Robust Web Extraction : An Approach Based on a Probabilistic Tree-Edit Model

Nilesh Dalvi
Yahoo! Research
Santa Clara, CA, USA
ndalvi@yahoo-inc.com

Philip Bohannon
Yahoo! Research
Santa Clara, CA, USA
plb@yahoo-inc.com

Fei Sha
U. of Southern California
Los Angeles, CA, USA
feisha@usc.edu

ABSTRACT

On script-generated web sites, many documents share common HTML tree structure, allowing *wrappers* to effectively extract information of interest. Of course, the scripts and thus the tree structure evolve over time, causing wrappers to break repeatedly, and resulting in a high cost of maintaining wrappers. In this paper, we explore a novel approach: we use temporal snapshots of web pages to develop a tree-edit model of HTML, and use this model to improve wrapper construction. We view the changes to the tree structure as suppositions of a series of edit operations: deleting nodes, inserting nodes and substituting labels of nodes. The tree structures evolve by choosing these edit operations stochastically.

Our model is attractive in that the probability that a source tree has evolved into a target tree can be estimated *efficiently*—in quadratic time in the size of the trees—making it a potentially useful tool for a variety of tree-evolution problems. We give an algorithm to learn the probabilistic model from training examples consisting of pairs of trees, and apply this algorithm to collections of web-page snapshots to derive HTML-specific tree edit models. Finally, we describe a novel wrapper-construction framework that takes the tree-edit model into account, and compare the quality of resulting wrappers to that of traditional wrappers on synthetic and real HTML document examples.

1. INTRODUCTION

On the Web, many sites use scripts to generate highly structured HTML, e.g. academic repositories, product catalogs and entertainment sites. In many cases, the structure can be represented as an XML document tree. Figure 1 displays an example script-generated document tree. This tree is simplified from the actual page representing the movie, "The Godfather," on a popular movie website. Since this HTML page is script-generated, other movie pages on the same site are likely to have a very similar tree-structure, making this structure a powerful feature to exploit for information extraction. For instance, to extract the director name from this page, we can use the following XPath expression,

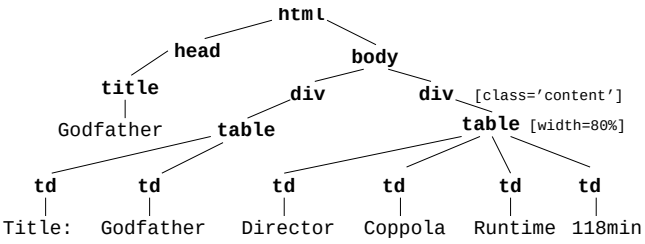

Figure 1: An HTML Webpage

$$W_1 \equiv \texttt{/html/body/div[2]/table/td[2]/text()}$$

which is a specification on how to traverse trees. In particular, the XPath expression W_1 above starts from the root, follows the tags `html`, `body`, the second `div` under `body`, `table` under it, the second `td` under `table` and then the `text()` under it.

In fact, the path expression W_1 is one form of a simple *wrapper*, and the problem of *inducing* wrappers from labeled examples has been extensively studied [18, 22, 13, 10, 12, 29, 4, 23, 2]. A key quality of a wrapper is its *effectiveness* with respect to the corpus of pages of the same type generated from the same script. For example, not only will W_1 work on this one page, it may work equally well on some or all of the remaining movie pages on the same site. The popularity of wrapper-based extraction can be attributed to the small number of training examples needed to induce an effective wrapper.

Wrapper Robustness Problem While wrapper induction works well for snapshots of web sites, in production use it suffers from a fundamental problem: the underlying webpages frequently change, even very slightly, but causing the wrapper to break and requiring them to be re-learned [23, 1, 17, 16]. For instance, consider the wrapper W_1 above. Even if W_1 were perfectly effective on the *current* site, it will break if the structure of the underlying webpages changes in any of the following ways: the first `div` is deleted or merged with the second `div`, a new `table` or `tr` is added under the second *div*, the order of *Director* and *Runtime* is changed, a new font element is added, and so on.

Although there are techniques [21, 8, 19] that detect wrap-

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

per breakage by learning *content models* for extracted data, repairing the wrappers often requires manual re-labeling, keeping the cost of wrapper maintenance high. Ideally, we desire wrappers that, in addition to being highly effective on the current site, would be as *robust* as possible to future changes in the site.

Fortunately, there are often several wrappers that can extract the same information. For example, each of the following XPath expressions extracts the director names and can be used as an alternative to W_1 (Note that `psib` is an abbreviation for the XPath axis `preceding-sibling`):

$$W_2 \equiv \texttt{//div[@class='content']/*/td[2]/text()}$$
$$W_3 \equiv \texttt{//table[@width='80\%'/td[2]/text()}$$
$$W_4 \equiv \texttt{//text()[psib::*[1] [text()='Director']]}$$

Intuitively, these patterns may be preferable to W_1, since they have a lesser dependence on the tree structure. For example, if the site script occasionally places `div` before the director to announce an award, as shown in Figure 2, then W_1 will not work, while W_3 and W_4 will and W_2 might. As a human, we have some notion of robustness in mind: even if W_1 were perfectly effective for a test set of pages, it would still seem risky to deploy it, and we might reason that the width of the table on which W_3 depends is less stable than the word `'Director'` (or vice versa), and thus W_4 is apparently more robust.

In general, given a set of alternative wrappers for an extraction task, it is not clear how to pick the most *robust* wrapper. In this paper, we address exactly this problem by introducing a *principled framework* for inducing robust wrappers. In fact, even though the importance of robust wrappers has been pointed out repeatedly in empirical studies [23, 1, 17, 16], we are not aware of any formal definitions of robust wrappers in prior work, nor algorithms for inducing them.

Formalizing Robustness: A Probabilistic Tree-Edit Model

The heart of our framework is a probabilistic *tree change model* that captures how webpages change over time. Given (HTML) trees S and T, the model gives the probability $\mathbf{P}(T \mid S)$ of S evolving to T in the future. It assigns a probability to individual tree edit operations like changing a `<b>` tag to an `<i>` tag, changing the *width* of a `<table>`, inserting a new `div` and deleting a `<br>`. It models tree evolution as a stochastic process where the individual edit operations are drawn according to their probability. We give efficient algorithms to compute $\mathbf{P}(T \mid S)$ given trees S and T. In addition, we develop an effective learning technique based on gradient ascent to learn the change model using historic data.

The probabilistic tree-edit model gives us a principled way to construct robust wrappers. We define the robustness of a wrapper w on a webpage S as the probability that w works on a webpage T where T is drawn randomly with probability $\mathbf{P}(T \mid S)$. This gives us a method to choose the most robust wrapper from the set of alternative wrappers.

While the probabilistic tree-edit model is a key component of our framework, the model is of independent interest. A lot of research has been carried out in the last decade [28, 26, 20] to develop learnable probabilistic edit models for strings, but the corresponding problem of modeling tree-structured data has remained under-addressed. [7] outlines several potential applications of such a tree-edit model that include studying evolution of phylogenetic trees, image recognition and music recognition. While previous work [5, 7] has tried to extend the string-edit model to apply to a tree-edit scenario, the resulting model is not properly normalized and thus does not yield a valid probabilistic distribution (cf. Section 8 for details).

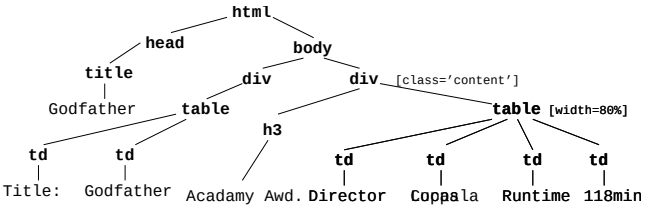

Figure 2: Second Snapshot of Webpage

Contributions In this paper, we make five fundamental contributions:

1. We formalize the previously heuristic notion of *wrapper robustness* and introduce a framework for constructing robust wrappers given a model of HTML volatility.

2. We develop a novel probabilistic tree edit model for modeling tree changes, which is of independent interest. We give efficient techniques to learn the model from sets of tree-pairs.

3. We propose the use of a probabilistic tree-edit model trained on archival web data as a principled and flexible source of information about HTML volatility.

4. We analyze the complexity of evaluating robustness of XPaths as wrappers.

5. We define a family of *minimal* wrappers suitable for robust extraction, and provide an algorithm to efficiently generate all minimal, effective XPath wrappers.

Finally, we demonstrate the effectiveness of incorporating a tree-edit based wrapper robustness model on wrappers of real web pages. For example, the robustness of a wrapper on *IMDB* as constructed by [2] is 40%, while the one chosen by our framework is 86% (a wrapper heuristically selected for robustness achieved 73%). At the average rate of script-change of two or three per year the expected lifetime of the robust wrapper beats that of the default wrapper by more than a year.

Outline In Section 2 we introduce our robust wrapper framework. We describe our tree edit model and parameter estimation in Sections 3 and 4, respectively. We discuss robust wrapper induction in Section 5, evaluating robustness of wrappers in Section 6 and present our experimental study in Section 7, discuss related work in Section 8 and conclude in Section 9.

2. ROBUST EXTRACTION FRAMEWORK

In this section, we give an overview of our robust extraction framework. The framework is depicted in Figure 3. We explain the main components here and describe them in details in the following sections.

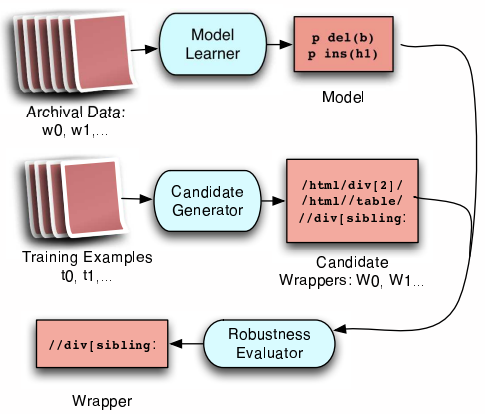

Figure 3: Robust Extraction Framework

Archival Data A webpage w, say the *IMDB* page for the film *Godfather*, undergoes various changes across its lifetime. Let $w^{(0)}, w^{(1)}, \cdots$ denote the various versions of w. The archival data is a collection of pairs $\{(w^{(t)}, w^{(t+1)})\}$ for various webpages w. Figure 2 shows a possible second snapshot of the page shown in Figure 1. Archival data can be obtained by monitoring a set of webpages over time. In this paper, for obtaining archival data, we use Internet Archive [3], a public website that regularly takes snapshots of the Web.

Model The salient component of our framework is a probabilistic model that gives a probability distribution over the possible next states of a webpage given its current snapshot, i.e. it gives the probability $\mathbf{P}(w^{(t+1)} = Y \mid w^{(t)} = X)$, which we more concisely write as $\mathbf{P}(Y \mid X)$. Note that, unlike some probabilistic models for string mutations [28] which are which are joint models of both X and Y, our model is a conditional model. Note that, to estimate the parameter of the joint model $P(X, Y)$, a learning criteria based on maximum likelihood estimation needs to learn both the conditional model $P(Y|X)$ and the marginal distribution of $P(X)$. When it is difficult to characterize $P(X)$ with a suitable model, maximum likelihood estimation does not always yield the best performance if one is mainly interested in answering questions such as: which is the most likely Y that is changed from X? In contrast, our approach focuses more directly on the conditional distribution $P(Y|X)$ and therefore, gives rise to models that are relevant to those questions. The difference of modeling and parameter estimation between $P(X, Y)$ and $P(Y|X)$ has also been extensively studied in the machine learning community [24], where they have shown that $P(Y|X)$ leads to lower asymptotic error rate in classification tasks.

The model is specified by a set of parameters, each defining the probability of an atomic edit operation, e.g. inserting a `<b>` tag, deleting an `<hr>`, and changing an `<h2>` to `<h3>`. We define the model formally in Section 3

Model Learner The model learner takes as input the archival data and learns a model that best fits the data. Recall that a model is specified by a set, θ, of parameters. The model learner learns the parameter values that maximize the probability of observing the archival data, i.e.

$$\arg\max_{\theta} \prod_{(X,Y) \in ArchivalData} \mathbf{P}_\theta(Y \mid X)$$

Training Data An extraction task is to extract some field of interest, say *film director*, from a set of structurally similar pages, say film pages from *IMDB* website. Training data for an extraction task consists of a small subset of the set of pages of interest along with annotations that specify the value of the field to be extracted.

Candidate Generator The candidate wrappers generator takes labeled training data over a corpus and generates a set of alternate wrappers. There has been extensive research [2, 23, 18, 22, 13, 10, 12, 29, 4] on learning wrappers from labeled data, with some focusing specifically on learning XPath rules [2, 23]. Any of the techniques can be used as part of candidate generation. In Section 5, we introduce a class of "minimal" XPath wrappers that are particularly suitable as candidates for robust wrappers.

Robustness Evaluator Finally, the robustness evaluator takes the set of candidate wrappers, evaluates the *robustness* of each using the probabilistic model learned on the archival data, and chooses the most robust wrapper. We define the robustness of a wrapper as the probability that it will continue to work on a future snapshot of the webpage. More formally, let W be a wrapper, X be a webpage and $\mathbf{P}$ be the probability model given by the model learner. Let $X \models W$ denote the fact that W succeeds on X. Then, the robustness of a wrapper on X is defined as

$$Rob_X(W) = \sum_{Y | Y \models W} \mathbf{P}(Y \mid X)$$

The wrapper that has the highest robustness is chosen among the set of candidate wrappers as the desired wrapper.

In Sections 3, 4, 5 and 6, we give details on each of these components.

3. CHANGE MODEL

Webpages can be viewed as *XML documents*, which we define next. Let $\mathcal{N}$ be the set of *nodes* which is infinite so as to accommodate arbitrarily large documents. Let Σ denote a finite set of *labels*. An XML document T is an ordered tree with nodes from $\mathcal{N}$ along with a function $\mathcal{L}$ that maps nodes of T to elements from Σ. Given a webpage, since we are primarily interested in structural changes, we replace all HTML text nodes with nodes having special label "TEXT".

A forest F is a finite ordered list of trees. Let $[u]$ denote the subtree of F rooted at u and let $\lfloor u \rfloor$ denote the forest consisting of the children sub-trees of u, i.e. $\lfloor u \rfloor$ is the forest obtained from $[u]$ by deleting u.

We define our change model in terms of a *conditional transducer*. A conditional transducer takes a tree and performs random edits on it to generate a new tree. It defines a conditional distribution that, given a tree, gives a probability distribution on the set of all possible trees as the likely next state of the tree.

While we are interested in XML tree structures, we find that it is more convenient to describe the process of tree changes in the more general term of forests. We will define an operator π given by a random process, which we call a *conditional transducer*, that takes a forest F and produces a new forest G. The transducer makes one pass over the

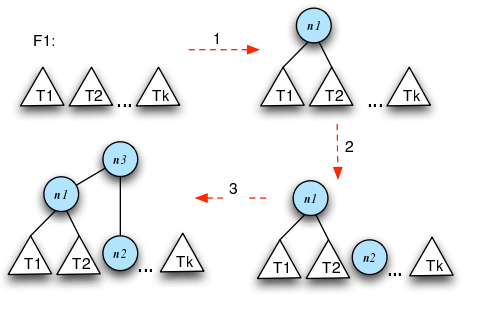

Figure 4: Example Insert Scenario

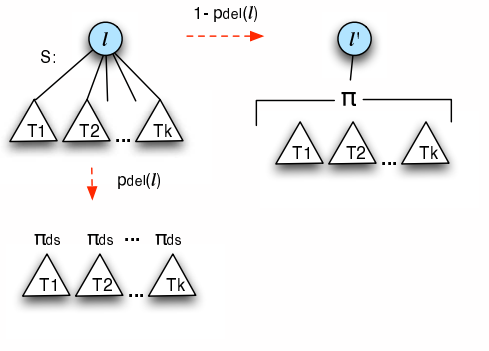

Figure 5: Action of π_{ds}

whole forest and at each position, it randomly decides to either delete a node, change its label, or insert new nodes.

The process π is defined recursively using two subprocesses π_{ins} and π_{ds} as follows.

Let $T_1, T_2, \cdots, T_K$ be the trees in F. Then

$$\pi(F) = \pi_{ins}(\pi_{ds}(T_1) \cdots \pi_{ds}(T_k)) \tag{1}$$

We define π_{ins} and π_{ds} next. The process π_{ins} takes a forest and performs a set of random inserts at the root level. It is defined by a parameter p_{stop} and a function $p_{ins} : \Sigma \mapsto [0, 1]$ such that $\sum_{l \in \Sigma} p_{ins}(l) = 1$. It is defined recursively as follows:

$$\pi_{ins}(U) \quad = \quad \begin{cases} U & \text{with probability } p_{stop} \\ \pi_{ins}(e_1(U)) & \text{otherwise} \end{cases}$$

where $e_1(U)$ is an insert operation that adds a node at the top of U chosen randomly from all such operations. Specifically, $e_1(U)$ first chooses randomly a label $l \in \Sigma$ with probability $p_{ins}(l)$ and creates a new node. Then it chooses uniformly a random subsequence of forests as children.

EXAMPLE 1. *Figure 4 shows a possible run of π_{ins} on forest F_1. In the first step, a new node n_1 is created and T_1, T_2 is the random subsequence of $T_1 \ldots T_k$ selected to be n_1's children. In step 2, n_2 is created and the empty subsequence between trees T_2 and T_3 is selected to be its children. Finally, in step 3, node n_3 is inserted, and the subsequence $n_1 \ldots n_2$ is selected for its children. (The labels of the new nodes are not shown, but are also randomly selected.)*

The operator π_{ds} takes a *tree* S and transforms it into a forest. It either deletes the root of the tree or changes its label and then recursively transforms the subtrees of the tree S. It is defined by two functions, $p_{del} : \Sigma \mapsto [0, 1]$ and $p_{sub} : \Sigma \times \Sigma \mapsto [0, 1]$ such that $\sum_l p_{sub}(l, l') = 1$ for all $l \in \Sigma$. Given a tree S, with l as the label of its root and $T_1, \cdots, T_k$ being the trees in $\lfloor S \rfloor$, define

$$\pi_{ds}(S) \quad = \quad \begin{cases} \pi_{ds}(T_1) \cdots \pi_{ds}(T_k) & \text{with probability } p_{del}(l) \\ e_2(\pi(\lfloor S \rfloor)) & \text{otherwise} \end{cases}$$

where $e_2(U)$ is an insertion operation that creates a new root node whose children are all trees returned by $\pi(\lfloor s \rfloor)$. The label l_{new} of the new root is chosen randomly with probability $p_{sub}(l, l_{new})$. These alternatives are illustrated in Figure 5.

To summarize, the generative process π is characterized by following parameters

$$\theta = (p_{stop}, \{p_{del}(l)\}, \{p_{ins}(l)\}, \{p_{sub}(l_1, l_2))\}$$

for $l, l_1, l_2 \in \Sigma$ along with the following conditions:

$$\begin{array}{ccc} 0 < & p_{stop} & < 1 \\ 0 \leq & p_{del}(l) & \leq 1 \\ p_{ins}(l) \geq 0, & & \sum_l p_{ins}(l) = 1 \\ p_{sub}(l_1, l_2) \geq 0, & & \sum_{l_2} p_{sub}(l_1, l_2) = 1 \end{array} \tag{2}$$

Given a set of parameters θ defining the transducer π, let $\mathbf{P}_\theta(Y \mid X)$ denote the probability that the process π applied to forest F stops and results in forest G. It is easy to show that:

THEOREM 1. *If θ satisfies all the conditions in Eq. (2), then $\mathbf{P}_\theta(\ \mid X)$ is a probability distribution on the set of all forests, i.e. $\sum_Y \mathbf{P}_\theta(Y \mid X) = 1$.*

4. MODEL LEARNER

We want to use archival data to learn a probabilistic model of change. Recall that archival data is a collection of pairs $\{(S, T)\}$, where S and T denote the old and new versions of a webpage at some point of time. Also recall that a model is specified in terms of a set of parameters θ. Our objective here is to learn a model, θ^*, that best fits the data, e.g. that maximizes the likelihood of observing the archival data. Thus, we want to compute the following

$$\theta^* = \arg\max_\theta \prod_{(T,S) \in ArchivalData} \mathbf{P}_\theta(T \mid S) \tag{3}$$

First we look at a simpler problem: given a model θ, and trees S and T, efficiently compute $\mathbf{P}_\theta(T \mid S)$. We call this the problem of *computing transformation probability*. While the problem is of independent interest as an alternative to tree-similarity metrics based on minimum edit-distance (e.g. [6]), we will also need it in our algorithm to estimate θ^*. Next, we give an efficient algorithm to compute transformation probabilities and then present the algorithm for model learning.

4.1 Computing Transformation Probabilities

The transducer π performs a sequence of edit operations consisting of insertions, deletions and substitutions to transform a tree S into another tree T. There can be multiple

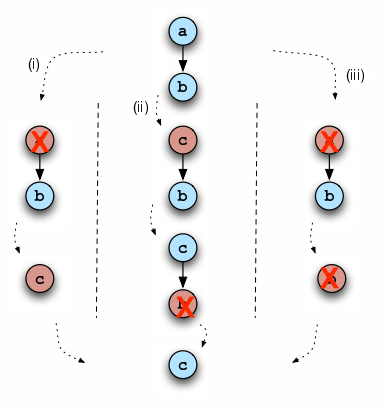

Figure 6: Multiple Edit Paths

possible ways to transform S into T. For instance, Figure 6 shows three ways that a tree with two nodes labeled a and b can be transformed into another tree with a single node c: (i) by deleting a and substituting b with c, or (ii) by substituting a with c and deleting b or (iii) by deleting a and b and inserting c. Since there can be exponentially many ways to transform a tree into another, directly enumerating them does not lead to an efficient solution. Instead, we use dynamic programming to efficiently compute the transformation probabilities.

Let F_s and F_t be the subforests of S and T respectively. Let $\mathsf{DP}_1(F_s, F_t)$ denote the probability that $\pi(F_s) = F_t$. Let u and v denote the roots of the rightmost trees in F_t and F_t respectively. Note that every node in F_t is either newly created by some π_{ins} operator or is the result of a substitution by some π_{sub} operator from some node in F_S. Let $\mathsf{DP}_2(F_s, F_t)$ denote the probability that $\pi(F_s) = F_t$ and v was generated by a substitution under π.

We next show how to compute DP_1 and DP_2 recursively. Consider $\mathsf{DP}_1(F_s, F_t)$. There are two cases: (i) The node v was the result of an insertion by π_{ins} operator. Let p be the probability that π_{ins} inserts the node v in $F_t - v$ to form F_t. Then, the probability of this case is $\mathsf{DP}_1(F_s, F_t - v) * p$. (ii) The node v was the result of a substitution. The probability of this case is $\mathsf{DP}_2(F_s, F_t)$. Hence, we have

$$\mathsf{DP}_1(F_s, F_t) = \mathsf{DP}_2(F_s, F_t) + p * \mathsf{DP}_1(F_s, F_t - v) \quad (4)$$

Now consider $\mathsf{DP}_2(F_s, F_t)$. Again, there are two cases: (i) v was substituted for u. In this case, we must have $F_s - [u]$ transform to $F_t - [v]$ and $\lfloor u \rfloor$ transform to $\lfloor v \rfloor$. Denoting $p_{sub}(label(u), label(v))$ with p_1, the total probability of this case is $p_1 * \mathsf{DP}_1(F_s - [u], F_t - [v]) * \mathsf{DP}_1(\lfloor u \rfloor, \lfloor v \rfloor)$. (ii) v was substituted for some node other than u. Then, it must be the case that u was deleted. Denoting $p_{del}(label(u))$ with p_2, the total probability of this case is $p_2 * \mathsf{DP}_2(F_s - u, F_t)$. Hence,

$$\begin{aligned} \mathsf{DP}_2(F_s, F_t) &= p_1 \mathsf{DP}_1(F_s - [u], F_t - [v]) \mathsf{DP}_1(\lfloor u \rfloor, \lfloor v \rfloor) \\ &+ p_2 \mathsf{DP}_2(F_s - u, F_t) \end{aligned} \quad (5)$$

The functions DP_1 and DP_2 can be computed using a dynamic programming algorithm based on Equation (4) and (5). In fact, we do not need to compute these functions for all pairs of subforests of S and T. We only need to compute them for *special subforests* as defined by Zhang and Shasha [32], who also show that the number of such subforests for a tree T is bounded by $|T| \min(D(T), L(T))$, where $D(T)$ is the *depth* of T and $L(T)$ is the *number of leaves*. Thus, we have:

THEOREM 2. *The probability* $\mathbf{P}_\theta(T \mid S)$ *can be computed in time* $O(|S||T| \min D(S), L(S) \min D(T), L(T))$.

Note that in the HTML domain, $D(T) \ll L(T)$ in practice, and seldom exceeds 20.

EXAMPLE 2. Recall the trees in Figure 6. Let T_1 be the tree with the nodes a and b, and let T_2 be the tree with single node c. Let us compute the probability that $\pi(T_1) = T_2$, which is denoted by $\mathsf{DP}_1(T_1, T_2)$. Applying Eq. (4), we get

$$\mathsf{DP}_1(T_1, T_2) = \mathsf{DP}_2(T_1, T_2) + p_{ins}(\mathsf{c}) * \mathsf{DP}_1(T_1, \emptyset)$$

$\mathsf{DP}_1(T_1, \emptyset)$ is the probability that the tree T_1 gets transformed into empty tree, which happens when each node gets deleted and there are no insertions. $\mathsf{DP}_1(T_1, \emptyset)$ evaluates to $p_{del}(\mathsf{a}) * p_{del}(\mathsf{b}) * p_{stop}$. To compute $\mathsf{DP}_2(T_1, T_2)$, we apply Eq. (5). Let T_3 denote the tree with single node b. Then,

$$\begin{aligned} \mathsf{DP}_2(T_1, T_2) &= p_{sub}(\mathsf{a}, \mathsf{c}) * \mathsf{DP}_1(\emptyset, \emptyset) * \mathsf{DP}_1(T_3, \emptyset) \\ &+ p_{del}(\mathsf{a}) * \mathsf{DP}_2(T_3, T_2) \end{aligned}$$

We have $\mathsf{DP}_1(\emptyset, \emptyset) = p_{stop}$ and $\mathsf{DP}_1(T_3, \emptyset) = p_{del}(\mathsf{b}) * p_{stop}$. To compute $\mathsf{DP}_2(T_3, T_2)$, we get

$$\begin{aligned} \mathsf{DP}_2(T_3, T_2) &= p_{sub}(\mathsf{b}, \mathsf{c}) * \mathsf{DP}_1(\emptyset, \emptyset) * \mathsf{DP}_1(\emptyset, \emptyset) \\ &+ p_{del}(\mathsf{b}) * \mathsf{DP}_2(\emptyset, T_2) \end{aligned}$$

The last term, $\mathsf{DP}_2(\emptyset, T_2)$, is 0 because there is no node in an empty tree that can be substituted. Combining everything, the total probability of transforming T_1 to T_2 is

$$\begin{aligned} \mathsf{DP}_1(T_1, T_2) &= p_{sub}(\mathsf{a}, \mathsf{c}) * p_{del}(\mathsf{b}) * p_{stop}^2 \\ &+ p_{sub}(\mathsf{b}, \mathsf{c}) * p_{del}(\mathsf{a}) * p_{stop}^2 \\ &+ p_{del}(\mathsf{a}) * p_{del}(\mathsf{b}) * p_{ins}(\mathsf{c}) * p_{stop} \end{aligned}$$

We see that the three terms in the above expression correspond precisely to the three paths for transforming T_1 to T_2 as shown in Figure 6.

4.1.1 Optimizations for Similar Trees.

An interesting special case of $\mathsf{DP}_1(S, T)$ is the case in which S and T are assumed to be *largely similar*, an assumption that largely holds for subsequent snapshots of HTML pages (especially those on which it is profitable to define wrappers). In fact the space and time needed to compute $\mathsf{DP}_1(S, T)$ can be substantially improved in practice by modifying the first case of Equation 5. The idea is to avoid exploring the case that the subtree of $[u]$ evolves to $[v]$ unless these trees are *substantially similar*; that is, by bounding $\mathsf{DP}_1([u], [v])$, and assuming that it is in fact 0 if S is substantially different from T.

We consider two simple metrics of *substantial difference*. One is the absolute number of changed nodes, which can be cheaply lower bounded by the difference in the size of the two trees. Thus, if the size of the two trees are substantially different, we declare their distance to be 0. The second metric is based on the assumption that certain strings that appear as leaf nodes in both S and T will be associated with the same node under any valid transformation of S to T. For

example, if an *IMDB* webpage contains a unique occurrence of the string "Francis Ford Coppola" and the new version of the page also contains the same string, then a transformation should match the two nodes. We call such strings *alignment strings*. In a preprocessing step, for each node in each tree, we compute the set of alignment strings under its subtree. We declare $\mathsf{DP}_1(S,T)$ to be 0 if S and T do not contain the same set of alignment strings. To identify alignment strings, we require that 1) these strings never appear more than once in any file in the corpus and 2) the *corpus* and *web document frequency* of the strings (taken as a set of tokens or phrases) is "extremely low" (e.g., containing 3 or more uncommon words).

4.2 Parameter estimation

As suggested in Eq. 3, we seek parameters that can maximize the likelihood of observed data. Specifically, let $\mathcal{T} = \{(S_n, T_n), n = 1, 2, \ldots, N$ denote N pairs of data in our archival collection of original webpages and transformed ones. We apply the criteria of maximum likelihood estimation to find the optimal θ^* such that the likelihood of observing the *training data* $\mathcal{T}$ is maximized:

$$\theta^* = \arg\max_\theta \mathbf{P}_\theta(T_1, T_2, \ldots, T_N \mid S_1, S_2, \ldots, S_N) \tag{6}$$

Assuming that the changes on these webpages are independent of other webpages, we arrive at a simplified estimation expression,

$$\theta^* = \arg\max_\theta \prod_{n=1}^{N} \mathbf{P}_\theta(T_n \mid S_n) \tag{7}$$

Note that using the notations of Sec 4.1, the probability $\mathbf{P}_\theta(T_n \mid S_n)$ is precisely $\mathsf{DP}_1(S_n, T_n)$. For algebraic convenience, we maximize the logarithm of the likelihood probability $\ell(\theta) = \prod_{n=1}^{N} \mathbf{P}_\theta(T_n \mid S_n)$ to compute the optimal parameter

$$\theta^* = \arg\max_\theta \sum_{n=1}^{N} \log \mathbf{P}_\theta(T_n \mid S_n) \tag{8}$$

The function $\log \ell(\theta)$, called the log-likelihood, is a very complicated nonlinear function of the parameter θ. Therefore, it is difficult to find a close form solution to Eq. (8). Instead, we use an iterative numerical method to find θ^*. The method is based on gradient-ascent, a common approach in nonlinear optimization [25]. We start by describing gradient-ascent methods in its general form and then address specific constraints when applied to our optimization problem of Eq. (8).

Gradient-ascent Suppose we have an estimate of the parameter θ^t at the time step t. How do we change it such that the new estimate θ^{t+1} for the next time step is closer to the optimal solution θ^*? Recall that the gradient of a function with respect to its variables is the direction that the function value increases the most, we follow this direction to update

$$\theta^{t+1} = \theta^t + \eta g(\theta^t) \tag{9}$$

where the gradient at the current estimate θ^t is defined by the evaluation of $g(\theta) = \partial \log \ell(\theta)/\partial\theta$ at θ^t. The variable η is a positive number, often referred as step size. It controls how far we want to follow the gradient direction $g(\theta^t)$. With suitably chosen step size, we are guaranteed that the *log*-likelihood is increased by updating the current estimate iteratively with Eq. (9)

$$\log \ell(\theta^1) < \log \ell(\theta^2) < \cdots < \log \ell(\theta^t) < \log \ell(\theta^{t+1}) < \cdots$$

Note that the *log*-likelihood is upper bounded $\log \ell(\theta) \le 0$ as it is just the logarithm of a product of probabilities. Therefore, the sequence in the above equation will converge (with a suitably chosen η) and so is the estimate of the parameter $\lim_{t\to\infty} \theta^t = \hat{\theta}$. If the *log*-likelihood $\log \ell(\theta)$ has one unique optimum θ^*, then the converging point of the update $\hat{\theta}$ is necessarily the same as θ^*. In many cases, however, the function $\log \ell(\theta)$ has multiple local optima. The update procedure thus converges to a local optimum and is not guaranteed to find the global optimum. In our experiments with synthetic data where the "ground-truth" parameters are known, convergence to local optima did not present an issue as our algorithm converges to estimates that are very close to the true parameters.

Computing the gradients A key step in applying the update Eq. (9) is to compute the gradient direction

$$g(\theta) = \frac{\partial \log \ell(\theta)}{\partial \theta} = \sum_{n=1}^{N} \frac{\partial \log \mathbf{P}_\theta(T_n \mid S_n)}{\partial \theta}$$

In the following, we sketch the steps for computing this quantity. The gradient can be computed efficiently with dynamic programming, in a way similar to the computing of the transformation probability in section 4.1. From instance, from Eq. 4, we obtain

$$\begin{aligned} \frac{\partial \mathsf{DP}_1(F_s, F_t)}{\partial \theta} &= \frac{\partial \mathsf{DP}_2(F_s, F_t)}{\partial \theta} + p * \frac{\partial \mathsf{DP}_1(F_s, F_t - v)}{\partial \theta} \\ &+ \frac{\partial p}{\partial \theta} \mathsf{DP}_1(F_s, F_t - v) \end{aligned} \tag{10}$$

where v is the root of the tree T_n and $T_n - v$ is the forest after v is removed. Thus, using another set of dynamic programming equations similar to Eq. (4) and (5), we can compute efficiently the quantity $\partial \log \mathbf{P}_\theta(T_n \mid S_n)/\partial\theta$, which is given $1/\mathsf{DP}_1(S_n, T_n)\partial \mathsf{DP}_1(S_n, T_n)/\partial\theta$.

Choosing step size η To assure convergence of the iterative update in Eq. (9), the step size η needs to be carefully chosen. A small η leads to slower convergence; a large η causes oscillations such that the monotonic increase of the objective function $\log \ell(\theta)$ is broken and the update fails to converge. In our experiments, we follow a standard procedure called *Armijo rule* [25] to dynamically adjust the η value during the iterative update. On our control experiments with synthetic data, this procedure works well and the update quickly converges to the truth parameter values.

Updating constrained parameters Note that the parameters θ are constrained by Eq. (2). In particular, the elements of θ are probabilities, constrained between 0 and 1. Thus, the update in Eq. (9) could lead to invalid values for the parameter θ, for instance, causing values to become negative or be greater than 1. Furthermore, for parameters that are constrained to sum to 1, such as the insertion probabilities $\sum_l p_{ins}(l) = 1$, the update breaks the normalization constraint. How can we overcome this problem?

We deploy a simple and effective trick called *variable reparameterization*. Concretely, for each element θ_i in the parameters θ, we introduce another variable α_i which is unconstrained and can be any real number. We reparameterize θ in terms of these auxiliary variables depending on how θ_i is constrained.

If a group of K parameters $\{\theta_{i_j}, j = 1, 2, \ldots, K\}$ are constrained such that $\sum_j \theta_{i_j} = 1$, we rewrite these θ_i as

$$\theta_{i_j} = \frac{e^{\alpha_{i_j}}}{\sum_{j=1}^{K} e^{\alpha_{i_j}}} \quad (11)$$

Note that the normalization constraint $\sum_j \theta_{i_j} = 1$ is automatically satisfied no matter what the values of α_{i_j} are. Additionally, because of the exponentiation, θ_{i_j} are guaranteed to be nonnegative for any α_{i_j}.

For some parameters θ_i such as the deletion probabilities $p_{del}(l)$, the only constraints are $0 \leq \theta_i \leq 1$. This is a special case of parameters with normalization constraints and can be handled similarly. Specifically, for these parameters, we introduce yet another "twin" variable β_i such that

$$\theta_i = \frac{e^{\alpha_i}}{e^{\alpha_i} + e^{\beta_i}} \quad (12)$$

Note that θ_i are now guaranteed to be between 0 and 1 for any α_i and β_i. Hence, this is a simplified case of Eq. (11) where $K = 2$.

To summarize, without loss of generality, we can represent every parameter θ_i in terms of a suitably chosen subset of α_i and β_i, in the form of Eq. (11). With a slight abuse of notation, we append those β_i to α_i and call the resulted (longer) parameter vector α. On an abstract level, the *log*-likelihood $\log \ell(\theta)$ now becomes a function of this α parameter.

To find the α, we follow the gradient-ascent method outlined previously. Specifically, the update takes the form of

$$\alpha^{t+1} = \alpha^t + \eta g(\alpha^t) \quad (13)$$

with *unconstrained* α. The key quantity to compute is again the gradient $g(\alpha) = \partial \log \ell(\theta)/\partial \alpha$ evaluated the current estimate α^t. Mindful of the reparameterization of Eq. (11), the gradient is computed through the chain rule, considering θ_{i_j} as an function of related α_{i_j}

$$\frac{\partial \log \mathbf{P}_\theta(T_n \mid S_n)}{\partial \alpha_{i_j}} = \sum_{k=1}^{K} \frac{\partial \log \mathbf{P}_\theta(T_n \mid S_n)}{\partial \theta_{i_k}} \frac{\partial \theta_{i_k}}{\partial \alpha_{i_j}} \quad (14)$$

Using Eq. (11), we get that $\partial \theta_{i_k}/\partial \alpha_{i_j}$ equals $\theta_{i_k}(\delta_{i_j i_k} - \theta_{i_j})$ where $\delta_{i_j i_k} = 1$ if $i_j = i_k$ and 0 otherwise.

5. GENERATING CANDIDATE WRAPPERS

We want to consider a set of alternative wrappers for our underlying extraction task to pick the most robust one according to our model. In order for this strategy to be successful, the candidate set should contain a variety of potentially robust wrappers. The previous work [2] on automatically learning XPath rules from labeled webpages works top-down, i.e. it starts from the specific paths in each webpage and generalizes them to a single XPath. Unfortunately, this results in the most specific XPath, which contains all possible predicates common across all webpage. The resulting XPath is complex and brittle, and not a suitable candidate for a robust wrapper.

In this section, we describe an algorithm that generates wrappers in a bottom-up fashion, by starting from the most general XPath that matches every node and specializing it till it matches only the target node in each document. This strategy is similar to wrapper-learning techniques such as *STALKER* [22] which consider tokens near to the target strings that should be extracted (but these techniques do not produce XPaths).

Let $\mathcal{D}$ be a set of labeled XML documents, where for each document, a subset of its nodes are labeled. For $D \in \mathcal{D}$, let $L(D)$ denote the subset of nodes of D which are labeled as target nodes. We want to generate XPath expressions w such that for each D, we have $w(D) = L(D)$. Given an XPath w, define

$$\begin{aligned} Precision(w) &= \sum_D (w(D) \cup L(D))/w(D) \\ Recall(w) &= \sum_D (w(D) \cup L(D))/L(D) \end{aligned}$$

We want to generate XPath expressions that have both precision and recall equal to 1.

Let w be an XPath. For illustration, we will use the following

$$w_0 = //table/*/td/text()$$

We define a one-step specialization of w to be an XPath obtained by any of the following operations on w :

1. converting a $*$ to a label-name. For instance, w_0 can be specialized to
$$//table/tr/td/text()$$
2. adding a predicate to some node in w. E.g.
$$//table[bgcolor =' red']/*/td/text()$$
3. adding child position information to some node in w. E.g.
$$//table/*/td[2]/text()$$
4. adding a $//*$ at the top of w. E.g.
$$//*/table/*/td[2]/text()$$

We say that $w_0 \rightsquigarrow w_1$ if w_1 is a one-step specialization of w_0 and we say that $w_0 \rightsquigarrow^* w_1$ if w_1 can be obtained from w_0 using a sequence of specializations.

The algorithm maintains a set P of partial wrappers. Each element of P is an XPath expressions which has a recall of 1, but precision less than 1. Initially P contains the single XPath "$//*$" that matches every node. The algorithm repeatedly applies specialization steps to XPaths in P to obtain new XPaths. XPaths are removed from P when their precision reaches 1 and added to the set of output Wrappers.

Given a set of documents $\mathcal{D}$ and an XPath w, we say that w is *minimal* if there is no other XPath w_0 such that $Precision(w_0) = Precision(w)$, $Recall(w_0) = Recall(w)$ and $w_0 \rightsquigarrow^* w$. Note that if w is a wrapper, i.e. it has precision and recall 1, and w is not minimal, then we can find a smaller XPath w_0 which is also a Wrapper. Since smaller XPaths are less likely to break, we are interested in enumerating all the minimal wrappers. The naive way of obtaining all minimal wrappers is to enumerate all wrappers and discard the ones which are not minimal. Instead, we use the following result to speed up the algorithm.

LEMMA 5.1. *Let X be any XPath expression, w be a wrapper such that $X \rightsquigarrow^* w$. If w is minimal, X is also minimal.*

Using this lemma, we can modify the algorithm to discard non-minimal XPath expressions in the set P after each specialization. The final algorithm for enumerating minimal wrappers is described in Algorithm 1.

Algorithm 1 ENUMERATE MINIMAL WRAPPERS
Input: A set of labeled webpages **Output:** S, a set of XPath wrappers

```
1: S ← ∅
2: P ← {"//*"}    (P is the set of partial wrappers)
3: while P ≠ ∅ do
4:    Let w be any XPath in P
5:    P ← P − w
6:    for all w0 s.t. w ⇝ w0 (one-step specialization) do
7:       if isMinimal(w0) and Recall(w0) = 1 and
         Precision(w0) = 1 then
8:          S = S ∪ w0
9:       end if
10:      if isMinimal(w0) and Recall(w0) = 1 and
         Precision(w0) < 1 then
11:         P = P ∪ w0
12:      end if
13:   end for
14: end while
```

THEOREM 3. *Algorithm 1 is sound and complete, i.e. it generates all minimal wrappers and only minimal wrappers.*

Using Anchor Texts One of the specialization operations we defined is adding a predicate to a node. We consider two kinds of predicates, $[attr = value]$ and $[xpath = "text"]$. The latter kind of predicates are based on the fact that there are often strong text cues near labeled nodes which can be exploited to generate robust wrappers. For instance, in the *IMDB* website, the node containing the director names is preceded by a node containing the text "Director : ".

We use anchor texts in two steps. In the first step, we identify all potential strings in the documents that can serve as anchor texts, where we define potential strings to be texts that appears in all the documents at the same location. Formally, given a node n in a document, let $path(n)$ denote the tag sequence from the root of the document to n. Then, an anchor text is a pair $(path, text)$, such that for each document in the corpus there is a text node n such that the content of n is *text* and $path(n) = path$, and we call n an *anchor node*. We find all the anchor nodes in the documents by a single pass over the corpus and maintaining a hash table of all the candidate anchor texts.

In the specialization step, given a node to add predicate to, we look for anchor nodes near its vicinity in the XML tree and for each such occurrence, we add the predicate $[xpath = "text"]$ where $xpath$ is the path used to reach the anchor node from the node under consideration and "*text*" is the content of the anchor node.

6. EVALUATING ROBUSTNESS OF WRAPPERS

Let Σ be a set of labels, $\mathcal{D}$ be the set of all XML document over Σ and θ be a probability model over $\mathcal{X}$.

Given a boolean property φ over $\mathcal{D}$, and a document $X \in \mathcal{D}$, we say that a document X models φ, denoted $X \models \varphi$, if the property φ is true on X. The *robustness* of φ on X, denoted $Rob_{X,\theta}(\varphi)$, is defined as the probability

$$Rob_{X,\theta}(\varphi) = \sum_{Y \mid X \models \varphi} \mathbf{P}_\theta(Y \mid X)$$

Thus, robustness of φ is the probability that φ holds on X if X changes randomly according to the change model θ.

A wrapper w canonically defines a Boolean property φ_w such that φ_w holds on all documents where the Wrapper correctly extracts the required field. We extend the notion of robustness to wrappers by viewing it as a Boolean property.

Given any XPath expression w, we can think of w as a Boolean property over $\mathcal{X}$ where $X \models w$ if the set of nodes selected by X on w is non-empty. For example, the XPath `//td/tr` defines a Boolean property that is true on all documents containing some node `td` with a child node `tr`. Also, given an XPath wrapper w, we can express the property φ_w itself as a Boolean XPath by adding a predicate to w that tests for the correctness of the extracted value. E.g., suppose we have the following wrapper

$$w = //\texttt{table}/\texttt{td}[5]/\texttt{tr}[3]/\texttt{text}()$$

for extracting *Director* from film pages and X is a webpage with *Director* = "*Francis Copolla*". Then, we can evaluate $Rob_{X,\theta}(\varphi_w)$ by evaluating $Rob_{X,\theta}(w')$, where w' is the Boolean XPath

$$w' = //\texttt{table}/\texttt{td}[5]/\texttt{tr}[3][\texttt{text}() = \text{"}\textit{Francis Copolla}\text{"}]$$

Hence, in the remainder of this section, we will restrict our discussion to the problem of evaluating the robustness of Boolean XPath properties.

Our first result shows that exact computation of robustness of XPaths is #P-hard [31].

THEOREM 4. *Given a model θ, a document X and an XPath w, computing $Rob_{X,\theta}(w)$ is #P-hard.*

PROOF. We will give a reduction from counting the number of satisfying assignments of a Monotone CNF Boolean formula, a problem known to be #P-complete [27].

Let $C = C_1 \wedge C_2 \cdots C_n$ be a CNF formula over a set of variables $X = \{x_1, \cdots, x_m\}$, where each C_i is a disjunction of a subset of variables in X. We want to compute the number of satisfying assignments of C, which we denote by $\#(C)$.

Construct a tree X with a root node R, a node N_i from each variable $x_i \in X$ and a node $M_{i,j}$ for each pair (x_i, C_j) such that the variable x_i appears in the clause C_j. The tree X has edges from R to N_i for each i and from N_i to $M_{i,j}$ for each j. Let the root node R have a label `A`, each N_i have the label `B` and each $M_{i,j}$ have the label $\texttt{C}_\texttt{j}$.

Let θ be such that the $p_{del}(\texttt{B}) = 1/2$ and all other deletion probabilities are 0. Also, all insertion and substitution probabilities are 0.

Consider the xpath w as follows:

$$w = /\texttt{R}[*/\texttt{C}_1][*/\texttt{C}_2]\cdots[*/\texttt{C}_\texttt{n}]$$

Claim: $Rob_{X,\theta}(w) = \#(C)/2^m$

To see why, note that the transducer π independently deletes each node N_i with probability 1/2. Thus, there are 2^m possible resulting trees, each with probability $1/2^m$.

Each resulting tree Y_i has a natural 1-1 correspondence with an assignment of C where x_i is `false` if N_i was deleted and `true` otherwise. Furthermore, the XPath w holds on a tree Y_i iff every C_j has at least one x_i contained in it which did not get deleted, i.e. it corresponds to a satisfying assignment of C. Thus, there are $\#(C)$ possible Y_i where w holds, each occurring with probability 2^{-m}. This proves the claim. Thus, we can compute the number of satisfying assignments of C from the robustness of w. The #P-hardness of computing $Rob_{x,\theta}(w)$ follows. □

While exact robustness computation of XPath expressions is computationally hard, note that we do need precise robustness for wrappers for two reasons. First, the parameters of the models themselves learnt using gradient search, and are not exact. Secondly, we only need the ability to choose the most robust wrapper. If two wrappers are very close to each other with respect to robustness, we can pick either of them. In light of this, we only need techniques to efficiently approximate the robustness of wrappers. We present a simple yet effective algorithm below.

Note that our probability model is defined in terms of a transducer π, which also gives us an efficient way to sample trees from the distribution $\mathbf{P}_\theta(\cdot \mid X)$ for a given document X. Based on this, we can use Monte-Carlo methods to approximate the robustness of a given wrapper, as described in Algorithm 2.

Algorithm 2 EVALUATE ROBUSTNESS
Input: An XPath w, a document X a model θ.
Output: $Rob_{X,\theta}(w)$

```
count = 0
for i in 1..N do
    Y = π(X)
    if w ⊨ Y then
        count = count + 1
    end if
end for
return count/N
```

7. EXPERIMENTAL EVALUATION

In this section we evaluate the effectiveness of our model-learning strategy on synthetic data and the improvements in robustness afforded by our techniques on a dataset of crawled pages.

Data Sets We experiment with three data sets from the web. The first two are hand-built collections of pages that have existed with the same URL for several years. The NOAA dataset consists of a set of 15 webpages from the website `www.noaa.gov` monitored over last 5 years. The FACULTY dataset consists of a set of 13 faculty homepages from a university monitored over last 5 years. The third, IMDB, data set consists of web pages from the IMDB [14] movie website, a script-generated site which contains a webpage for each movie. We use three pages from IMDB to train wrappers, and a subset of 100 pages to test effectiveness. To obtain the older snapshots of webpages, we use *Internet Archive* [3], a public website that periodically takes snapshots of the Web and archives it.

Implementation We have implemented our framework in Java. All our experiments are run on an Intel Xeon® 2.13GHz machine running Linux with 4GB RAM. We use the *Tidy* [30] utility to clean up and parse HTML pages into trees. As described in Sec 3, we replace HTML text nodes with nodes labeled "TEXT".

Insert		Delete	
a	0.0030	td	0.0366
br	0.0019	br	0.0355
td	0.0014	a	0.0271
span	0.0008	tr	0.0096
tr	0.0008	b	0.0064
b	0.0007	span	0.0064
li	0.0006	li	0.0030

Figure 7: Top θ_N probabilities for NOAA

Insert		Delete	
a	0.0141	a	0.0971
br	0.0073	p	0.0828
li	0.0055	li	0.0581
p	0.0037	br	0.0172
b	0.0024	b	0.0074
i	0.0015	ul	0.0046
img	0.0012	img	0.0038

Figure 8: Top θ_F probabilities for FACULTY

Experiments We perform three experiments. In the first experiment, we learn a change model on real web data given by our datasets. We report our finding in Sec. 7.1. In the second experiment, we use the change model to generate robust wrappers on IMDB and evaluate the robustness of various approaches. This is described in Sec 7.2. In the third experiment, we evaluate the accuracy and efficiency of our model learning algorithm. This experiment is described in Sec 7.3.

7.1 Change Model on Real Web Data

We apply our learning algorithm with all the optimizations as described in Sec. 4.1.1 on NOAA and FACULTY to get models θ_N and θ_F respectively. Each iteration of gradient ascent completed in around 10 mins and the search converged in 20 iterations. We were unable to train a model on the IMDB data set due to large amounts of noise. We discuss this issue in more details in Sec 7.4.

Figure 7 and Figure 8 show the top insertion and deletion probabilities as learned on the two datasets. While these data sets are relatively small, we can make several interesting observations. First, in each dataset, the same set of tags appear in the top list for insertions and deletions, i.e. the tags which are more likely to be deleted are also the tags more likely to be inserted. This supports the intuition that some tags, like `a` and `br`, are inherently more volatile than other tags that don't appear in the top, like `frame` and `div`. Tags like `h1` and `h2` come in the middle. Secondly, we observe that across the two datasets, almost the same sets of tags appear in the top of the list, leading us to hypothesize that the most volatile tags may not depend heavily on the data set under consideration.

While the models learned on the two datasets were quite similar, we did notice some startling differences. For in-

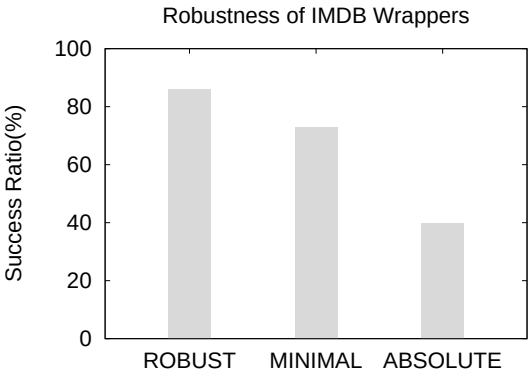

Figure 9: Robustness of *IMDB* Wrappers

stance, while the tags `tr` and `td` appear very prominently in NOAA, they have very small probabilities of insertion and deletion in FACULTY. An inspection reveals that while the *NOAA* website relies a lot on tables to present its data, webpages in FACULTY rarely use tables. They instead rely on lists (and hence, `ul` and `li` tags), e.g. lists of students, courses, publications etc. Thus, while the volatility of tags is a general feature, there could be some blatant exceptions. The second difference is that the absolute probabilities are much higher in FACULTY webpages. This shows that the homepages in FACULTY are more dynamic than the government webpages in NOAA.

We finally train a combined model θ_{NF} on the combined data set for use in ranking wrapper robustness.

7.2 Generating Robust Wrappers

For this experiment, we set up the following extraction task. We consider the IMDB dataset, which contains a webpage for each movie. We want to generate a wrapper rule that extracts the movie director from each webpage. Recall that a wrapper candidate generator takes a training data and generates a set of alternative wrappers. For training data, we randomly select a small subset of movie pages, $D = \{d_1, \cdots, d_t\}$ and label them with their director information. Then we apply the techniques described in Sec 5 to generate wrappers. We found that $|D| = 3$ was sufficient to learn effective wrappers [1]. We consider the change model θ_{NF} that we learnt in Sec 7.1, and apply our framework to choose a robust wrapper among various possible wrappers on this dataset.

To evaluate the robustness of our framework, we perform an empirical evaluation. We use *Internet Archive* to obtain all the previous versions of *IMDB* webpages. Let $D^{(t)}$ denote the snapshot of the pages in the training data at time instance t. The robustness is measured by looking at how often a wrapper learned using $D^{(t)}$ continues to work at *IMDB* pages at time instance $t + 1$. Note that we are using archival data here to evaluate the robustness or wrappers, not to learn a change model.

We compare three different approaches for generating wrappers: ROBUST, MINIMAL and ABSOLUTE. ROBUST denotes the wrapper that has the highest robustness, as predicted by our framework, among all minimal wrappers. MINIMAL denotes a minimal wrapper picked among the set of minimal wrappers without robustness considerations. ABSOLUTE denotes the wrapper consisting of absolute XPath from the root to the node of interest.

We look at 15 previous snapshots of *IMDB* at 2 months intervals and count the number of times the wrapper continue to work on the next snapshot for each of the three approaches. Figure 9 shows the result. We observe that choosing minimal wrappers substantially increased the robustness over absolute XPaths. Further, the wrapper generated by our framework is indeed more robust than a minimal wrapper picked without robustness considerations.

We also list the set of all minimal wrappers for a particular snapshot of *IMDB* ordered according to their robustness. Fig. 10 shows the list of wrappers. The list matches well with human intuition. For instance, the most robust wrapper simply looks at the text following "*Director:*" while the next most robust wrapper starts from the text "*Overview*" and goes two steps down. Towards the end, we have a wrapper that look at third `div` tag in the page and the third `div` below it, which is much more likely to break. Also interesting is the fact that when *IMDB* page changed next time, the first two wrappers in the list continued to work while all the remaining wrappers failed.

7.3 Evaluation of Model Learner

In this section, we study the effectiveness of model learner in correctly learning a model.

Evaluation Methodology We perform a controlled experiment. We fix a set of labels Σ and a change model θ over Σ, use θ to generate tree changes where each node is labeled from Σ, then test the ability of our parameter estimation algorithm to recover θ from the generated pages. In particular, we generate *training data* $\mathcal{T}$ as follows. Let π denote the transducer corresponding to the set of parameters θ. We consider an arbitrary set of trees $X_1, \cdots, X_N$ over Σ, and apply the transducer on each of them to get $\mathcal{T} = \{(Y_1, X_1), \cdots (Y_N, X_N)\}$, where $Y_i = \pi(X_i)$ for each i. This gives us the training set $\mathcal{T}$. Our experiment consists of using $\mathcal{T}$ to learn a model and then compare it with the true underlying model θ. Let θ^t denote the model we learn after running t steps of gradient ascent. We want to compare θ^t with the true model θ and study its behavior with t.

To compare θ with θ^t, we adopt the standard approach of using a *validation set*. A validation set V, similar to the training set $\mathcal{T}$, is a set of tree pairs drawn from the same underlying distribution but which are not used in the training. We compare the likelihood of V predicted by the learnt model θ^t with that predicted by the true θ. Specifically, we compare $-\ell(\theta^t)$ with $-\ell(\theta)$, where ℓ is the log-likelihood function as defined in Section 4.2. The lesser the difference, the closer the two models. Note that an alternative for comparing the two models would be to look at relative errors of individual model parameters. But a model can be more sensitive to certain parameters while being more tolerant to errors in other parameters, and it is not clear how to combine the errors in individual parameters. The validation set

[1] We tested the effectiveness of the wrappers on 100 randomly selected pages from IMDB from the same time snapshot, and found them to be effective. The only exception was a small fraction of webpages that had multiple directors. While these pages can be covered by building a second wrapper, for experimental purposes, we ignored these pages.

Robustness	Wrapper
0.400	//*[preceding-sibling::*[position()=1][text()='Director:']]/text()
0.370	//*[preceding-sibling::*[position()=1][text()='Overview']]/a/text()
0.370	//*[h5/text() = 'Director:']/a/text()
0.310	//*[@id='tn15content']/div[position()=3]/a/text()
0.250	//*[preceding-sibling::*[position()=1][text()='advertisement']]/div[position()=3]/a/text()
0.250	//*[h3/text() = 'Overview']/div[position()=3]/a/text()
0.230	//div[position()=3][@class='info']/a/text()
0.190	//*[@id='tn15main']/*/div[position()=3]/a/text()
0.180	//*[preceding-sibling::*[position()=1][h6/text()='Quicklinks']]/*/div[position()=3]/a/text()
0.180	//*[h3/text() = 'Fun Stuff']/div[position()=3]/a/text()
0.170	//*[preceding-sibling::*[position()=1][a/text()='trivia']]/*/div[position()=3]/a/text()
0.160	//*[preceding-sibling::*[position()=1][h6/text()='Top Links']]/*/div[position()=3]/a/text()
0.150	//*[preceding-sibling::*[position()=1][a/text()='official sites']]/*/div[position()=3]/a/text()
0.130	//div[position()=4]/*/div[position()=3]/a/text()
0.080	//div[position()=3]/div[position()=3]/a/text()
0.060	//*[preceding-sibling::*[position()=3][div/text()='SHOP']]/*/div[position()= 3]/a/text()

Figure 10: List of all minimal wrappers for extracting *Director* on *IMDB*

based approach overcomes this issue, and compares the true predictive accuracy of the models.

Evaluation Setup Now we give the details of our experimental setup based on the above methodology. We fix the following set of labels

$$\Sigma = \{\texttt{root}, \texttt{a}_1, \texttt{a}_2, \texttt{a}_3, \texttt{a}_4, \texttt{a}_5, \texttt{b}\}$$

We consider a change model θ where `root` is immutable, $\texttt{a}_\texttt{i}$ can get deleted or substituted and `b` can get inserted. More precisely, we consider the following. For `root`, $p_{del}(\texttt{root}) = p_{ins}(\texttt{root}) = 0$ and $p_{sub}(\texttt{root}, l) = 1$ if $l = \texttt{root}$ and 0 otherwise. For label $\texttt{a}_\texttt{i} (1 \leq i \leq 5)$, the parameters are:

$$p_{del}(\texttt{a}_\texttt{i}) = 0.4 \qquad p_{ins}(\texttt{a}_\texttt{i}) = 0$$

$$p_{sub}(\texttt{a}_\texttt{i}, l) = \begin{cases} 0.3 & l = \texttt{a}_\texttt{i} \\ 0.3 & l = \texttt{a}_{\texttt{i+1 mod 5}} \\ 0 & \text{otherwise} \end{cases}$$

Finally, for label `b`, $p_{ins}(b) = 1$, $p_{del}(b) = 0$ and $p_{sub}(b, l) = 1$ if $l = \texttt{b}$ and 0 otherwise. The stopping probability is $p_{stop} = 0.8$. Note that the resulting set of parameters, θ, satisfies all the conditions in Eq. (2) and define a valid change model.

Let π denote the transducer corresponding to the set of parameters θ. We generate the training set $\mathcal{T}$ by applying the transducer on a set of trees, as explained in the evaluation methodology. We generate 400 pairs, i.e. the size of $\mathcal{T}$ is 400. We also generate a validation set V of size 400.

We use $\mathcal{T}$ to learn the change model. Since Σ has 7 labels, we need to learn 56 independent parameters[2]. We use the gradient ascent method describe in Sec 4.2 to find the set of parameters that maximize the likelihood of observing $\mathcal{T}$.

Results Figure 11 plots the value of $-\ell(\theta^t)$ on the validation set V for various t, where ℓ is the log-likelihood function as defined in Section 4.2. We observe that the gradient ascent search converges to the true model rapidly. There is a sharp drop in error in first 5 iterations, and after around 40 iterations, the model is already very close to the true model. Also observe that there is a very small gap between the two curves that seems to remain even in the asymptotic case. This is due to the sampling errors that creep in when training data is generated. E.g. even if the probability of deletion of $\texttt{a}_1$ is 0.4, it will not get deleted exactly 40 out of 100 times in the training set.

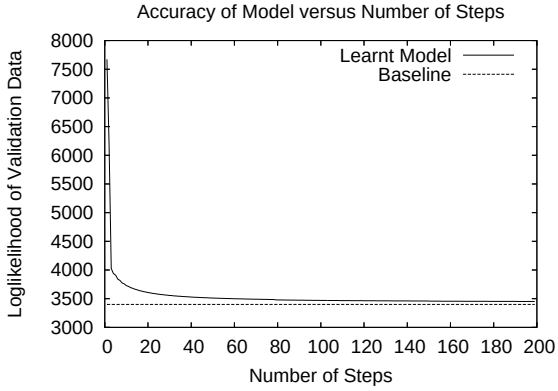

Figure 11: Accuracy of Model Learner

In this experiments, all the trees were small with an average of 16 nodes per tree. The model learner was extremely efficient on this dataset. Each iteration of gradient ascent took around 250ms.

7.4 Discussion

Change Model For The Web Change models can be learnt at various levels of granularities: individual webpage level, website level, domain level or at the other extreme a single model for the entire web. While learning models at individual webpage level might be an overkill and infeasible due to unavailability of archival data, learning a single general model for the web might hinder its effectiveness in producing the most robust wrapper. An interesting problem

[2]In general, if there are n labels, there are n^2+n independent parameters in the model: 1 p_{stop} value, n p_{del} values, $n-1$ p_{ins} values (not n, since they all add up to 1) and $n^2 - n$ p_{sub} values (again, since they add up to 1 for each label).

to explore is how the change model differs across different parts of the web, and how much effect does the specificity of model has on the robustness of generated wrappers. Our experiments suggest that certain characteristics of models might hold universally across the web, e.g. certain *HTML* tags are inherently more volatile. At the same time, there can be blatant exceptions across websites and domains. A detailed study of behavior of change models across the web is an interesting direction to explore, but is beyond the scope of this work.

Handling Noise We describe here the challenges in learning a model for IMDB. IMDB dataset differs from the other two datasets in terms of having more *dynamic content* : it contains a section for third-party advertisements, which contains a random fragment of HTML in each snapshot, forthcoming announcements, user comments and discussion boards which, again, are very dynamic and result in random content in each snapshot. This incurs a lot of noise in our learning algorithms, which consider the pages to be "too different" to align. Since our algorithms are optimized for trees which are similar, the noise not only causes errors in learning changes but also makes the time complexity prohibitively high. An interesting future direction would be to isolate the main stable content of webpages and focus on the changes there. There is a parallel line of research on webpage segmentation for discovering the most important blocks on a webpage. Incorporating this into our framework is beyond the scope of this work.

8. RELATED WORK

Very little work directly addresses building *more robust wrappers*. Two studies [17, 1] experimentally evaluate the robustness of hand-built wrappers by testing them on later versions of the same page. Other work discusses the need for more robust wrappers [23, 2], or in one case for more fragile wrappers (so that change can be detected easily). In general, the result is the suggestion to use relative XPaths and to normalize variant tags (for example, changing "b" and "i" tags to "font" [9]).

Our probabilistic tree model is related to two bodies of existing work, *probabilistic tree transducers* and *tree edit models*. The work on probabilistic tree transducers (see [5, 11, 15] and citations) focuses on machine translation between languages when "non-local" effects are needed. In this case, sentences can be modeled by their parse trees, and probabilistic tree transducers on the parse trees can be trained to translate sentences. Unlike HTML trees, these grammar trees have a *small, fixed* number of children.

A number of papers have focused on finding the *edit distance*, or shortest editing script that changes a source tree to a target tree (see [6] and citations). There are also weighted versions of tree edit distances that assign different weights to different edit operations. However, these models do not define a probability distribution: they do not compute the probability of a source tree changing to the target tree, but only the shortest/lightest path according to the weights. To see the difference, consider the two tree transformations of Figure 12. Clearly, each transformation has an edit distance of 1; in the first case, corresponding to deletion of the leftmost `L1` child, and in the second case corresponding to the deletion of *any* of the n nodes. Clearly, the second transformation is far more probable (if individual edits have the same probability). The lack of a probability distribution also

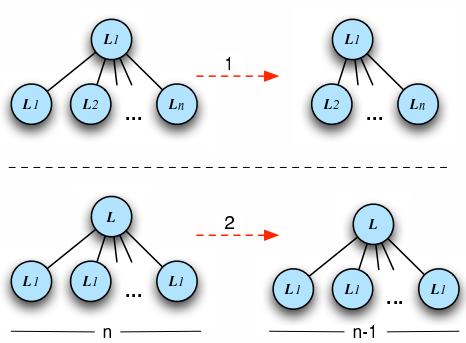

Figure 12: Edit Distance vs. Probabilistic Model

makes it difficult to define a principled learning component that can learn the weights of various edit operations.

Probabilistic edit models do exist for strings [28, 26, 20]. Ristad et al. [28] consider a string edit model where the source string is scanned from left to right and at each step, a probabilistic decision is made to either insert a new character, delete the current character or substitute the current character. A key property of this model is that both the source and target strings can be recovered from the edit script. For example, if the edit script is the following sequence of operations

$$ins(\texttt{a}), ins(\texttt{b}), sub(\texttt{d}, \texttt{c}), sub(\texttt{e}, \texttt{f}), del(\texttt{g}), ins(\texttt{h})$$

then the source string must be `deg` and target string must be `abcfh`. As a result, a probability distribution on the set of all edit scripts also induces a probability distribution on the set of all string pairs. While this gives a generative model $\mathbf{P}(S, T)$, there is also work on conditional models for strings [20].

There is a line of work that tries to extend this probabilistic string edit model to trees [5, 7]. However, it incorrectly assumes that even for trees, both the source tree and target tree can be recovered from the edit script. This fails due to the two-dimensional structure of trees. For instance, consider two trees S_1 and S_2, written in prefix notation as (`a`(`a`)(`a`)) and (`a`(`a`(`a`))) respectively. Thus, S_1 is a node with two children and S_2 is a path of length 3. Then, the same script, $del(\texttt{a}), del(\texttt{a}), del(\texttt{a})$ will take both S_1 and S_2 to the empty tree.

Evaluating wrapper robustness is complementary to wrapper repair [21, 19, 8]. The idea here is generally to use *content models* of the desired data to learn or repair wrappers. Repair is only effective in cases where content models are effective, but content models are often not effective. For example, a content model will not generally be able to distinguish the name of a director from that of an actor. Wrapper induction techniques focus on finding a small number of wrappers from a few examples [18, 22, 13, 10, 12, 29, 4, 23, 2]. Any of these techniques, whether manual or automatic, can benefit from a robustness metric on the resulting wrappers, especially when it is desirable to learn a wrapper on very few pages leaving the wrapper construction system to choose between a large number of apparently equally good wrappers.

9. CONCLUSION

In this paper, we introduce a formal model of wrapper robustness to capture and substantially extend past heuristic approaches. We devise a discriminative model for tree similarity based on an edit model that overcomes technical problems around probability normalization in previous efforts. We expect that this tree distance model will have wide applicability outside the domain of wrapper robustness. We prove a number of theoretical results to characterize the problem of finding robust XPath wrappers, and introduce a new family of *minimal* XPath wrappers that are likely to include robust wrappers.

We expect a variety of future work to arise from this. First, we plan to integrate robustness evaluation with candidate generation to improve performance. Second, we expect to investigate ways to prune trees to improve performance without sacrificing model quality. Third, we plan to investigate other applications for the tree edit model and learning algorithm.

10. REFERENCES

[1] Mari Abe and Masahiro Hori. Robust pointing by xpath language: Authoring support and empirical evaluation. *Applications and the Internet, IEEE/IPSJ International Symposium on*, 0:156, 2003.

[2] Tobias Anton. Xpath-wrapper induction by generating tree traversal patterns. In *LWA*, pages 126–133, 2005.

[3] Internet archive: *http://www.archive.org/*.

[4] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *VLDB*, pages 119–128, 2001.

[5] Marc Bernard, Amaury Habrard, and Marc Sebban. Learning stochastic tree edit distance. In *ECML*, pages 42–53, 2006.

[6] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[7] Laurent Boyer, Amaury Habrard, and Marc Sebban. Learning metrics between tree structured data: Application to image recognition. In *ECML*, pages 54–66, 2007.

[8] Boris Chidlovskii, Bruno Roustant, and Marc Brette. Documentum eci self-repairing wrappers: performance analysis. In *SIGMOD*, pages 708–717, 2006.

[9] William W. Cohen, Matthew Hurst, and Lee S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 232–241, New York, NY, USA, 2002. ACM.

[10] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, pages 109–118, 2001.

[11] J. Graehl and K. Knight. Training tree transducers. In *HLT-NAACL*, pages 105–112, 2004.

[12] Wei Han, David Buttler, and Calton Pu. Wrapping web data into XML. *SIGMOD Record*, 30(3):33–38, 2001.

[13] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[14] Internet movie database: *http://www.imdb.com/*.

[15] Kevin Knight and JonathaGraehl. An overview of probabilistic tree transducers for natural language processing. In *Proceedings of the CICLing*, 2005.

[16] Marek Kowalkiewicz, Tomasz Kaczmarek, and Witold Abramowicz. Myportal: robust extraction and aggregation of web content. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1219–1222. VLDB Endowment, 2006.

[17] Marek Kowalkiewicz, Maria E. Orlowska, Tomasz Kaczmarek, and Witold Abramowicz. Robust web content extraction. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 887–888, New York, NY, USA, 2006. ACM.

[18] Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *IJCAI*, pages 729–737, 1997.

[19] Kristina Lerman, Steven N. Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:2003, 2003.

[20] A. McCallum, K. Bellare, and P. Pereira. A conditional random field for discriminatively-trained finite-state string edit distance. In *UAI*, 2005.

[21] Robert McCann, Bedoor AlShebli, Quoc Le, Hoa Nguyen, Long Vu, and AnHai Doan. Mapping maintenance for data integration systems. In *VLDB*, pages 1018–1029, 2005.

[22] I. Muslea, S. Minton, and C. Knoblock. Stalker: Learning extraction rules for semistructured, 1998.

[23] Jussi Myllymaki and Jared Jackson. Robust web data extraction with xml path expressions. Technical report, IBM Research Report RJ 10245, May 2002.

[24] Andrew Ng and Michael Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in Neural Information Processing Systems 2001 (NIPS 14)*, 2002.

[25] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999.

[26] Jose Oncina and Marc Sebban. Learning stochastic edit distance: Application in handwritten character recognition. *Pattern Recogn.*, 39(9):1575–1587, 2006.

[27] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.

[28] Eric Sven Ristad and Peter N. Yianilos. Learning string edit distance. In *ICML*, pages 287–295, 1997.

[29] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using w4f. In *VLDB*, pages 738–741, 1999.

[30] http://sourceforge.net/projects/jtidy.

[31] L. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8:410–421, 1979.

[32] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.