Journal of Machine Learning Research 4 (2003) 773-818 Submitted 5/03; Published 10/03

Path Kernels and Multiplicative Updates

Eiji Takimoto[*] T2@ECEI.TOHOKU.AC.JP
Graduate School of Information Sciences
Tohoku University, Sendai, 980-8579, Japan

Manfred K. Warmuth MANFRED@CSE.UCSC.EDU
Computer Science Department
University of California, Santa Cruz
CA 95064, USA

Editors: Ralf Herbrich and Thore Graepel

Abstract

Kernels are typically applied to linear algorithms whose weight vector is a linear combination of the feature vectors of the examples. On-line versions of these algorithms are sometimes called "additive updates" because they add a multiple of the last feature vector to the current weight vector.

In this paper we have found a way to use special convolution kernels to efficiently implement "multiplicative" updates. The kernels are defined by a directed graph. Each edge contributes an input. The inputs along a path form a product feature and all such products build the feature vector associated with the inputs. We also have a set of probabilities on the edges so that the outflow from each vertex is one. We then discuss multiplicative updates on these graphs where the prediction is essentially a kernel computation and the update contributes a factor to each edge. After adding the factors to the edges, the total outflow out of each vertex is not one any more. However some clever algorithms re-normalize the weights on the paths so that the total outflow out of each vertex is one again. Finally, we show that if the digraph is built from a regular expressions, then this can be used for speeding up the kernel and re-normalization computations.

We reformulate a large number of multiplicative update algorithms using path kernels and characterize the applicability of our method. The examples include efficient algorithms for learning disjunctions and a recent algorithm that predicts as well as the best pruning of a series parallel digraphs.

Keywords: Kernels, Multiplicative Updates, On-Line Algorithms, Series Parallel Digraphs.

1. Introduction

There is a large class of linear algorithms, such as the Linear Least Squares algorithm and Support Vector Machines, whose weight vector is a linear combination of the input vectors. Related on-line algorithms, such as the Perceptron algorithm and the Widrow Hoff algorithm, maintain a weight vector that is a linear combination of the past input vectors. The on-line weight update of these algorithm is *additive* in that a multiple of the last instance is added to the current weight vector.

The linear models are greatly enhanced by mapping the input vectors x to feature vectors $\Phi(x)$. The features may be non-linear, and the number of features is typically much larger than the input dimension. Now the above algorithms all use a linear model in feature space defined by a weight vector W of feature weights that is a linear combination of the expanded inputs $\Phi(x_q)$ $(1 \leq q \leq t)$

[*]. Part of this work was done while Eiji Takimoto visited the University of California at Santa Cruz.

of the training examples. Given an input vector x, the linear prediction $W \cdot \Phi(x)$ can be computed via the dot products $\Phi(x_q) \cdot \Phi(x)$ and these dot products can often be computed efficiently via an associated kernel function $K(x_q, x) = \Phi(x_q) \cdot \Phi(x)$.

In this paper we give kernel methods for *multiplicative* algorithms. Now the componentwise logarithm of the feature weight vector W is constant plus a linear combination of the expanded instances. In the on-line versions of these updates, the feature weights are multiplied by factors and then the weight vector is renormalized. This second family of algorithms is motivated by using an entropic regularization on the feature weights (Kivinen and Warmuth, 1997) rather than the square Euclidean distance used for the additive update algorithms.[1] A general theory based on Mercer's theorem has been developed that characterizes kernels usable for additive algorithms (see e.g. Cristianini and Shawe-Taylor 2000). The kernels usable for multiplicative algorithms are much more restrictive. In particular, the features must be products. We will show that multiplicative updates mesh nicely with path kernels. These kernels are defined by a directed graph. There is one feature per source to sink path and the weight/feature associated with a path is the product of the weights of the edges along the path. The number of paths is typically exponential in the number of edges. The algorithms can easily be described by "direct" algorithms that maintain exponentially many path-weights. The algorithms are then simulated by "indirect" algorithms that maintain only one weight per edge. More precisely, the weight vector W on the paths is represented as $\Phi(w)$, where w are the weights on the edges and $\Phi(\cdot)$ is the feature map associated with the path kernel. Thus the indirect algorithm updates w, instead of directly updating the feature weights W. The prediction and the update of the edge weights become efficient kernel computations.

There is a lot of precedent for simulating inefficient direct algorithms (Helmbold and Schapire, 1997, Maass and Warmuth, 1998, Helmbold, Panizza, and Warmuth, 2002, Takimoto and Warmuth, 2002) by efficient indirect algorithms. In this paper we hope to give a unifying view and make the connection to path kernels. The key requirement will be that the loss of a path decomposes into a sum of the loss of the edges of the path. We will re-express many of the previously developed indirect algorithms using our methods.

As discussed before, for additive algorithms the vector of feature weights has the form $W = \sum_{q=1}^{t} \alpha_q \Phi(x_q)$, where the α_q are the linear coefficients of the expanded instances. In the case of Support Vector Machines, optimizing the α_q for a batch of examples is a non-negative quadratic optimization problem. Various algorithms can be used for finding the optimum coefficients. For example, Cristianini et al. (1999) does this using multiplicative updates (motivated in terms of an entropic distance function on the α_q instead of the feature weights). An alternate "multiplicative" update algorithm for optimizing the α_q (not motivated by an entropic regularization) is given by Shu et al. (2003). In contrast, in this paper we discuss multiplicative algorithms of the feature weights, i.e. the logarithm of the feature weights is a constant plus a linear combination of the expanded instances (see Kivinen et al. 1997 for more discussion).

Paper outline: In the next section we define path kernels and discuss how to compute them for general directed graphs. One method is to solve a system of equations. In Appendix A we show that the system has a unique solution if some minimal assumptions hold. We then give a simple hierarchically constructed digraph in Section 3, whose associated kernel initiated this research. In Section 4, we generalize this example and define path sets for a family of hierarchically constructed digraphs corresponding to regular expressions. We show how the hierarchical construction facilitates the ker-

1. See Kivinen et al. (1997) for a geometric characterization of the additive algorithms based on a rotation invariance.

nel computation (Sections 4.1 and 4.2). In Section 5, we discuss how path kernels can be used to represent probabilistic weights and how the predictions of the algorithms can be expressed as kernel computations. In Section 5.1, we give some key properties of the probabilistic edge weights that we would like to maintain. In particular, the total outflow from each vertex (other than the sink) should be one and the weight W_P of each path P must be the product of its edge weights, i.e. $W_P = \prod_{e \in P} w_e$. We show in Appendix B that the edge weights fulfilling these properties are unique. The updates we consider in this paper always have the following form: Each edge is multiplied by a factor and then the total path weight is renormalized. We show in Section 5.2 that this form of the updates appears in multiplicative updates when the loss of a path decomposes into a sum over the edges of the path. We then introduce the Weight Pushing algorithm of Mehryar Mohri (1998) in Section 6 which re-establishes the properties of the edge weights after each edge received an update factor. Efficient implementation of this algorithm can make use of the hierarchical construction of the graphs (see Appendix C).

We then apply our methods to a dynamic routing problem (Section 7) and to an on-line shortest path problem (Section 8). We prove bounds for our algorithm that decay with the length of the longest path in the graph. However, we also show that for the hierarchically constructed digraph given in Section 3, the longest path does not enter into the bound. In Section 9, we discuss how the set of paths associated with this graph can be used to motivate the Binary Exponentiated Gradient (BEG) algorithm for learning disjunctions (Helmbold et al., 2002) and show how this efficient algorithm for learning disjunctions becomes a special case of our methods. Finally, we rewrite the algorithms for predicting as well as the best pruning of a series parallel digraph using our methods (Section 10) and conclude with some open problems (Section 11).

Relationship to previous work: Our main contribution is the use of kernels for implementing multiplicative updates of the feature weights. The path kernels we use are similar to previous kernels introduced by Haussler (1999) and Watkins (1999).[2] Here we focus on the efficient computation of the path kernels based on the corresponding regular expressions or syntax trees. Our key new idea is to use the path kernels to implicitly represent exponentially many probabilistic weights on the features/paths by only maintaining weights on the edges. Multiplicative updates are ideally suited for updating the path weights since they contribute factors to the edge weights. We characterize exactly the requirements for such algorithms. Also a key insight is the use of Weight Pushing algorithm for maintaining probabilistic weights on the edges. We show how to efficiently implement this algorithm on syntax trees. The applications to the dynamic routing and on-line shortest path problem are new. The sections on learning disjunctions and on-line algorithms for predicting as well as the best pruning of the series parallel digraph are mainly rewritings of previously existing algorithms in terms of the new common framework of path kernels.

2. Path Kernels

Assume we have a directed graph G with a source and a sink vertices. The source may have incoming edges but the sink does not have outgoing edges. Inputs to the edges are specified by a vector $x \in \mathcal{R}^n$, where n is the number of inputs and is fixed. If edge e receives input x_i, then we denote this as $x_e = x_i$. So this notation hides a fixed assignment from the edges $E(G)$ of the graph G to the input indices $\{1, \ldots, n\}$. The assignment is fixed for each graph. So if x' is a second input vector and $x_e = x_i$, then $x'_e = x'_i$ as well. Edges may also receive constants as inputs, denoted as $x_e = 1$. In that

2. Our kernels are also special cases of the "rational kernels" recently introduced by Cortes et al. (2002).

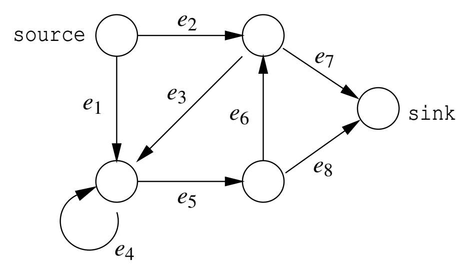


Figure 1: An example digraph. Edge e_i receives input x_i. The input vector $x = (x_1, \ldots, x_8)$ is expanded to the feature vector $\Phi(x) = (x_2x_7, x_1x_5x_8, x_2x_3x_5x_8, x_1x_4x_5x_8, x_1x_5x_6x_7, x_2x_3x_4x_5x_8, x_2x_3x_5x_6x_7, x_1x_4x_4x_5x_8, x_1x_5x_6x_3x_5x_8, \ldots)$. The order of the features is arbitrary but fixed.

case, $x'_e = 1$ as well. The number of inputs n may be less than the number of edges, i.e. edges may share inputs. But in the simplest case (for example in Figure 1), $n = |E(G)|$ and edge e_i receives input x_i.

The input vector x is expanded to a feature vector that has one feature for each source-to-sink path. The feature X_P associated with path P is the product of the inputs of its edges, i.e. $X_P = \prod_{e \in P} x_e$ (see Figure 1). (Throughout the paper, we use upper case letters for the product features and lower case letters for inputs.) We let $\Phi(x)$ be the vector of all path features. Given a second input vector x' on the edges, we define the *path kernel* of a directed graph as follows:

$$K(x, x') = \Phi(x) \cdot \Phi(x') = \sum_P \prod_{e \in P} x_e x'_e.$$

Similar related kernels that are built from regular expressions or pair-HMMs were introduced by Haussler (1999), Watkins (1999) for the purpose of characterizing the similarity between strings.

We would like to have efficient algorithms for computing kernels. For this reason we first generalize the definition to sums over all paths starting at any fixed vertex rather than the source. For any vertex u, let $\mathcal{P}(u)$ denote the set of paths from the vertex u to the sink. Assume there are two input vectors x and x' to the edges. Then for any vertex u, let

$$K_u(x, x') = \sum_{P \in \mathcal{P}(u)} \prod_{e \in P} x_e \prod_{e \in P} x'_e = \sum_{P \in \mathcal{P}(u)} \prod_{e \in P} x_e x'_e.$$

Clearly $K_{\mathtt{source}}(x, x')$ gives the dot product $\Phi(x) \cdot \Phi(x')$ that is associated with the whole graph G. For any vertex u other than the sink we have:

$$K_u(x, x') = \sum_{u' : (u, u') \in E(G)} x_{(u,u')} x'_{(u,u')} K_{u'}(x, x'). \tag{2.1}$$

The computation of K depends on the complexity of the graph G. If G is acyclic, then the functions $K_u(x, x')$ can be recursively calculated in a bottom up order using (2.1) and

$$K_{\mathtt{sink}}(x, x') = 1. \tag{2.2}$$

Clearly this takes time linear in the number of edges.

When G is an arbitrary digraph (with cycles and infinite path sets), then (2.1) and (2.2) form a system of linear equations which can be solved by a standard method, e.g. Gaussian elimination. In this paper we only need the case when all inputs to the edges are non-negative and $K(x,x')$ is finite. For that case we show in Appendix A that the solution is unique. Alternatively K_u can be computed via dynamic programming using essentially the Floyd-Warshal all-pairs shortest path algorithm (Mohri, 1998). The cost of these algorithms is essentially cubic in the number of vertices of G. Speed-ups are possible for sparse graphs. A more efficient method is given later in Section 4.1 for the case when the graph, viewed as a DFA, has a concise regular expression.

3. A Hierarchically Constructed Digraph That Motivates the Subset Kernel

In this section we discuss the kernel associated with the digraph given in Figure 2 and use it as a motivating example for what is to follow in great detail in the next section. This kernel was the initial focal point of our research.

First observe that the paths of this graph can be described using the following simple regular expression

$$(e_1+e_{n+1})(e_2+e_{n+2})\dots(e_n+e_{2n}). \tag{3.1}$$

Assume the bottom edges e_i receive input value x_i and all top edges e_{n+i} receive input one. The feature X_P is the product of the inputs along the path P. So $X_P = \prod_{i\in A} x_i$, where A is the subset of indices in $\{1,\dots,n\}$ corresponding to the bottom edges in P. If you now consider two input values x_i and x'_i to the bottom edges, then $\Phi(x)$ and $\Phi(x')$ have one feature for each of the 2^n subsets/monomials over n variables, and the dot product defines a kernel

$$K(x,x') = \Phi(x)\cdot\Phi(x') = \sum_{A\subseteq\{1,\dots,n\}} \prod_{i\in A} x_i x'_i = \prod_{i=1}^{n}(1+x_i x'_i). \tag{3.2}$$

We call this the *subset kernel*.[3] This kernel was introduced by Kivinen and Warmuth (1997) and is also sometimes called the *monomial kernel* (Khardon et al., 2001). Note that it computes a sum over 2^n subsets in $O(n)$ time and this computation is closely related to the above regular expression: Replace e_i by $x_i x'_i$, e_{n+i} by $x_{n+i}x'_{n+i} = 1$, the regular $+$ by the arithmetic $+$, and the regular concatenation by the arithmetic multiplication. Also note that the fundamental unit in the graph of Figure 2 is a pair of vertices connected by a top and bottom edge. The whole graph may be seen as a "sequential composition" of n of such units.

We will use this kernel again in Section 9 to motivate an efficient algorithm for learning disjunctions. In the next section we discuss general schemes for building graphs and kernels from regular expressions.

4. Regular Expressions for Digraphs

Generalizing the above example, we consider a digraph as an automaton by identifying the source and the sink with the start and the accept state, respectively. The set of all source-sink paths is a

3. If the top edges receive inputs x_{n+i} and x'_{n+i}, respectively, then $K(x,x') = \sum_{A\subseteq\{1,\dots,n\}} \prod_{i\in A} x_i x'_i \prod_{i\notin A} x_{n+i}x'_{n+i} = \prod_{i=1}^{n}(x_i x'_i + x_{n+i}x'_{n+i})$.

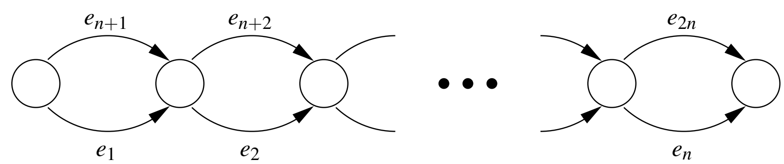


Figure 2: The digraph that defines the subset kernel: The top edges receive input one and function as ε edges.

regular language and can be expressed as a regular expression. For example the set of paths of the digraph given in Figure 1 is expressed as

$$e_2e_7 + (e_1 + e_2e_3)e_4^*e_5(e_6e_3e_4^*e_5)^*(e_8 + e_6e_7).$$

We assume in this paper that each edge symbol identifies a single edge and thus there is never any confusion about how words map to paths. Note that in the above regular expression some symbols appear more than once. So when we convert multiple occurrences of the same symbol to edges we use differently named edges for each occurrence but assign all of them the same input.

Since path features are products, the edges e that are always assigned the constant one (i.e. $x_e = 1$) function as the ε symbol in the regular expression. On the other hand, as we will see later, when we consider probabilistic weights on edges, ε edges are not always assigned weight one.

The convolution kernel based on regular expressions was introduced by Haussler (1999). We will show that computing regular expression kernels is linear in the size of the regular expression that represents the given digraph G. Recall that the methods for computing kernels introduced in Section 2 take $O(n^3)$ time, where n is the number of vertices of G. So there is a speed-up when there is a regular expression of size $O(n^3)$. Even though the size of the smallest regular expression can be exponential in n, we will see that there are small regular expressions for many practical kernels.

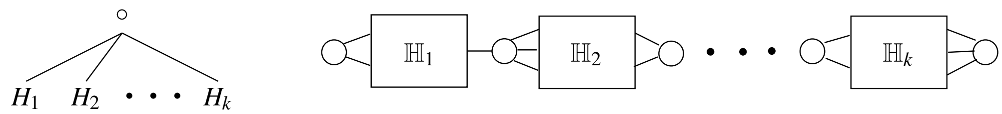
For the sake of simplicity we assume throughout the paper that the source has no incoming and the sink no outgoing edges (One can always add new source and sink vertices and connect them to the old ones via ε edges).

4.1 Series Parallel Digraphs

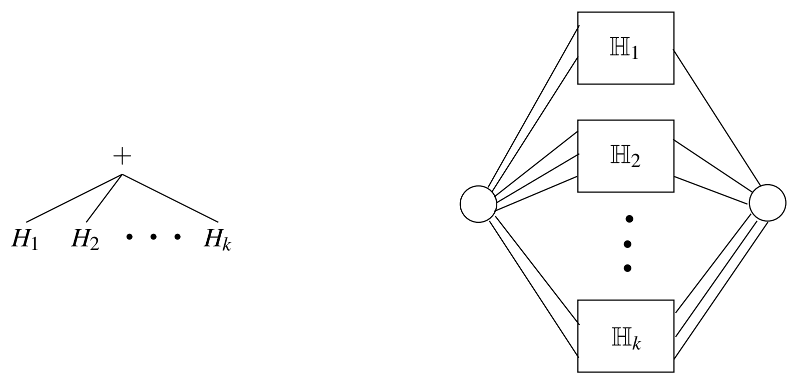
First let us consider the simple case where regular expressions do not have the $*$-operation. In this case the prescribed graphs (corresponding to regular expressions with the operations union $(+)$ and concatenation $(\circ)$) are *series parallel digraphs* (SP digraphs, for short) (Valdes et al., 1982).

For a regular expression H, let $\mathbb{H}$ denote the SP digraph that H represents. Furthermore, we sometimes write $\mathbb{H}(\mathtt{s},\mathtt{t})$ to explicitly specify the source $\mathtt{s}$ and the sink $\mathtt{t}$. Now we clarify how a regular expression with operations $+$ and $\circ$ recursively defines a SP digraph (see Figure 3 for a schematic description). A symbol e defines the SP digraph $\mathbb{H}(\mathtt{s},\mathtt{t})$ consisting of a single edge with label e, initial vertex $\mathtt{s}$ and terminal vertex $\mathtt{t}$. Let $H_1,\ldots,H_k$ be regular expressions and $\mathbb{H}_1(\mathtt{s}_1,\mathtt{t}_1),\ldots,\mathbb{H}_k(\mathtt{s}_k,\mathtt{t}_k)$ be the corresponding SP digraphs, respectively. The concatenation of regular expressions $H_1,\ldots,H_k$, denoted by $H = H_1 \circ\cdots\circ H_k$, corresponds to a *series composition* of the SP digraphs $\mathbb{H}_1(\mathtt{s}_1,\mathtt{t}_1),\ldots,\mathbb{H}_k(\mathtt{s}_k,\mathtt{t}_k)$. We denote this series composition as $\mathbb{H}(\mathtt{s},\mathtt{t})$. The source $\mathtt{s}$ and sink $\mathtt{t}$ of this graph are identified with $\mathtt{s}_1$ and $\mathtt{t}_k$, respectively, and for any $1 \le i \le k-1$,

(a) A symbol corresponds to a single edge.



(b) Concatenation corresponds to the series composition.



(c) Union corresponds to the parallel composition.

Figure 3: Regular expressions (as syntax trees on the left) and their corresponding SP digraphs (right).

the sink $\mathtt{t}_i$ and the source $\mathtt{s}_{i+1}$ are merged into one internal vertex. Finally, the union of the regular expression $H_1, \ldots, H_k$, denoted by $H = H_1 + \cdots + H_k$, corresponds to the *parallel composition* $\mathbb{H}(\mathtt{s},\mathtt{t})$ of the corresponding SP digraphs $\mathbb{H}_1(\mathtt{s}_1,\mathtt{t}_1), \ldots, \mathbb{H}_k(\mathtt{s}_k,\mathtt{t}_k)$, where all sources are merged into the source $\mathtt{s}$ and all sinks are merged into the sink $\mathtt{t}$. In Figure 4 we give an example SP digraph with its regular expression (represented by a syntax tree).

The syntax tree is used to compute the dot product $\Phi(x) \cdot \Phi(x')$. We represent the feature vector $\Phi(x)$ by the syntax tree where the leaves e are replaced by their assignments x_e. The feature vector $\Phi(x')$ is the same except now the assignments x'_e are used. For computing the dot product, we replace the leaves labeled edge e by the input product $x_e x'_e$, the union $+$ by the arithmetic plus $+$ and the concatenation $\circ$ by the arithmetic multiplication $\times$. Now the value of the dot product is computed by a postorder traversal of the tree. This takes time linear in the size of the tree (number of edges). See Figure 5 for an example. For this figure, e_i receives input x_i and x'_i, respectively.

In general, let H_0 denote the given regular expression. Note that each internal node H of the syntax tree for H_0 corresponds to a regular expression H (we use the same symbol for both meanings) and this in turn represents a component $\mathbb{H}$ of the entire SP digraph $\mathbb{H}_0$. For an internal node H that represents a component $\mathbb{H}(\mathtt{s},\mathtt{t})$, let $\mathcal{P}^H$ denote the set of all paths from $\mathtt{s}$ to $\mathtt{t}$ in $\mathbb{H}$. In

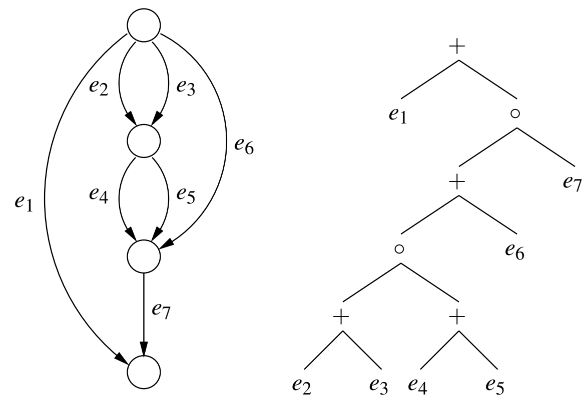


Figure 4: An example of a SP digraph and its syntax tree.

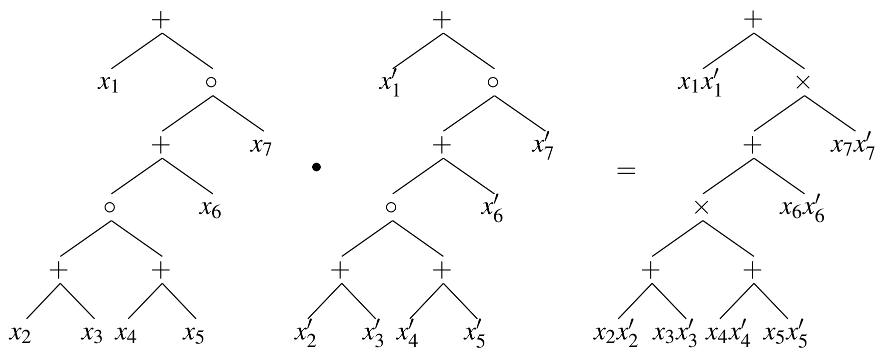


Figure 5: The computation of dot product $\Phi(x)\cdot\Phi(x')$: The trees on the left represent the feature vectors $\Phi(x)$ and $\Phi(x')$, respectively; the tree on the right represents the computation of the dot product, where the regular union $+$ is replaced by the arithmetic $+$ and the regular concatenation $\circ$ becomes the arithmetic multiplication $\times$.

other words, $\mathcal{P}^H$ is the language generated by the regular expressions H. Furthermore, we define the kernel associated with H as

$$K^H(x,x') = \Phi^H(x)\cdot\Phi^H(x') = \sum_{P\in\mathcal{P}^H}\prod_{e\in P} x_e x'_e,$$

where $\Phi^H(x)$ is the feature vector defined by a regular expression H with leaf assignments x. Now it is straightforward to see that this kernel is recursively calculated as follows:

$$K^H(x,x') = \begin{cases} x_e x'_e & \text{if } H = e \text{ for some edge symbol } e, \\ \prod_{i=1}^k K^{H_i}(x,x') & \text{if } H = H_1\circ\cdots\circ H_k, \\ \sum_{i=1}^k K^{H_i}(x,x') & \text{if } H = H_1+\cdots+H_k. \end{cases}$$

Although SP digraphs seem a very restricted class of acyclic digraphs, they can define some important kernels such as the polynomial kernel: $\Phi(x)\cdot\Phi(x') = (1+x\cdot x')^k$, for some degree k.

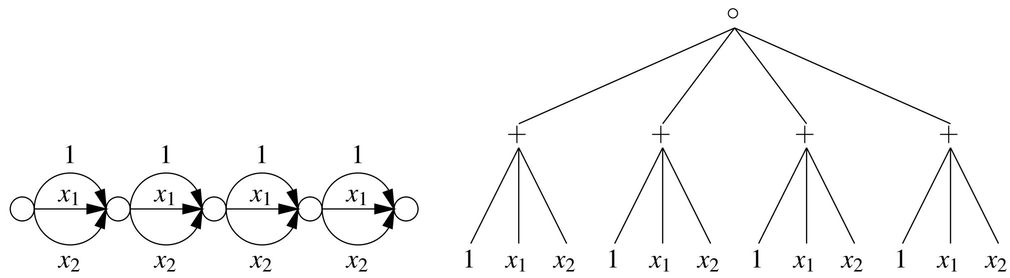


Figure 6: Some inputs to distinct edges might be the same. The feature vector is $\Phi(x) = (1, 4x_1, 4x_2, 6x_1^2, 12x_1x_2, 6x_2^2, 4x_1^3, 12x_1^2x_2, 12x_1x_2^2, 4x_2^3, x_1^4, 4x_1^3x_2, 6x_1^2x_2^2, 4x_1x_2^3, x_2^4)$. This defines the polynomial kernel $K(x,x') = (1 + x \cdot x')^4$.

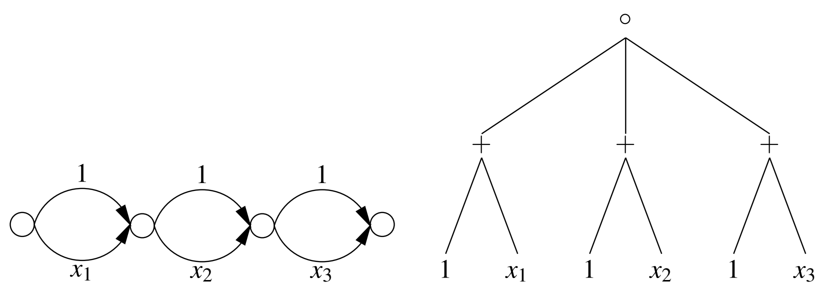


Figure 7: One feature per subset of $\{1,2,3\}$: $\Phi(x) = (1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, x_1x_2x_3)$. This defines the subset kernel $K(x,x') = \prod_{i=1}^{3}(1 + x_ix_i')$.

This is the path kernel defined by the regular expression

$$(e_{1,0} + e_{1,1} + \cdots + e_{1,n})(e_{2,0} + e_{2,1} + \cdots + e_{2,n}) \cdots (e_{k,0} + e_{k,1} + \cdots + e_{k,n}),$$

where all edges $e_{*,0}$ receive input one, and all edges $e_{*,i}$ (with $i \geq 1$) receive inputs x_i and x_i', respectively. The tree in Figure 6 represents the feature vector $\Phi(x)$ for the polynomial kernel with $n = 2$ and $k = 4$.

A related example is given in Figure 2 which is the digraph associated with the subset kernel (3.2). Now the bottom edges e_i receive input values x_i and x_i', respectively, and the top edges e_{n+i} always receive input one (see Figure 7: left). In this case $\Phi(x)$ has one feature for each of the 2^n subsets/monomials over n variables. The syntax diagram representing $\Phi(x)$ is given in Figure 7: right.

Since the inputs to the edges might not be distinct (see example in Figure 6), we can use shorthands for the union and concatenation of identical subgraphs that receive the same assignment. Specifically, let $\Phi^{(n\circ)H}(x)$ and $\Phi^{(n+)H}(x)$ denote the feature vectors defined by the n-fold concatenation and the n-fold union of H, respectively, with the same assignments x to its leaves (see Figure 8 for an example). Clearly,

$$K^{(n\circ)H}(x,x') = \Phi^{(n\circ)H}(x) \cdot \Phi^{(n\circ)H}(x') = \left(K^H(x,x')\right)^n$$

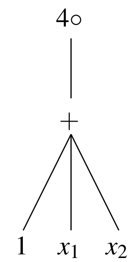


Figure 8: Shorthand for identical subgraphs. This defines the same feature vector as in Figure 6.

and

$$K^{(n+)H}(x,x') = \Phi^{(n+)H}(x) \cdot \Phi^{(n+)H}(x') = nK^H(x,x').$$

In the table below we summarize the correspondence between regular operators and arithmetic operations.

regular		arithmetic	
$\circ$	concatenation	$\times$	multiplication
$+$	union	$+$	sum
$n\circ$	n-fold concatenation	$\wedge n$	power n
$n+$	n-fold union	$\times n$	times n

4.2 Allowing $*$-operations

Now we consider the general case where regular expressions have $*$-operations. First let us clarify how the digraph is defined by $*$-operations. Let $\mathbb{H}_1(\mathtt{s}_1, \mathtt{t}_1)$ be any digraph that is defined by a regular expression H_1. Then we define the digraph for $H = H_1^*$ as in Figure 9: add a new source $\mathtt{s}$, an internal vertex u_0 and a new sink $\mathtt{t}$, and connect with ε-edges from $\mathtt{s}$ to u_0, from $\mathtt{t}_1$ to u_0, from u_0 to $\mathtt{t}$ and from u_0 to $\mathtt{s}_1$. For convenience we call them ε_1, ε_2, ε_3 and ε_H, respectively. The last edge ε_H is called the *repeat edge* of $\mathbb{H}$ and will play an important role. Note that the ε-edges are newly introduced and there are no such symbols in the given regular expression $H = H_1^*$. Actually we can eliminate the ε-edges by merging the five vertices $\mathtt{s}$, u_0, $\mathtt{s}_1$, $\mathtt{t}_1$ and $\mathtt{t}$ into one vertex. We introduce the dummy ε-edges for the following three reasons.

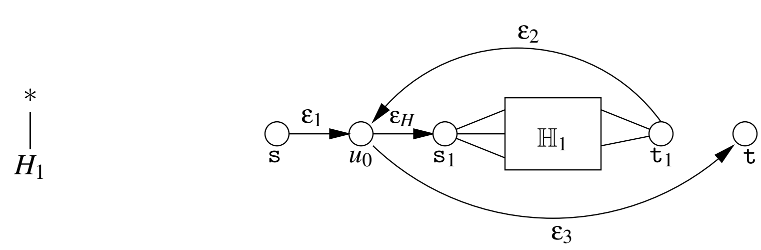


Figure 9: The digraph defined by $H = H_1^*$. $\mathtt{s}_1$ and $\mathtt{t}_1$ are the source and the sink of $\mathbb{H}_1$, respectively. A new internal vertex u_0 is introduced. The edge ε_H is called the repeat edge of $\mathbb{H}$.

1. The properties that s has no incoming edges and t has no outgoing edges are preserved.

2. There remains a one-to-one correspondence between the set of paths in the component $\mathbb{H}$ and the language that the regular expression H produces. In particular, the trajectory of a path P in $\mathbb{H}$ excluding ε transitions forms a symbol sequence that H produces.

3. As we will see later, the kernel computation and weight updates for H_1 can be made independent of the larger component.

The kernel $K^H(x,x') = \sum_{P\in\mathcal{P}^H}\prod_{e\in P} x_e x'_e$ can be calculated as before by traversing the syntax tree (now with operations $+$, $\circ$ and $*$) in postorder. The local operation done when completing the traversal of a $*$-node is as follows: When $H = H_1^*$ for some regular expression H_1, then

$$K^H(x,x') = x_{\varepsilon_1}x'_{\varepsilon_1}\sum_{k=0}^{\infty}\left(x_{\varepsilon_H}x'_{\varepsilon_H}K^{H_1}(x,x')x_{\varepsilon_2}x'_{\varepsilon_2}\right)^k x_{\varepsilon_3}x'_{\varepsilon_3} = \frac{x_{\varepsilon_1}x'_{\varepsilon_1}x_{\varepsilon_3}x'_{\varepsilon_3}}{1 - x_{\varepsilon_H}x'_{\varepsilon_H}x_{\varepsilon_2}x'_{\varepsilon_2}K^{H_1}(x,x')}$$

if $x_{\varepsilon_H}x'_{\varepsilon_H}x_{\varepsilon_2}x'_{\varepsilon_2}K^{H_1}(x,x') < 1$ and $K^H(x,x') = \infty$ otherwise.

5. Using Path Kernels to Represent Weights

In this paper we describe algorithms that use the path weights as linear weights. The *direct* representation of the weights is the weight vector W which has one component W_P per path P. The *indirect* representation of the weights is a weight vector w on the edges for which $W = \Phi(w)$. If the graph has cycles, then the dimension of W is countably infinite, and in the acyclic case the dimension of W is typically exponential in the dimension of w.

The predictions of the algorithms are determined by kernel computations. In the simplest case there is a set of inputs x_e to the edges and path P predicts with $X_P = \prod_{e\in P} x_e$. The algorithm combines the predictions of the paths by predicting with the weighted average of the predictions of the paths or with a squashing function applied on top of this average. A typical squashing function is a threshold function or a sigmoid function. The weighted average becomes the following dot product:

$$\sum_P W_P X_P = \sum_P\left(\prod_{e\in P} w_e\right)\left(\prod_{e\in P} x_e\right) = \Phi(w)\cdot\Phi(x) = K(w,x).$$

In a slightly more involved case, the prediction of path P is the sum of the predictions of its edges, i.e. $\sum_{e\in P} x_e$. Now the weighted average can be rewritten as

$$\sum_P W_P \sum_{e\in P} x_e = \sum_P\sum_{e\in P}\left(\prod_{e'\in P} w_{e'}\right)x_e = \sum_{e\in E(G)} x_e \sum_P\left(\prod_{e'\in P} w_{e'}\right) I(e\in P), \tag{5.1}$$

where $I(\text{true}) = 1$ and $I(\text{false}) = 0$. For edge e, let u^e be edge weights defined as

$$u^e_{e'} = \begin{cases} 1 & \text{if } e' \neq e, \\ 0 & \text{if } e' = e. \end{cases}$$

Then, we can see that

$$I(e\in P) = 1 - \prod_{e'\in P} u^e_{e'}.$$

By plugging this into the r.h.s. of (5.1) we can again rewrite the weighted average using kernel computations:

$$\sum_P W_P \sum_{e \in P} x_e = \sum_{e \in E(G)} x_e \sum_P \left(\prod_{e' \in P} w_{e'} - \prod_{e' \in P} w_{e'} u^e_{e'} \right) = \sum_{e \in E(G)} x_e \left(K(w, 1) - K(w, u^e) \right), \tag{5.2}$$

where 1 denotes the vector whose components are all 1. As before, the prediction of the algorithm might be a squashed version of this average.

5.1 Probabilistic Weights

We will use the path kernel to represent a probabilistic weight vector on the paths, i.e. $W = \Phi(w)$ is a probability vector. Thus we want three properties for the weights on the set of paths (by default, all weights in this paper are non-negative):

P1 The weights should be in *product form*. That is,

$$W_P = \prod_{e \in P} w_e,$$

where w_e are edge weights.

P2 The outflow from each vertex u should be one. That is, for any vertex u of G,

$$\sum_{u' : (u,u') \in E(G)} w_{(u,u')} = 1,$$

where $E(G)$ denotes the set of edges of G.

P3 The total path weight is one. That is,

$$\sum_P W_P = 1.$$

Note that the sum of Property P3 is over all paths of G from the source to the sink. The three properties make it trivial to generate random paths: Start at the source and iteratively pick an outgoing edge from the current vertex according to the prescribed edge probabilities until the sink is reached. Property P3 guarantees that any random walk eventually goes to the sink. In other words, any vertex u that is reachable from the source via a (partial) path of non-zero probability must also reach the sink via such a path.

Note that our use of kernels (from this section onward) is highly unusual. When we used the notation $\Phi(x) \cdot \Phi(x')$ before in Sections 2–4, then x and x' were always the same type of object (input vectors corresponding to two different examples). In particular, the underlying assignment of inputs to edges was the same for both x and x' (see discussion at the beginning of Section 2). However now, when we write $\Phi(w) \cdot \Phi(x)$, then $\Phi(.)$ is still the same feature map defined by a fixed digraph, but the vectors w and x are not the same type of object any more. Here w is a vector of edge weights and x a vector of edge inputs, and their dot product might be the prediction of an algorithm. Also, the Properties P1–3 hold for w, but not necessarily for x. For the rest of this paper, we will use this type of "asymmetric" dot products between feature vectors. From now on, the vectors w and x always have dimension equal to the number of edges of the underlying graph and there is no common assignment to inputs.

5.2 Decomposable Multiplicative Updates

In this paper we restrict ourselves to updates that multiply each edge weight by a factor and then the total weight of all paths is renormalized. Let W_P be the old weight for path P and $\tilde{W}_P$ the updated weight. Assume that the old weights are in product form, i.e. $W_P = \prod_{e \in P} w_e$ and let b_e be the update factor for edge e. Then updates must have the following form:

$$\tilde{W}_P = \frac{W_P \prod_{e \in P} b_e}{\sum_P W_P \prod_{e \in P} b_e} = \frac{\prod_{e \in P} w_e b_e}{\Phi(w) \cdot \Phi(b)}. \tag{5.3}$$

The normalization needs to be non-zero and finite. That is, we need the following property on the edge factors b_e:

P4 The edge factors b_e are non-negative and

$$0 < \sum_P W_P \prod_{e \in P} b_e < \infty.$$

Typically, the edge factors satisfy $0 < b_e < 1$, and in this case Property P4 is satisfied.

Note that the updated weights are not in product form any more. In the next section we will give an algorithm that translates the above update (5.3) into an update of the edge weights so that the new weights again have the product form and satisfy Properties P1–3 again. In this section we discuss (at a high level) general update families that give rise to updates of the above form.

Consider the following update on the path weights

$$\tilde{W}_P = \frac{W_P \exp(-\eta \ell_P)}{\sum_P W_P \exp(-\eta \ell_P)}, \tag{5.4}$$

where η is a non-negative learning rate and ℓ_P is the loss of path P in the current trial. This is known as the *loss update* of the expert framework for on-line learning, where the paths are the experts. One weight is maintained per expert/path and the weight of each expert decays exponentially with the loss of the expert. The Bayes Update can be seen as a special case, when the loss is the log loss (DeSantis et al., 1988). Another special case is the Weighted Majority algorithm (Littlestone and Warmuth, 1994) (discrete loss). General loss functions were first discussed in the seminal work of Vovk (1990). See Kivinen and Warmuth (1999) for an overview.

The loss update has the required form (5.3), if ℓ_P decomposes into a sum over the losses l_e of the edges of P, i.e.

$$\ell_P = \sum_{e \in P} l_e \text{ and } b_e = \exp(-\eta l_e). \tag{5.5}$$

For maintaining Property P4, it is sufficient to assume that for all paths P, the losses ℓ_P are both upper and lower bounded. The applications of Sections 7, 8 and 10 use the loss update, where the loss of a path decomposes into a sum. In many cases, however, the loss does not decompose into a sum. In Section 9, we discuss this issue in the context of learning disjunctions.

The loss update is a special case of the Exponentiated Gradient (EG) update (Kivinen and Warmuth, 1997), which is of the form

$$\tilde{W}_P = W_P \exp\left(-\eta \frac{\partial \lambda}{\partial W_P}\right) / Z, \tag{5.6}$$

where λ is the current loss of the algorithm (which depends on the weights W) and Z renormalizes the weights to one. If $\lambda = \sum_P W_P \ell_P$, then $\frac{\partial \lambda}{\partial W_P} = \ell_P$, and the EG update becomes the loss update (5.4).

So to obtain EG updates of the required form (5.3), we want the gradient of the loss to decompose into a sum over the edges. The canonical case is the following. Assume that inputs x are given to the edges, the prediction of a path P is $\sum_{e \in P} x_e$, and the prediction of the algorithm is given by

$$\hat{y} = \sigma(\hat{a}), \text{ where } \hat{a} = \sum_P W_P \sum_{e \in P} x_e \tag{5.7}$$

and $\sigma(.)$ is a differentiable squashing function (Recall that we showed in (5.2) how to write the weighted path prediction $\hat{a}$ as a sum of kernel computations.) Assume the loss λ measures the discrepancy between the prediction $\hat{y}$ and a target label y. That is, $\lambda = \ell(y, \hat{y})$, where ℓ is a function $\mathcal{R} \times \mathcal{R} \to \mathcal{R}_{\geq 0}$ that is differentiable in the second argument. (A typical example is the square loss, i.e. $\lambda = \ell(y, \hat{y}) = (y - \hat{y})^2 = (y - \sigma(\hat{a}))^2$.) With this form of the loss, the EG update becomes

$$\tilde{W}_P = W_P \exp\left(-\eta \lambda' \sum_{e \in P} x_e\right) / Z, \tag{5.8}$$

where

$$\lambda' = \left. \frac{\partial \ell(y, \sigma(a))}{\partial a} \right|_{a = \hat{a}}.$$

For example, in the case of the square loss,[4] we have $\lambda' = 2\sigma'(\hat{a})(\hat{y} - y)$. Since the derivative λ' is constant with respect to P, the exponent decomposes into a sum over the edges. Thus in this canonical case, the EG update (5.8) has the required form (5.3), where the edge factors are

$$b_e = \exp\left(-\eta \lambda' x_e\right). \tag{5.9}$$

In Section 9 we will use this form of the EG update for designing efficient disjunction learning algorithms. The discrete loss for disjunctions does not decompose into a sum. However by using a prediction of the form (5.7) and an appropriate loss function for our algorithm, we are in the lucky situation where the gradient of this loss decomposes into a sum and our efficient methods are again applicable.

The term "multiplicative update" is an informal term where each updated weight is proportional to the corresponding previous weight times a factor. A more precise definition is in terms of the regularization function used to derive the updates: *Multiplicative updates* must be derivable with a relative entropy or an unnormalized relative entropy as the regularization (see Kivinen and Warmuth 1997 for a discussion, where these update are called EG and EGU updates, respectively). The Unnormalized Exponentiated Gradient (EGU) update has the same form as the EG update (5.6) except that the weights are not normalized.

Note that for the methodology of this paper, the update must have form (5.3), and if the algorithm uses a prediction, then it must be efficiently computable via for example kernel computations.

4. The update can be used in slightly more general contexts when the $\sigma(.)$ is not differentiable and $\ell(.,.)$ is not differentiable in its second argument. For example in Section 9, we will use a threshold function and the hinge loss.

6. Weight Pushing Algorithm

The additional factors b on the edges (from the multiplicative update (5.3)) mess up the three Properties P1–3. However there is an algorithm that rearranges the weights on the edges so that the relative weights on the path remain unchanged but again have the three properties. This algorithm is called the *Weight Pushing algorithm* developed by Mehryar Mohri (1998) in the context of speech recognition.

The generalized path kernels K_u will be used to re-normalize the path weights with the Weight Pushing algorithm. Assume that the edge weights w_e and the path weights W_P fulfill the three Properties P1–3. Our goal is to find new edge weights $\tilde{w}_e$ so that the three properties are maintained.

A straightforward way to normalize the edge weights would be to divide the weights of the edges leaving a vertex by the total weight of all edges leaving that vertex. However this usually does not give the correct path weights (5.3) unless the product of the normalization factors along different paths is the same. Instead, the Weight Pushing algorithm (Mohri, 1998) assigns the following new weight to the edge $e = (u, u')$:

$$\tilde{w}_e = \frac{w_e b_e K_{u'}(w,b)}{K_u(w,b)}. \tag{6.1}$$

Below we show that the three properties are maintained for the updated weights $\tilde{W}_P$ and $\tilde{w}_e$.

Theorem 1 *Assume that the path weights W and the edge weights w fulfill the three Properties P1, P2 and P3. Let $\tilde{W}_P$ be the updated path weight given by (5.3) and $\tilde{w}_e$ be the new edge weights given by (6.1). Here we assume that Property P4 holds. Then, $\tilde{W}$ and $\tilde{w}$ fulfill the three Properties P1, P2 and P3.*

Proof Property P3 follows from the definition (5.3) of $\tilde{W}_P$. Here Property P4 is needed to assure that the normalization is positive. From (2.1), it is easy to see that the new weights $\tilde{w}_e$ are normalized, i.e. Property P2 holds for any non-sink vertex u,

$$\sum_{u':(u,u')\in E(G)} \tilde{w}_{(u,u')} = 1.$$

Finally, Property P1 is proven as follows: Let $P = \{(u_0,u_1),(u_2,u_3),\ldots,(u_{k-1},u_k)\}$ be any path from the source u_0 to the sink u_k. Then by starting from (5.3) we get

$$\begin{aligned}
\tilde{W}_P &= \frac{W_P \prod_{e\in P} b_e}{\sum_P W_P \prod_{e\in P} b_e} \\
&= \frac{\prod_{e\in P} w_e b_e}{K_{\texttt{source}}(w,b)} \\
&= \frac{\prod_{e\in P} w_e b_e K_{\texttt{sink}}(w,b)}{K_{\texttt{source}}(w,b)} \\
&= \prod_{i=1}^{k} \frac{w_{(u_{i-1},u_i)} b_{(u_{i-1},u_i)} K_{u_i}(w,b)}{K_{u_{i-1}}(w,b)} \\
&= \prod_{e\in P} \tilde{w}_e.
\end{aligned}$$

■

Recall that in Section 4 we showed how to speed-up the kernel computation when the digraph is represented as a regular expression. In Appendix C we will implement the Weight Pushing algorithm on syntax diagrams for regular expression. The first algorithm is linear in the size of the regular expression. We then give sub linear algorithms for computing the dot product $\Phi(w)\cdot\Phi(x)$ when most of the x_e are one, and for implementing the Weight Pushing algorithm when most of the factors b_e are one.

Before we begin with applications of our methods in the next section, we describe the Weight Pushing algorithm for the subset kernel given in Section 3: See regular expression (3.1) and the digraph of Figure 2. This example might give an idea how the Weight Pushing algorithm can be implemented on regular expressions. Assume that the factors $b_{e_{n+i}}$ are all one. By Property P2 we have $w_{e_{n+i}} = 1 - w_{e_i}$. So each pair of edges e_i and e_{n+i} contributes a factor $1 - w_{e_i} + w_{e_i} b_{e_i}$ in the kernel computation (see the footnote of p. 777). More generally, if u_j is the vertex at which the j-th pair is starting, then

$$K_{u_j}(w,b) = \sum_{P\in\mathcal{P}(u_j)} \prod_{e\in P} w_e b_e = \prod_{i=j}^{n} (1 - w_{e_i} + w_{e_i} b_{e_i}).$$

With this we see that the ratio of kernels in (6.1) cancel, except for the factor belonging to the j-th pair, and

$$\tilde{w}_{e_j} = w_{e_j} b_{e_j} / (1 - w_{e_j} + w_{e_j} b_{e_j}) \text{ and } \tilde{w}_{e_{n+j}} = (1 - w_{e_j}) / (1 - w_{e_j} + w_{e_j} b_{e_j}). \tag{6.2}$$

This re-normalization of the weights will be used in the BEG algorithm for learning disjunctions (Section 9).

7. A Dynamic Routing Problem

In the subsequent sections we show various applications of our method. In particular, in this and the next sections we discuss two on-line network routing problems. Assume that we want to send packets from the source to the sink (destination) of a given digraph (network). For each trial t, we assign transition probabilities $w_{t,e}$ to the edges that define a probabilistic routing. Starting from the source we choose a random path P to the sink according to the transition probabilities and try to send the packet along the path. But some edges (links) may be very slow due to network congestion. The goal is to find a protocol that is competitive with the optimal static routing chosen in the hindsight. Note that we make no assumptions on how the traffic changes in time. In other words we seek guarantees that hold for arbitrary network traffic. There are several ways to define the "resistance" of an edge as well as the throughput of a protocol. In this section, the resistance is the success probability of transferring the packet along the edge, and the throughput of a protocol is measured by the total success probability of sending all packets from the source to the sink. In the next section, the resistance and the throughput are defined in terms of the time it takes for a packet to traverse a link rather than the success probability.

There is a large body of research on competitive routing. Typically, edges have limited capacities and the problem is to find an "efficient" routing protocol subject to the capacity constraints. In most cases, a reliable network is assumed in the sense that the network parameters do not adaptively change (see e.g. Leonardi 1998, Awerbuch et al. 2001, Aiello et al. 2003). So some aspects of our model are new and may be more appropriate for networks operating in a strongly adversarial environment. Below we give the problem formally.

In each trial $t = 1,2,\ldots,T$, the following happens:

1. At the beginning of the trial, the algorithm has transition probabilities $w_{t,e} \in [0,1]$ for all edges e, such that Properties P1–3 hold.

2. Conductances $d_{t,e} \in [0,1]$ for all edges are given. Let X_t denote the event that the current packet is successfully sent from the source to the sink. Assuming independence between conductances of individual edges, the probability that the current packet is sent along a particular path P becomes
$$\Pr(X_t \mid X_1,\ldots,X_{t-1},P) = \prod_{e\in P} d_{t,e}. \tag{7.1}$$

3. A random path P is chosen with probability $W_{t,P} = \prod_{e\in P} w_{t,e}$. The success probability at this trial is
$$a_t = \sum_P W_{t,P} \Pr(X_t \mid X_1,\ldots,X_{t-1},P) = \Phi(w_t)\cdot\Phi(d_t).$$

4. The path weights $W_{t,P}$ are updated indirectly to $W_{t+1,P}$ by updating the edge weights $w_{t,e}$ to $w_{t+1,e}$, while maintaining Properties P1–3.

The goal is to make the total success probability $\prod_{t=1}^T a_t$ as large as possible.

In the feature space we can employ the Bayes algorithm. The initial weights are interpreted as a prior for path P, i.e. $\Pr(P) = W_{1,P} = \prod_{e\in P} w_{1,e}$. We assume that the initial edge weights $w_{1,e}$ are chosen so that Properties P1–3 are satisfied and $W_{1,P} > 0$ for all paths P. Then the Bayes algorithm sets the weight $W_{t+1,P}$ as a posterior of P given the input packets $X_1,\ldots,X_t$ observed so far. Specifically, assuming $W_{t,P} = \Pr(P \mid X_1 \ldots X_{t-1})$, then the Bayes algorithm updates the path weights as

$$\begin{aligned} W_{t+1,P} &= \Pr(P \mid X_1,\ldots,X_t) \\ &= \frac{W_{t,P}\Pr(X_t \mid X_1,\ldots,X_{t-1},P)}{\sum_P W_{t,P}\Pr(X_t \mid X_1,\ldots,X_{t-1},P)} \\ &= \frac{W_{t,P}\prod_{e\in P} d_{t,e}}{\sum_P W_{t,P}\prod_{e\in P} d_{t,e}}. \end{aligned} \tag{7.2}$$

Here we assume that at least one static routing P has positive success probability, i.e., $\Pr(X_1,\ldots,X_T \mid P) > 0$. This assumption implies that Property P4 holds and the denominator of (7.2) is always positive. Note that this update has the required form (5.3), where the $d_{t,e}$ function as update factors to the edges. Thus the Weight Pushing algorithm can be used to update the edge weights $w_{t,e}$ to $w_{t+1,e}$. Below we show that the Bayes algorithm achieves the best possible competitive ratio against the optimal static routing.

Theorem 2 *The Bayes algorithm guarantees the following performance.*

$$\prod_{t=1}^T a_t = \sum_P W_{1,P}\Pr(X_1,\ldots,X_T \mid P) \geq \max_P W_{1,P}\Pr(X_1,\ldots,X_T \mid P).$$

On the other hand, for any protocol there exist conductances such that

$$\prod_{t=1}^{T} a_t \le \max_P W_{1,P} \Pr(X_1,\ldots,X_T \mid P).$$

Proof First we analyze the throughput of the Bayes algorithm. Repeatedly applying Bayes rule, we have

$$\prod_{t=1}^{T} a_t = \prod_{t=1}^{T} \frac{\Pr(X_1,\ldots,X_t)}{\Pr(X_1,\ldots,X_{t-1})} = \sum_P W_{1,P} \Pr(X_1,\ldots,X_T \mid P),$$

which implies the first part of the theorem.

Next we consider a strategy for the adversary. Fix a simple path P^* arbitrarily, and for any trial t, let $d_{t,e} = 1$ if $e \in P^*$ and $d_{t,e} = 0$ otherwise. Then, since $a_t = W_{t,P^*}$ for any t, we have

$$\prod_{t=1}^{T} a_t = \prod_{t=1}^{T} W_{t,P^*} \le W_{1,P^*} = \max_P W_{1,P} \Pr(X_1,\ldots,X_T \mid P).$$

■

Note that the bound $\sum_P W_{1,P} \Pr(X_1,\ldots,X_T \mid P)$ for the Bayes algorithm is usually much larger than that for the static routing with the initial prior, which is expressed as $\prod_{t=1}^{T} \sum_P W_{1,P} \Pr(X_t \mid X_1,\ldots,X_{t-1},P)$. For example, consider the conductances used to show the lower bound of the above theorem. In this case, the Bayes algorithm has constant bound W_{1,P^*}, whereas the static routing has exponentially smaller bound W_{1,P^*}^T.

Note that in this simple example, the algorithm did not produce a prediction in each trial. However, note that the success probability can be expressed as a kernel computation. Also, if we define the loss of path P at trial t as $\ell_{t,P} = -\ln \Pr(X_t \mid X_1 \ldots X_{t-1}, P)$, then by (7.1) this loss decomposes into a sum over the edges, i.e. $\ell_{t,P} = \sum_{e \in P} -\ln(d_{t,e})$. Now, for learning rate $\eta = 1$, Bayes rule (7.2) becomes an example of a decomposable loss update (5.4), (5.5), where the independence assumption on the acceptance probabilities of the edges (7.1) caused the negative log likelihood to decompose into a sum.

8. On-line Shortest Path Problem

In this section we let $d_{t,e}$ denote the time it takes the t-th packet to travel along the edge e. The throughput of a protocol is measured by the total amount of time it takes to send all packets from the source to the sink. Equivalently, we can interpret $d_{t,e}$ as the distance of edge e at trial t. Our overall goal is to make the total length of travel from the source to the sink not much longer than the shortest path of the network based on the cumulative distances of all packets for each edge. We call this problem the on-line shortest path problem. We prove a bound for an algorithm and then return to the digraph that defines the subset kernel. For this type of graph there is an improved bound. For our bounds to be applicable, we require in this section that the digraph G defining the network is acyclic. This assures that all paths have bounded length.

In each trial $t = 1, 2, \ldots, T$, the following happens:

1. At the beginning of the trial, the algorithm has transition probabilities $w_{t,e} \in [0,1]$ for all edges e, such that Properties P1–3 hold.

2. Distances $d_{t,e} \in [0,1]$ for all edges are given.

3. The algorithm incurs a loss λ_t which is defined as the expected length of path of G. That is,

$$\lambda_t = \sum_P W_{t,P} \ell_{t,P},$$

where

$$\ell_{t,P} = \sum_{e \in P} d_{t,e} \tag{8.1}$$

is the length of the path P, and this is interpreted as the loss of P.

4. The path weights $W_{t,P}$ are updated indirectly to $W_{t+1,P}$ by updating the edge weights $w_{t,e}$ to $w_{t+1,e}$, respectively, while maintaining Properties P1–3.

Note that the length $\ell_{t,P}$ of path P at each trial is upper bounded by the number of edges in P. Letting D denote the depth (maximum number of edges of P) of G, we have $\ell_{t,P} \in [0,D]$. Note that the path P minimizing the total length

$$\sum_{t=1}^{T} \ell_{t,P} = \sum_{t=1}^{T} \sum_{e \in P} d_{t,e} = \sum_{e \in P} \sum_{t=1}^{T} d_{t,e}$$

can be interpreted as the shortest path based on the cumulative distances of edges: $\sum_{t=1}^{T} d_{t,e}$. The goal of the algorithm is to make its total loss $\sum_{t=1}^{T} \lambda_t$ not much larger than the length of the shortest path.

Considering each path P as an expert, we can view the problem above as a dynamic resource allocation problem introduced by Freund and Schapire (1997). So we can apply their *Hedge algorithm* which is a reformulation of the *Weighted Majority algorithm* (see WMC and WMR of Littlestone and Warmuth, 1994). Let $W_{1,P} = \prod_{e \in P} w_{1,e}$ be initial weights for paths/experts. Note that since the graph is acyclic with a unique sink vertex, the initial weights $W_{1,P}$ sum to 1 and Property P3 is satisfied. At each trial t, when given losses $\ell_{t,P} \in [0,D]$ for paths/experts, the algorithm incurs loss

$$\lambda_t = \sum_P W_{t,P} \ell_{t,P}$$

and updates weights according to

$$W_{t+1,P} = \frac{W_{t,P} \beta^{\ell_{t,P}/D}}{\sum_{P'} W_{t,P'} \beta^{\ell_{t,P'}/D}}, \tag{8.2}$$

where $0 \le \beta < 1$ is a parameter. Note that this is the loss update (5.4), where we use the β parameter instead of the learning rate parameter η (and $\eta = (-\ln \beta)/D$). Since the loss (8.1) decomposes, the update has the required form (5.3), and the Weight Pushing algorithm can be used to update the edge weights using the update factors $b_{t,e} = \beta^{d_{t,e}/D}$. Property P3 for the weights $W_{t,P}$ guarantees the existence of a path for which $W_{t,P} > 0$. Since the update factors are all positive, the normalization in the above update is positive and Property P4 is satisfied. As in the example of the last section, the Hedge algorithm does not predict. But by (5.2), the loss λ_t can be computed using kernel computations.

For the rest of this section we discuss bounds that hold for the Hedge algorithm. It is shown by Littlestone and Warmuth (1994) and Freund and Schapire (1997) that for any sequence of loss vectors for the experts, the Hedge algorithm guarantees[5] that

$$\sum_{t=1}^{T} \lambda_t \le \min_P \left(\frac{\ln(1/\beta)}{1-\beta} \sum_{t=1}^{T} \ell_{t,P} + \frac{D}{1-\beta} \ln \frac{1}{W_{1,P}} \right). \tag{8.3}$$

Below we give a proof of (8.3) with a more sophisticated form of the bound.

Fix an arbitrary probability vector U on the paths. Let $d(U,W_t) = \sum_P U_P \ln(U_P/W_{t,P})$ denote the relative entropy between U and W_t. Looking at the progress $d(U,W_t) - d(U,W_{t+1})$ for one trial, we have

$$\begin{aligned} d(U,W_t) - d(U,W_{t+1}) &= \sum_P U_P \ln(W_{t+1,P}/W_{t,P}) \\ &= \sum_P U_P \ln \frac{\beta^{\ell_{t,P}/D}}{\sum_{P'} W_{t,P'} \beta^{\ell_{t,P'}/D}} \\ &= \frac{\ln \beta}{D} \sum_P U_P \ell_{t,P} - \ln \sum_P W_{t,P} \beta^{\ell_{t,P}/D}. \end{aligned}$$

Since $\ell_{t,P}/D \in [0,1]$ and $0 \le \beta < 1$, we have $\beta^{\ell_{t,P}/D} \le 1 - (1-\beta)\ell_{t,P}/D$. Plugging this into the second term, we have

$$\begin{aligned} \ln \sum_P W_{t,P} \beta^{\ell_{t,P}/D} &\le \ln \left(1 - (1-\beta) \sum_P W_{t,P} \ell_{t,P}/D \right) \\ &= \ln(1 - (1-\beta)\lambda_t/D) \\ &\le -(1-\beta)\lambda_t/D. \end{aligned}$$

Thus

$$d(U,W_t) - d(U,W_{t+1}) \ge \frac{\ln \beta}{D} \sum_P U_P \ell_{t,P} + \frac{1-\beta}{D} \lambda_t.$$

Summing this over all t's we get

$$d(U,W_1) \ge d(U,W_1) - d(U,W_{T+1}) \ge \frac{\ln \beta}{D} \sum_P U_P \sum_{t=1}^{T} \ell_{t,P} + \frac{1-\beta}{D} \sum_{t=1}^{T} \lambda_t,$$

or equivalently,

$$\sum_{t=1}^{T} \lambda_t \le \frac{\ln(1/\beta)}{1-\beta} \sum_P U_P \sum_{t=1}^{T} \ell_{t,P} + \frac{D}{1-\beta} d(U,W_1). \tag{8.4}$$

If U is the unit vector that attains the minimum of the right hand side, we have (8.3).

5. In the references the losses of experts are upper bounded by one at each trial, while in our case they are upper bounded by D. So the second term of the loss bound in (8.3) has the factor D.

8.1 Improved Bounds for Hierarchically Constructed Digraphs

Unfortunately, the loss bound (8.4) depends on the depth D of G. Thus for graphs of large depth or for cyclic graphs, the bound is vacuous. In some cases, however, we can prove bounds where the depth D is replaced with a much smaller depth than the depth of the entire graph G.

Theorem 3 *Assume that a graph G is a series composition of acyclic digraphs $H_1,\dots,H_n$. Note that each component graph H_i is not necessarily a SP digraph. Let D be the maximum depth of a component. Then, the Hedge algorithm using the update rule (8.2) guarantees that*

$$\sum_{t=1}^{T}\lambda_t \le \frac{\ln(1/\beta)}{1-\beta}\sum_P U_P \sum_{t=1}^{T}\ell_{t,P} + \frac{D}{1-\beta}d(U,W_1).$$

Proof We use the convention that $\Phi^{H_i}(x)$ denotes the feature vector based on the paths of H_i. As in the case of SP digraphs, we have

$$\Phi(w_t)\cdot\Phi(b_t) = \prod_{i=1}^{n}\Phi^{H_i}(w_t)\cdot\Phi^{H_i}(b_t),$$

where $b_{t,e} = \beta^{d_{t,e}/D}$. Therefore

$$\begin{aligned}
\ln\sum_P W_{t,P}\beta^{\ell_{t,P}/D} &= \ln\Phi(w_t)\cdot\Phi(b_t)\\
&= \sum_{i=1}^{n}\ln\Phi^{H_i}(w_t)\cdot\Phi^{H_i}(b_t)\\
&= \sum_{i=1}^{n}\ln\sum_{P\in\mathcal{P}^{H_i}} W_{t,P}\beta^{\ell_{t,P}/D}\\
&\le \sum_{i=1}^{n} -(1-\beta)\lambda_{t,i}/D\\
&= -(1-\beta)\lambda_t/D,
\end{aligned}$$

where $\lambda_{t,i} = \sum_{P\in\mathcal{P}^{H_i}} W_{t,P}\ell_{t,P}$ is the expected length of the partial paths through H_i and $\lambda_t = \sum_{i=1}^{n}\lambda_{t,i}$. Using the same technique as in the previous proof, we get the theorem. ■

For example, let us return to the digraph that defines the subset kernel (Figure 2). The graph is a series composition of n two-vertex graphs, where there are two parallel edges connecting the pairs of vertices. Note that although the depth of the whole graph is n, each component graph has depth constant one. So, the Hedge update (8.2) with $D = 1$ leads to a bound that does not depend on the depth of the entire graph.

9. Learning Disjunctions and Conjunctions

In this section we give a high-level discussion of on-line algorithms for learning disjunctions and conjunctions. All algorithms we present are known. We start with inefficient algorithms that have excellent mistake bounds. However, we show how upper bounding the discrete loss by a loss that decomposes into a sum lets us apply the path kernel methodology introduced in this paper. By choosing an appropriate digraph for each case, the weight pushing algorithm realizes all known *efficient* algorithms for learning disjunctions. The price for the gained efficiency is a slight degradation of the mistake bounds.

At each trial a binary instance vector $x_t \in \{0,1\}^n$ is given to the algorithm. After the algorithm produces a binary prediction $\hat{y}_t$, it receive a binary label y_t and incurs discrete loss $|\hat{y}_t - y|$ (also binary). The goal is to design algorithms whose loss (or number of prediction mistakes) is not much worse than the loss of the best conjunction or disjunction of a given size k, where k is typically much smaller than the number of variables n. The Winnow algorithm was the first efficient algorithm for learning disjunction (Littlestone, 1988). Here we motivate two alternate algorithms and also briefly discuss Winnow. All these algorithms spend $O(n)$ time to predict and update their weights and their loss bounds grow linearly in k and logarithmic in n.

We begin by discussing an algorithm for learning conjunctions. Recall the subset kernel of Kivinen and Warmuth (1997) introduced in Section 3 (the path kernel associated with Figure 2). Assume that the bottom edge e_i receive input $x_{t,i}$ and the top edges e_{n+i} all receive input one and function as ε edges. Each path feature $X_{t,P}$ is the product over the inputs along the path. More precisely, $X_{t,P} = X_{t,A} = \prod_{i\in A} x_{t,i}$, where A is the subset of indices in $\{1,\ldots,n\}$ corresponding to the bottom edges of P. So the subset feature $X_{t,A}$ predicts as the conjunction over the variables with indices in A and $|y - X_{t,A}|$ is the discrete loss of conjunction A on the example (x_t, y_t).

One approach is to maintain a weight $W_{t,A}$ per subset A and predict with $\hat{y}_t = \sigma_\theta(a_t)$, where $a_t = \sum_A W_{t,A} X_{t,A}$ and $\sigma_\theta(.)$ is the $\{0,1\}$-valued threshold function with threshold θ. Here we choose $\theta = 1/2$. Assume the weights are updated multiplicatively using the loss update, i.e.

$$W_{t+1,A} = \frac{W_{t,A}\exp(-\eta|y_t - X_{t,A}|))}{Z_t}, \text{ where } Z_t \text{ normalizes the total weight to 1.}$$

Since the loss function is the discrete loss, this is an application of the Weighted Majority algorithm (Littlestone and Warmuth, 1994). Using methods similar to what was used in Section 8, it is easy to prove[6] a mistake bound of $O(k\ln n + m_*)$, where m_* is the number of mistakes of the best conjunction of size k.

Unfortunately, the algorithm is inefficient since it maintains 2^n weights. Moreover, it was essentially shown by Khardon, Roth, and Servedio (2001) that computing the predictions for this type of update[7] is #P-hard. Our efficient methods do not apply because the discrete loss $|y_t - \prod_{i\in A} x_{t,i}|$ does not decompose into a sum.

So how do we obtain an efficient learning algorithm for conjunctions? One way is to use additive algorithms such as the Perceptron algorithm together with the subset kernel. The additive algorithms for solving the same problem are efficient, but the bounds are much weaker (linear in n instead of logarithmic) (see discussion by Kivinen and Warmuth 1997, Kivinen et al. 1997, Khardon et al. 2001).

We now develop efficient multiplicative algorithms. For convenience, let us switch to learning disjunctions (Because of de Morgan's law, the learning problems for conjunctions and disjunctions

6. Choose the prior $W_{1,A}$ so that for each size, the total weight of all conjunction of that size is $1/n$.
7. Khardon, Roth, and Servedio (2001) call the subset kernel the "monomial kernel". Hardness was shown for computing the predictions of Winnow when there is one weight per conjunction. This is essentially an unnormalized version of the update discussed here.

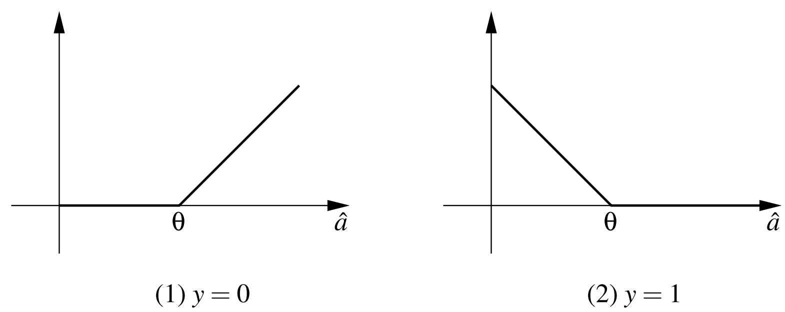


Figure 10: The hinge loss $\lambda = |y - \sigma_\theta(\hat{a})|\ |\theta - \hat{a}|$ as functions of $\hat{a}$.

are equivalent.) Now the loss of disjunction A on example (x_t, y_t) becomes

$$\ell_{t,A} = |y_t - I(\sum_{i \in A} x_{t,i} \geq 1)|,$$

where $I(\text{true}) = 1$ and $I(\text{false}) = 0$. Again, this discrete loss does not decompose and we cannot implement the loss update efficiently.

However, we now modify our setup. We let each disjunction predict with $\sum_{i \in A} x_{t,i}$ instead of $I(\sum_{i \in A} x_{t,i} \geq 1)$. The weighted average prediction is now $\hat{a}_t = \sum_A W_{t,A} \sum_{i \in A} x_{t,i}$ and the algorithm predicts with $\hat{y}_t = \sigma_\theta(\hat{a}_t)$, where the threshold θ is suitably chosen. For a moment let us assign the following loss to the algorithm

$$\lambda_t = |y_t - \sigma_\theta(\hat{a}_t)|\ |\theta - \hat{a}_t| = \ell(y_t, \hat{y}_t).$$

So when $\hat{y}_t = y_t$, then this loss is zero. But when $y_t \neq \hat{y}_t$, it is linear in $\hat{a}_t$. This is the *linear hinge loss* used for motivating Support Vector Machines (Cristianini and Shawe-Taylor, 2000, Gentile and Warmuth, 1998). See Figure 10 to see how the hinge loss behaves with respect to the linear activation $\hat{a}_t$. Note that the weighted average prediction $\hat{a}_t$ and the hinge loss λ_t are of the canonical form we discussed in Section 5.2. So the derivative $\frac{\partial \lambda_t}{\partial W_{t,A}} = \lambda'_t \sum_{i \in A} x_{t,i}$ decomposes into a sum with $\lambda'_t = \partial \ell(y_t, \sigma_\theta(\hat{a})/\partial \hat{a}|_{\hat{a}=\hat{a}_t} = \hat{y}_t - y_t$ (see Figure 10), and the EG update (5.8) has the required form (5.3), with $b_{t,e} = \exp(-\eta(\hat{y}_t - y_t)x_{t,e})$. Now we can indirectly represent the weight vector W_t for the 2^n subsets by maintaining a weight vector w_t on the $2n$ edges, so that $W_t = \Phi(w_t)$ where $\Phi(.)$ is the feature map of the subset kernel. We efficiently update the weights w_t using the Weight Pushing algorithm on the digraph of Figure 2 (See update (6.2) and discussion at the end of Section 6). The weighted average $\hat{a}_t$ can be expressed in term of kernel computations (5.2). However, for the the special case of the subset kernel it follows from (5.1) that $\hat{a}_t = \sum_i w_{t,e_i} \cdot x_{t,i}$. Hence the prediction $\hat{y}_t$ is easy to compute. The algorithm described above is called the Binary Exponentiated Gradient algorithm (BEG algorithm, for short) for learning disjunctions (Bylander, 1997, Helmbold, Panizza, and Warmuth, 2002).

Recall that the mistake bounds provable for the inefficient disjunction learning algorithm are linear in m_*, the minimum number of mistakes of any k-literal disjunction. All efficient disjunction learning algorithm are linear in a_* instead, where a_* is the minimum number of bits/attributes that have to be flipped in all examples (x_t, y_t), so that there is a disjunction of size k that agrees with all labels y_t. More precisely the mistake bound of the efficient disjunction learning algorithm all have the form $O(k \ln n + a_*)$. This includes BEG when the thresholds and learning rate is appropriately chosen (see Theorem 5 of Helmbold et al. 2002). Note that a_* might be up to a factor of k larger than m_*. So there is a price we have to pay for moving from the discrete loss to the decomposable hinge loss which leads to the more efficient algorithms.

The hinge loss can also be used to derive the normalized Winnow algorithm.[8] However now a slightly different kernel must be used corresponding to the regular expression

$$(e_{1,1} + e_{1,2} + \dots e_{1,n})(e_{2,1} + e_{2,2} + \dots e_{2,n}) \dots (e_{n,1} + e_{n,2} + \dots e_{n,n}),$$

where the edges $e_{*,i}$ receive input x_i (i.e. $x_{e_{*,i}} = x_i$). A path $P = \{e_{1,i_1}, e_{2,i_2}, \dots, e_{n,i_n}\}$ corresponds to the disjunction with index set $\{i_1, i_2, \dots, i_n\}$. Note that now many paths represent the same disjunction. We follow the same derivation as for BEG. Instead of letting path P predict with $I(\sum_{e \in P} x_{t,e} \geq 1)$ and using the discrete loss

$$\ell_{t,P} = |y_t - I(\sum_{e \in P} x_{t,e} \geq 1)|,$$

we use a different prediction and the hinge loss. We let the path P predict with $\sum_{e \in P}^n x_{t,e}$, and use the loss

$$\lambda_t = |y_t - \sigma_\theta(\hat{a}_t)|\, |\theta - \hat{a}_t|, \text{ where } \hat{a}_t = \sum_P W_{t,P} \sum_{e \in P} x_{t,e}.$$

Again the loss decomposes. Assume that all edges $e_{j,i}$ are assigned the same initial weight $w_{1,e_{j,i}} = 1/n$. Then, since for any trial t all edges $e_{*,i}$ receive the same factor $b_{t,e_{*,i}} = \exp(-\eta(\hat{y} - y)x_{t,i})$, the Weight Pushing algorithm keeps their weights $w_{t,e_{*,i}}$ the same. Let $w_{t,i}$ denote these weights. Now the Weight Pushing algorithm updates the weights as follows:

$$w_{t+1,i} = \frac{w_{t,i} b_{t,i}}{\sum_{j=1}^n w_{t,j} b_{t,j}}, \text{ where } b_{t,i} = \exp\left(-\eta(\hat{y} - y)x_{t,i}\right).$$

Also the prediction simplifies to the following:

$$\begin{aligned} \hat{a}_t &= n \sum_i \left(w_{t,i} x_{t,i} \left(\sum_j w_{t,j} \right)^{n-1} \right) \qquad (9.1) \\ &= n \sum_i w_{t,i} x_{t,i}, \end{aligned}$$

because $\sum_j w_{t,j} = 1$ for Normalized Winnow. The constant n can be incorporated into the threshold. See Theorem 9 of Helmbold et al. (2002) for the settings of the learning rate and threshold that lead to the $O(k \log n + a_*)$ bound for Normalized Winnow, where a_* is as before the minimum number of bits/attributes that have to be flipped in all examples (x_t, y_t), so that there is a disjunction of size k that agrees with all labels y_t.

The Winnow algorithm can also be derived using the hinge loss and the same kernel as Normalized Winnow. The only difference is that the weights are not normalized and the Weight Pushing algorithm is not needed. However now, we do not know how to motivate the prediction of the Winnow algorithm because (9.1) does not simplify (since $\sum_i w_{t,i}$ might not be one).

8. This algorithm is due to Nick Littlestone and was discussed for the first time with the authors in 1995.

The weights of the BEG algorithm have a probabilistic interpretation as representing n independent Bernoulli coins. Similarly, normalized Winnow corresponds to an n fold multinomial distributions. No probabilistic interpretation is known for the original Winnow[9] (see Helmbold et al. 2002 for a discussion).

Finally, for most of this section we focused on efficient algorithms for learning disjunctions of monotone literals. Going from the discrete loss to the hinge loss allowed us to use one weight per literal instead of one weight per disjunction. When the base functions are conjunctions instead of single literals then the same motivation leads to algorithms for learning Boolean functions in Disjunctive Normal Form (DNF). Now going from the discrete loss to the hinge loss allows us to use one weight per conjunctions instead of one weight per DNF formula. For example, the algorithm we started with in this section with its threshold moved to $\theta = \frac{1}{k}$ now becomes the Normalized Winnow algorithm for learning monotone DNF formulas with up to k terms. Its mistake bound is $O(kn + a_*)$, but it maintains 2^n weights. See Maass and Warmuth (1998), Khardon et al. (2001) for related discussions.

10. Predicting Nearly as Well as the Best Pruning

One of the main representations of Machine Learning is decision trees. Frequently a large tree is produced initially and then this tree is pruned for the purpose of obtaining a better predictor. A pruning is produced by deleting some vertices in the tree and with them all their successors. Although there are exponentially many prunings, a recent method developed in coding theory and machine learning makes it possible to maintain one weight per pruning. In particular, Helmbold and Schapire (1997) use this method to design an elegant multiplicative algorithm that is guaranteed to predict nearly as well as the best pruning of a decision tree in the on-line prediction setting. Recently, the authors (Takimoto and Warmuth, 2002) generalize the pruning problem to the much more general class of acyclic planar digraphs. The key property of planar digraphs G that is used in the previous paper is the following: There is a dual digraph G^D such that prunings of the original G are paths in the dual G^D and vice versa. In particular, for SP digraphs (a subclass of planar digraphs), the dual digraphs are easily obtained by swapping the union and the concatenation operations in the syntax tree. In this section we restate this result in terms of path kernels on SP digraphs.

A pruning of a SP digraph G is a minimal set of edges (a cut) that interrupts all paths from the source to the sink. More precisely, a pruning R of G is a set of edges of G such that R intersects with any path $P \in \mathcal{P}(G)$ at exactly one edge $P \cap R$ (see Figure 11). In the example of Figure 4, all the prunings of the graph are $\{e_1, e_2, e_3, e_6\}$, $\{e_1, e_4, e_5, e_6\}$, $\{e_1, e_7\}$. In general, we let $\mathcal{R}(G)$ be the set of all prunings of R. Now we give the problem formally. In the following we fix a loss function $\ell : [0,1] \times [0,1] \to [0,\infty]$, say, the square loss $\ell(y, \hat{y}) = (y - \hat{y})^2$.

In each trial $t = 1, 2, \ldots, T$, the following happens:

1. Prediction values $x_{t,e} \in [0,1]$ are given to the edges e of a single path $P_t \in \mathcal{P}(G)$ called the *prediction path*. Typically this path is generated by some decision process that passes down the graph. Here we do not need to be concerned with how the prediction path P_t is generated.

2. Each pruning $R \in \mathcal{R}(G)$ is assumed to predict as the edge that cuts the path P_t. That is, R predicts $x_{t,P_t \cap R}$.

9. Winnow belongs to the family of Unnormalized Exponentiated Gradient algorithms (Kivinen and Warmuth, 1997).

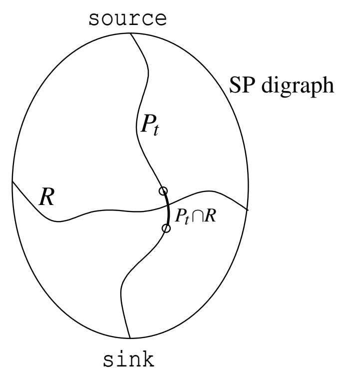


Figure 11: Any pruning R of the digraph intersects with path P_t at exactly one edge, denoted by $P_t \cap R$.

3. The algorithm maintains a weight per pruning and makes its own prediction $\hat{y}_t \in [0,1]$ based on the weighted average of the predictions of all prunings (details are given below).

4. An outcome $y_t \in [0,1]$ is observed.

5. The algorithm incurs loss $\lambda_t = \ell(y_t, \hat{y}_t)$. Similarly a pruning R incurs loss $\ell_{t,R} = \ell(y_t, x_{t,P_t \cap R})$.

The goal is to make the total loss of the algorithm not much larger than the total loss of the best pruning.

First we describe an algorithm that works on the feature space. The algorithm is based on the Weighted Averaging algorithm (the WA algorithm, for short). The WA algorithm maintains a weight $W_{t,R}$ per pruning R. (Here we use the upper case letter $W_{t,R}$ to denote the weights of prunings. In a moment we will see that these weights become product features.) In each trial t, the algorithm predicts with the weighted average of the predictions of the prunings. That is,

$$\hat{y}_t = \sum_R W_{t,R}\, x_{t,P_t \cap R}.$$

After the outcome y_t is observed, the weights on the prunings are updated using the familiar loss update

$$W_{t+1,R} = \frac{W_{t,R} \exp(-\eta \ell_{t,R})}{\sum_R W_{t,R} \exp(-\eta \ell_{t,R})},$$

where η is a non-negative learning rate. The WA algorithm guarantees the following performance (Kivinen and Warmuth, 1999): Assume that for any fixed $y \in [0,1]$, the function $f_y : [0,1] \to [0,1]$ defined as

$$f_y(x) = \exp(-\eta \ell(y,x)) \tag{10.1}$$

is concave. Then, it holds that

$$\sum_{t=1}^{T} \lambda_t \le \min_R \left(\sum_{t=1}^{T} \ell_{t,R} + (1/\eta) \ln \frac{1}{W_{1,R}} \right)$$

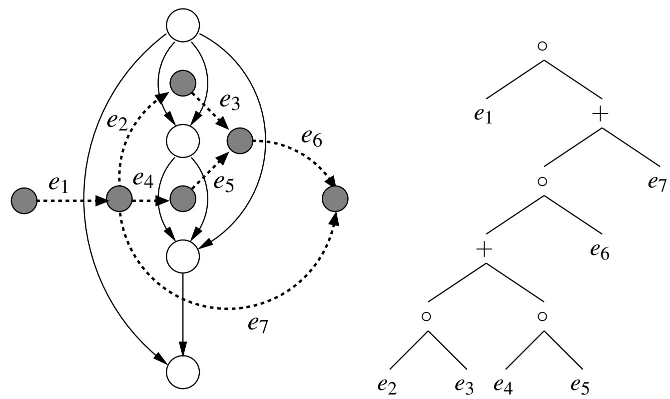


Figure 12: The dual SP digraph G^D of G of Figure 4 and its syntax tree. G is represented by open circles and solid lines, and G^D is by closed circles and dashed lines. Observe that each edge of G^D intersects with exactly one edge of G. We use the same symbol for these edges. The syntax tree of G^D is obtained by swapping the $+$ and $\circ$ in the syntax tree of G.

where $W_{1,R}$ is the initial weight for R. Note that the condition on the concavity of the functions (10.1) is satisfied by many convex loss functions for an appropriate choice of η. For example, the square loss satisfies the condition with $\eta \leq 1/2$. The higher the learning rate η, the better the bound. Higher learning rates are permissible with the fancier prediction functions developed by Vovk (1990). Here we only discuss the simplest case of predicting with the weighted average (see Kivinen and Warmuth (1999) for an overview). More sophisticated version of the above bound using relative entropies (as done in Section 8) are easily obtainable.

Now we give an efficient implementation of this algorithm in terms of path kernels. Consider the dual SP digraph G^D which is defined by swapping the $+$ and $\circ$ operations in the syntax tree. Note that prunings and paths in G are swapped in G^D, namely, a pruning R in G is a path in G^D and the path P in G is a pruning in G^D (Takimoto and Warmuth, 2002). See Figure 12 for an example.

The efficient implementation maintains weights w_t on the edges of G^D such that the weight of path R in G^D is $W_{t,R} = \prod_{e \in R} w_{t,e}$. If $\Phi^D(.)$ is the feature map associated with the path kernel of G^D, then we use this map to indirectly represent the weights on the prunings of G via weights on the edges, i.e. $W_t = \Phi^D(w_t)$. We can also represent the predictions of the prunings using the same feature map. At trial t, the inputs $x_{t,e}$ are assigned only to the edges of path P_t. We extend the inputs to all edges by letting $x_{t,e} = 1$ for $e \notin P_t$ and use x_t to denote the entire input vector over the edges. Now the value of feature $X_{t,R}$ is its prediction, i.e.

$$X_{t,R} = \Phi^D(x_t)_R = \prod_{e \in R} x_{t,e} = x_{t,P_t \cap R}.$$

So the prediction $\hat{y}_t$ of the algorithm becomes $\hat{y}_t = W_t \cdot X_t = \Phi^D(w_t) \cdot \Phi^D(x_t)$, which is efficiently computed by the path kernel of G^D.

If we define the loss of edge e at trial t as $l_{t,e} = \ell(y_t, x_{t,e})$, if $e \in P_t$, and $l_{t,e} = 0$, if $e \notin P_t$, then the loss of path R decomposes into a sum: $\ell_{t,R} = \ell(y_t, x_{t,P_t \cap R}) = \sum_{e \in R} l_{t,e}$. Thus the above loss update has the required form (5.3)

$$W_{t+1,R} = \frac{W_{t,R}\prod_{e\in R}\exp(-\eta\sum_{e\in R} l_{t,e})}{\sum_R W_{t,R}\exp(-\eta\sum_{e\in R} l_{t,e})} = \frac{\prod_{e\in R} w_{t,e} b_{t,e}}{\Phi(w_t)\cdot\Phi(b_t)},$$

where $b_{t,e} = \exp(-\eta l_{t,e})$, if $e \in P_t$, and one otherwise. Hence instead of directly maintaining the weights W_t on the prunings, we can efficiently update the edge weights w_t using the Weight Pushing algorithm on the digraph G^D.

In Appendix C.3 we give an efficient implementation for computing kernels and the Weight Pushing algorithm. The algorithm does not maintain a vector of edge weights w for the given digraph (here G^D) but instead maintains a vector of edge weights μ for the syntax tree of the regular expression H_0 that describes the path set of the digraph. More precisely, for each union node $H = H_1 + \cdots + H_k$ of the syntax tree, we maintain weights μ_i^H for $1 \le i \le k$ (Here k is called the degree of H). These new weights implicitly define a probability vector W on the path set of the digraph via the following stochastic process for traversing the syntax tree for H_0: Start from the root; if we are at a concatenation node, then we go to all of its children; if we are at a union node H, then choose a child node H_i with probability μ_i^H and go to H_i; finally all the leaves we visit form a path $R \in \mathcal{P}^{H_0}$, and the weight W_R is defined as the probability that the path R is chosen by this process. (Recall that H_0 generates the path set of G^D that are the pruning set of G.) In Appendix C.3 we treat the general case where $*$-operations are allowed as well.

For a given input x to the edges, a node H of the syntax tree is *relevant* with respect to x if H has a descendant leaf with input not equal one. We will show in the appendix that the algorithm computes kernels in time linear in the number of relevant nodes and computes the new weights $\tilde{\mu}$ in time linear in the sum of the degrees of relevant union nodes. (Recall that we have probabilistic weights μ_i^H only for union nodes H. Even if only one child node of H is relevant, the weights μ_i^H of all children of H will be affected by the update. This is why the time for updating weights is proportional to the sum of the degrees of relevant union nodes.)

We now return to the on-line pruning problem of this section. Note that we work on the syntax tree for the dual graph G^D. Recall that for each trial t the algorithm is given the prediction path P_t in G which is a pruning of G^D. We claim that for a given pruning P_t of G^D, the sum of the degrees of relevant union nodes of the syntax tree is $O(|P_t|)$. To show this, we give a proof for the dual version of the claim, which we hope is easier to understand. Now the claim can be restated as follows: For a given path P_t of the primal graph G, the sum of the degrees of relevant *concatenation* nodes of the syntax tree for G is $O(|P_t|)$. Without loss of generality we assume that all the inputs x_e on the path P_t ($e \in P_t$) are not one since this assumption only increases the relevant nodes. For any node H of the syntax tree, let $\deg(H)$ denote the sum of the degrees of relevant concatenation nodes of the subtree rooted at H. Furthermore, let $P^H \in \mathcal{P}^H$ denote the partial path of P_t that goes through the component $\mathbb{H}$, that is, $P^H = E(\mathbb{H}) \cap P_t$, where $E(\mathbb{H})$ denotes the edge set of $\mathbb{H}$. Note that P^H becomes a path of $\mathbb{H}$. Below we show by induction that for any relevant node H, $\deg(H) \le 2|P^H| - 2$, which proves the claim. For the base case, where $H = e$ for some edge symbol e, the claim trivially holds. If $H = H_1 + \cdots + H_k$, then the path P^H is a path $P^{H_i} \in \mathcal{P}^{H_i}$ and $\deg(H) = \deg(H_i)$ for some H_i. So by the induction hypothesis the claim holds. Finally assume $H = H_1 \circ \cdots \circ H_k$. Note that without loss of generality we assume $k \ge 2$. In this case, the path P^H goes through all the components $\mathbb{H}_1, \ldots, \mathbb{H}_k$ and thus P^H is of the form $P^H = P^{H_1} \cup \cdots \cup P^{H_k}$. So all the children H_i of H are also relevant. Clearly, $\deg(H) = k + \sum_{i=1}^k \deg(H_i)$. By the induction hypothesis, this is at most $k + \sum_{i=1}^k \left(2|P^{H_i}| - 2\right) = 2|P^H| - k \le 2|P^H| - 2$, which completes the claim.

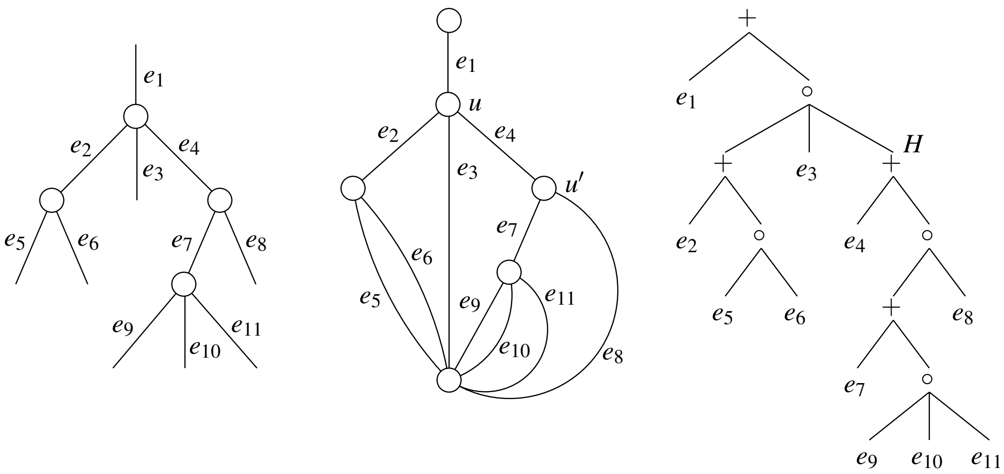


Figure 13: A tree (left) can be viewed as a SP digraph G (middle) by introducing the source and the sink and merging all leaves into one sink. The syntax tree of the dual G^D is given (right). Each union node (for example H) corresponds to an internal edge (u,u') of G.

In summary, applying the algorithm given in Appendix C.3 to the on-line pruning problem of this section, we can compute the prediction value $\hat{y}_t$ and the new weights μ_{t+1} in $O(|P_t|)$ time. This can be significantly faster than the original Weight Pushing algorithm, especially when the given SP digraph has a small depth.

Finally we note that any tree can be interpreted as a SP digraph by merging all leaves into one sink. Moreover, if the given SP digraph G is a tree in this sense, we give an interesting interpretation for the weights μ (see Figure 13). For simplicity we assume that any vertex other than the root of the tree G has degree more than one. Then each union node H of the syntax tree for G^D has always two children H_1 and H_2. One child, H_1, is a leaf labeled with an internal edge (u,u') of G and the other child, H_2, corresponds to the subtree rooted at u'. The weight μ_1^H can be interpreted as the probability of pruning the tree G at this edge (u,u') and $\mu_2^H = 1-\mu_1^H$ as the probability of choosing a pruning in the subtree below edge (u,u'). Curiously, the resultant algorithm for trees turns out to be exactly the same as the one motivated by dynamic programming (Takimoto et al., 2001).

10.1 Pruning for Probabilistic Path Inputs

In this section we extend the on-line pruning problem for SP digraphs to the situation where prediction path P_t is chosen probabilistically. More precisely, in trial t the algorithm observes prediction values $x_{t,e} \in [0,1]$ assigned to all edges rather than along a single prediction path, together with the set of edge weights $\nu_{t,e} \in [0,1]$ that satisfies Properties P1, P2 and P3. The value $x_{t,e}$ is interpreted as the prediction of edge e at trial t and the edge weights ν_t induce a probability distribution on all paths P of G. That is, the probability of prediction path P at trial t is given by $\Phi(\nu_t)_P = \prod_{e\in P} \nu_{t,e}$. Now the prediction $X_{t,R}$ of a pruning R becomes a random variable and it takes value $x_{t,P\cap R}$ with probability $\Phi(\nu_t)_P$. Accordingly, when an outcome y_t is presented, the loss $\ell_{t,R}$ of R at trial t is defined as the *expected loss*, i.e.,

$$\ell_{t,R} = \mathbb{E}\left[\ell(y_t, X_{t,R})\right] = \sum_P \Phi(\nu_t)_P\, \ell\left(y_t, x_{t,P\cap R}\right).$$

Since any path P intersects with the pruning R at exactly one edge, the set of paths that go through $e \in R$ and the set of paths that go through $e' \in R$ are disjoint if $e \neq e'$. Therefore, we have

$$\begin{aligned}
\ell_{t,R} &= \sum_{e\in R}\sum_P \Phi(\nu_t)_P \ell(y_t, x_{t,e}) I(e\in P) \\
&= \sum_{e\in R} p_{t,e}\ell(y_t, x_{t,e}),
\end{aligned} \tag{10.2}$$

where

$$p_{t,e} = \sum_P \Phi(\nu_t)_P I(e\in P)$$

is the probability that the prediction path goes through the particular edge e. Note that for any pruning R,

$$\sum_{e\in R} p_{t,e} = 1. \tag{10.3}$$

Similarly, the expected prediction of pruning R is

$$\mathbb{E}[X_{t,R}] = \sum_P \Phi(\nu_t)_P\, x_{t,P\cap R} = \sum_{e\in R} p_{t,e} x_{t,e}. \tag{10.4}$$

As shown in Section 5, $p_{t,e}$ can be expressed with kernel computations. That is, letting u^e be the edge weights defined as

$$u^e_{e'} = \begin{cases} 1 & \text{if } e' \neq e \\ 0 & \text{if } e' = e, \end{cases}$$

we have

$$p_{t,e} = \sum_P \prod_{e'\in P} \nu_{t,e'} \left(1 - \prod_{e'\in P} u^e_{e'}\right) = K(\nu_t, 1) - K(\nu_t, u^e). \tag{10.5}$$

Again our goal is to produce predictions $\hat{y}_t$ so that the total loss $\sum_{t=1}^T \ell(y_t, \hat{y}_t)$ is not much larger than the loss $\sum_{t=1}^T \ell_{t,R}$ of the best pruning R.

Recall that a path P of G is a pruning of G^D and a pruning R of G is a path of G^D. In each trial $t = 1, \ldots, T$,

1. Prediction values $x_{t,e} \in [0,1]$ and weights $\nu_{t,e} \in [0,1]$ are given to all edges e of G^D, where ν_t fulfills Properties P1, P2 and P3 for the primal graph G. The weight vector ν_t on the edges assigns probability $\prod_{e\in P} \nu_{t,e}$ to the prediction path P of G and $\Phi(\nu_t)$ represents the probability vector on all such paths.

2. The algorithm predicts $\hat{y}_t \in [0,1]$.

3. An outcome $y_t \in [0,1]$ is observed.

4. The algorithm incurs loss $\lambda_t = \ell(y_t, \hat{y}_t)$ and each path R of G^D incurs loss $\ell_{t,R}$ which is given by (10.2).

In this setting we show that a slight modification of the WA algorithm has the same loss bound as in the deterministic case. It is easy to extent the methods to the fancier predictions given by Vovk (1990) that give improved bounds (as long as they predict with a function of the weighted average).

The algorithm maintains edge weights w_t so that they represent weights $W_{t,R} = \prod_{e\in R} w_{t,e}$ for paths R of G^D, i.e. $W_t = \Phi^D(w_t)$. Now the prediction $\hat{y}_t$ of the algorithm at trial t is the weighted average of the *expected* predictions (10.4) of paths. That is,

$$\hat{y}_t = \sum_R W_{t,R}\mathbb{E}[X_{t,R}] = \sum_R W_{t,R}\sum_e p_{t,e}x_{t,e}. \tag{10.6}$$

Using (10.5) and (5.1), the prediction $\hat{y}_t$ can be expressed with kernel computations.

When an outcome y_t is given, the algorithm updates its weights so that

$$W_{t+1,R} = \frac{W_{t,R}\exp(-\eta\ell_{t,R})}{\sum_R W_{t,R}\exp(-\eta\ell_{t,R})} \tag{10.7}$$

holds for any pruning R. Since $\ell_{t,R}$ decomposes into a sum over edges in R, this update can be efficiently simulated using the Weigh Pushing algorithm for updating the edge weights w_t on the digraph G^D.

The below theorem gives the loss bound for the algorithm.

Theorem 4 *Let $\ell : [0,1]\times[0,1] \to [0,\infty]$ be a loss function such that ℓ is convex with respect to the second argument and the function f_y given by (10.1) is concave. Then, the algorithm using (10.6) and (10.7) for prediction and update, respectively, guarantees the following performance: For any probability vector U on the prunings of the digraph G, it holds that*

$$\sum_{t=1}^T \lambda_t \le \sum_R U_R \sum_{t=1}^T \ell_{t,R} + (1/\eta)d(U,W_1),$$

where d is the relative entropy and W_1 is the initial probability vector on the prunings.

Proof Let U be an arbitrarily probability vector on the prunings of G. Looking at the progress $d(U,W_t) - d(U,W_{t+1})$ for one trial, we have

$$\begin{aligned}
d(U,W_t) - d(U,W_{t+1}) &= \sum_R U_R \ln\frac{W_{t+1,R}}{W_{t,R}} \\
&= \sum_R U_R \ln\frac{\exp(-\eta\ell_{t,R})}{\sum_R W_{t,R}\exp(-\eta\ell_{t,R})} \\
&= -\eta\sum_R U_R\ell_{t,R} - \ln\sum_R W_{t,R}\exp(-\eta\ell_{t,R}) \\
&= -\eta\sum_R U_R\ell_{t,R} - \ln\sum_R W_{t,R}\exp\left(-\eta\sum_{e\in R} p_{t,e}\ell(y_t,x_{t,e})\right). \qquad (10.8)
\end{aligned}$$

By the convexity of L and (10.3), we have

$$\sum_{e\in R} p_{t,e}\ell(y_t,x_{t,e}) \ge L\left(y_t, \sum_{e\in R} p_{t,e}x_{t,e}\right) = \ell(y_t,\mathbb{E}[X_{t,R}]).$$

From this and the concavity of the function f_y, it follows that

$$\begin{aligned}
\sum_R W_{t,R} \exp\left(-\eta \sum_{e\in R} p_{t,e}\ell(y_t, x_{t,e})\right) &\le \sum_R W_{t,R} \exp\Big(-\eta L(y_t, \mathbb{E}[X_{t,R}])\Big) \\
&= \sum_R W_{t,R} f_{y_t}(\mathbb{E}[X_{t,R}]) \\
&\le f_{y_t}\left(\sum_R W_{t,R}\mathbb{E}[X_{t,R}]\right) \\
&= f_{y_t}(\hat{y}_t) \\
&= \exp(-\eta\ell(y_t, \hat{y}_t)) \\
&= \exp(-\eta\lambda_t).
\end{aligned}$$

Plugging this into (10.8), we have

$$d(U, W_t) - d(U, W_{t+1}) \ge \eta\left(-\sum_R U_R \ell_{t,R} + \lambda_t\right).$$

Summing this inequality over $t = 1, \ldots, T$, we immediately have the theorem. ■

Note that if we take U as the unit vectors that puts all probability on a single pruning R, then we obtain the following simpler version of the bound:

$$\sum_{t=1}^{T} \lambda_t \le \min_R \left(\sum_{t=1}^{T} \ell_{t,R} + (1/\eta)\ln(1/w_{1,R})\right).$$

11. Conclusion

In this paper we showed that path kernels can be used to indirectly maintain exponentially many path weights. Multiplicative updates give factors to the edges and the Weight Pushing algorithm renormalizes the edge weights so that the outflow out of each vertex remains one. We also showed that it is often convenient to express the path sets as regular expressions, leading to more efficient implementations of path kernels and the Weight Pushing algorithm. We gave the path kernels that interpret the BEG and the normalized Winnow algorithms for learning disjunctions as direct algorithms over exponentially many paths. A number of other examples were given for implementing multiplicative algorithms over exponentially many weights.

In Section 5 we specified the requirements needed for our method of using path kernels: The weight update must have the form (5.3), and if the algorithm predicts, then its prediction must be efficiently computable via for example kernel computations. We gave a number of examples of our methods.

The motivation and analysis of various additive and multiplicative linear threshold algorithms based on the hinge loss was done before by Gentile and Warmuth (1998). In this paper, we start with inefficient algorithms (with exponentially many weights) and show that the transition to the hinge loss leads to multiplicative updates that can be simulated implicitly (because the gradient of this loss decomposes into a sum).

Multiplicative algorithms belong to the EG family of updates and in this paper we found special kernels that can be used to efficiently implement updates from that family. A key requirement was that the features are products. The question is whether there are special kernels for other families of updates. Recently updates have been found that interpolate between the EG family and and the additive updates (Gentile and Littlestone, 1999). These are called the p-norm updates. It is an open question whether there are kernels that allow us to efficiently implement the p-norm updates over exponentially many variables.

Acknowledgments

We thank Sandra Panizza for fruitful discussions and Fernando Pereira for pointing out that we had rediscovered the Weight Pushing algorithm of Mehryar Mohri. Manfred Warmuth acknowledges the support of NSF grant CCR-9821087.

Appendix A. On the Uniqueness of the Solution to the Linear Equations for Kernels

Recall that we want to compute

$$K_u(x,x') = \sum_{P\in\mathcal{P}(u)} \prod_{e\in P} x_e x'_e.$$

Consider the following linear equations:

$$k_u = \begin{cases} 1 & \text{if } u = \texttt{sink}, \\ \displaystyle\sum_{u':(u,u')\in E(G)} x_{(u,u')} x'_{(u,u')} k_{u'} & \text{if } u \neq \texttt{sink}. \end{cases} \tag{A.1}$$

Here k_u are the linear variables and clearly $k_u = K_u(x,x')$ is a solution. In this appendix we show that this solution is essentially unique, if $x,x' \geq 0$ and $K(x,x')$ finite.

A vertex u is *source-reachable* (with respect to x and x') if there exists a path P from the source to u such that $\prod_{e\in P} x_e x'_e > 0$. *Sink-reachability* is defined similarly. In other words, a vertex u is sink-reachable, if $K_u(x,x') > 0$ and not sink-reachable if $K_u(x,x') = 0$. To make the solution of the above equations unique, we use the following additional constraints:

$$k_u = 0, \text{for any vertex } u \text{ that is not sink-reachable.} \tag{A.2}$$

Theorem 5 *Assume that $x,x' \geq 0$ and $K(x,x') < \infty$. Let k be a solution to (A.1) and (A.2). Then for any source-reachable vertex u,*

$$k_u = K_u(x,x') = \sum_{P\in\mathcal{P}(u)} \prod_{e\in P} a_e.$$

Proof First we assume that the source is not sink-reachable, i.e., $K(x,x') = 0$. In this case, any source reachable vertex u is also not sink-reachable. So the vertex u is eliminated and we have the right solution for u: $k_u = K_u(x,x') = 0$.

Next we assume that the source is sink-reachable. Plugging (A.2) into (A.1) we get a system of linear equations restricted to the variables k_u with sink-reachable vertices u. Let n denote the number of the sink-reachable vertices and we use integers $1,\ldots,n$ to specify such vertices. We assume that the source and the sink is the first and the last (nth) vertices, respectively. Let $k = (k_1,\ldots,k_n)'$ be the column vector whose ith component is the solution k_i for vertex i, where the prime ($'$) denotes the transposition. Similarly, let A be the $n \times n$ matrix whose (i, j) component $a_{i,j}$ is defined as $a_{i,j} = x_{(i,j)} x'_{(i,j)}$ if (i, j) is an edge of G and $a_{i,j} = 0$ otherwise. Note that $a_{n,j} = 0$ for all j, since the sink has no outgoing edges. Now the linear equations (A.1) become

$$k = Ak + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \tag{A.3}$$

with the condition that $A \geq 0$ and $K_1(x,x') < \infty$. Let $e_n = (0,\dots,0,1)'$. Expanding (A.3) we have

$$\begin{aligned} k &= Ak + e_n \\ &= A(Ak + e_n) + e_n \\ &\vdots \\ &= \sum_{s=0}^{\infty} A^s e_n + A^{\infty} k, \end{aligned}$$

where $A^{\infty} = \lim_{s\to\infty} A^s$. So it suffices to show that the first term coincides with $K(x,x') = (K_1(x,x'), \dots, K_n(x,x'))'$ and for any source-reachable vertex i, the ith row of A^{∞} is the zero vector $(0,\dots,0)$.

First we show that $\sum_{s=0}^{\infty} A^s e_n = K(x,x')$. Let $a^s_{i,j}$ denote the (i, j) component of A^s. Since $a^s_{n,j} = 0$ for all j and $s \geq 1$, the nth component of the left hand side is 1. Here we used the fact that A^0 is the identity matrix. On the other hand, we have $K_n(x,x') = 1$ by definition. So the equality holds for the nth component. For $i < n$, it is easy to see that

$$\begin{aligned} K_i(x,x') &= \sum_{P\in\mathcal{P}(i)} \prod_{e\in P} a_e \\ &= \sum_{s=1}^{\infty} \sum_{\substack{P\in\mathcal{P}(i) \\ |P|=s}} \prod_{e\in P} a_e \\ &= \sum_{s=1}^{\infty} a^s_{i,n}, \end{aligned}$$

which clearly coincides with the ith component of $\sum_{s=0}^{\infty} A^s e_n$.

Next we show that for a source-reachable vertex i and any vertex j, $a^{\infty}_{i,j} = \lim_{s\to\infty} a^s_{i,j} = 0$. Let i be a source-reachable vertex. That is, there exists a path P_0 from the source to i such that $\prod_{e\in P_0} a_e > 0$. Fix an arbitrary $j \neq i$. Since only sink-reachable vertices remain in the equations, there exists a path P_1 from j to the sink such that $\prod_{e\in P_1} a_e > 0$. Let $\mathcal{P}(i, j)$ denote the set of paths from i to j that do not pass the vertex j before arriving at j, and let

$$K_{i,j}(x,x') = \sum_{P\in\mathcal{P}(i,j)} \prod_{e\in P} a_e.$$

Restricting the prefix and the suffix of paths to P_0 and P_1, respectively, we have a subset of all source-sink paths. This, together with the assumption that $K_1(x,x') < \infty$, implies that

$$\infty > K_1(x,x') \geq \left(\prod_{e\in P_0} a_e\right) \left(\sum_{P\in\mathcal{P}(i,j)} \prod_{e\in P} a_e\right) \left(\prod_{e\in P_1} a_e\right)$$

$$= \left(\prod_{e\in P_0} a_e\right)\left(\prod_{e\in P_1} a_e\right) K_{i,j}(x,x').$$

Since the first two factors are positive, we have $K_{i,j}(x,x') < \infty$. Now we observe that

$$\begin{aligned} K_{i,j}(x,x') &= \sum_{P\in\mathcal{P}(i,j)} \prod_{e\in P} a_e \\ &= \sum_{s=1}^{\infty} \sum_{\substack{P\in\mathcal{P}(i,j)\\ |P|=s}} \prod_{e\in P} a_e \\ &= \sum_{s=1}^{\infty} a_{i,j}^s, \end{aligned}$$

which is finite. Since $A \geq 0$, it must hold that $a_{i,j}^{\infty} = 0$, as required. ■

Appendix B. On the Uniqueness of Edge Weights

In this section we show that no two distinct edge weights represent the same path weights. More precisely the claim is described as follows.

Theorem 6 *Let w and w' be edge weights and W and W' be the corresponding path weights. We assume that Properties P1–3 hold. That is, $W_P = \prod_{e\in P} w_e$ and $W'_P = \prod_{e\in P} w'_e$ for any path P; $\sum_{u':(u,u')} w_{(u,u')} = 1$ and $\sum_{u':(u,u')} w'_{(u,u')} = 1$ for any vertex u; and $\sum_P W_P = 1$ and $\sum_P W'_P = 1$. Assume that $W = W'$. Then for all edges $e = (u,u')$ on paths P for which W_P or W'_P are positive, we have $w_e = w'_e$.*

Proof The proof we give below is based on the entropy decomposition argument developed by Singer and Warmuth (1997). Consider the relative entropy between W and W': $d(W,W') = \sum_P W_P \ln(W_P/W'_P)$. We rewrite this relative entropy as follows:

$$\begin{aligned} d(W,W') &= \sum_P W_P \ln \prod_{e\in P} \frac{w_e}{w'_e} \\ &= \sum_P W_P \sum_{e\in P} \ln \frac{w_e}{w'_e} \\ &= \sum_P W_P \sum_e \#_e(P) \ln \frac{w_e}{w'_e}, \end{aligned}$$

where the second sum is over all edges and $\#_e(P)$ denotes the number of occurrences of edge e in the path P. If $e = (u,u')$, then the properties assure that the expectation of $\#_e(P)$ under W, denoted $\mathbb{E}_W[\#_e(P)]$, is w_e times the expected number of visits to the vertex u in the path P, denoted $\mathbb{E}_W[\#_u(P)]$. That is, $\mathbb{E}_W[\#_e(P)] = w_e \mathbb{E}_W[\#_u(P)]$. It follows that

$$\begin{aligned} d(W,W') &= \sum_e \ln \frac{w_e}{w'_e} \sum_P W_P \#_e(P) \\ &= \sum_e \ln \frac{w_e}{w'_e} \mathbb{E}_W[\#_e(P)] \end{aligned}$$

$$\begin{aligned}
&= \sum_{e=(u,u')} \left(w_e \ln \frac{w_e}{w'_e} \right) \mathbb{E}_W[\#_u(P)] \\
&= \sum_u \mathbb{E}_W[\#_u(P)] \sum_{u'} w_{(u,u')} \ln \frac{w_{(u,u')}}{w'_{(u,u')}} \\
&= \sum_u \mathbb{E}_W[\#_u(P)] d(w_{(u,\cdot)}, w'_{(u,\cdot)}),
\end{aligned}$$

where $w_{(u,\cdot)}$ and $w'_{(u,\cdot)}$ are the probability distributions over the edges outgoing from u. Since $d(W,W')=0$, we must have $w_{(u,\cdot)} = w'_{(u,\cdot)}$ for any u such that $\mathbb{E}_W[\#_u(P)] > 0$. That is, $w_{(u,\cdot)} = w'_{(u,\cdot)}$ for any u that lies on a path P for which W_P is positive. ■

Appendix C. Weight Pushing Algorithm on Syntax Trees

Now we give an efficient implementation for the Weight Pushing algorithm that computes (6.1), namely,

$$\tilde{w}_e = \frac{w_e b_e K_{u'}(w,b)}{K_u(w,b)} \tag{C.1}$$

for any edge $e=(u,u')$. The algorithm does a pass over the syntax tree of a given regular expression H_0 and runs in time linear in the size of the regular expression. (A further speedup is given in Section C.3, where we present an algorithm that is sub linear in the size of the regular expression.) Recall that each node H of the syntax tree corresponds to a regular expression H which represents a component $\mathbb{H}$ of the entire graph $\mathbb{H}_0$. We use the same symbol H for internal nodes of the syntax tree and the corresponding regular expression.

C.1 Weight Pushing Algorithm for SP Digraphs

First we assume that the given regular expression H_0 does not have $*$-operation. That is, we give an update rule for edge weights for SP digraphs. The idea is to compute edge weights $\tilde{w}^H$ recursively for each component $\mathbb{H}$ so that $\tilde{w}^H$ is the same as the weights that the Weight Pushing algorithm would produce when applied on $\mathbb{H}$. If edge e is outgoing from the source of the component $\mathbb{H}$, then we call e a *source edge* of $\mathbb{H}$. For any edge e in $\mathbb{H}$, let $\tilde{w}^H_e$ be defined recursively as

$$\tilde{w}^H_e = \begin{cases} 1 & \text{if } H = e \text{ for some edge symbol } e, \\ \frac{K^{H_i}(w,b)}{K^H(w,b)} \tilde{w}^{H_i}_e & \text{if } H = H_1 + \cdots + H_k \text{ and } e \text{ is a source edge of } \mathbb{H}_i, \\ \tilde{w}^{H_i}_e & \text{if } H = H_1 \circ \cdots \circ H_k \text{ and } e \text{ is an edge of } \mathbb{H}_i. \end{cases} \tag{C.2}$$

Finally the weights on the edges are given by $\tilde{w}_e = \tilde{w}^{H_0}_e$.

The next theorem shows that this update assures (C.1) for any component $\mathbb{H}$. To describe this, we need to extend the kernel K^H to the sums over paths starting from an arbitrary vertex u in $\mathbb{H}$ as before. For a component $\mathbb{H}(\mathtt{s},\mathtt{t})$ and a vertex u of $\mathbb{H}$, let $\mathcal{P}^H(u)$ denote the set of all paths from u to $\mathtt{t}$ in $\mathbb{H}$. Furthermore, we define the kernel associated with H and vertex u of $\mathbb{H}$ as

$$K^H_u(w,b) = \sum_{P \in \mathcal{P}^H(u)} \prod_{e \in P} w_e b_e.$$

Note that $\mathcal{P}^H = \mathcal{P}^H(\mathtt{s})$ and hence $K^H(w,b) = K^H_{\mathtt{s}}(w,b)$.

Theorem 7 *Let H_0 be a regular expression that has no $*$-operations. For any component graph $\mathbb{H}$ of $\mathbb{H}_0$ and any edge $e=(u,u')$ of $\mathbb{H}$, the update rule (C.2) satisfies*

$$\tilde{w}_e^H = \frac{w_e b_e K_{u'}^H(w,b)}{K_u^H(w,b)}.$$

Proof We show the theorem by an induction on the depth of the syntax tree.

In the case where H consists of a single edge $e=(u,u')$, then by the above formula $\tilde{w}_e^H=1$ as required, because for the sink u' of the edge, $K_{u'}^e(w,b)=1$, and for the source u of the edge, $K_u^e(w,b)=w_e b_e$.

Consider the case where $H=H_1\circ\cdots\circ H_k$. Assume that $e=(u,u')$ is an edge of $\mathbb{H}_i$. Since any path $P\in\mathcal{P}^H(u)$ is a union $P=P_i\cup\cdots\cup P_k$ for some $P_i\in\mathcal{P}^{H_i}(u)$ and $P_j\in\mathcal{P}^{H_j}$ for $i+1\le j\le k$, it follows that

$$K_u^H(w,b)=K_u^{H_i}(w,b)\prod_{j=i+1}^{k}K^{H_j}(w,b) \tag{C.3}$$

and similarly

$$K_{u'}^H(w,b)=K_{u'}^{H_i}(w,b)\prod_{j=i+1}^{k}K^{H_j}(w,b). \tag{C.4}$$

So

$$\begin{aligned}
\tilde{w}_e^H &= \tilde{w}_e^{H_i} && \text{by the definition of } \tilde{w}_e^H\\
&= \frac{w_e b_e K_{u'}^{H_i}(w,b)}{K_u^{H_i}(w,b)} && \text{by the induction hypothesis}\\
&= \frac{w_e b_e K_{u'}^{H}(w,b)}{K_u^{H}(w,b)} && \text{by (C.3) and (C.4),}
\end{aligned}$$

as required.

Finally consider the case where $H=H_1+\cdots+H_k$ and $e=(u,u')$ is an edge of $\mathbb{H}_i$. If u is not the source of $\mathbb{H}$, then since any path $P\in\mathcal{P}^H(u)$ is a path in $\mathcal{P}^{H_i}(u)$, it follows that $K_u^H(w,b)=K_u^{H_i}(w,b)$ and similarly $K_{u'}^H(w,b)=K_{u'}^{H_i}(w,b)$ (which holds when u is the source). So the same argument as above shows that the theorem holds. If u is the source of H, then

$$\begin{aligned}
\tilde{w}_e^H &= \frac{K_u^{H_i}(w,b)}{K_u^H(w,b)}\tilde{w}_e^{H_i} && \text{by the definition of } \tilde{w}_e^H\\
&= \frac{K_u^{H_i}(w,b)}{K_u^H(w,b)}\frac{w_e b_e K_{u'}^{H_i}(w,b)}{K_u^{H_i}(w,b)} && \text{by the induction hypothesis}\\
&= \frac{w_e b_e K_{u'}^{H_i}(w,b)}{K_u^{H}(w,b)}\\
&= \frac{w_e b_e K_{u'}^{H}(w,b)}{K_u^{H}(w,b)} && \text{since } K_{u'}^H(w,b)=K_{u'}^{H_i}(w,b),
\end{aligned}$$

which completes the proof. ∎

It is not hard to see that the weights $\tilde{w}$ can be calculated in time linear in the size of the regular expression. Note that a SP digraph is acyclic and so, as shown in Section 2, we already have a linear time implementation for the Weight Pushing algorithm. But the syntax tree based algorithm can be easily extended to a linear time algorithm for the general case when $*$-operations are allowed (see the next section). Moreover if the edge factors b_t are "sparse" in some sense, the algorithm can skip the redundant computation and run significantly faster. For details see Section C.3.

C.2 Allowing $*$-operations

Now we consider the general case where $*$-operation is allowed. That is, H_0 is now an arbitrary regular expression. Again the update rule for the edge weights w is computed through $\tilde{w}_e^H$. The definition of $\tilde{w}_e^H$ is the same as (C.2) when H is a concatenation, a union or a single edge. So here we can restrict ourselves to the case when $H = H_1^*$. Recall how the digraph $\mathbb{H}$ is defined by the $*$-operation is defined in Figure 9. We can assume that the edge factors b_e for ε-edges of that figure are one. The new weights for edges in $\mathbb{H}$ are given by:

$$\tilde{w}_e^H = \begin{cases} 1 & \text{if } e \text{ is labeled } \varepsilon_1 \text{ or } \varepsilon_2, \\ w_{\varepsilon_H} K^{H_1}(w,b) & \text{if } e \text{ is labeled } \varepsilon_H, \\ 1 - w_{\varepsilon_H} K^{H_1}(w,b) & \text{if } e \text{ is labeled } \varepsilon_3, \\ \tilde{w}_e^{H_1} & \text{otherwise.} \end{cases} \tag{C.5}$$

Note that here for some ε edges e may have $w_e \neq 1$.

We now show that adding the case $H = H_1^*$ with the above update to (C.2) simulates the Weight Pushing algorithm for any regular expression H.

Theorem 8 *For any regular expression H and any edge $e = (u,u')$ of $\mathbb{H}$, the update rules (C.2) and (C.5) establishes*

$$\tilde{w}_e^H = \frac{w_e b_e K_{u'}^H(w,b)}{K_u^H(w,b)}.$$

Proof It suffices to add to the proof of Theorem 7 the induction step for the $*$-operation. Assume that $H = H_1^*$ for some regular expression H_1 and the claim holds for H_1. The corresponding digraph is given in Figure 9.

First we notice that

$$K^H(w,b) = w_{\varepsilon_1} K_{u_0}^H(w,b) \tag{C.6}$$

and for any $u \notin \{u_0, \mathtt{s}\}$,

$$K_u^H(w,b) = K_u^{H_1}(w,b) K_{u_0}^H(w,b) \tag{C.7}$$

If $e = \varepsilon_1 = (\mathtt{s}, u_0)$, then (C.6) immediately assures the theorem:

$$\frac{w_e b_e K_{u_0}^H(w,b)}{K_{\mathtt{s}}^H(w,b)} = \frac{w_{\varepsilon_1} K_{u_0}^H(w,b)}{K^H(w,b)} = 1 = \tilde{w}_e^H.$$

If $e = \varepsilon_2$, then the theorem trivially holds. Next assume that $e = \varepsilon_H = (u_0, \mathtt{s}_1)$. It follows that

$$\begin{aligned} \tilde{w}_e^H &= w_e K^{H_1}(w,b) && \text{by the update rule (C.5) for } w_{\varepsilon_H} \\ &= w_e K_{\mathtt{s}_1}^{H_1}(w,b) \\ &= \frac{w_e b_e K_{\mathtt{s}_1}^H(w,b)}{K_{u_0}^H(w,b)} && \text{by (C.7) with } u = \mathtt{s}_1, \end{aligned}$$

as required. Note that we used the fact that all ε edges receive factor $b_e = 1$. It is trivial that the claim holds for $e = \varepsilon_3$ because the Weight Pushing algorithm guarantees that $\tilde{w}_{\varepsilon_3}^H = 1 - \tilde{w}_{\varepsilon_H}^H$.

Finally consider the case where $e = (u, u')$ is an edge of $\mathbb{H}_1$. In this case, since both u and u' are not in $\{\mathtt{s}, u_0\}$, we have

$$\begin{aligned}
\tilde{w}_e^H &= \tilde{w}_e^{H_1} && \text{by the update rule} \\
&= \frac{w_e b_e K_{u'}^{H_1}(w,b)}{K_u^{H_1}(w,b)} && \text{by the induction hypothesis} \\
&= \frac{w_e b_e K_{u'}^{H}(w,b)/K_{u_0}^{H}(w,b)}{K_u^{H}(w,b)/K_{u_0}^{H}(w,b)} && \text{by (C.7)} \\
&= \frac{w_e b_e K_{u'}^{H}(w,b)}{K_u^{H}(w,b)},
\end{aligned}$$

which completes the proof. ∎

C.3 Further Speedup of the Weight Pushing Algorithm

In this section we give another implementation for computing kernels and the weight pushing algorithm. It turns out that if inputs x and update factors b are "sparse" in the sense that most edges receive input $x_e = 1$ and factor $b_e = 1$, then the new algorithm runs significantly faster. For the loss update, $b_e = 1$ for all edges e with loss zero at this trial (see (5.5) and preceding discussion). For the EG update, $b_e = 1$ for all edges e for which $x_e = 0$ (see (5.9) and preceding discussion). This kind of sparseness may naturally happen in many applications. For example, in the dynamic routing problem discussed in Section 7, this corresponds to a reliable network where most edges accepts the packet with probability 1. Another example is an on-line pruning problem where only the edges along a single path are relevant, which is discussed in Section 10.

We now describe the speedup of this section in a precise form. Assume that a regular expression H_0 is given. We consider inputs x to be given to the leaves (edge symbols) of the syntax tree for H_0, rather than to the edges of the digraph $\mathbb{H}_0$. If a node H of the syntax tree has a leaf with label e in its descendants such that $x_e \neq 1$ (equivalently, if the component $\mathbb{H}$ contains an edge e with $x_e \neq 1$), then we say that the node H is *relevant with respect to* x. Let $V(x)$ denote the set of all union and star nodes (concatenation nodes and leaves are not counted) that are relevant with respect to x. In the new implementation, we no longer maintain edge weights w of the digraph $\mathbb{H}_0$ but maintain weights, denoted μ, for the edges of the syntax tree for H_0, so that μ implicitly represents the path weights W. That is, $W = \Psi(\mu)$, where Ψ is the new feature map described in the next subsection. So far the dot product is computed in terms of the kernel $K(w,x)$ by maintaining weights w such that $W = \Phi(w)$. But now the dot product is computed in terms of a function of μ and x which is defined as

$$\mathcal{K}(\mu, x) = \Psi(\mu) \cdot \Phi(x) = \sum_P W_P \prod_{e \in P} x_e.$$

We call this the *pseudo-kernel* and give an algorithm for computing $\mathcal{K}(\mu, x)$ in time linear in $|V(x)|$. Moreover, we give an algorithm that, when given edge factors b which are now assigned to the leaves of the syntax tree of H_0, updates weights μ so that the new weights $\tilde{\mu}$ represents the updated path weights: $\tilde{W} = \Psi(\tilde{\mu})$. The update algorithm runs in time linear in $\sum_{H \in V(b)} \deg(H)$, where $\deg(H)$ is the degree (number of children) of node H in the syntax tree. Note that this sum is not always linear in $|V(b)|$.

```
Traverse(H)
{
    if H = e, then P = {e};
    else if H = H_1 + ··· + H_k, then
        choose H_i with probability μ_i^H;
        P = Traverse(H_i);
    else if H = H_1 ∘ ··· ∘ H_k, then
        P = Traverse(H_1) ∪ ··· ∪ Traverse(H_k);
    else if H = H_1^*, then
        P = ∅;
        repeat
            let c = { 1  with probability μ^H,
                    { 0  with probability 1 − μ^H;
            if c = 1 then P = P ∪ Traverse(H_1);
        until c = 0;
    return P;
}
```

Figure 14: Algorithm Traverse: return a path $P \in \mathcal{P}^H$ with probability W_P^H.

C.3.1 FEATURE MAP Ψ

Here we consider a path as a sequence of edge symbols that the regular expression H_0 produces. In other words paths do not contain ε-edge symbols. For an internal node H of the syntax tree, let $\mathcal{P}^H$ now denote the language (words over the edge symbols except for epsilon edges) that H produces. In the following we assume that W is a probability vector on the paths in $\mathcal{P}^{H_0}$.

First we show how the edge weights μ for the syntax tree (rooted at H_0) implicitly represent the path weights W. Specifically, for each union node $H = H_1 + \cdots + H_k$ of the syntax tree, we maintain weights $\mu_i^H \in [0,1]$ for each edge (H, H_i) so that $\sum_{i=1}^k \mu_i^H = 1$, and for each star node $H = H_1^*$, we maintain $\mu^H \in [0,1)$. The weights μ implicitly specify the path weights W in the way described below.

Consider the following stochastic process for traversing the syntax tree. From the weights μ, the process produces a random path $P \in \mathcal{P}^{H_0}$. That is, it defines a probabilistic map Ψ from μ to the set of probability vectors on $\mathcal{P}^{H_0}$: Start from the root; if we are at a concatenation node, then we go to all of its children; if we are at a union node H, then choose a child node H_i with probability μ_i^H and go to H_i; If we are at a star node $H = H_1^*$, then repeat the following: with probability μ^H traverse H_1 and with probability $1-\mu^H$ exit the repeat statement. Finally all the leaves we visit form a path P, and the weight W_P is defined as the probability that the path P is chosen by this process. For more detail, see algorithm Traverse in Figure 14. It is easy to see that Traverse(H) returns a path P in $\mathcal{P}^H$. Let W_P^H denote the probability that Traverse(H) returns P. In particular, let $W_P = W_P^{H_0}$ for the given regular expression H_0. The construction immediately shows that the weight W_P^H of a path $P \in \mathcal{P}^H$

can be computed recursively as follows.

$$W_P^H = \begin{cases} 1 & \text{if } H = e \text{ (thus } P = \{e\}), \\ \mu_i^H W_P^{H_i} & \text{if } H = H_1 + \cdots + H_k \text{ and } P \in \mathcal{P}^{H_i}, \\ \prod_{i=1}^k W_{P_i}^{H_i} & \text{if } H = H_1 \circ \cdots \circ H_k \text{ and } P = P_1 \cup \cdots \cup P_k \text{ with} \\ & P_i \in \mathcal{P}^{H_i} \text{ for } 1 \le i \le k, \\ \prod_{i=1}^k \left(\mu^H W_{P_i}^{H_1}\right)(1-\mu^H) & \text{if } H = H_1^* \text{ and } P = P_1 \cup \cdots \cup P_k \text{ with } P_i \in \mathcal{P}^{H_1} \\ & \text{for } 1 \le i \le k \text{ for some } k \ge 0. \end{cases} \tag{C.8}$$

Obviously the weights W^H are probabilistic, that is,

$$\sum_{P \in \mathcal{P}^H} W_P^H = 1.$$

Actually, it can by shown that the range of the mapping $\Psi(.)$ consists of all probability vectors W that fulfill Properties P1–3 (which is also the range of $\Phi(.)$). Also $\Psi(.)$ is injective as well (For the mapping $\Phi(.)$, this was shown in Appendix B).

C.3.2 COMPUTING THE PSEUDO-KERNEL $\mathcal{K}(\mu,x)$

In this section we give an algorithm for computing the pseudo-kernel $\mathcal{K}$. As in the case of the kernel K, we define the pseudo-kernel associated with each node H as

$$\mathcal{K}^H(\mu,x) = \sum_{P \in \mathcal{P}^H} W_P^H \prod_{e \in P} x_e \tag{C.9}$$

and compute $\mathcal{K}(\mu,x) = \mathcal{K}^{H_0}(\mu,x)$ recursively. The recursion is given as follows.

$$\mathcal{K}^H(\mu,x) = \begin{cases} x_e & \text{if } H = e, \\ \sum_{i=1}^k \mu_i^H \mathcal{K}^{H_i}(\mu,x) & \text{if } H = H_1 + \cdots + H_k, \\ \prod_{i=1}^k \mathcal{K}^{H_i}(\mu,x) & \text{if } H = H_1 \circ \cdots \circ H_k, \\ \dfrac{1-\mu^H}{1-\mu^H \mathcal{K}^{H_1}(\mu,x)} & \text{if } H = H_1^*. \end{cases} \tag{C.10}$$

Lemma 9 *The function $\mathcal{K}^H(\mu,x)$ defined recursively (C.10) is the pseudo-kernel defined in (C.9).*

Proof We show the lemma by the induction on the depth of the syntax tree. For the base case where $H = e$ for some edge symbol e, the lemma clearly holds. For the case where $H = H_1 + \cdots + H_k$,

$$\begin{aligned} \mathcal{K}^H(\mu,x) &= \sum_{i=1}^k \mu_i^H \mathcal{K}^{H_i}(\mu,x) \\ &= \sum_{i=1}^k \mu_i^H \sum_{P \in \mathcal{P}^{H_i}} W_P^{H_i} \prod_{e \in P} x_e \quad \text{by the induction hypothesis} \\ &= \sum_{i=1}^k \sum_{P \in \mathcal{P}^{H_i}} W_P^H \prod_{e \in P} x_e \quad \text{by (C.8)} \\ &= \sum_{P \in \mathcal{P}^H} W_P^H \prod_{e \in P} x_e. \end{aligned}$$

For the case where $H = H_1 \circ \cdots \circ H_k$,

$$\begin{aligned}
\mathcal{K}^H(\mu,x) &= \prod_{i=1}^k \mathcal{K}^{H_i}(\mu,x) \\
&= \prod_{i=1}^k \sum_{P\in\mathcal{P}^{H_i}} W_P^{H_i} \prod_{e\in P} x_e && \text{by the induction hypothesis} \\
&= \sum_{P_1\in\mathcal{P}^{H_1}} \cdots \sum_{P_k\in\mathcal{P}^{H_k}} \prod_{i=1}^k \left(W_{P_i}^{H_i} \prod_{e\in P_i} x_e \right) \\
&= \sum_{P\in\mathcal{P}^H} W_P^H \prod_{e\in P} x_e && \text{by (C.8).}
\end{aligned}$$

Finally consider the case where $H = H_1^*$. In this case, by using the formula $1/(1-a) = a^0 + a^1 + a^2 + \cdots$, we have

$$\begin{aligned}
\mathcal{K}^H(\mu,x) &= \frac{1-\mu^H}{1-\mu^H \mathcal{K}^{H_1}(\mu,x)} \\
&= (1-\mu^H) \sum_{k=0}^{\infty} \left(\mu^H \mathcal{K}^{H_1}(\mu,x)\right)^k \\
&= (1-\mu^H) \sum_{k=0}^{\infty} \left(\mu^H \sum_{P\in\mathcal{P}^{H_1}} W_P^{H_1} \prod_{e\in P} x_e \right)^k && \text{by the induction hypothesis} \\
&= (1-\mu^H) \sum_{k=0}^{\infty} \sum_{P_1\in\mathcal{P}^{H_1}} \cdots \sum_{P_k\in\mathcal{P}^{H_1}} \prod_{i=1}^k \left(\mu^H W_{P_i}^{H_1} \prod_{e\in P_i} x_e \right) \\
&= \sum_{k=0}^{\infty} \sum_{P\in\left(\mathcal{P}^{H_1}\right)^k} W_P^H \prod_{e\in P} x_e && \text{by (C.8)} \\
&= \sum_{P\in\mathcal{P}^H} W_P^H \prod_{e\in P} x_e
\end{aligned}$$

which completes the proof. ■

If H is irrelevant (that is, $x_e = 1$ for all leaves e of H), then $\mathcal{K}^H(\mu,x) = \mathcal{K}^H(\mu,1) = \sum_{P\in\mathcal{P}^H} W_P^H = 1$. Therefore, the recursions (C.10) can be computed by traversing the relevant nodes only. More precisely, if $H = H_1 \circ \cdots \circ H_k$, then

$$\mathcal{K}^H(\mu,x) = \prod_{H_i:\ \text{relevant}} \mathcal{K}^{H_i}(\mu,x)$$

and if $H = H_1 + \cdots + H_k$, then

$$\begin{aligned}
\mathcal{K}^H(\mu,x) &= \sum_{i=1}^k \mu_i^H \mathcal{K}^{H_i}(\mu,x) \\
&= \sum_{H_i:\ \text{relevant}} \mu_i^H \mathcal{K}^{H_i}(\mu,x) + \sum_{H_i:\ \text{irrelevant}} \mu_i^H \\
&= 1 - \sum_{H_i:\ \text{relevant}} \mu_i^H \left(1 - \mathcal{K}^{H_i}(\mu,x)\right).
\end{aligned}$$

Clearly, we get $\mathcal{K}^H(\mu,x)$ for all relevant nodes H in time linear in the number of relevant nodes.

C.3.3 UPDATE RULE FOR μ

Next we consider how to update μ. Assume that edge factors b are given. For union nodes H, the new weights $\tilde{\mu}_i^H$ are simply calculated by

$$\tilde{\mu}_i^H = \frac{\mu_i^H \mathcal{K}^{H_i}(\mu,b)}{\mathcal{K}^H(\mu,b)}. \tag{C.11}$$

For star nodes H, the new weights $\tilde{\mu}^H$ are given by

$$\tilde{\mu}^H = \mu^H \mathcal{K}^{H_1}(\mu,b). \tag{C.12}$$

Recall that if H is irrelevant, then $K^H(\mu,b) = K^H(\mu,1) = 1$. Moreover if H is irrelevant, then its children H_i are also irrelevant. So for any union node $H \notin V(b)$, $\tilde{\mu}_i^H = \mu_i^H$ for all children H_i of H, and for any star node $H \notin V(b)$, $\tilde{\mu}^H = \mu^H$. This implies that we do not need to update weights for all irrelevant node $H \notin V(b)$. Note that if $H = H_1 + \cdots + H_k$ is a relevant union node, then we have to calculate new weights $\tilde{\mu}_i^H$ for all $1 \leq i \leq k$ even if only one child is relevant. This is why the updating takes time $O\left(\sum_{H\in V(b)} \deg(H)\right)$.

Now we show that this update rule simulates the multiplicative update (5.3) for path weights.

Lemma 10 *Assume that edge factors b are given. Let $\tilde{\mu}$ be the new weights obtained by the rule (C.11) and (C.12). Then the path weights $\tilde{W} = \Psi(\tilde{\mu})$ satisfy*

$$\tilde{W}_P^H = \frac{W_P^H \prod_{e\in P} b_e}{\sum_{P\in\mathcal{P}^H} W_P^H \prod_{e\in P} b_e} = \frac{W_P^H \prod_{e\in P} b_e}{\mathcal{K}^H(\mu,b)}$$

for any node H of the syntax tree and any path $P \in \mathcal{P}^H$.

Proof Let $\tilde{W}^H$ be defined recursively as in (C.8) with μ being replaced by $\tilde{\mu}$. Note that by definition $\tilde{W}^{H_0} = \tilde{W} = \Psi(\tilde{\mu})$. We show the theorem by an induction on the depth of the syntax tree.

For the base case where $H = e$, since $\mathcal{P}^H$ consists of a single edge e, it follows that

$$\tilde{W}_P^H = 1 = \frac{W_P^H \prod_{e\in P} b_e}{\sum_{P\in\mathcal{P}^H} W_P^H \prod_{e\in P} b_e}$$

for any $P \in \mathcal{P}^H$.

Consider the case where $H = H_1 + \cdots + H_k$ and $P \in \mathcal{P}^{H_i}$. Then it follows that

$$\begin{aligned}
\tilde{W}_P^H &= \tilde{\mu}_i^H \tilde{W}_P^{H_i} && \text{by (C.8)} \\
&= \frac{\mu_i^H \tilde{W}_P^{H_i} \mathcal{K}^{H_i}(\mu,b)}{\mathcal{K}^H(\mu,b)} && \text{by (C.11)} \\
&= \frac{\mu_i^H \mathcal{K}^{H_i}(\mu,b)}{\mathcal{K}^H(\mu,b)} \frac{W_P^{H_i} \prod_{e\in P} b_e}{\mathcal{K}^{H_i}(\mu,b)} && \text{by the induction hypothesis} \\
&= \frac{W_P^H \prod_{e\in P} b_e}{\mathcal{K}^H(\mu,b)} && \text{by (C.8).}
\end{aligned}$$

For the case where $H = H_1 \circ \cdots \circ H_k$ and $P = P_1 \cup \cdots \cup P_k$ with $P_i \in \mathcal{P}^{H_i}$ for $1 \le i \le k$,

$$\begin{aligned}
\tilde{W}_P^H &= \prod_{i=1}^k \tilde{W}_{P_i}^{H_i} && \text{by (C.8)} \\
&= \prod_{i=1}^k \frac{W_{P_i}^{H_i} \prod_{e \in P_i} b_e}{\mathcal{K}^{H_i}(\mu, b)} && \text{by the induction hypothesis} \\
&= \frac{W_P^H \prod_{e \in P} b_e}{\mathcal{K}^H(\mu, b)} && \text{by (C.8) and (C.10).}
\end{aligned}$$

Finally consider the case where $H = H_1^*$ and $P = P_1 \cup \cdots P_k$ with $P_i \in \mathcal{P}^{H_1}$ for $1 \le i \le k$.

$$\begin{aligned}
\tilde{W}_P^H &= (1 - \tilde{\mu}^H) \prod_{i=1}^k \left(\tilde{\mu}^H \tilde{W}_{P_i}^{H_1} \right) && \text{by (C.8)} \\
&= \left(1 - \mu^H \mathcal{K}^{H_1}(\mu, b)\right) \prod_{i=1}^k \frac{\mu^H \mathcal{K}^{H_1}(\mu, b) W_{P_i}^{H_1} \prod_{e \in P_i} b_e}{\mathcal{K}^{H_1}(\mu, b)} && \text{by (C.12) and the induction hypothesis} \\
&= \frac{(1 - \mu^H) \prod_{i=1}^k \left(\mu^H W_{P_i}^{H_1} \prod_{e \in P_i} b_e \right)}{\mathcal{K}^H(\mu, b)} && \text{by (C.10)} \\
&= \frac{W_P^H \prod_{e \in P} b_e}{\mathcal{K}^H(\mu, b)} && \text{by (C.8).}
\end{aligned}$$

■

References

W. Aiello, R. Ostrovsky, E. Kushilevitz, and A. Rosén. Dynamic routing on networks with fixed-size buffers. In *14th ACM-SIAM Symposium on Discrete Algorithms*, pages 771–780, 2003.

B. Awerbuch, Y. Azar, S. A. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. *Journal of Computer and System Sciences*, 62(3):385–397, 2001.

T. Bylander. The binary exponentiated gradient algorithm for learning linear functions. In *Proceedings of the Tenth Annual Conference on Computational Learning Theory*, pages 184–192. ACM Press, New York, NY, 1997.

C. Cortes, P. Haffner, and M. Mohri. Rational kernels. In *Advances in Neural Information Processing Systems*, volume 15. MIT Press, London, 2002.

N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Methods*. Cambridge University Press, Cambridge, England, 2000.

N. Cristianini, C. Campbell, and J. Shawe-Taylor. A multiplicative updating algorithm for training support vector machine. In *6th European Symposium on Artificial Neural Networks (ESANN)*, pages 189–194, 1999.

A. DeSantis, G. Markowsky, and M. N. Wegman. Learning probabilistic prediction functions. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 110–119. IEEE Computer Society Press, Los Alamitos, CA, 1988.

Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

C. Gentile and N. Littlestone. The robustness of the p-norm algorithms. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, pages 1–11. ACM Press, New York, 1999.

C. Gentile and M. K. Warmuth. Hinge loss and average margin. In *Advances in Neural Information Processing Systems 11*, pages 225–231, MIT Press, London, UK, 1998.

D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, Univ. of Calif. Computer Research Lab, Santa Cruz, CA, 1999.

D. P. Helmbold, S. Panizza, and M. K. Warmuth. Direct and indirect algorithms for on-line learning of disjunctions. *Theoretical Computer Science*, 284(1):109–142, July 2002.

D. P. Helmbold and R. E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Machine Learning*, 27(01):51–68, 1997.

R. Khardon, D. Roth, and R. Servedio. Efficiency versus convergence of Boolean kernels for on-line learning algorithms. In *Advances in Neural Information Processing Systems 14*, pages 423–430. MIT Press, London, UK, 2001.

J. Kivinen and M. K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. *Information and Computation*, 132(1):1–64, January 1997.

J. Kivinen, M. K. Warmuth, and P. Auer. The perceptron algorithm vs. winnow: linear vs. logarithmic mistake bounds when few input variables are relevant. *Artificial Intelligence*, 97:325–343, December 1997.

J. Kivinen and M. K. Warmuth. Averaging expert predictions. In *Proc. 4th European Conference on Computational Learning Theory*, volume 1572 of *Lecture Notes in Artificial Intelligence*, pages 153–167. Springer-Verlag, 1999.

S. Leonardi. On-line network routing. In *Online Algorithms - LNCS 1442*, pages 242–267, 1998.

N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

W. Maass and M. K. Warmuth. Efficient learning with virtual threshold gates. *Information and Computation*, 141(1):66–83, February 1998.

M. Mohri. General algebraic frameworks and algorithms for shortest distance problems. Technical Report 981219-10TM, AT&T Labs-Research, 1998.

F. Shu, L. K. Saul, and D. D. Lee. Multiplicative updates for large margin classifiers. In *16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003*, pages 188–202. Springer-Verlag, 2003.

Y. Singer and M. K. Warmuth. Training algorithms for hidden Markov models using entropy based distance functions. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9.*, pages 641–647. MIT Press, London, UK, 1997.

E. Takimoto, A. Maruoka, and V. Vovk. Predicting nearly as well as the best pruning of a decision tree through dynamic programming scheme. *Theoretical Computer Science*, 261(1):179–209, 2001.

E. Takimoto and M. K. Warmuth. Predicting nearly as well as the best pruning of a planar decision graph. *Theoretical Computer Science*, 288(2):217–235, 2002.

J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982.

V. Vovk. Aggregating strategies. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 371–383. Morgan Kaufmann, 1990.

C. Watkins. Dynamic alignment kernels. Technical Report CSD-TR-98-11, Royal Holloway, University of London, 1999.