based on the base cases described in Section 3 from Line 3 to 10, we prepare our vector $\mathbb{V}$ that will record results from the previous infix calculation. At the beginning, $\mathbb{V}$ should hold $\mathbb{IM}(\mathcal{F}(\lambda)) = \mathbb{I}$. In the first step, we eliminate state q_1 of the automaton. Since state q_0 leads to state q_1 with a λ-transition, $\mathcal{F}$ at this step corresponds to $\mathcal{F}(\lambda)$ and $\mathcal{F}(\lambda) \cdot \Sigma^* = \Sigma^*$, which trivially has infix probability 1. At stage k of the for loop, we compute $\mathcal{F}(w_1 w_2 \cdots w_{k-1}) \backslash \mathcal{F}(w_1 w_2 \cdots w_k)$ on Line 13. We multiply this by $\mathbb{V}$, which holds $\mathcal{F}(w_1 w_2 \cdots w_{k-1})$, which makes $\mathbb{V}$ now store $\mathcal{F}(w_1 w_2 \cdots w_k)$. On Line 14, we emit the infix probability of $w_1 w_2 \cdots w_k$ by evaluating $\mathbb{VM}_{\mathcal{P}}(\Sigma^*)\mathbb{F}_{\mathcal{P}}$. From Line 15 to 18, we update our state elimination table. On Line 19, we use the sliding window technique and copy our updated table into T which allows for T' to be safely overwritten in the next iteration. This process repeats until we have removed all states q_i for $1 \leq i \leq n+1$, and therefore have computed the infix probability of each prefix of w.

The loop starting on Line 12 executes $O(|w|)$ times since the automaton has $|w| + 3$ states. At each iteration, we recompute the state-elimination table, which has size $O(|w|^2)$. For each element of the table we must perform one matrix addition, two matrix multiplications, and one matrix inverse on the matrices from the PFA, which has overall time complexity $O(|Q|^m)$. Thus, the runtime of Algorithm 2 is $O(|w|(|w|^2|Q|^m)) = O(|w|^3|Q|^m)$, where m is the matrix multiplication constant. Throughout the computa-

States	$\lvert Q\rvert = 614$		$\lvert Q\rvert = 1028$		$\lvert Q\rvert = 1455$	
Infix Length	Incremental	Intersection	Incremental	Intersection	Incremental	Intersection
1	0.226	0.147	0.857	0.468	2.383	1.079
2	0.272	0.316	1.072	1.235	3.000	3.112
3	0.334	0.637	1.327	2.634	3.693	6.997
4	0.399	1.133	1.586	4.864	4.442	13.250
5	0.465	1.934	1.855	8.104	5.124	22.357
6	0.527	3.375	2.088	12.562	5.815	35.065
7	0.584	4.129	2.347	18.414	6.593	51.709
8	0.649	5.791	2.591	25.614	7.224	72.512
9	0.711	7.879	2.851	34.959	7.950	99.347
Total	4.169	25.342	16.574	108.853	46.224	305.428

Table 1: Experimental results (in seconds) for the incremental regular expression method and the intersection method. The average of 10 runs is reported. For the incremental test, we measure the time for the step of building the state elimination table and outputting the current infix probability.

tion, we generate a series of tables of size $O(|w| \times |w|)$ with elements of size $O(|Q_{\mathcal{P}}| \times |Q_{\mathcal{P}}|)$. However, since we use the sliding window technique, we only save $O(1)$ of these tables, leading to an overall space complexity of $O((|w||Q_{\mathcal{P}}|)^2)$. Note that the space complexity matches that of the intersection method, but the time complexity of the incremental method is asymptotically faster; $O(|w|^3|Q_{\mathcal{P}}|^m)$ compared to $O(|w|(|w||Q_{\mathcal{P}}|)^m) = O(|w|^{m+1}|Q_{\mathcal{P}}|^m)$.

6 Experimental Results

We verify the effectiveness of our method using real-world data. We present 3 n-grams[3] (two 2-grams and one 3-gram) and their 3 PFAs with 614, 1028, and 1455 states extracted from the Brown Corpus tagged with the Penn Tree Bank tagset using the NLTK library (Bird and Loper, 2004). As a preprocessing step, we convert all punctuation to the `PUNC` tag for a total of 35 tags. The n-grams are built by computing the probability of reading a given tag under the condition that we have just seen a specific sequence of n tags. We then randomly select a sequence of 9 tags as our string across all experiments. For each of the n-grams, we calculate the infix probability of each prefix of our input string using the incremental regular expression method and the intersection method. We record the average of 10 runs per infix calculation. All experiments were written in Python 3.5 and the automata and matrices were implemented using NumPy. The experiments were run on an AMD Ryzen 7 1700 (3.0 GHz) 8-Core Processor with 16GB of RAM.

In the worst-case (the case when $|Q| = 1028$ in Table 1), the cumulative time for the incremental shows an 556.77% speed-up. The largest individual infix probability calculation is in the experiment with $|Q| = 1455$ when calculating the infix of length 9, where the incremental method obtains a 560.76% speed-up. These results show that the incremental method is not only theoretically faster, but also much faster in practice when compared to the intersection method.

The runtime gap is quite large compared to what is expected from the theoretical analysis of the two algorithms. Additionally, in theory, the time for each step of the incremental calculation should remain essentially constant, but the experimental results show that it grows at a slow rate. These are most likely implementation issues stemming from the large memory requirements involved in the calculations.

The experimental results show that the incremental infix method vastly outperforms the naive intersection method. The ability to memoize previous calculations allows the incremental approach to calculate the next infix probability much faster than simply restarting the entire calculation. This empirically verifies the asymptotic analysis of the two approaches.

7 Conclusions

We have presented a new method for solving the open problem of incrementally computing the infix probabilities of a given string with respect to a PFA. Our method utilizes unambiguous regular expressions and is distinguished from the previous

[3]We have a similar performance improvement for all other test cases that we extract from the dataset.

methods in that it does not alter the structure of the PFA during the evaluation. Similarly to the algorithm presented in (Nederhof and Satta, 2011a), this method imposes no restrictions on the input string.

We have showed that our method is asymptotically faster than the previously best-known method. Furthermore, we have experimentally evaluated the performance of our algorithm on a real life dataset and have observed that the proposed algorithm performs significantly better than the intersection method in all cases.

Future directions of this line of research are to determine a method for two sided incremental infix computation—that is, given a computation for the infix of w, compute the infix of wa or aw at will. Currently, it is only possible to do one sided incremental infix calculations. Computing the infix incrementally in an online fashion—in which we do not know the entire string ahead of time and receive new characters in a stream—would be another improvement. We believe that the current method can be modified to work in an online setting, possibly with an increase in runtime. Furthermore, finding new classes of problems that can benefit from a similar incremental calculation is also interesting. Finally, extending this method to work for PCFGs and more complex probabilistic models is an important open problem.

Acknowledgements

The authors thank the reviewers for their detailed and constructive comments.

This work was supported by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (2018-0-00247, 2018-0-00276).

References

Steven Bird and Edward Loper. 2004. NLTK: The natural language toolkit. In *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*.

Ronald Book, Shimon Even, Sheila Greiback, and Gene Ott. 1971. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20:149–153.

Dogan Can and Shrikanth Narayanan. 2015. A dynamic programming algorithm for computing n-gram posteriors from lattices. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 2388–2397.

Anna Corazza, Renato De Mori, Roberto Gretter, and Giorgio Satta. 1991. Computation of probabilities for an island-driven parser. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):936–950.

Ana L. N. Fred. 2000. Computation of substring probabilities in stochastic grammars. In *Grammatical Inference: Algorithms and Applications*, pages 103–114.

Adrià de Gispert, Graeme Blackwood, Gonzalo Iglesias, and William Byrne. 2013. N-gram posterior probability confidence measures for statistical machine translation: an empirical study. *Machine Translation*, 2:85–114.

Binod Gyawali, Gabriela Ramírez-de-la-Rosa, and Thamar Solorio. 2013. Native language identification: a simple n-gram based approach. In *Proceedings of the Eighth Workshop on Innovative Use of NLP for Building Educational Applications, BEA@NAACL-HLT*, pages 224–231.

John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350.

Robert McNaughton and Hisao Yamada. 1960. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9:39–47.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Mark-Jan Nederhof and Giorgio Satta. 2011a. Computation of infix probabilities for probabilistic context-free grammars. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1213–1221.

Mark-Jan Nederhof and Giorgio Satta. 2011b. Prefix probabilities for linear context-free rewriting systems. In *Proceedings of the 12th International Conference on Parsing*, pages 151–162.

Corinna Ng, Ross Wilkinson, and Justin Zobel. 2000. Experiments in spoken document retrieval using phoneme n-grams. *Speech Communication*, 32(1-2):61–77.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201.

Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356.

Enrique Vidal, Franck Thollard, Colin de la Higuera, Francisco Casacuberta, and Rafael C. Carrasco. 2005a. Probabilistic finite-state machines–part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1013–1025.

Enrique Vidal, Franck Thollard, Colin de la Higuera, Francisco Casacuberta, and Rafael C. Carrasco. 2005b. Probabilistic finite-state machines–part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1026–1039.

Theresa Wilson and Stephan Raaijmakers. 2008. Comparing word, character, and phoneme n-grams for subjective utterance recognition. In *INTERSPEECH 2008, 9th Annual Conference of the International Speech Communication Association*, pages 1614–1617.