An Algebra for Feature Extraction

Vivek Srikumar
School of Computing
University of Utah
svivek@cs.utah.edu

Abstract

Though feature extraction is a necessary first step in statistical NLP, it is often seen as a mere preprocessing step. Yet, it can dominate computation time, both during training, and especially at deployment. In this paper, we formalize feature extraction from an algebraic perspective. Our formalization allows us to define a message passing algorithm that can restructure feature templates to be more computationally efficient. We show via experiments on text chunking and relation extraction that this restructuring does indeed speed up feature extraction in practice by reducing redundant computation.

1 Introduction

Often, the first step in building statistical NLP models involves feature extraction. It is well understood that the right choice of features can substantially improve classifier performance. However, from the computational point of view, the process of feature extraction is typically treated, at best as the preprocessing step of caching featurized inputs over entire datasets, and at worst, as 'somebody else's problem'. While such approaches work for training, when trained models are deployed, the computational cost of feature extraction cannot be ignored.

In this paper, we present the first (to our knowledge) algebraic characterization of the process of feature extraction. We formalize feature extractors as arbitrary functions that map objects (words, sentences, etc) to a vector space and show that this set forms a commutative semiring with respect to feature addition and feature conjunction.

An immediate consequence of the semiring characterization is a computational one. Every semiring admits the *Generalized Distributive Law (GDL) Algorithm* (Aji and McEliece, 2000) that exploits the distributive property to provide computational speedups. Perhaps the most common manifestation of this algorithm in NLP is in the form of inference algorithms for factor graphs and Bayesian networks like the max-product, max-sum and sum-product algorithms (*e.g.* Goodman, 1999; Kschischang et al., 2001). When applied to feature extractors, the GDL algorithm can refactor a feature extractor into a faster one by reducing redundant computation. In this paper, we propose a junction tree construction to allow such refactoring. Since the refactoring is done at the feature template level, the actual computational savings grow as classifiers encounter more examples.

We demonstrate the practical utility of our approach by factorizing existing feature sets for text chunking and relation extraction. We show that, by reducing the number of operations performed, we can obtain significant savings in the time taken to extract features.

To summarize, the main contribution of this paper is the recognition that feature extractors form a commutative semiring over addition and conjunction. We demonstrate a practical consequence of this characterization in the form of a mechanism for automatically refactoring any feature extractor into a faster one. Finally, we show the empirical usefulness of our approach on relation extraction and text chunking tasks.

2 Problem Definition

Before formal definitions, let us first see a running example.

2.1 Motivating Example

Consider the frequently used unigram, bigram and trigram features. Each of these is a template that specifies a feature representation for a word. In

fact, the `bigram` and `trigram` templates themselves are compositional by definition. A `bigram` is simply the conjunction of a word `w` and previous word, which we will denote as $\texttt{w}_{-1}$; *i.e.*, $\texttt{bigram} = \texttt{w}_{-1}\&\texttt{w}$. Similarity, a `trigram` is the conjunction of $\texttt{w}_{-2}$ and `bigram`.

These templates are a function that operate on inputs. Given a sentence, say *John ate alone*, and a target word, say *alone*, they will produce indicators for the strings $\texttt{w}$=*alone*, $\texttt{w}_{-1}$=*ate*&$\texttt{w}$=*alone* and $\texttt{w}_{-2}$=*John*&$\texttt{w}_{-1}$=*ate*&$\texttt{w}$=*alone* respectively. Equivalently, each template maps an input to a vector. Here, the three vectors will be basis vectors associated with the feature strings.

Observe that the function that extracts the target word (*i.e.*, `w`) has to be executed in all three feature templates. Similarly, $\texttt{w}_{-1}$ has to be extracted to compute both the bigrams and the trigrams. *Can we optimize feature computation by automatically detecting such repetitions?*

2.2 Definitions and Preliminaries

Let X be a set of inputs to a classification problem at hand; *e.g.*, X could be words, sentences, etc. Let $\mathcal{V}$ be a possibly infinite dimensional vector space that represents the feature space. Feature extractors are functions that map the input space X to the feature space $\mathcal{V}$ to produce feature vectors for inputs. Let $\mathcal{F}$ represent the set of feature functions, defined as the set $\{\texttt{f} : X \to \mathcal{V}\}$. We will use the typewriter font to denote feature functions like `w` and `bigram`.

To round up the definitions, we will name two special feature extractors in $\mathcal{F}$. The feature extractor $\mathbb{0}$ maps all inputs to the zero vector. The feature extractor $\mathbb{1}$ maps all inputs to a bias feature vector. Without loss of generality, we will designate the basis vector $i_0 \in \mathcal{V}$ as the bias feature vector.

In this paper, we are concerned about two generally well understood operators on feature functions – addition and conjunction. However, let us see formal definitions for completeness.

Feature Addition. Given two feature extractors $\texttt{f}_1, \texttt{f}_2 \in \mathcal{F}$, feature addition (denoted by $+$) produces a feature extractor $\texttt{f}_1 + \texttt{f}_2$ that adds up the images of $\texttt{f}_1$ and $\texttt{f}_2$. That is, for any example $x \in X$, we have

$$(\texttt{f}_1 + \texttt{f}_2)(x) = \texttt{f}_1(x) + \texttt{f}_2(x) \qquad (1)$$

For example, the feature extractor $\texttt{w} + \texttt{w}_{-1}$ will map the word *alone* to a vector that is one for the basis elements $\texttt{w}$=*alone* and $\texttt{w}_{-1}$=*went*. This vector is the sum of the indicator vectors produced by the two operands `w` and $\texttt{w}_{-1}$.

Feature Conjunction. Given two feature extractors $\texttt{f}_1, \texttt{f}_2 \in \mathcal{F}$, their conjunction (denoted by $\&$) can be interpreted as an extension of Boolean conjunction. Indicator features like `bigram` are predicates for certain observations. Conjoining indicator features for two predicates is equivalent to an indicator feature for the Boolean conjunction of the predicates. More generally, with feature extractors that produce real valued vectors, the conjunction will produce their tensor product. The equivalence of feature conjunctions to tensor products has been explored and exploited in recent literature for various NLP tasks (Lei et al., 2014; Srikumar and Manning, 2014; Gormley et al., 2015; Lei et al., 2015).

We can further generalize this with an additional observation that is crucial for the rest of this paper. We argue that the conjunction operator produces *symmetric tensor products* rather than general tensor products. To see why, consider the `bigram` example. Though we defined the `bigram` feature as the conjunction of $\texttt{w}_{-1}$ and `w`, their ordering is irrelevant from classification perspective – the eventual goal is to associate weights with this combination of features. This observation allows us to formally define the conjunction operator as:

$$(\texttt{f}_1\&\texttt{f}_2)(x) = \text{vec}\left(\texttt{f}_1(x) \odot \texttt{f}_2(x)\right) \qquad (2)$$

Here, $\text{vec}(\cdot)$ stands for vectorize, which simply converts the resulting tensor into a vector and $\odot$ denotes the symmetric tensor product, introduced by Ryan (1980, Proposition 1.1). A symmetric tensor product is defined to be the average of the tensor products of all possible permutations of the operands, and thus, unlike a simple tensor product, is invariant to permutation of is operands. Informally, if we think of a tensor as a mapping from an ordered sequence of keys to real numbers, then, symmetric tensor product can be thought of as a mapping from a *set* of keys to numbers.

3 An Algebra for Feature Extraction

In this section, we will see that the set of feature extractors $\mathcal{F}$ form a commutative semiring with respect to addition and conjunction. First, let us revisit the definition of a commutative semiring.

Definition 1. *A* commutative semiring *is an algebraic structure consisting of a set K and two bi-*

nary operations $\oplus$ and $\otimes$ (addition and multiplication respectively) such that:

S1. $(K, \oplus)$ *is a commutative monoid:* $\oplus$ *is associative and commutative, and the set* K *contains a unique additive identity* 0 *such that* $\forall x \in K$*, we have* $0 \oplus x = x \oplus 0 = x$.

S2. $(K, \otimes)$ *is a commutative monoid:* $\otimes$ *is associative and commutative, and the set* K *contains a unique multiplicative identity* 1 *such that* $\forall x \in K$*, we have* $1 \otimes x = x \otimes 1 = x$.

S3. Multiplication distributes over addition on both sides. That is, for any $x, y, z \in K$*, we have* $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ *and* $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$.

S4. The additive identity is an annihilating element with respect to multiplication. That is, for any $x \in K$*, we have* $x \otimes 0 = 0 = 0 \otimes x$.

We refer the reader to Golan (2013) for a broad-ranging survey of semiring theory. We can now state and prove the main result of this paper.

Theorem 1. *Let* X *be any set and let* $\mathcal{F}$ *denote the set of feature extractors defined on the set. Then,* $(\mathcal{F}, +, \&)$ *is a commutative semiring.*

Proof. We will show that the properties of a commutative semiring hold for $(\mathcal{F}, +, \&)$ using the definitions of the operators from §2.2. Let $\mathtt{f}_1, \mathtt{f}_2$ and $\mathtt{f} \in \mathcal{F}$ be feature extractors.

S1. For any example $x \in X$, we have $(\mathtt{f}_1 + \mathtt{f}_2)(x) = \mathtt{f}_1(x) + \mathtt{f}_2(y)$. The right hand side denotes vector addition, which is associative and commutative. The $\mathbb{0}$ feature extractor is the additive identify because it produces the zero vector for any input. Thus, $(\mathcal{F}, +)$ is a commutative monoid.

S2. To show that the conjunction operator is associative over feature extractors, it suffices to observe that the tensor product (and hence the symmetric tensor product) is associative. Furthermore, the symmetric tensor product is commutative by definition, because it is invariant to permutation of its operands.
Finally, the bias feature extractor, $\mathbb{1}$, that maps all inputs to the bias vector i_0, is the multiplicative identity. To see this, consider the conjunction $\mathtt{f}\&\mathbb{1}$, applied to an input x:

$$\begin{aligned}(\mathtt{f}\&\mathbb{1})(x) &= \text{vec}(\mathtt{f}(x) \odot \mathbb{1}(x)) \\ &= \text{vec}(\mathtt{f}(x) \odot i_0)\end{aligned}$$

The product term within the $\text{vec}(\cdot)$ in the final expression is a symmetric tensor, defined by basis vectors that are sets of the form $\{i_0, i_0\}, \{i_1, i_0\}, \cdots$. Each basis $\{i_j, i_0\}$ is associated with a feature value $\mathtt{f}(x)_j$. Thus, the vectorized form of this tensor will contain the *same elements* as $\mathtt{f}(x)$, perhaps mapped to different bases. The mapping from $\mathtt{f}(x)$ to the final vector is independent of the input x because the bias feature extractor is independent of x. Without loss of generality, we can fix this mapping to be the identity mapping, thereby rendering the final vectorized form equal to $\mathtt{f}(x)$. That is, $\mathtt{f}\&\mathbb{1} = \mathtt{f}$.
Thus, $(\mathcal{F}, \&)$ is a commutative monoid.

S3. Since tensor products distribute over addition, we get the distributive property.

S4. By definition, conjoining with the $\mathbb{0}$ feature extractor annihilates all feature functions because $\mathbb{0}$ maps all inputs to the zero vector.

■

4 From Algebra to an Algorithm

The fact that feature extractors form a commutative semiring has a computational consequence. The *generalized distributive law (GDL) algorithm* (Aji and McEliece, 2000) exploits the properties of a commutative semiring to potentially reduce the computational effort for marginalizing sums of products. The GDL algorithm manifests itself as the Viterbi, Baum-Welch, Floyd-Warshall and belief propagation algorithms, and the Fast Fourier and Hadamard transforms. Each corresponds to a different commutative semiring and a specific associated marginalization problem.

Here, we briefly describe the general marginalization problem from Aji and McEliece (2000) to introduce notation and also highlight the analogies to inference in factor graphs. Let $x_1, x_2, \cdots, x_n$ denote a collection of variables that can take values from finite sets $A_1, A_2, \cdots, A_n$ respectively. Let boldface $\mathbf{x}$ denote the entire set of variables. These variables are akin to inference variables in factor graphs that may be assigned values or marginalized away.

Let $(K, \oplus, \otimes)$ denote a commutative semiring. Suppose α_i is a function that maps a subset of the variables $\{x_{i_1}, x_{i_2}, \cdots\}$ to the set K. The subset of variables that constitute the domain of α_i is called the *local domain* of the corresponding *local function*. Local domains and local functions are analogous to factors and factor potentials in a factor graph. With a collection of local domains, each associated with a function α_i, the "marginalize the

product" problem is that of computing:

$$\sum_{\mathbf{x}} \prod_{i} \alpha_i \left(x_{i_1}, x_{i_2}, \cdots\right) \tag{3}$$

Here, the sum and product use the semiring operators. The summation is over all possible valid assignments of the variables $\mathbf{x}$ over the cross product of the sets $A_1, A_2, \cdots, A_n$. This problem generalizes the familiar max-product or sum-product settings. Indeed, the GDL algorithm is a generalization of the message passing (Pearl, 2014) for efficiently computing marginals.

To make feature extraction efficient using the GDL algorithm, in the next section, we will define a marginalization problem in terms of the semiring operators by specifying the variables involved, the local domains and local functions. Instead of describing the algorithm in the general setting, we will instantiate it on the semiring at hand.

5 Marginalizing Feature Extractors

First, let us see why we can expect any benefit from the GDL algorithm by revisiting our running example (unigrams, bigrams and trigrams), written below using the semiring operations:

$$\mathtt{f} = \mathtt{w} + (\mathtt{w}_{-1} \& \mathtt{w}) + (\mathtt{w}_{-2} \& \mathtt{w}_{-1} \& \mathtt{w}) \tag{4}$$

When applied to a token, $\mathtt{f}$ performs two additions and three conjunctions. However, by applying the distributive property, we can refactor it as follows to reduce the number of operations:

$$\mathtt{f}' = (\mathbb{1} + (\mathbb{1} + \mathtt{w}_{-2}) \,\&\mathtt{w}_{-1}) \,\&\mathtt{w} \tag{5}$$

The refactored version $\mathtt{f}'$ – equivalent to the original one – only performs two additions and two conjunctions, offering a computational saving of one operation. This refactoring is done at the level of feature templates (*i.e.*, feature extractors); the actual savings are realized when the feature vectors are computed by applying this feature function to an input. Thus, the simplification, though seemingly modest at the template level, can lead to a substantial speed improvements when the features vectors are actually manifested from data.

The GDL algorithm instantiated with the feature extractor semiring, automates such factorization at a symbolic level. In the rest of this section, first (§5.1), we will write our problem as a marginalization problem, as in Equation (3). Then (§5.2), we will construct a junction tree to apply the message passing algorithm.

5.1 Canonicalizing Feature Extractors

To frame feature simplification as marginalization, we need to first write any feature extractor as a *canonical* sum of products that is amenable for factorization (*i.e.*, as in (3)). To do so, in this section, we will define: (a) the variables involved, (b) the local domains (*i.e.*, subsets of variables contributing to each product term), and, (c) a local function for each local domain (*i.e.*, the α_i's).

Variables. First, we write a feature extractor as a sum of products. Our running example (4) is already one. If we had an expression like $\mathtt{f}_1 \& (\mathtt{f}_2 + \mathtt{f}_3)$, we can expand it into $\mathtt{f}_1 \& \mathtt{f}_2 + \mathtt{f}_1 \& \mathtt{f}_3$. From the sum of products, we identify the *base feature extractors* (*i.e.*, ones not composed of other feature extractors) and define a variable x_i for each. In our example, we have $\mathtt{w}$, $\mathtt{w}_{-1}$ and $\mathtt{w}_{-2}$.

Next, recall from §4 that each variable x_i can take values from a finite set A_i. If a base feature extractor $\mathtt{f}_i$ corresponds to the variable x_i, then, we define x_i's domain to be the set $A_i = \{\mathbb{1}, \mathtt{f}_i\}$. That is, each variable can either be the bias feature extractor or the feature extractor associated with it. Our example gives three variables x_1, x_2, x_3 with domains $A_1 = \{\mathbb{1}, \mathtt{w}\}$, $A_2 = \{\mathbb{1}, \mathtt{w}_{-1}\}$, $A_3 = \{\mathbb{1}, \mathtt{w}_{-2}\}$ respectively.

Local domains. Local domains are subsets of the variables defined above. They are the domains of functions that constitute products in the canonical form of a feature extractor. We define the following local domains, each illustrated with the corresponding instantiation in our running example:

1. A singleton set for each variable: $\{x_1\}$, $\{x_2\}$, and $\{x_3\}$.
2. One local domain consisting of all the variables: The set $\{x_1, x_2, x_3\}$.
3. One local domain consisting of no variables: The empty set $\{\}$.
4. One local domain for each subset of base feature extractors that participate in at least two conjunctions in the sum-of-products (*i.e.*, the ones that can be factored away): Only $\{x_1, x_2\}$ in our example, because only $\mathtt{w}$ and $\mathtt{w}_{-1}$ participate in two conjunctions in (4).

Local functions. Each local domain is associated with a function that maps variable assignments to feature extractors. These functions (called local kernels by Aji and McEliece (2000)) are like potential functions in a factor graph. We define two kinds of local functions, driven by the goal of de-

signing a marginalization problem that pushes towards simpler feature functions.

1. We associate the *identity function* with all singleton local domains, and the constant function that returns the bias $\mathbb{1}$ with the empty domain $\{\}$.
2. With all other local domains, we associate an *indicator function*, denoted by z. For a local domain, z is an indicator for those assignments of the variables involved, whose conjunctions are present in any product term in sum-of-products. In our running example, the function $z(x_1, x_2)$ is the indicator for (x_1, x_2) belonging to the set $\{(\mathtt{w}, \mathbb{1}), (\mathtt{w}, \mathtt{w}_{-1})\}$, represented by the table:

x_1	x_2	$z(x_1, x_2)$
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$
$\mathbb{1}$	$\mathtt{w}_{-1}$	$\mathbb{0}$
$\mathtt{w}$	$\mathbb{1}$	$\mathbb{1}$
$\mathtt{w}$	$\mathtt{w}_{-1}$	$\mathbb{1}$

   The indicator returns the semiring's multiplicative and additive identities. The value of z above for inputs $(\mathtt{w}, \mathbb{1})$ is $\mathbb{1}$ because the first term in (4) that defines the feature extractor contains $\mathtt{w}$, but not $\mathtt{w}_{-1}$. On the other hand, the input $(\mathbb{1}, \mathbb{1})$ is mapped to $\mathbb{0}$ because every product term contains either $\mathtt{w}$ or $\mathtt{w}_{-1}$. For the local domain $\{x_1, x_2, x_3\}$, the local function is the indicator for the set $\{(\mathtt{w}, \mathbb{1}, \mathbb{1}), (\mathtt{w}, \mathtt{w}_{-1}, \mathbb{1}), (\mathtt{w}, \mathtt{w}_{-1}, \mathtt{w}_{-2})\}$, corresponding to each product term.

In summary, for the running example we have:

Local domain	Local function
$\{x_1\}$	x_1
$\{x_2\}$	x_2
$\{x_3\}$	x_3
$\{x_1, x_2, x_3\}$	$z(x_1, x_2, x_3)$
$\{\}$	$\mathbb{1}$
$\{x_1, x_2\}$	$z(x_1, x_2)$

The procedure described here aims to convert *any* feature function into a canonical form that can be factorized using the GDL algorithm. Indeed, using local domains and functions specified above, any feature extractor can we written as a canonical sum of products as in (3). For example, using the table above, our running example is identical to

$$\sum_{x_1, x_2, x_3} z(x_1, x_2, x_3) \& z(x_1, x_2) \& x_1 \& x_2 \& x_3 \quad (6)$$

Here, the summation is over the cross product of the A_i's. The choice of the z functions ensures that only those conjunctions that were in the original feature extractor remain.

This section shows one approach for canonicalization; the local domains and functions are a design choice that may be optimized in future work. We should also point out that, while this process is notationally tedious, its actual computational cost is negligible, especially given that it is to be performed only once at the template level.

5.2 Simplifying feature extractors

As mentioned in §4, a commutative semiring can allow us to employ the GDL algorithm to efficiently compute a sum of products. Starting from a canonical sum-of-products expression such as the one in (6), this process is similar to variable elimination for Bayesian networks. The junction tree algorithm is a general scheme to avoid redundant computation in such networks (Cowell, 2006). To formalize this, we will first build a junction tree and then define the messages sent from the leaves to the root. The final message at the root will give us the simplified feature function.

Constructing a Junction Tree. First, we will construct a junction tree using the local domains from § 5.1. In any junction tree, the edges should satisfy the *running intersection property*: *i.e.*, if a variable x_i is in two nodes in the tree, then it should be in every node in the path connecting them. To build a junction tree, we will first create a graph whose nodes are the local domains. The edges of this graph connect pairs of nodes if the variables in one are a subset of the other. For simplicity, we will assume that our nodes are arranged in a lattice as shown in Figure 1, with edges connecting nodes in subsequent levels. For example, there is no edge connecting nodes `B` and `C`.

Every spanning tree of this lattice is a junction tree. Which one should we consider? Let us examine the properties that we need. First, the root of the tree should correspond to the empty local domain $\{\}$ because messages arriving at this node will accumulate all products. Second, as we will see, feature extractors farther from the root will appear in inner terms in the factorized form. That is, frequent or more expensive feature extractors should be incentivized to appear higher in the tree.

To capture these preferences, we frame the task of constructing the junction tree as a maximum spanning tree problem over the graph, with edge weights incorporating the preferences. One natural weighting function is the computational expense of the base feature extractors associated with that edge. For example, the weight associated with the edge connecting nodes `E` and `D` in the fig-

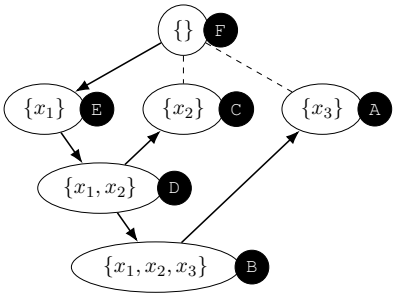

Figure 1: The junction tree for our running example. The process of constructing the junction tree is described in the text. Here, we show both the tree and the graph from which it is constructed; dashed lines show edges are not in the tree. Filled circles denote the names of the nodes. The local domain $\{x_1\}$ is connected to the empty local domain because the feature `w` corresponding to it is most frequent.

ure can be the average cost of the `w` and $\texttt{w}_{-1}$ feature extractors. If computational costs are unavailable, we can use the number times a feature extractor appears in the expression to be simplified. Under this criterion, in our example, edges connecting `E` to its neighbors will be weighted highest.

Once we have a spanning tree, we make the edges directed so that the empty set is the root. Figure 1 shows the junction tree obtained for our running example.

Message Passing for Feature Simplification. Given the junction tree, we can use a standard message passing scheme for factorization. The goal is to collect information at each node in the tree from its children all the way to the root.

Suppose v_i, v_j denote two nodes in the tree. Since nodes are associated with sets of variables, their intersections $v_i \cap v_j$ and differences $v_i \setminus v_j$ are defined. For example, in the example, $\texttt{A} \cap \texttt{B} = \{x_3\}$ and $\texttt{B} \setminus \texttt{D} = \{x_3\}$. We will denote children of a node v_i in the junction tree by $C(v_i)$.

The message from any node v_i to its parent v_j is a function that maps the variables $v_i \cap v_j$ to a feature extractor by marginalizing out all variables that are in v_i but not in v_j. Formally, we define the message μ_{ij} from a node v_i to a node v_j as:

$$\mu_{ij}\left(v_i \cap v_j\right) = \sum_{v_j \setminus v_i} \alpha_i\left(v_i\right) \prod_{v_k \in C(v_i)} \mu_{ki}\left(v_k \cap v_i\right). \quad (7)$$

Here, α_i is the local function at node v_i. To complete the formal definition of the algorithm, we note that by performing post-order traversal of the junction tree, we will accumulate all messages at the root of the tree, that corresponds to the empty set of variables. The incoming message at this node represents the factorized feature extractor.

Algorithm 1 briefly summarizes the entire simplification process. The proof of correctness of the algorithm follows from the fact that the range of all the local functions is a commutative semiring, namely the feature extractor semiring. We refer the reader to (Aji and McEliece, 2000, Appendix A) for details.

Algorithm 1 The Generalized Distributive Law Algorithm for simplifying a feature extractor `f`. See the text for details.

1: Convert `f` into a canonical sum of products representation (§ 5.1).
2: Construct a junction tree whose nodes are local domains.
3: **for** edge (v_j, v_i) in the post-order traversal of the tree **do**
4: Receive a message μ_{ij} at v_j using (7).
5: **end for**
6: **return** the incoming message at the root

Example run of message propagation. As an illustration, let us apply it to our running example.

1. The first message is from `A` to `B`. Since `A` has no children and its local function is the identity function, we have $\mu_{AB}(x) = x$. Similarly, we have $\mu_{CD}(x) = x$.
2. The message from `B` to `D` has to marginalize out the variable x_3. That is, we have $\mu_{BD}(x_1, x_2) = \sum_{x_3} z(x_1, x_2, x_3)\mu_{AB}(x_3)$. The summation is over the domain of x_3, namely $\{\mathbb{1}, \texttt{w}_{-2}\}$. By substituting for z and μ_{AB}, and simplifying, we get the message:

x_1	x_2	$\mu_{BD}(x_1, x_2)$
$\mathbb{1}$	$\mathbb{1}$	$\mathbb{0}$
$\mathbb{1}$	$\texttt{w}_{-1}$	$\mathbb{0}$
$\texttt{w}$	$\mathbb{1}$	$\mathbb{1}$
$\texttt{w}$	$\texttt{w}_{-1}$	$\mathbb{1} + \texttt{w}_{-2}$

3. The message from `D` to `E` marginalizes out the variable x_2 to give us $\mu_{DE}(x_1) = \sum_{x_2} z(x_1, x_2)\mu_{CD}(x_2)\mu_{BD}(x_1, x_2)$. Here, the summation is over the domain of x_2, namely $\{\mathbb{1}, \texttt{w}_{-1}\}$. We can simplify the message as:

x_1	$\mu_{DE}(x_1)$
$\mathbb{1}$	$\mathbb{0}$
$\texttt{w}$	$\mathbb{1} + (\mathbb{1} + \texttt{w}_{-2})\,\&\texttt{w}_{-1}$

4. Finally, the message from `E` to the root `F` marginalizes out the variable x_1 by summing over its domain $\{\mathbb{1}, \texttt{w}\}$ to give us the message $(\mathbb{1} + (\mathbb{1} + \texttt{w}_{-2})\,\&\texttt{w}_{-1})\,\&\texttt{w}$.

The message received at the root is the factorized feature extractor. Note that the final form is identical to (5) at the beginning of §5.

Discussion. An optimal refactoring algorithm would produce a feature extractor that is both correct and fastest. The algorithm above has the former guarantee. While it does reduce the number of operations performed, the closeness of the refac-

tored feature function to the fastest one depends on the heuristic used to weight edges for identifying the junction tree. Changing the heuristic can change the junction tree, thus changing the final factorized function. We found via experiments that using the number of times a feature extractor occurs in the sum-of-products to weight edges is promising. A formal study of optimality of factorization is an avenue of future research.

6 Experiments

We show the practical usefulness of feature function refactoring using text chunking and relation extraction. In both cases, the question we seek to evaluate empirically is: *Does the feature function refactoring algorithm improve feature extraction time?* We should point out that our goal is not to measure accuracy of prediction, but the efficiency of feature extraction. Indeed, we are guaranteed that refactoring will not change accuracy; factorized feature extractors produce the *same* feature vectors as the original ones.

In all experiments, we compare a feature extractor and its refactored variant. For the factorization, we incentivized the junction tree to factor out base feature extractors that occurred most frequently in the feature extractor. For both tasks, we use existing feature representations that we briefly describe. We refer the reader to the original work that developed the feature representations for further details. For both the original and the factorized feature extractors, we report (a) the number of additions and conjunctions at the template level, and, (b) the time for feature extraction on the entire dataset. For the time measurements, we report average times for the original and factorized feature extractors over five paired runs to average out variations in system load.[1]

6.1 Text Chunking

We use data from the CoNLL 2000 shared task (Tjong Kim Sang and Buchholz, 2000) of text chunking and the feature set described by Martins et al. (2011), consisting of the following templates extracted at each word: (1) Up to 3-grams of POS tags within a window of size ten centered at the word, (2) up to 3-grams of words, within a window of size six centered at the word, and (3) up to 2-grams of word shapes, within a window of size four centered at the word. In all, there are 96 feature templates.

[1]We performed all our experiments on a server with 128GB RAM and 24 CPU cores, each clocking at 2600 MHz.

Setting	Size		Average feature extraction time (ms)
	+	&	
Original	47	75	17776.6
Factorized	47	54	4294.2

Table 1: Comparison of the original and factorized feature extractors for the text chunking task. The time improvement is statistically significant using the paired t-test at $p < 0.01$.

Setting	Size		Average feature extraction time (ms)
	+	&	
Original	43	19	8173.0
Factorized	43	11	6276.4

Table 2: Comparison of the original and factorized feature extractors for the relation extraction task. We measured time using 3191 training mention pairs. The time improvement is statistically significant using the paired t-test at $p < 0.01$.

We factorized the feature representation using Algorithm 1. Table 1 reports the number of operations (addition and conjunction) in the templates in the original and factorized versions of the feature extractor. The table also reports feature extraction time taken from the entire training set of 8,936 sentences, corresponding to 211,727 tokens. First, we see that the factorization reduces the number of feature conjunction operations. Thus, to produce exactly the same feature vector, the factorized feature extractor does less work. The time results show that this computational gain is not merely a theoretical one; it also manifests itself practically.

6.2 Relation Extraction

Our second experiment is based on the task of relation extraction using the English section of the ACE 2005 corpus (Walker et al., 2006). The goal is to identify semantic relations between two entity mentions in text. We use the feature representation developed by Zhou et al. (2005) as part of an investigation of how various lexical, syntactic and semantic sources of information affect the relation extraction task. To this end, the feature set consists of word level information about mentions, their entity types, their relationships with chunks, path features from parse trees, and semantic features based on WordNet and various word lists. Given the complexity of the features, we do not describe them here and refer the reader to the original work for details. Note that compared to the chunking features, these features are more diverse in their computational costs.

We report the results of our experiments in Ta-

ble 2. As before, we see that the number of conjunction operations decreases after factorization. Curiously, however, despite the complexity of the feature set, the actual number of operations is smaller than text chunking. Due to this, we see a more modest, yet significant decrease in the time for feature extraction after factorization.

7 Related Work and Discussion

Simplifying Expressions. The problem of simplifying expressions with an eye on computational efficiency is the focus of logic synthesis (cf. Hachtel and Somenzi, 2006), albeit largely geared towards analyzing and verifying digital circuits. Logic synthesis is NP-hard in general. In our case, the hardness is hidden in the fact that our approach does not guarantee that we will find the smallest (or most efficient) factorization. The junction tree construction determines the factorization quality.

Semirings in NLP. Semirings abound in NLP, though primarily as devices to design efficient inference algorithms for various graphical models (*e.g.* Wainwright and Jordan, 2008; Sutton et al., 2012). Goodman (1999) synthesized various parsing algorithms in terms of semiring operations. Since then, we have seen several explorations of the interplay between weighted dynamic programs and semirings for inference in tasks such as parsing and machine translation (*e.g.* Eisner et al., 2005; Li and Eisner, 2009; Lopez, 2009; Gimpel and Smith, 2009). Allauzen et al. (2003) developed efficient algorithms for constructing statistical language models by exploiting the algebraic structure of the probability semiring.

Feature Extraction and Modeling Languages. Much work around features in NLP is aimed at improving classifier accuracy. There is some work on developing languages to better construct feature spaces (Cumby and Roth, 2002; Broda et al., 2013; Sammons et al., 2016), but they do not formalize feature extraction from an algebraic perspective. We expect that the algorithm proposed in this paper can be integrated into such feature construction languages, and also into libraries geared towards designing feature rich models (*e.g.* McCallum et al., 2009; Chang et al., 2015).

Representation vs. Speed. As the recent successes (Goodfellow et al., 2016) of distributed representations show, the representational capacity of a feature space is of primary importance. Indeed, several recent lines of work that use distributed representations have independently identified the connection between conjunctions (of features or factors in a factor graph) and tensor products (Lei et al., 2014; Srikumar and Manning, 2014; Gormley et al., 2015; Yu et al., 2015; Lei et al., 2015; Primadhanty et al., 2015). They typically impose sparsity or low-rank requirements to induce better representations for learning. In this paper, we use the connection between tensor products and conjunctions to prove algebraic properties of feature extractors, leading to speed improvements via factorization.

In this context, we note that in both our experiments, the number of conjunctions are reduced by factorization. We argue that this is an important saving because conjunctions can be a more expensive operation. This is especially true when dealing with dense feature representations, as is increasingly common with word vectors and neural networks, because conjunctions of dense feature vectors are tensor products, which can be slow.

Finally, while training classifiers can be time consuming, when trained classifiers are deployed, feature extraction will dominate computation time over the classifier's lifetime. However, the prediction step includes both feature extraction and computing inner products between features and weights. Many features may be associated with zero weights because of sparsity-inducing learning (*e.g.* Andrew and Gao, 2007; Martins et al., 2011; Strubell et al., 2015). Since these two aspects are orthogonal to each other, the factorization algorithm presented in this paper can be used to speed up extraction of those features that have non-zero weights.

8 Conclusion

In this paper, we studied the process of feature extraction using an algebraic lens. We showed that the set of feature extractors form a commutative semiring over addition and conjunction. We exploited this characterization to develop a factorization algorithm that simplifies feature extractors to be more computationally efficient. We demonstrated the practical value of the refactoring algorithm by speeding up feature extraction for text chunking and relation extraction tasks.

Acknowledgments

The author thanks the anonymous reviewers for their insightful comments and feedback.

References

Srinivas M Aji and Robert J McEliece. 2000. The generalized distributive law. *IEEE Transactions on Information Theory* 46(2).

Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2003. Generalized algorithms for constructing statistical language models. In *ACL*.

Galen Andrew and Jianfeng Gao. 2007. Scalable training of L_1-regularized log-linear models. In *ICML*.

Bartosz Broda, Paweł Kędzia, Michał Marcińczuk, Adam Radziszewski, Radosław Ramocki, and Adam Wardyński. 2013. Fextor: A feature extraction framework for natural language processing: A case study in word sense disambiguation, relation recognition and anaphora resolution. In *Computational Linguistics*, Springer, pages 41–62.

Kai-Wei Chang, Shyam Upadhyay, Ming-Wei Chang, Vivek Srikumar, and Dan Roth. 2015. IllinoisSL: A JAVA library for Structured Prediction. *arXiv preprint arXiv:1509.07179* .

Robert G Cowell. 2006. *Probabilistic networks and expert systems: Exact computational methods for Bayesian networks*. Springer Science & Business Media.

Chad M Cumby and Dan Roth. 2002. Learning with feature description logics. In *Inductive logic programming*, Springer.

Jason Eisner, Eric Goldlust, and Noah A Smith. 2005. Compiling Comp Ling: Practical weighted dynamic programming and the Dyna language. In *HLT-EMNLP*.

Kevin Gimpel and Noah A Smith. 2009. Cube summing, approximate inference with non-local features, and dynamic programming without semirings. In *EACL*.

Jonathan S Golan. 2013. *Semirings and their Applications*. Springer Science & Business Media.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics* 25(4):573–605.

Matthew R. Gormley, Mo Yu, and Mark Dredze. 2015. Improved relation extraction with feature-rich compositional embedding models. In *EMNLP*.

Gary D Hachtel and Fabio Somenzi. 2006. *Logic synthesis and verification algorithms*. Springer Science & Business Media.

Frank R Kschischang, Brendan J Frey, and H-A Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory* 47(2):498–519.

Tao Lei, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-rank tensors for scoring dependency structures. In *ACL*.

Tao Lei, Yuan Zhang, Lluís Màrquez, Alessandro Moschitti, and Regina Barzilay. 2015. High-order lowrank tensors for semantic role labeling. In *NAACL*.

Zhifei Li and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *EMNLP*.

Adam Lopez. 2009. Translation as weighted deduction. In *EACL*.

André FT Martins, Noah A Smith, Pedro MQ Aguiar, and Mário AT Figueiredo. 2011. Structured sparsity in structured prediction. In *CoNLL*.

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*.

Judea Pearl. 2014. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Audi Primadhanty, Xavier Carreras, and Ariadna Quattoni. 2015. Low-rank regularization for sparse conjunctive feature spaces: An application to named entity classification. In *ACL*.

Raymond A Ryan. 1980. *Applications of topological tensor products to infinite dimensional holomorphy*. Ph.D. thesis, Trinity College.

Mark Sammons, Christos Christodoulopoulos, Parisa Kordjamshidi, Daniel Khashabi, Vivek Srikumar, and Dan Roth. 2016. EDISON: Feature Extraction for NLP. In *LREC*.

Vivek Srikumar and Christopher D. Manning. 2014. Learning distributed representations for structured output prediction. In *NIPS*.

Emma Strubell, Luke Vilnis, Kate Silverstein, and Andrew McCallum. 2015. Learning Dynamic Feature Selection for Fast Sequential Prediction. In *ACL*.

Charles Sutton, Andrew McCallum, et al. 2012. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning* 4(4):267–373.

Erik F Tjong Kim Sang and Sabine Buchholz. 2000. Introduction to the CoNLL-2000 shared task: Chunking. In *CoNLL*.

Martin J Wainwright and Michael I Jordan. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning* 1(1–2):1–305.

Christopher Walker, Stephanie Strassel, Julie Medero, and Kazuaki Maeda. 2006. ACE 2005 multilingual training corpus. *Linguistic Data Consortium, Philadelphia* 57.

Mo Yu, Matthew R. Gormley, and Mark Dredze. 2015. Combining Word Embeddings and Feature Embeddings for Fine-grained Relation Extraction. In *NAACL*.

GuoDong Zhou, Jian Su, Jie Zhang, and Min Zhang. 2005. Exploring various knowledge in relation extraction. In *ACL*.