Ad Hoc Data and the Token Ambiguity Problem

Qian Xi[1], Kathleen Fisher[2], David Walker[1], and Kenny Q. Zhu[1]

[1] Princeton University
[2] AT&T Research

Abstract. PADS is a declarative language used to describe the syntax and semantic properties of *ad hoc data sources* such as financial transactions, server logs and scientific data sets. The PADS compiler reads these descriptions and generates a suite of useful data processing tools such as format translators, parsers, printers and even a query engine, all customized to the ad hoc data format in question. Recently, however, to further improve the productivity of programmers that manage ad hoc data sources, we have turned to using PADS as an *intermediate language* in a system that first infers a PADS description directly from example data and then passes that description to the original compiler for tool generation. A key subproblem in the inference engine is the *token ambiguity problem* — the problem of determining which substrings in the example data correspond to complex tokens such as dates, URLs, or comments. In order to solve the token ambiguity problem, the paper studies the relative effectiveness of three different statistical models for tokenizing ad hoc data. It also shows how to incorporate these models into a general and effective format inference algorithm. In addition to using a declarative language (PADS) as a key intermediate form, we have implemented the system as a whole in ML.

1 Introduction

An *ad hoc data format* is any data format for which useful data processing tools do not exist. Examples of ad hoc data formats include web server logs, genomic data sets, astronomical readings, financial transaction reports, agricultural data and more.

PADS [7,20] is a declarative language that describes the syntax and semantics of ad hoc data formats. The PADS compiler, developed in ML, reads these declarative descriptions and produces a series of programming libraries (parser, printer, validator and visitor) and end-to-end tools (XML translator, query engine, reformatter, error monitor, *etc.*). Consequently, PADS can dramatically improve the productivity of data analysts who work with ad hoc data. However, PADS is not (yet) a silver bullet. It takes time for new users to learn the language syntax and even experienced users can take hours or days to develop descriptions for complex formats. Hence, to further improve programmer productivity, we have developed a system called LEARNPADS that automatically generates end-to-end data processing tools directly from example data [9,8]. It uses machine learning techniques to infer a PADS description and then it passes that description on to the PADS compiler. The compiler will in turn produce its suite of custom data processing tools. Hence PADS now serves as a declarative intermediate language in the tool generation process.

A. Gill and T. Swift (Eds.): PADL 2009, LNCS 5418, pp. 91–106, 2009.

Our past experiments [9] have shown that LEARNPADS is highly effective when the set of tokens it uses matches the tokens used in the unknown data set. For instance, when the unknown data set contains URLs, dates and messages the inference system will work very well when its tokenizer contains the correct corresponding definitions for URLs, dates and messages used in the file. If the tokenizer does not contain these elements, inference is still possible, but the inferred descriptions are generally much more complex than they would be otherwise.

The challenge then is to develop a general-purpose tokenizer containing a wide variety of abstractions like URLs, dates, messages, phone numbers, file paths and more. The key problem is that when using the conventional approach to building a tokenizer (*i.e.,* regular expressions), as we did in our previous work, the definitions of basic tokens overlap tremendously. For example, "`January 24, 2008`" includes a word made up of letters, a couple of numbers, some spaces and English-like punctuation such as the "`,`". Does that mean this string should be treated as an arbitrary text fragment or is it a date? Perhaps "`January`" an element of an string-based enumeration unconnected to integers `24` and `2008`? Perhaps the entire phrase should be merged with surrounding characters rather than treated in isolation? Doing a good job of format inference involves identifying that the string of characters `J-a-n-...-0-8` should be treated as an indivisible token and that it is in fact a date. More generally, an effective format inference engine for ad hoc data solves the *Token Ambiguity Problem* – the problem of determining which substrings of a data file correspond to which token definitions in the presence of syntactic ambiguity.

In this paper, we describe our attempts to solve the token ambiguity problem. In particular, we make the following contributions:

- We redesign our format inference algorithm [9] to take advantage of information generated from an arbitrary statistical token model. This advance allows the algorithm to process a set of ambiguous parses, selecting the most likely parses that match global criteria.
- We instantiate the arbitrary statistical token model with Hidden Markov Models (HMMs), Hierarchical Maximum Entropy Models (HMEMs) and Support Vector Machines (SVMs) and evaluate their relative effectiveness empirically. We also compare the effectiveness of these models to our previous approach, which used regular expressions and conventional prioritized, longest match for disambiguation.
- We augment our algorithm with an additional phase to analyze the complexity of inferred descriptions and to simplify them when description complexity exceeds a threshold relative to the underlying data complexity.

2 The Token Ambiguity Problem

Consider the log files generated by `yum`, a common software package manager. These log files consist of a series of lines, each of which is broken into several distinct fields: date, time, action taken, package name and version. Single spaces separate the fields. For instance:

```
Penum action {
  install Pfrom("Installed");
  update Pfrom("Updated");
  erase Pfrom("Erased");
};
Pstruct version_hdr {
  Pint major; ':';
}
Pstruct sp_version {
  ' ';
  Popt version_hdr h_opt;
  Pid version;
}
```

```
Precord Pstruct entry_t {
        Pdate date;
  ' ';  Ptime time;
  ' ';  action m;
  ": "; Pid package;
        Popt sp_version sv;
};
Psource Parray yum {
        entry_t[];
};
```

Fig. 1. Ideal PADS description of `yum.txt` format

```
May 02 06:19:57 Updated: openssl.i686 0.9.7a-43.8
Jul 16 12:37:13 Erased: dhcp-devel
Dec 10 04:07:51 Updated: openldap.x86_64 2.2.13-4
...
```

Figure 1 shows an *ideal* PADS description of `yum.txt` written by a human expert. The description is structured as a series of C-like type declarations. There are *base types* like `Pdate` (a date), `Ptime` (a time) and `Pint` (an integer). There are also *structured types* such as `Penum` (one of several strings), `Pstruct` (a sequence of items with different types, separated by punctuation symbols), `Popt` (an optional type) and `Parray` (a sequence of items with the same type). PADS descriptions are often easiest read from bottom to top, so the best place to start examining the figure is the last declaration in the right-hand column. There, the declaration says that the entire source file (as indicated by the `Psource` annotation) is an array type called `yum`. The elements of the array are items with type `entry_t`. Next, we can examine the type `entry_t` and observe that it is a new-line terminated record (as indicated by the `Precord` annotation) and it contains a series of fields including a date, followed by a space, followed by a time, followed by an action (which is another user-defined type), followed by a colon and a space, *etc.* We leave the reader to peruse the rest of the figure.

Unfortunately, when we ran our original format inference algorithm [9] on this data source, rather than inferring a compact 23-line description, our algorithm returned a verbose 179-line description that was difficult to understand and even harder to work with. After investigation, we discovered the problem. The data can be tokenized in many ways, and the inference system was using a set of regular expressions to do the tokenization that was a poor match for this data set. More concretely, consider the string "`2.2.13-4`." This string may be parsed by any of the following token sequences:

```
Option 1: [int] [.] [int] [.] [int] [-] [int]
Option 2: [float] [.] [int] [-] [int]
Option 3: [int] [.] [float] [-] [int]
Option 4: [id]
```

The best choice for this format is Option 4, `id`, because `id` can be used to parse the data found at this point in all lines of the `yum` format. Unfortunately, the simplistic disambiguation rules for the original system chose Option 2. Moreover, other lines are tokenized in different ways. For instance, `dhcp-devel`, which also could have been an `id` is tokenized as `[word]` and `0.9.7a-43.8` is tokenized as `[float] [.] [int] [char] [-] [float]`. As each distinct tokenization of similar data regions is introduced, the inference engine attempts to find common patterns and unify them. However, in this case, unification was unsuccessful and the result was an overly complex format.

The original inference algorithm disambiguates between overlapping tokens by using the same strategy as common lexer-generators: It tries each token in a predefined order and picks the first, longest token that matches. While effective for some data sources, this simple policy makes fixed tokenization decisions up front, does not take contextual information into account, and restricts the use of complex tokens like `id`, `url` and `message` that *shadow* simpler ones.

3 The Format Inference Algorithm

Our new format inference algorithm consists of four stages: (1) building a statistical token model from labeled training data; (2) dividing the text into newline-separated *chunks* of data and finding all possible tokenizations of each chunk; (3) inferring a *candidate structure* using the statistical model and the tokenizations; and (4) applying rewriting rules to improve the candidate structure. Because this algorithm shares the general structure of our earlier work [9], we focus on the salient differences here.

Training the statistical models. To speed up the training cycle, we created a tool capable of reading any PADS description and labelling the described data with the tokens specified in the description. This way, all data for which we have PADS descriptions can serve as a training suite. As we add more descriptions, our training data improves. Currently, the training suite is biased towards systems data, and includes tokens for integers, floats, times, dates, IP addresses, hostnames, file paths, URLs, words, ids and punctuation. Parsing of tokens continues to use longest match semantics and hence the string "43.8" can be parsed by sequences such as `[int] [.] [int]` or `[int] [.] [float]` or `[float]`, but not by `[float] [.] [int]` or `[float] [.] [float]`. We have experimented with a number of statistical models for tokenization, which we discuss in Section 4.

Tokenization. When inferring a description, the algorithm computes the set of all possible tokenizations of each data chunk. Because these sequences share subsequences, we organize them into a directed acyclic graph called a SEQSET. For example, Figure 2 shows the SEQSET for the substring "2.2.13-4".

Each edge in the SEQSET represents an occurrence of a token in the data, while each vertex marks a location in the input. If a token edge ends at a vertex v, then v indicates the position immediately after the last character in the token. The first vertex in a SEQSET marks the position before the first character in its outgoing edges.

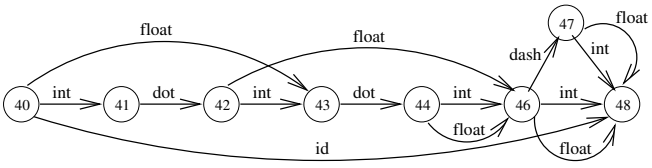


Fig. 2. SEQSET from parsing string "2.2.13-4"

```
type description  (* abstract syntax of pads description *)
type seqset       (* the seqset data structure *)
type seqsets = seqset list

(* A top-level description guess *)
datatype prophecy =
   BaseProphecy   of description
 | StructProphecy of seqsets list
 | ArrayProphecy  of seqsets * seqsets * seqsets
 | UnionProphecy  of seqsets list

(* Guesses the best top-level description *)
fun oracle : seqsets -> prophecy

(* Implements a generic inference algorithm *)
fun discover (sqs:seqsets) : description =
 case (oracle sqs) of
   BaseProphecy b => b

 | StructProphecy sqss =>
     let Ts = map discover sqss in
     struct { Ts }

 | ArrayProphecy (sqsfirst,sqsbody,sqslast) =>
     let Tfirst = discover sqsfirst in
     let Tbody  = discover sqsbody  in
     let Tlast  = discover sqslast  in
     struct { Tfirst; array { Tbody }; Tlast; }

 | UnionProphecy sqss =>
     let Ts = map discover sqss in
     union { Ts }
```

Fig. 3. A generic structure-discovery algorithm in Pseudo-ML

Structure discovery. The structure discovery phase uses a *top-down*, *divide-and-conquer* algorithm outlined in Figure 3 in the pseudo-ML function `discover`. Each invocation of `discover` calls the `oracle` function to guess the structure of the data represented by the current set of SEQSETs. The oracle can prophesy either a *base type*, a *struct*, an *array* or a *union*. The `oracle` function also partitions the input SEQSETs into sets

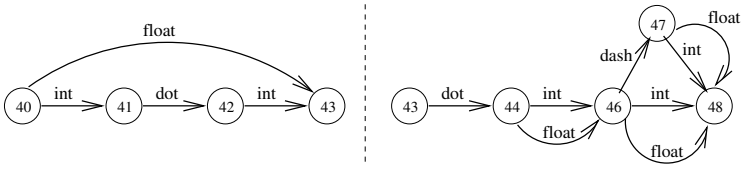


Fig. 4. Cutting SEQSET for "2.2.13-4" after the first float token

of sub-SEQSETs, each of which corresponds to a component in the guessed structure. The `discover` function then recursively constructs the structure of each set of sub-SEQSETs.

How does the `oracle` produce its prophecy? First, it uses the trained statistical model to assign probabilities to the edges in the input SEQSETs. Next, it computes for each SEQSET the *most probable token sequence* (MPTS) among all the possible paths using a modified *Viterbi* algorithm [22], which we discuss in Section 4. Then, based on the statistics of the tokens in the MPTSs, the oracle predicts the structure of the current collection of SEQSETs using the heuristics designed for our earlier algorithm [9].

As an example, consider applying the oracle to determine the top-level structure of the first line in `yum.txt`. It would predict the following:

```
struct {date;  ' '; time; ' '; word; ':'; ' '; id; TBD}
```

i.e., a `struct` containing nine sub-structures including `TBD`, which is a sub-structure whose form will be determined recursively. At this point, the `oracle` partitions every SEQSET in the input into nine parts, corresponding to sub-structure boundaries, *i.e.*, at the vertices after tokens `date`, `space`, `time`, *etc.* During partitioning, the oracle removes SEQSET edges that cross partition boundaries because such edges are irrelevant for the next round of structure discovery. For example, if the oracle cuts after the first `float` token in the SEQSET in Figure 2, then it removes the `id` edge and the `float` edge between vertices 42 and 46, creating the two new SEQSETs in Figure 4. Finally, the `oracle` function returns the predicted structure as a "prophecy" along with the partitioned SEQSETs.

Format refinement with blob-finding. The refinement phase, which follows structure discovery, tries to improve the initial rough structure by applying a series of rewriting rules. We have modified the earlier algorithm to use a "blob-finding" rule. This rule tries to identify data segments with highly complex, structured descriptions where none of the individual pieces of the description describe much of the data. Intuitively, such occurrences correspond to places where the data contained a high degree of variation, and the inference algorithm built a description that enumerated all the possible variations in painstaking detail. The blob rule replaces such complexity with a single *blob* token. A typical example of this kind of data is free-form text comments that sometimes appear at the end of each line in a log file. The blob-finding rule reduces the overall complexity of the resulting description and hence makes it more readable.

The format refinement algorithm applies the blob-finding rule in a bottom-up fashion. It converts into a blob each sub-structure that it deems overly complex and for

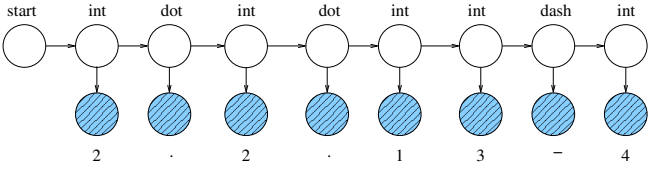
which it can find a terminating pattern. The PADS parser uses the terminating pattern to find the extent of the blob. The algorithm merges adjacent blobs.

To decide whether a given structure is a blob, the algorithm computes the *variance* of the structure, which measures the total number of union/switch/enum branches and different array lengths in the structure. When the ratio between the variance and the amount of the data described by the structure exceeds a threshold, the algorithm decides to convert the structure to a blob if it can find a terminating sequence.

4 Statistical Models

A key component of the format inference algorithm described in the previous section is a selection of the best token sequence from each SEQSET. To prioritize sequences, the algorithm assigns probabilities using a statistical token model. This section describes three such models that we have experimented with.

Character-by-character Hidden Markov Model (HMM). The first model we investigate is the classic first-order, character-by-character Hidden Markov Model (HMM) [22]. An HMM is a statistical model that includes one set of states whose values we can observe and a second set whose values are *hidden* and we wish to infer. The hidden states determine, with some probability, the values of the observable states. In our case, we can observe the sequence of characters in the input string and wish to infer the token that is associated with each character. The model assumes the probability that we see a particular character depends upon its associated token and moreover, since the HMM is first-order, the probability of observing a particular token depends upon the previous token but no other earlier tokens. The picture below illustrates the process of generating the character sequence "2.2.13-4" from a token sequence. Hidden HMM states are white and observables are shaded. Notice particularly that the adjacent digits "1" and "3" are generated from two consecutive instances of the token `int`, when in a true token sequence, both characters are generated from a single `int` token. A postpass will clean this up, but such situations are dealt with more effectively by the HMEMs described in the following subsection.



Finally, since our training data is limited, we employ one further approximation in our model. Instead of modelling every individual character separately, we classify characters using a set of boolean features including features for whether the character is (a) a digit, (b) an upper-case alphabetic letter, (c) white space, or (d) a particular punctuation character such as a period. We call the feature vectors involving (a)-(d) *observations*.

Let $\mathbf{T}_i$ denote the i^{th} hidden state; its value ranges over the set of all token names. Let $\mathbf{C}_i$ denote the observation emitted by hidden state $\mathbf{T}_i$. Three parameters determine

the model: the transition matrix $\mathbf{P}(T_i|T_{i-1})$, the sensor matrix $\mathbf{P}(C_i|T_i)$ and the initial probabilities $\mathbf{P}(T_i|begin)$. We compute these parameters from the training data as follows:

$$\mathbf{P}(T_i|T_{i-1}) = \frac{\text{occurrences where } T_i \text{ follows } T_{i-1}}{\text{occurrences of } T_{i-1}} \tag{1}$$

$$\mathbf{P}(C_i|T_i) = \frac{\text{occurrences of } C_i \text{ annotated with } T_i}{\text{occurrences of } T_i} \tag{2}$$

$$\mathbf{P}(T_1|begin) = \frac{\text{occurrences of } T_1 \text{ being first token}}{\text{number of training chunks}} \tag{3}$$

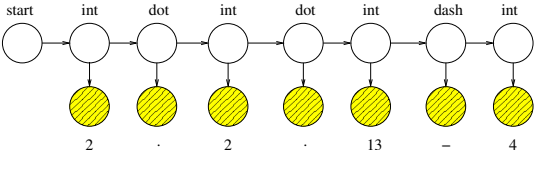
Given these parameters and a fixed input, we want to find the token sequence with the highest probability, *i.e.*, from the input sequence $C_1, C_2, ..., C_n$, we want to find the token sequence $T_1, T_2, ..., T_n$ that maximizes the conditional probability $\mathbf{P}(T_1, T_2, ..., T_n|C_1, C_2, ..., C_n)$. This probability is defined as usual:

$$\begin{aligned}\mathbf{P}(T_1, T_2, ..., T_n|C_1, C_2, ..., C_n) &\propto \mathbf{P}(T_1, T_2, ..., T_n, C_1, C_2, ..., C_n) \\ &= \mathbf{P}(T_1|begin) \cdot \prod_{i=2}^{n} \mathbf{P}(T_i|T_{i-1})\end{aligned} \tag{4}$$

To calculate the highest probability token sequence from this model, we run a slightly modified variant of the Viterbi algorithm over the SEQSET.

Because the character-by-character HMM is first-order and employs only single character features, it cannot capture complex features in the data such as a substring "`http://`" which indicates a strong likelihood of being part of a URL. One obvious solution is increasing the order of the HMM. However, since the token length is variable in our application, it is not clear what the order should be. In addition, increasing the order also increases the complexity exponentially. Instead, in the next sections, we pursue two hybrid methods that incorporate existing classification techniques into the HMM framework.

Hierarchical Maximum Entropy Model (HMEM). The character-by-character HMM extracts a set of features from each character to create an observation and then runs a standard HMM over these observations. In contrast, the Hierarchical Maximum Entropy Model (HMEM), which we will explore next, extracts a set of features from each substring, uses the Maximum Entropy (ME) procedure [24,19] to produce an observation and runs a standard HMM over these new kinds of observations. Using the sequence "2.2.13-4" as our example again, the corresponding HMEM may be drawn as follows:



Formally, let $\mathbf{T}_i$ be the i^{th} hidden state or token in the sequence (denoted by a white node in picture above) and let $\mathbf{S}_i$ be the substring annotated by $\mathbf{T}_i$. Suppose the number of tokens in the chunk is l; then the target probability is as follows.

$$\mathbf{P}(T_1, T_2, ..., T_n|S_1, S_2, ..., S_l) \propto \mathbf{P}(T_1|begin) \cdot \prod_{i=2}^{l} \mathbf{P}(T_i|T_{i-1}) \cdot \prod_{i=1}^{l} \mathbf{P}(S_i|T_i) \quad (5)$$

Equations (1) and (3) allow us to calculate the transition matrix and the initial probability. We can compute $\mathbf{P}(S_i|T_i)$ using Bayes Rule,

$$\mathbf{P}(S_i|T_i) = \frac{\mathbf{P}(T_i|S_i) \cdot \mathbf{P}(S_i)}{\mathbf{P}(T_i)} \quad (6)$$

Finally, since obtaining accurate estimates of $\mathbf{P}(S_i)$ and $\mathbf{P}(T_i)$ appears to require more training data than we currently have, we have further approximated by simply using $\mathbf{P}(T_i|S_i)$ to estimate $\mathbf{P}(S_i|T_i)$. Estimation of $P(T_i|S_i)$ through the ME procedure involves using the following features (among others): (a) total number of characters in the string, (b) the number of occurrences of certain punctuation characters, (c) the total number of punctuation characters in the string, (d) the presence of certain substrings such as "`am`", "`pm`", "`January`", "`Jan`", "`january`", and (e) the presence of digit sequences. When we substitute $\mathbf{P}(T_i|S_i)$ for $\mathbf{P}(S_i|T_i)$ in equation (5), we obtain the following:

$$\mathbf{P}(T_1, T_2, ..., T_n|S_1, S_2, ..., S_l) \propto \mathbf{P}(T_1|begin) \cdot \prod_{i=2}^{l} \mathbf{P}(T_i|T_{i-1}) \cdot \prod_{i=1}^{l} \mathbf{P}(T_i|S_i) \quad (7)$$

Finally, notice that in equation (7), the number of tokens in a sequence will determine the number of terms in the product. Consequently, a sequence with more tokens will produce more terms, which our experiments have shown produces a significant bias towards shorter token sequences. To avoid such bias, we modify Equation (7) to use the average log likelihood.

$$\begin{aligned} &\log \mathbf{P}(T_1, T_2, ..., T_n|S_1, S_2, ..., S_l) \\ &\propto \frac{\log \mathbf{P}(T_1|begin) + \sum_{i=2}^{l} \log \mathbf{P}(T_i|T_{i-1}) + \sum_{i=1}^{l} \log \mathbf{P}(T_i|S_i)}{l} \end{aligned} \quad (8)$$

Using average log likelihood guarantees that the algorithm will not select shorter token sequences unless the average value of all conditional probabilities $\mathbf{P}(T_i|S_i)$ exceeds a threshold.

To find the highest probability sequence for a chunk under this model, we implemented a modified Viterbi algorithm that takes into account the number of tokens in the sequence. In what follows, let the number of characters in the chunk be n and the number of tokens be l. Let C_i be the character at position i, and PT_i be the partial token that emits the character C_i. Then $\mathbf{P}(PT_1, PT_2, ..., PT_i|C_1, C_2, ..., C_i, k)$ is the probability of a partial token sequence $PT_1, PT_2, ..., PT_i$ conditioned on a substring of characters $C_1, C_2, ..., C_i$, collectively emitted by a sequence of k tokens. Now, let T_i be a token

that ends at position i and let S_i be the corresponding substring. The probability of the most likely partial token sequence up to position i is

$$\max_{PT_1,...,PT_i} \log \mathbf{P}(PT_1, PT_2, ..., PT_i, PT_{i+1}|C_1, C_2, ..., C_{i+1}, k+1) \propto$$

$$\begin{cases} \log \mathbf{P}(S_{i+1}|T_{i+1}) + \max\limits_{T_{i+1-\delta}} (\log \mathbf{P}(T_{i+1}|T_{i+1-\delta}) + \\ \qquad \max\limits_{PT_1,...,PT_{i-1}} \log \mathbf{P}(PT_1, ..., PT_i|C_1, ..., C_i, k)), \\ \qquad \text{if } i+1 \text{ is the end of an edge in SEQSET, } \delta \text{ is the length of token } T_{i+1}; \\ \max\limits_{PT1,...,PT_i} \log \mathbf{P}(PT_1, ..., PT_i|C1, ..., C_i, k+1) \\ \qquad \text{otherwise.} \end{cases} \quad (9)$$

The left-hand-side of (9), known as a *forward message*, contains the token sequence up to a position i in the chunk as well as the lengths of the tokens. At the last position n, we compute l from

$$\max_{l} \ \log \frac{\mathbf{P}(TP_1, TP_2, ..., TP_n|C_1, C_2, ..., C_n, l)}{l} \quad (10)$$

and select the last token in the most likely token sequences. After tracing backwards through the chain of messages, we obtain the most likely token sequences. The modified Viterbi algorithm is linear to the number of characters n in the chunk.

We saw there were some problems with the basic HMM model that motivated the use of the HMEM model. What further problems plague the HMEMs? The most worrisome problem is that the HMEM is a generative model that simulates the procedure of generating the data, and estimates the target conditional probability by a joint probability. Therefore, it is biased towards tokens with more occurrences in the training data. In practice, we found that when particular tokens appear infrequently in our training data, the algorithm would never identify them, even when they had clear distinguishing features. These difficulties motivated us to explore the effectiveness of Hierarchical Support Vector Machines (HSVM), which use a discriminative model as opposed to a generative one.

4.1 Hierarchical Support Vector Machines (HSVM)

An HSVM is exactly the same as an HMEM except it uses a Support Vector Machine (SVM) [5] as opposed to Maximum Entropy to classify tokens. Basically, an SVM measures the target conditional probability $\mathbf{P}(T_i|S_i)$ by generating hyperplanes that divide the feature vector space according to the positions of training data points. The hyperplanes are positioned so that the data points (feature vectors in our case) are separated into classes with the maximum margin between any two classes. The data points that lie on the margins (or boundaries) of each class are called *support vectors*.

5 Evaluation

We use sample files from twenty different ad hoc data sources to evaluate our overall inference algorithm and the different approaches to probabilistic tokenization. These data

sources, many of which are published on the web [20], are mostly system-generated log files of various kinds and a few ASCII spreadsheets describing business transactions. These files range in size from a few dozen lines to a few thousand.

To test a given tokenization approach on a particular sample file, we first construct a statistical model from the other nineteen sample files using the given approach. We then use the resulting model to infer a description for the selected file. We repeat this process for all three tokenization approaches (HMM, HMEM, and HSVM) and all twenty sample files. We use three metrics described in the following sections to evaluate the results: *token accuracy*, *quality of description* and *execution time*.

Token accuracy. To evaluate tokenization accuracy for a model M on a given sample file, we compare the most likely sequence of tokens predicted by M, denoted S_m, with the ideal token sequence, denoted S. We define S to be the sequence of tokens generated by the hand-written PADS description of the file. We define three kinds of error rates, all normalized by $|S|$, the total number of tokens in S:

$$\text{token error} = \frac{\text{number of misidentified tokens in } S_m}{|S|}$$

$$\text{token group error} = \frac{\text{number of misidentified groups in } S_m}{|S|}$$

$$\text{token boundary error} = \frac{\text{number of misidentified boundaries in } S_m}{|S|}$$

The token error rate measures the number of times a token appears in S but the same token does not appear in the same place in S_m. A *token group* is a set of token types that have similar feature vectors and hence are hard to distinguish, *e.g.*, `hex string` and `id`, which both consist of alpha-numeric characters. The token group error rate measures the number of times a token from a particular token group appears in S but no token from the same group appears in the same location in S_m. Intuitively, if the algorithm mistakes a token for another token in the same token group, it is doing better than choosing a completely unrelated token type. The *token boundary* error rate measures the number of times there is a boundary between tokens in S but no corresponding boundary in S_m. This relatively coarse measure is interesting because boundaries are important to structure discovery. Even if the tokens are incorrectly identified, if the boundaries are correct, the correct structure can be still discovered.

Table 1 lists the token error, token group error, and token boundary error rates of the twenty benchmarks. The results from the original LEARNPADS system are presented in columns marked by `lex`. The original system produces high error rates for many files because the lexer is unable to resolve overlapping tokens effectively. HMM relies heavily on transition probabilities, which require a lot of data to compute to a useful precision. Because we currently have insufficient data, HMM generally does not perform as well as HMEM and HSVM. In the case of `asl.log`, `corald.log` and `coralwebsrv.log`, HMM's failure to detect some punctuation characters causes the entire token sequences to be misaligned and hence gives very high error rates.

Table 1. Tokenization errors

Data source	Token Error (%)				Token Group Error (%)				Token Boundary Error (%)			
	lex	HMM	HMEM	HSVM	lex	HMM	HMEM	HSVM	lex	HMM	HMEM	HSVM
1967Transactions	30	30	18.93	18.93	11.07	11.07	0	0	11.07	11.07	0	0
ai.3000	70.23	15.79	18.98	11.20	70.23	14.68	17.26	10.27	53.53	12.34	4.79	4.00
yum.txt	19.44	13.33	21.80	0	19.17	11.73	21.80	0	19.17	11.49	21.80	0
rpmpkgs.txt	99.66	2.71	15.01	0.34	99.66	2.14	14.67	0	99.66	0.23	14.67	0
railroad.txt	51.94	9.47	6.48	5.58	51.94	9.36	5.93	5.58	46.08	8.77	5.41	5.58
dibbler.1000	15.72	43.40	11.91	0.00	15.72	36.78	11.91	0.00	4.54	13.33	13.15	0.00
asl.log	89.92	98.91	8.94	5.83	89.63	98.91	8.94	5.83	83.28	98.54	6.27	3.29
scrollkeeper.log	18.58	28.48	18.67	9.86	18.58	18.77	8.96	0.12	18.58	17.83	8.96	0.12
page_log	77.72	15.29	0	7.52	72.76	15.29	0	7.52	64.70	5.64	0	5.64
MER_T01_01.csv	84.56	23.09	31.32	15.40	84.56	23.09	31.22	15.40	84.56	7.71	13.20	0.02
crashreporter	51.89	7.91	4.99	0.19	51.85	7.91	4.96	0.14	51.34	7.91	4.92	0.14
ls-l.txt	33.73	18.70	19.96	6.65	33.73	18.23	19.96	6.65	19.70	7.45	19.76	6.45
windowserver_last	73.31	14.98	10.16	3.24	71.50	14.98	10.07	3.15	69.18	11.16	8.05	3.14
netstat-an	13.89	17.83	9.61	9.01	12.51	15.44	5.95	5.95	12.51	14.90	5.80	5.20
boot.txt	10.67	25.40	9.37	2.77	3.99	25.10	9.14	2.43	3.34	14.48	8.27	1.69
quarterlyincome	82.99	5.52	1.98	1.98	82.99	4.22	1.53	1.54	77.53	1.54	1.53	1.54
corald.log	84.86	100	5.67	3.02	83.11	98.25	3.93	1.27	81.76	97.80	1.27	1.27
coraldnssrv.log	91.04	18.17	10.64	5.23	91.04	18.17	9.33	5.22	83.07	14.37	4.11	3.92
probed.log	1.74	27.99	16.50	16.50	1.74	27.99	16.50	16.50	1.75	27.98	16.42	16.42
coralwebsrv.log	86.67	100	8.75	23.99	86.67	100	8.75	23.99	81.90	98.33	8.75	23.81

Quality of description. To assess description quality quantitatively, we use the *Minimum Description Length Principle* (MDL) [13], which postulates that a useful measure of description quality is the sum of the cost in bits of transmitting the description (the type cost) and the cost in bits of transmitting the data *given the description* (the data cost). In general, the type cost measures the complexity of the description, while the data cost measures how loosely a given description explains the data. Increasing the type cost generally reduces the data cost, and *vice versa*. The objective is to minimize both. Table 2 shows the percentage change in the type and data costs of the descriptions produced by the new algorithm using each of the three tokenization schemes when compared to the same costs produced by the original LEARNPADS system. In both cases, the measurements were taken before the refinement case.

For most of the data sources, the probabilistic tokenization scheme improved the quality of the description by reducing both the type and the data costs. In the files `dibbler.1000`, `netstat-an` and `coralwebsrv.log`, a few misidentified tokens cause the resulting descriptions to differ significantly from the ones produced by the original system.

In another experiment, a human expert judged how each description compared to the original LEARNPADS results, focusing on the readability of the descriptions, *i.e.*, whether the descriptions present the structure of the data sources clearly. In this experiment, the judge rated the descriptions one by one, on a scale from -2 (meaning the description is too concise and it loses much useful information) to 2 (meaning the description is too precise and the structure is unclear). The score of a good description is therefore close to 0, which means the description provides sufficient information for

Table 2. Increase (+%) or decrease (-%) in type cost and data cost before refinement

Data source	Type Cost			Data Cost		
	HMM	HMEM	HSVM	HMM	HMEM	HSVM
1967Transactions	-39.661	-27.03	-27.03	-2.80	-2.80	-2.80
ai.3000	-26.27	+4.44	-19.27	-3.16	-6.85	-12.68
yum.txt	-57.60	+50.93	-76.27	-1.55	-7.93	-1.05
rpmpkgs.txt	-92.03	-76.29	-91.86	+1.47	-0.00	+1.47
railroad.txt	-31.86	-20.88	-22.93	-29.54	-29.22	-29.16
dibbler.1000	+611.22	+17.83	+7.03	-19.88	-22.11	-22.10
asl.log	-75.71	-22.33	-25.54	+8.57	-15.13	-17.53
scrollkeeper.log	-14.55	-58.86	-21.18	-7.77	-9.98	-11.36
page_log	0	0	0	-11.46	-11.67	-11.67
MER_T01_01.csv	-8.59	-12.74	-12.74	-25.59	-24.15	-24.14
crashreporter	+4.03	-8.66	-12.73	-9.38	-9.41	-12.45
ls-l.txt	-74.61	-51.32	-39.30	+0.10	-7.26	-2.18
windowserver_last	-62.84	-33.29	-56.18	+6.93	-11.12	-9.87
netstat-an	+147.07	-12.00	-21.63	+14.18	+6.74	+7.65
boot.txt	-72.60	-38.95	-71.29	+5.26	-6.54	-5.03
quarterlyincome	-18.36	-18.36	-18.36	-32.04	-32.51	-32.51
corald.log	-4.75	-5.53	-5.53	-27.28	-29.81	-29.81
coraldnssrv.log	-1.86	-2.03	-5.86	+59.53	+59.53	+59.53
probed.log	-14.61	-33.48	-33.48	+59.53	+63.18	+63.18
coralwebsrv.log	-8.75	+94.58	-71.55	-49.30	-15.91	+13.36

Table 3. Qualitative comparison of descriptions learned using probabilistic tokenization to descriptions learned by original LEARNPADS algorithm

Data source	lex	HMM	HMEM	HSVM	Data source	lex	HMM	HMEM	HSVM
1967Transactions	0	0	0	0	crashreporter	2	0	1	1
ai.3000	1	1	1	0	ls-l.txt	2	0	1	1
yum.txt	2	-1	1	0	windowserver_last	2	0	1	1
rpmpkgs.txt	2	-1	-2	0	netstat-an	2	-2	0	0
railroad.txt	2	1	1	1	boot.txt	2	-1	1	1
dibbler.1000	0	2	0	0	quarterlyincome	1	1	1	1
asl.log	2	-2	2	2	corald.log	0	1	1	0
scrollkeeper.log	1	2	1	1	coraldnssrv.log	0	1	1	-1
page_log	0	0	0	0	probed.log	0	0	0	0
MER_T01_01.csv	0	1	0	0	coralwebsrv.log	0	1	1	-1

the user to understand the data source and the user can easily understand the structure from the description. Table 3 shows that on average, HMEM and HSVM outperform the original system denoted by `lex`.

Execution time. Compared to the original system, statistical inference requires extra time to construct SEQSETs and compute probabilities. We measured the execution times on a 2.2 GHz Intel Xeon processor with 5 GB of memory. The original algorithm takes

anywhere from under 10 seconds to 25 minutes to infer a description, while the new system requires a few seconds to several hours, depending on the amount of test data and the statistical model used. In general, the character-by-character HMM model is the fastest, while HSVM is most time-consuming.

We have performed a number of experiments (not shown due to space constraints) that demonstrate that execution time is proportional to the number of lines in the data source. Moreover, we have found that for most descriptions, a relatively small representative sample of the data is sufficient for learning its structure with high accuracy. For instance, out of the twenty benchmarks we have, seven data sources have more than 500 records. Preliminary results show that for these seven data sources, we can generate descriptions from just 10% of the data that can parse 95% of records correctly.

6 Related Work

In the last two decades, there has been extensive work on classic grammar induction problems [25,11,3,1,6], XML schema inference [3,10], information extraction [17,15,2], and other related areas such as natural language processing [4,14] and bioinformatics [16]. Machine learning techniques have played a very important role in these areas. Our earlier paper [9] contains an extensive comparison of our basic format inference algorithm to others that have appeared in the literature.

One of the most closely related pieces of work to this paper is Soderland's WHISK system [23], which extracts useful information from semi-structured text such as stylized advertisements from an online community service called Craig's List [12]. In the WHISK system, the user is presented with a few online ads as training data and is asked to label which bits of information to extract. Then the system learns extraction rules from labeled data and uses them to retrieve more wanted information from a much larger collection of data. The WHISK system differs from our system in several ways. First, WHISK, as well as other information extraction systems, have a clear and fixed token set, defined by words, numbers, punctuations, HTML tags and user pre-specified semantic classes, etc. Second, WHISK only focuses on certain bits of information, namely, single or multiple fields in records, whereas we not only identify useful fields, but also obtain the organization and relations of these fields by generating the complete description of the entire data file. Last, in WHISK, the extraction rules learned from a particular domain can only be used on data from the same domain. For example, rules learned from sample on-line rental ads are only relevant to other rental ads, and cannot be applied to software job postings. But the statistical token models we learned in our system can be applied to many different types of data, as shown in the experiments we have done in Section 5.

Also closely related is the work on text table extraction by Pinto and others [21]. Text tables can be viewed as special ad hoc data with a tabular layout. There are often clear delimiters between columns in the table, and table rows are well defined with new line characters as their boundaries. Because of its tabular nature, the data studied has less variation in general. The goal of their work is to identify tables embedded in free text and the types of table rows such as header, sub-header and data row, etc, whereas we are learning the entire structure of the data. To this end, Pinto et al. use Conditional Random

Fields (CRFs) [18], a statistical model that is useful in learning from sequence data with overlapping features. Their system extracts features from white space characters, text between white spaces and punctuations. Although not explicitly stated, words, numbers and punctuations are used as fixed set of tokens.

To summarize, problems studied by previous efforts in grammar induction and information extraction do not typically suffer from token ambiguities that we see in ad hoc data, because tags cleanly divide XML and web-based data, while spaces and known punctuation symbols separate natural language text. In contrast, the separators and token types found in ad hoc data sources such as web logs and financial records are far more variable and ambiguous.

7 Conclusion

Ad hoc data is unpredictable, poorly documented, filled with errors, and yet ubiquitous. It poses tremendous challenges to the data analysts that must analyze, vet and transform it into useful information. Our goal is to alleviate the burden, risk and confusion associated with ad hoc data by using the declarative PADS language and system.

In this paper, we describe our continuing efforts to develop a format inference engine for the PADS language. In particular, we show how to redesign our format inference algorithm so that it can take advantage of information generated from an arbitrary statistical token model and we study the effectiveness of three candidate models: Hidden Markov Models (HMMs), Hierarchical Maximum Entropy Models (HMEMs) and Support Vector Machines (SVMs). We show that each model in succession is generally more accurate than the last, but at an increased performance cost.

Acknowledgement. This material is based upon work supported by the NSF under grants 0612147 and 0615062. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Angluin, D.: Inference of reversible languages. Journal of the ACM 29(3), 741–765 (1982)
2. Arasu, A., Garcia-Molina, H.: Extracting structured data from web pages. In: SIGMOD, pp. 337–348 (2003)
3. Bex, G.J., Neven, F., Schwentick, T., Tuyls, K.: Inference of concise DTDs from XML data. In: VLDB, pp. 115–126 (2006)
4. Borkar, V., Deshmukh, K., Sarawagi, S.: Automatic segmentation of text into structured records. In: SIGMOD, New York, NY, USA, pp. 175–186 (2001)
5. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines. Software (2001), `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
6. Chen, S.F.: Bayesian grammar induction for language modeling. In: Proceedings of the 33rd Annual Meeting of the ACL, pp. 228–235 (1995)
7. Fisher, K., Gruber, R.: PADS: A domain specific language for processing ad hoc data. In: PLDI, pp. 295–304 (June 2005)

8. Fisher, K., Walker, D., Zhu, K.Q.: LearnPADS: Automatic tool generation from ad hoc data. In: SIGMOD (June 2008)
9. Fisher, K., Walker, D., Zhu, K.Q., White, P.: From dirt to shovels: Fully automatic tool generation from ad hoc data. In: POPL (January 2008)
10. Garofalakis, M.N., Gionis, A., Rastogi, R., Seshadri, S., Shim, K.: XTRACT: A system for extracting document type descriptors from XML documents. In: SIGMOD, pp. 165–176 (2000)
11. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
12. Craig's List (2008), http://www.craigslist.org/
13. Grünwald, P.D.: The Minimum Description Length Principle. MIT Press, Cambridge (2007)
14. Heeman, P.A., Allen, J.F.: Speech repairs, intonational phrases and discourse markers: Modeling speakers' utterances in spoken dialog. Computational Linguistics 25(4), 527–571 (1999)
15. Hong, T.W.: Grammatical Inference for Information Extraction and Visualisation on the Web. Ph.D. Thesis, Imperial College, London (2002)
16. Kulp, D., Haussler, D., Reese, M.G., Eeckman, F.H.: A generalized hidden markov model for the recognition of human genes in DNA. In: Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology, pp. 134–141 (1996)
17. Kushmerick, N.: Wrapper induction for information extraction. PhD thesis, University of Washington, Department of Computer Science and Engineering (1997)
18. Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: ICML, pp. 282–289 (2001)
19. MEGA model optimization package (2007), http://www.cs.utah.edu/~hal/megam/
20. PADS project (2007), http://www.padsproj.org/
21. Pinto, D., McCallum, A., Wei, X., Croft, W.B.: Table extraction using conditional random fields. In: SIGIR, New York, NY, USA, pp. 235–242 (2003)
22. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2) (February 1989)
23. Soderland, S.: Learning information extraction rules for semi-structured and free text. Machine Learning 34(1-3), 233–272 (1999)
24. Adam, L., Berger, T., Vincent, J., Della Pietra, Stephen, A.: A maximum entropy approach to natural language processing. Computational Linguistics 22(1) (March 1996)
25. Vidal, E.: Grammatical inference: An introduction survey. In: ICGI, pp. 1–4 (1994)