Predicting Structure in Handwritten Algebra Data From Low Level Features

James A. Duyck
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jduyck@andrew.cmu.edu

Geoffrey J. Gordon
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
ggordon@cs.cmu.edu

Abstract

Background A goal in computer vision and natural language processing is to predict high level structure over handwritten algebra data. This is typically done by clustering or otherwise identifying symbols, then applying a known or learned grammar to generate dependency trees. However, these methods are often not generic and require domain knowledge.

Aim The goal of this paper is to determine whether it is practical to predict dependency trees from low level features of handwritten algebra data without the use of symbol identification, and to determine which algorithms are best suited for this task.

Data Our data consists of 400 pages of handwritten algebra data collected by one of our researchers.

Methods We use transition-based parsing to generate dependency trees, and various no-regret online imitation learning algorithms to predict transitions.

Results We find that a simple algorithm called greedy transition-based parsing, which has regretful learning and greedy decoding, gives similar or better results compared to the no-regret algorithms. This is surprising, as theory suggests no-regret algorithms should outperform regretful algorithms. We discuss possible explanations for this discrepancy.

Conclusions We conclude that it is practical to predict dependency trees over handwritten algebra data without the use of symbol identification.

1 Introduction

In problems involving handwriting, there has been a large focus on clustering. The MNIST database of handwritten digits [1] lists many approaches for clustering handwritten digits, with error rates as low as 0.23%. Similarly common are parsers that derive high order structure such as dependency trees using indicators of symbols or words as features. Examples of such features include the CoNLL datasets [2] and the Penn Treebank [3]. This paper focuses on a less well-studied topic, that of learning high order structure from low level features.

In this paper we examine handwritten algebraic equations, and attempt to find dependency trees. Rather than clustering symbols and and applying parsing algorithms to the cluster identities, we apply algorithms for finding dependency trees to low level features. We chose this model for several reasons. First of all, this model is not well studied. Most models in the literature follow either a pipeline approach, first performing recognition and then parsing, or a joint model where recognition and parsing are simultaneously performed. By approaching the problem from a different perspective, and attempting to learn structure without explicit recognition, we hope to gain new insight into the properties of the data. Additionally, models that perform recognition and parsing separately

typically require both supervision and domain knowledge. Human effort is usually required to label the symbols to be recognized, and domain knowledge is often used to construct grammars used in parsing. By studying models that do not perform explicit recognition, we hope to eventually build models that require less supervision and less domain knowledge. This will lead to systems with more general applications.

We chose to study dependency trees because they are a straightforward data structure to work with, which still being structurally interesting. There exists an easily implemented parsing method called transition-based parsing, described in section 3.1, which can represent a dependency tree as a time series of transitions between states. These states and transitions provide natural features and outputs for online learning algorithms, including both regretful and no-regret algorithms.

1.1 Research goals

Our research group is interested in an important problem in AI research: to uncover high level structure and make predictions from low level features. In particular, we are interested in time series data. We hope to develop generic structure prediction algorithms that require less supervision and less domain knowledge.

The goal of this paper is to determine whether it is feasible to predict structure from low level features, and to identify the best algorithms for doing so. We examine whether it is practical to find dependency trees over handwritten algebra data without explicit symbol recognition. We conclude that training a greedy transition-based parsing, a regretful algorithm in which we train a support vector machine to predict transitions and use greedy decoding, gives similar or better results compared to the no-regret approaches we examine. This is theoretically surprising, as no-regret algorithms provide guarantees that regretful algorithms do not, but reflects other results in parsing showing that simple methods can be more effective [4] than more complex methods.

This work may have applications in auto-completion of handwriting. In future work, we hope to use learned structure to predict future features of time series data. This work also has applications in education, where a problem of interest is to learn how students solve problems. High level, representative models are necessary to understand the complex process of learning. We hope to use the high level structures we predict as a basis for modeling human learning.

1.2 Paper structure

This paper is intended to be self-contained, so we summarize some other papers when describing the algorithms that we use. First we present a summary of related work in section 2. Next we describe the algorithms and approaches that we implement in section 3, including a discussion of transition-based parsing systems in subsection 3.1, a discussion of imitation learning algorithms for predicting transitions in subsection 3.2, and a discussion of efficiency in terms of time complexity in subsection 3.3. In section 4 we describe in detail our dataset, and in section 5 we describe the experiments we performed. We present our results in section 6, and discuss the implications of those results in section 7.

2 Related work

Typical systems for parsing handwriting proceed by performing some form of recognition on characters, then determining some sort of structure over over the characters, usually incorporating some form of domain knowledge. An example of such a system is presented by Belaid and Haton (1984) [5]. They identify characters using domain knowledge, for example the locations where strokes cross. They then interpret the symbols using a grammar constructed with domain knowledge of mathematics. Garain and Chaudhuri (2004) [6] employ a similar method.

Belaid and Haton use the interpretation step to correct errors in the identification step by ensuring that a set of character identities is in the language of the grammar. Similar corrections are also common in the literature. Toyozumi et al. (2004) [7] use a similar method to improve segmentation — separating pen strokes from different symbols without breaking a symbol into multiple parts — by calculating probabilities using a predefined grammar. Shi et al. (2007) [8] and Luo et al.

(2008) [9] use graphical models over strokes to find the likeliest symbol segmentation and identities prior to performing structure recognition.

Another common area of research is simultaneously performing symbol recognition and syntax analysis. Such methods are presented by Yamamoto et al. (2006) [10] and Awal et al. (2009) [11], (2014) [12].

State-of-the-art methods typically follow the same pattern. Álvaro et al. (2014) [13] use hidden Markov models for symbol recognition and stochastic 2D context-free grammars for structure recognition, achieving best-in-class at the CROHME 2011 Competition on Recognition of Online Handwritten Mathematical Expressions [14].

This paper attempts to recognize structure without explicitly performing symbol recognition, a problem that is not well studied in the literature.

3 Approaches

In this work, we attempt to predict dependency trees from handwriting features. Rather than predicting a dependency tree directly, we examine **transition-based dependency parsing** [15]. In this model a sequence of transitions, for example setting one symbol to be the parent of another, is applied to a sequence of words or symbols.

We predict the sequence of predictions via **imitation learning**. In imitation learning, an **expert** provides demonstrations of good behavior, which are used to control the behavior of a system [16]. Our goal is to train classifiers to predict the next transition, given the features of symbols and expert-provided transitions for sample features. The algorithms we describe are online in the sense that they predict each transition after the previous one is applied, but we allow the use of future symbols as features of the classifiers.

Our system then consists of two components. The first component is a transition-based dependency parsing systems. This is used as an expert that provides the training classes for our classifiers. We implement two transition systems described by Goldberg and Nivre: the **arc-eager system** and the **arc-hybrid system** [15]. These are described in subsection 3.1.

The second component that we require is an imitation learning method. We implement several **no-regret online learning** methods described by Ross, Gordon, and Bagnell: **forward training**, **SMILe**, and **DAgger** [16]. These are described in subsection 3.2. We also implement **greedy transtion-based parsing**, which is first introduced in subsection 3.1. We will also discuss the time complexities of these algorithms in section 3.3.

3.1 Parsing Systems

In the interests of making this paper self-contained, this section is a summary of and largely paraphrases the relevant portions of Goldberg and Nivre (2013) [15].

The parsing systems developed by Goldberg and Nivre contain two features relevant to our work. First of all, they encode dependency trees as linear time series of transitions. Having a time series and a set of classes to predict makes the problem amenable to imitation learning such as DAgger. Secondly, they provide experts to determine the next transition given the ground truth dependency tree.

First we will describe the arc-eager and arc-hybrid transition systems. Then we will discuss **greedy transition-based parsing** and why it is often insufficient for imitation learning. We then establish several concepts used by Goldberg and Nivre. Finally, we describe how the transition-based dependency parsing systems can be used for imitation learning.

3.1.1 Transition systems

We want to construct transition systems that can construct any **projective dependency tree** by applying transitions to an ordered sequence. A projective dependency tree is a (non-binary) tree with vertices in bijection with the elements of an ordered sequence, such that if a vertex is to the left of a sibling vertex, then its descendents are to the left of the sibling's descendents.

Configurations Transition systems for parsing maintain a **configuration** consisting of a **stack** σ, a buffer β, and a set of dependency arcs A. We denote a configuration $c = (\sigma, \beta, A)$. We may also write the top element(s) of the stack and the first element(s) of the buffer to the right and left respectively, as in $c = (\sigma|s, b|\beta, A)$. We write arcs in A as (h, d), where h is the head and d is the dependent. The system is initialized as $\sigma = \varnothing$, $A = \varnothing$, and $\beta = w_1, ..., w_n$ where w_i is the i^{th} word or symbol.

We use $\phi(c)$ to denote the features of a configuration c, which we will use to train classifiers. Specifically, we use features of the symbols in each of σ, β, and A.

A transition system then requires a set of transitions that can be applied to the initial buffer β_0 to reach any dependency tree. We describe two such transition systems. Other transition systems are possible, but these two can be shown to have properties that are useful for imitation learning (see subsection 3.1.3).

Arc-eager transition system The arc-eager system has four transitions. As stated in Goldberg and Nivre [15], they are:

$\textsc{Shift}[(\sigma, b|\beta, A)] = (\sigma|b, \beta, A)$
$\textsc{Right}_{lb}[(\sigma|s, b|\beta, A)] = (\sigma|s|b, \beta, A \cup \{(s, b)\})$
$\textsc{Left}_{lb}[(\sigma|s, b|\beta, A)] = (\sigma, b|\beta, A \cup \{(b, s)\})$
$\textsc{Reduce}[(\sigma|s, \beta, A)] = (\sigma, \beta, A)$

These transitions require that b or s exist as necessary, so for example SHIFT requires that the buffer is non-empty. To maintain tree structure, $\textsc{Left}_{lb}$ requires that s does not already have a parent, and REDUCE requires that s does already have a parent. Projectivity is maintained because all left dependents must be collected before a SHIFT or $\textsc{Right}_l$ operation, and all right dependents after. Conversely, all trees may be constructed recursively by processing all left subtrees, collecting the heads of left subtress as dependents with $\textsc{Left}_{lb}$, performing a SHIFT if the head of the current subtree is a left depedent and a $\textsc{Right}_{lb}$ otherwise, then processing the right subtrees.

Arc-hybrid transition system The arc-hybrid system has three transitions. As stated in Goldberg and Nivre [15], they are:

$\textsc{Shift}[(\sigma, b|\beta, A)] = (\sigma|b, \beta, A)$
$\textsc{Right}_{lb}[(\sigma|s_1|s_0, \beta, A)] = (\sigma|s_1, \beta, A \cup \{(s_1, s_0)\})$
$\textsc{Left}_{lb}[(\sigma|s, b|\beta, A)] = (\sigma, b|\beta, A \cup \{(b, s)\})$

As in the arc-eager system, transitions require that b or s exist as necessary. All left dependents must be collected before a SHIFT operation, and all right dependents after, so as in the arc-eager system projectivity is maintained and any projective tree can be constructed.

3.1.2 Greedy transition-based parsing, and problems therewith

Since we can construct any projective dependency tree with either transition system, a naive method of imitation learning called **greedy transition-based parsing** is as follows.

1. Convert the ground-truth dependency tree for each training sequence W into a sequence of transitions $t_1, ..., t_m$.
2. Starting from the initial configuration c_0, apply each transition t_i to get the updated configuration c_i.
3. Train a classifier to predict t_i from features $\phi(c_{i-1})$.

The problem with this method is that all t_i are predicted by an expert, so when a classifier is trained, it only sees the features $\phi(c_{i-1})$ of c_i resulting from perfect sequences of transitions. However, to predict a dependency tree, the classifier predicts each transition from an imperfect sequence of previous transitions. This discrepancy between training and testing can lead to error. More explicitly, the no-regret online learning algorithms, which deal with this problem, require expert training data on non-expert sequences. For more details see subsection 3.2.

Since we need to be able to train classifiers on configurations resulting from non-optimal sequences of transitions, we require parsing systems that can produce optimal expert transitions, given ground

truth dependency trees and configurations resulting from non-optimal sequences of transitions. The following subsection establishes such systems based on the arc-eager and arc-hybrid transition systems.

3.1.3 Dynamic oracles

The method for producing labels in the greedy transition algorithm above can be referred to as a **static oracle**. A static oracle o_s maps a tree T to a sequence of transitions $t_1, ..., t_m$. Alternatively, we write $o_s(c_i, T) = t_{i+1}$ or $o_s(t; c_i, T) = \mathbf{true}$ for $t = t_{i+1}$ and is $\mathbf{false}$ otherwise. o_s is **correct** if $A_m = T$ for $c_m = (\sigma_m, \beta_m, A_m)$ where $c_m = t_m(t_{m-1}(...(t_1(c_0))))$.

A **non-deterministic** oracle is an oracle o_n that may return a set of transitions $o_n(c_i, T)$ such that if T is reachable from c_i, then for any $t_{i+1} \in o_n(c_i, T)$, T is reachable from $t_{i+1}(c_i)$.

A **dynamic** non-deterministic oracle is a non-deterministic oracle o_d such that if T is not reachable from c_i, $o_d(c_i, T)$ outputs a set of transitions that are optimal by some optimality condition. We will describe the optimality condition, then establish dynamic oracles for the arc-eager and arc-hybrid systems.

Optimality condition We begin by defining a cost function $C(A, T)$ on a set of arcs A and a tree T. In this case we use the Hamming loss for unlabeled dependency arcs, which is the symmetric difference of arcs in T and in A. Equivalently, since T and A contain the same number of arcs, we use the number of arcs in T but not in A. We then define the optimality condition that t_{i+1} is optimal for (c_i, T) if $\min_{A_m : c_i \rightsquigarrow A_m} C(A_m, T) = \min_{A'_m : t_{i+1}(c_i) \rightsquigarrow A'_m} C(A'_m, T)$, where $c \rightsquigarrow A$ means that A is reachable from c. In other words, t_{i+1} is optimal if the end arc set A'_m that closest to T and reachable from $t_{i+1}(c_i)$ is no farther from T then the end arc set A_m that closest to T and reachable from c_i. A dynamic oracle o_d is an oracle such that $o_d(c_i, T) = \{t_{i+1} \mid t_{i+1} \text{ is optimal for } (c_i, T)\}$.

Arc reachability and arc decomposition We now define several concepts that we will use to find dynamic oracles for the arc-eager and arc-hybrid systems.

A dependency arc (h, d) is **reachable** from configuration c_i if there exists a sequence of transitions $t_{i+1}, ..., t_k$ such that $(h, d) \in A_k$ where A_k is the arc set of $t_k(...(t_{i+1}(c_i)))$. We write this as $c_i \rightsquigarrow (h, d)$.

A set A of arcs is **reachable** if there is a sequence of transitions $t_{i+1}, ..., t_k$ such that $A \subseteq A_k$. We write this as $c_i \rightsquigarrow A$.

A set A of arcs is **tree consistent** if there exists a projective dependency tree T such that $A \subseteq T$.

A transition system is **arc decomposable** if, for tree consistent $A = \{(h_1, d_1), ..., (h_n, d_n)\} \subseteq T$, we have $c_i \rightsquigarrow (h_1, d_1), ..., c_i \rightsquigarrow (h_n, d_n) \Rightarrow c_i \rightsquigarrow A$.

Arc decomposability is a useful property.

Suppose configuration $c_i \rightsquigarrow A'_m$, where A'_m is the minimizer $A'_m = \underset{A_m : c_i \rightsquigarrow A_m}{\arg\min} C(A_m, T)$. Suppose also that after applying t_{i+1}, the resulting configuration $t_{i+1}(c_i) \rightsquigarrow A''_m$, where A''_m is the minimizer $A''_m = \underset{A_m : t_{i+1}(c_i) \rightsquigarrow A_m}{\arg\min} C(A_m, T)$. Then arc decomposability implies that we can decompose the change in cost $\Delta C(t_{i+1}; c_i, T) = C(A''_m, T) - C(A'_m, T)$ into a sum over arcs in T as $\sum_{(h,d) \in T} \left(\min_{A_m : t_{i+1}(c_i) \rightsquigarrow A_m} C(A'_m, (h, d)) - \min_{A_m : c_i \rightsquigarrow A_m} C(A_m, (h, d)) \right)$. Then to calculate $\Delta C(t_{i+1})$, all we need is to calculate the number of arcs that are no longer reachable after applying transition t_{i+1}.

We state without proof that the arc-eager and arc-hybrid systems are arc decomposable. Readers may refer to Goldberg and Nivre [15] for more details. We will now construct the corresponding dynamic oracles.

Arc-eager dynamic oracle Since the arc-eager system is arc decomposable, to calculate $\Delta C(t_{i+1})$ the increase in cost function after applying t_{i+1}, all we need is to calculate the number of

ground-truth arcs that are no longer reachable after applying transition t_{i+1}. Given a configuration $c_i = (\sigma|s, b|\beta, A)$ and T, $\Delta C(t_{i+1}; c_i, T)$ can be calculated as

- $t_{i+1} = \text{LEFT}_{lb}$: The number of base-truth dependents and heads of s in β.
- $t_{i+1} = \text{RIGHT}_{lb}$: The number of base-truth heads of b in $\sigma \cup \beta$ plus the number of base-truth dependents of b in σ that do not already have a (wrong) head in A.
- $t_{i+1} = \text{REDUCE}$: The number of base-truth dependents of s in $b|\beta$.
- $t_{i+1} = \text{SHIFT}$: The number of base-truth heads of b in $\sigma|s$ plus the number of base-truth dependents in $\sigma|s$ that do not already have a (wrong) head in A.

Then a dynamic oracle for the arc-eager system is one that returns $o_d(c_i, T) = \{t_{i+1} \mid \Delta C(t_{i+1}; c_i, T) = 0\}$.

Arc-hybrid dynamic oracle Since the arc-hybrid system is arc decomposable, to calculate $\Delta C(t_{i+1}; c_i, T)$ the increase in cost function after applying t_{i+1}, all we need is to calculate the number of arcs that are no longer reachable after applying transition t_{i+1}. Given a configuration $c_i = (\sigma|s_1|s_0, b|\beta, A)$ and base truth dependency tree T, ΔC can be calculated as

- $t_{i+1} = \text{LEFT}_{lb}$: The number of base-truth dependents of s_0 in $b|\beta$ plus the number of heads of s_0 in $\{s_1\} \cup \beta$.
- $t_{i+1} = \text{RIGHT}_{lb}$: The number of base-truth heads and dependents of s_0 in $b|\beta$.
- $t_{i+1} = \text{SHIFT}$: The number of base-truth heads of b in $\sigma|s_1$ plus the number of base-truth dependents in $\sigma|s_1|s_0$.

Then a dynamic oracle for the arc-hybrid system is one that returns $o_d(c_i, T) = \{t_{i+1} \mid \Delta C(t_{i+1}; c_i, T) = 0\}$.

3.2 Imitation learning

In the interests of making this paper self-contained, this section is a summary of and largely paraphrases the relevant portions of Ross, Gordon, and Bagnell (2010) [16] and Ross and Bagnell (2010) [17].

In the **imitation learning** paradigm, an expert provides demonstrations of good or optimal behavior, and these demonstrations are used to learn a controller. Such a controller might be a classifier trained to predict expert behavior from observations. In the context of **sequence prediction**, each element of a sequence of observations depends on the output of the controller on the previous observation. However, since the output of the controller is a prediction of expert behavior depending on the previous observation, the observations are not i.i.d. as is commonly assumed.

Since observations are not i.i.d., naively training a classifier under i.i.d. assumptions can lead to poor performance. In fact, there are systems such that if the probability of a mistake on an expert sequence is ϵ, the overall expected number of mistakes is $O(T^2\epsilon)$ where T is the length of the sequence of predictions [17]. Intuitively, after a mistake is made, the controlled may see observations unlike those it was trained under, so the probability of a mistake becomes $O(1)$. Then since the probability of a mistake in the first $O(T)$ steps is $O(T\epsilon)$, and the expected number of mistakes in the $O(T)$ steps after the first mistake is $O(T)$, the expected total number of mistakes is $O(T^2\epsilon)$.

The typical method to removing the i.i.d. assumption is to train a sequence of classifiers, training each on the output of the previous. Here we describe several approaches to imitation learning described in Ross, Gordon, and Bagnell (2010) [16]. Since a theoretical discussion is not necessary for implementation, we state some results but refer readers to that paper for proofs.

Recall that in subsection 3.1.3, we established experts o_d that identify whether a transition t_{i+1} is optimal given configuration c_i and the ground truth dependency tree T, where c_i may have been generated by a sequence of non-optimal transitions. When we mention experts in the following subsections, these are the experts we are implicitly referring to.

3.2.1 Greedy transition-based parsing

Greedy transition-based parsing was described in subsection 3.1.2. It is a method that uses the i.i.d. assumption and is used as a baseline for the other methods.

3.2.2 Forward training

In forward training, a classifier π_i^i is trained at each time step $i = 1..T$ based on the output of the previous classifier π_{i-1}^{i-1} on the training set. Then at test time each classifier π_i^i is applied to the configuration c_i to get the next transition t_{i+1}. Typically we maintain a set of classifiers $\{\pi_1^i, ..., \pi_T^i\}$, where $\pi_1^i, ..., \pi_i^i$ is the set of classifiers we have trained up to the i^{th} step, and $\pi_{i+1}^i, ..., \pi_T^i$ are expert classifiers. This allows us to run sequences of actions to completion, so we can train each classifier to optimize a cost function evaluated at the last step.

The forward training algorithm as stated in [17] and using expert π^*, is given in algorithm 1.

Initialize $\pi_1^0, ..., \pi_T^0$ to be π^*;
for $i = 1$ **to** T **do**
  Sample T-step trajectories by following π^{i-1};
  Get dataset $\mathcal{D} = \{(c_i, \pi^*(c_i))\}$ of configurations, transitions given by expert at step i;
  Train classifier $\pi_i^i = \arg\min_\pi I\!E_{c\sim\mathcal{D}}(e_\pi(c))$ where $I\!E_{c\sim\mathcal{D}}(e_\pi(c))$ is the expectation over $\mathcal{D}$ that π gives the optimal transition;
  $\pi_j^i = \pi_j^{i-1}$ for all $j \neq i$.
end
return $\pi_1^T, ..., \pi_T^T$

Algorithm 1: The forward training algorithm trains a classifier for each time step based on the previous classifier.

In our case, the expert is $\pi^* = o_d$ the dynamic oracle. Since the dynamic oracle gives the transition that optimizes cost evaluated at the end step, we can simplify the forward training algorithm by training new classifiers as we increment time steps, as in algorithm 2.

Initialize $\mathcal{D}_0 = \mathcal{D}_0^* = \{c_0\}$ the set of initial configurations of the training data;
for $i = 1$ **to** T **do**
  Sample $\mathcal{D}_{i-1} = \{c_{i-1}\}$ by following π_{i-1};
  Get dataset $\mathcal{D}_i^* = \{(c_{i-1}, \pi^*(c_i))\}$ of configurations, transitions given by dynamic oracle at step i;
  Train classifier $\pi_i = \arg\min_\pi I\!E_{c\sim\mathcal{D}}(I(\pi(c_i) = \pi^*(c_i)))$;
end
return $\pi_1, ..., \pi_T$

Algorithm 2: The simplified forward training algorithm trains a classifier for each time step based on the previous classifier. Note that it does not require entire training sequences to be run for each new classifier, but it can train classifiers as it increments time steps.

For forward training using dynamic oracles, the Hamming cost function is minimized when the classifier minimizes the imitation loss $I(\pi(c_i) = \pi^*(c_i))$. Then as proven in [16] the expected Hamming loss is $O(T\bar{\epsilon})$, where $\bar{\epsilon}$ is the mean probability of making a mistake over steps.

3.2.3 Stochastic mixed iterative learning (SMILe)

SMILe works by training a collection of classifiers. The first classifier is trained on the output of the expert, and each subsequent classifier is trained on the output of a probabilistic combination of the previous classifiers. Here we use the notation $\pi = \sum_n p_n \pi_n$ to represent a classifier that selects one of $\pi_1, \pi_2, ...$ with probability $p_1, p_2, ...$, then applies that classifier to the given configuration features. Over iterations, the probability that the expert is used to select the next transition decreases. Note that the expert is still always used to select the labels that the classifiers are trained on, just not the trajectories.

The SMILe algorithm is given in algorithm 3.

Initialize $\pi_0 = \pi^*$;
for $n = 1$ **to** N **do**
  Sample trajectories $\{c_0, c_1, ...\}$ by following π_{n-1};
  Get dataset $\mathcal{D} = \{(c_i, \pi^*(c_i))\}$ of configurations visited by π_{n-1}, transitions given by expert at all time steps i;
  Train classifier $\hat{\pi}_n = \arg\min_\pi I\!E_{c_i \sim \mathcal{D}, i \in 1..T}(I(\hat{\pi}_n(c_i) = \pi^*(c_i)))$;
  $\pi_n = (1-\alpha)^n \pi_* + \alpha \sum_{j=1}^{n} (1-\alpha)^{j-1} \hat{\pi}_j$;
end
Remove expert classifier: $\tilde{\pi}_N = \frac{\pi_N - (1-\alpha)^N \pi^*}{1-(1-\alpha)^N}$;
return $\tilde{\pi}_N = \alpha \sum_{j=1}^{N} \frac{(1-\alpha)^{j-1}}{1-(1-\alpha)^N} \hat{\pi}_j$

Algorithm 3: SMILe trains each classifier on a probabilistic combination of the previous classifiers, where the expert has decreasing impact on the trajectory but is still used to produce the target classes for classification.

If $\alpha = O(\frac{1}{T^2})$ and $N = O(T^2 \log T)$, then as stated in [16] the expected Hamming loss is $O(T\bar{\epsilon})$ for some problems, where again $\bar{\epsilon}$ is the mean probability of making a mistake over steps.

Ross and Bagnell [17] present a second, equivalent implementation of SMILe that is more efficient in terms of the number of training samples required. This second version has as a parameter m the number of samples required to train a classifier, and requires $O(Tm)$ training sequences. For brevity, we do not describe the second algorithm here. Both versions were implemented, but only the second was used for testing due to higher efficiency.

3.2.4 Dataset Aggregation (DAgger)

Note that although Goldberg and Nivre develop a learning scheme based on DAgger [15], we only examine the original DAgger algorithm [16].

In DAgger, a dataset of all trajectories seen so far is maintained. At each iteration, a probabilistic combination of the expert and the latest classifier is used to generate more sequences. Then the next classifier is trained on the full dataset to predict the transitions given by the expert over all paths.

Unlike in SMILe, trajectories only result from a combination of the one classifier and the expert. As in SMILe, the probability of using the expert to choose transitions decreases over iterations. The DAgger algorithm only returns one classifier, not a combination of classifiers. The DAgger algorithm is given in algorithm 4.

Initialize $\mathcal{D} = \varnothing$;
Initialize $\hat{\pi}_1$;
for $n = 1$ **to** N **do**
  $\pi_n = \beta_n \pi^* + (1-\beta_n)\hat{\pi}_n$;
  Sample trajectories $\{c_0, c_1, ...\}$ by following π_n;
  Get dataset $\mathcal{D}_n = \{(c_i, \pi^*(c_i))\}$ of configurations visited by π_n, transitions given by expert at all time steps i;
  Aggregate datasets $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_n$;
  Train classifier $\hat{\pi}_{n+1} = \arg\min_\pi I\!E_{c_i \sim \mathcal{D}, i \in I\!N}(I(\hat{\pi}_i(c_i) = \pi^*(c_i)))$;
end
return *best* $\hat{\pi}_n$ *on validation.*

Algorithm 4: DAgger trains each classifier on the output of probabilistic combinations of all previous classifiers and the expert. β_n is typically decreasing, ex. $\beta_n = \alpha^{n-1}$. In this case $\hat{\pi}_1$ need not be specified.

If $N = O(uT)$, where u is an upper bound on the Hamming cost of any transition, then as stated in [16] the expected Hamming loss is $O(uT\epsilon)$ where ϵ is the probability of making a mistake.

3.3 Complexity

In this section we discuss the time complexities of the imitation learning algorithms discussed in subsection 3.2. We define the following quantities used to calculate computational complexity.

- T: the average length of a time series to be predicted.
- n_{ts}: the number of separate time series used in training.
- N: the number of models (classifiers) trained by SMILe or DAgger.
- n_c: the number of classes to predict (4 for arc-eager and 3 for arc-hybrid).
- n_{it}: the number of iterations used by a black box classifier.
- n_{sv}: the number of support vectors used by a black box SVM classifier.
- n_f: the number of features used in training (i.e. the length of $\phi(c_i)$ as described in subsection 3.1.1).
- l: the number of training instances. This typically depends on other values.
- k: the number of testing instances. This typically depends on other values.

All methods use a (Gaussian) radial basis function kernel SVM as a black box classifier. We use the LIBSVM library, written by Chang and Lin [18]. See section 5.1 for version details. Training an SVM in LIBSVM has a time complexity of $O(n_{it}n_f l)$. Chang and Lin state that empirically, n_{it} may be super linear in l. Here we assume that $n_{it} = O(l)$, so the time complexity of training an SVM is $O(n_f l^2)$. We do not vary the number of features in this paper, so we may assume the time complexity is $O(l^2)$.

To classify using a kernel SVM, each testing instance must be compared against each support vector, for a total time complexity of $O(n_f k n_{sv})$. Assuming n_f to be constant and n_{sv} to be at most linear in l, we simplify this as $O(kl)$. Note that due to overhead, we are also concerned with the minimizing the number of calls to `svm-predict`.

To train a multi-class classifier, we train a classifier on each pair of classes. Then we must train $O(n_c^2)$ classifiers and predict $O(n_c^2)$ times to identify a class, but this will not affect the relative complexity between methods, so we ignore this factor.

3.3.1 Greedy transition-based parsing

In greedy transition-based parsing, we train a single SVM on all available data. In each time series, all configurations are calculated using the dynamic oracle as an expert, in negligible time. There are n_{ts} time series of average length T, so there are $l = O(n_{ts}T)$ transitions/configurations to use as training instances. Then the time complexity is $O(n_{ts}^2 T^2)$.

3.3.2 Forward training

In forward training, an SVM is trained for each time step $i = 1..T$. At each time step the training data is the set of transitions/configurations for each time series at that time step. Then there are $l = O(n_{ts})$ training instances, so the time to train one SVM is $O(n_{ts}^2)$. Then the total time to train all SVMs is $O(n_{ts}^2 T)$.

In addition, our training procedure requires that at each time step we predict the transitions to be applied to configurations. We can predict the next transition for all of the time series at once, so we require $O(T)$ calls to `svm-predict`. Since each SVM is trained on $l = O(n_{ts})$ instances and predicts $k = O(n_{ts})$ transitions, the time complexity to predict at each step is $O(n_{ts}^2)$ per time step, or $O(n_{ts}^2 T)$ total.

3.3.3 Stochastic mixed iterative learning (SMILe)

Ross and Bagnell [17] describe an implementation of SMILe that is more efficient in terms of number of training samples required than the version presented in subsection3.2.3. We implemented both versions, but only present results for the faster version. If a classifier requires m sequences to train, this implementation requires $O(mT)$ training sequences for $\alpha = O(\frac{1}{T^2})$ to achieve their guarantees

```
c2 ( v0 + c1 ) = d1
v0 + c1 = d0
v0 = c0
```

(a) Form 1

```
c2 ( v0 + c1 ) = d1
c2 v0 + d2 = d1
c2 v0 = d0
v0 = c0
```

(b) Form 2

Figure 1: Two forms of equations/solutions were used. Variables `v` and integer constants `c` were chosen at random; integer constants `d` were calculated.

discussed in subsection 3.2.3. Then the total time complexity required for predicting transitions is $O(m^2T^2)$. However, for such α a large number ($O(T^2(\log T)^{3/2})$) of classifiers must be trained, so for practical reasons a larger α is used. This implementation uses m and α as inputs rather than N and α. Since each classifier is trained on $O(mT)$ samples, each classifier requires $O(m^2T^2)$ time to train. Then we can upper bound the time required to train all classifiers by $O(m^2T^4(\log T)^{3/2})$.

3.3.4 Dataset Aggregation (DAgger)

Note that in DAgger, we the total data size increases linearly with the iterations. This would lead to long training times, so to avoid this we considered two possibilities. The first was to split data over iterations, so each $\mathcal{D}_n$ would be constructed from $\frac{1}{N}$ of the total data. The second was to construct each $\mathcal{D}_n$ from the full set of training data, and then to downsample by choosing only one version of each training sequence at random from one $\mathcal{D}_n$. We found that splitting the data yielded poor results, so we only present the results after downsampling.

After downsampling, we train an SVM on n_{ts} time series worth of data, which is $O(n_{ts}T)$ transitions. Since we train $O(N)$ SVMs, the total SVM training time complexity is $O(Nn_{ts}^2T^2)$.

For $O(N)$ iterations we predict $k = O(n_{ts})$ transitions using a single SVM trained on $l = O(n_{ts}T)$ training instances for each of $O(T)$ time steps, for a total of $O(NT)$ calls to `svm-predict` and $O(Nn_{ts}^2T^2)$ total prediction time complexity.

Note that N is typically much lower for DAgger than for SMILe.

4 Dataset

The dataset we have chosen to work with is a dataset of handwritten algebraic equations and solutions. The fact that we are using algebra means that the data is highly structured. The fact that the data is handwritten means that there are rich low level features that can be used to predict structure.

To generate the data, first we generated random ground-truth algebraic equations and solutions following set patterns, described in subsection 4.1. The equations were then hand-copied to produce the handwriting dataset, described in subsection 4.2.

4.1 Ground truth generation

Each algebraic solution generated took one of the forms shown in Figure 1, chosen uniformly at random. Variables `v` were chosen uniformly from {`x,y,z`}. Integer constants `c` were chosen at uniformly at random from `[-5, 5]`. Integer constants `d` were calculated from the integer constants `c`. Examples of algebraic solutions can be seen in Figure 2.

Note that some generated solutions include canceling zeros. For example in Form 2 of Figure 1, the equation `0 x = 0` could be solved as `x = 3`, if `c0 = 3` and `c2 = 0`. Since we are only attempting to learn structure in this paper, these solutions were allowed.

4.2 Handwriting collection

To collect handwriting data, a researcher copied 400 algebraic solutions generated as in subsection 4.1 into a PDF document using a drawing tablet (details below). Each algebraic solution was copied onto a separate page. The researcher was instructed to copy all symbols from the ground

```
-5 ( x + -1 ) = -20
x + -1 = 4
x = 5
```

(a) Form 1

```
4 ( x + 1 ) = -16
4 x + 4 = -16
4 x = -20
x = -5
```

(b) Form 2

Figure 2: These are examples of equations used as ground truth.

truth, except that expressions of the form + − were simply copied as −. The researcher was also instructed to align the equations as they normally would while solving equations, rather than for example keeping the equations left aligned. An example of handwriting corresponding to the base truth in Figure 2a is shown in Figure 3.

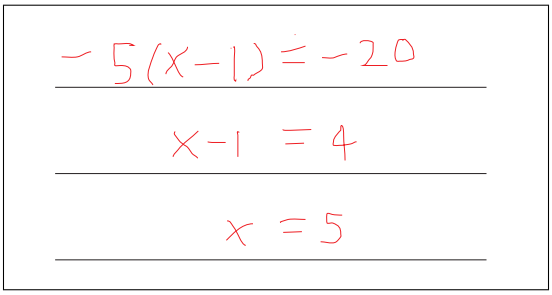

Figure 3: This figure shows an example of handwriting corresponding to the equations in Figure 2a.

After the handwriting was collected, the base truth was corrected to match any copying errors in the handwriting. This includes replacing + − with −, which was done at runtime during preprocessing (subsection 5.2).

4.2.1 Tools

The algebraic solutions were written using a "Wacom Intuos5 Touch Medium Pen Tablet", model number "PTH-650". The algebraic solutions were written into a PDF using the program "Adobe Acrobat Pro DC", version number "2015.009.20071". The solutions were written using the "Draw" tool on the "Comment" toolbar.

The blank PDFs consisted of 100 pages, each with 6 lines 100 pixels apart, ranging from 150 to 650 pixels from the bottom of the page, and 550 pixels wide starting at 50 pixels from the left edge. The first equation was written on top of the first line. The PDFs were standard letter size, with a resolution of 72 pixels per inch. The Intuos tablet digitizes pen strokes with a resolution well under one pixel, so the data contains fractional stroke coordinates.

After the handwriting was collected, the PDFs were converted to PDF version 1.3 (compatible with Adobe Acrobat 4.0 and later) using Adobe Acrobat Pro DC. This was done because PDF version 1.3 is easier for MATLAB to parse.

5 Experiments

This section describes the experiments that were performed on the handwriting data. First we describe tools and libraries used in subsection 5.1. We discuss preprocessing performed on the data in subsection 5.2. Then we describe how algorithms were tuned in subsection 5.3. We then describe the main tests performed in subsection 5.4. We discuss evaluation metrics in subsection 5.5.

5.1 Tools and libraries

The data base truth generation and some initial preprocessing was written in Python™ version 2.7. The rest of our code was written in MATLAB® Release 2014b for 64-bit Windows [19] with

Computer Vision System Toolbox version 6.1 (R2014b). We also used the LIBSVM library version 3.20, written by Chang and Lin [18], to train support vector machines.

The code was run on an Intel® Core™ i7 processor.

5.2 Preprocessing

Preprocessing involved three main parts. First dependency trees were generated from the base truth. This is described in subsection 5.2.1. Next, ground truth transition sequences for the trees were derived and saved for later use. This is described in subsection 5.2.2. Finally, features were extracted from the handwriting data. This is described in subsection 5.2.3.

5.2.1 Dependency trees

Algebraic solutions were converted into dependency trees by repeatedly splitting strings into pairs of substrings. Each substring formed a subtree, with the head of one dependent on the head of the other. Note that there are other possible rule sets, but this one was chosen as intuitively reflecting the structure of algebra. We use the notation $a \rightarrow b$ to denote a tree with the head of subtree b dependent on the head of subtree a. We built dependency trees according to the recursive function $f(s)$, defined on the string s with the following rules, applied in order.

1. If s is "$l_1 ... l_k$" for lines "l_1", ..., "l_k", return $f($"$l_1 ... l_{k-1}$"$) \rightarrow f($"l_k"$)$.
2. If s is "$a = b$", return $f($"a"$) \leftarrow$ " $=$ " $\rightarrow f($"b"$)$.
3. If s is "$a(b)$", return $f($"a"$) \leftarrow f($"(b)"$)$.
4. If s is "(b)", return $f($"$(b$"$) \leftarrow f($"$)$"$)$.
5. If s is "$(b$", return "$($" $\rightarrow f($"b"$)$.
6. If s is "$a + b$", return $f($"a"$) \leftarrow$ " $+$ " $\rightarrow f($"b"$)$.
7. If s is "$a - b$", return $f($"a"$) \leftarrow$ " $-$ " $\rightarrow f($"b"$)$.
8. If s is " $- a$", return " $-$ " $\leftarrow f($"a"$)$.
9. If s is "av", for $v \in \{$"x", "y", "z"$\}$, return $f($"a"$) \leftarrow$ "v".
10. If s is "ab", where "b" is a digit, return $f($"a"$) \rightarrow$ "b".
11. If s is "a", where "a" is a single symbol, return "a".

For example, the equations in Figure 2a would be rendered as shown in Figure 4.

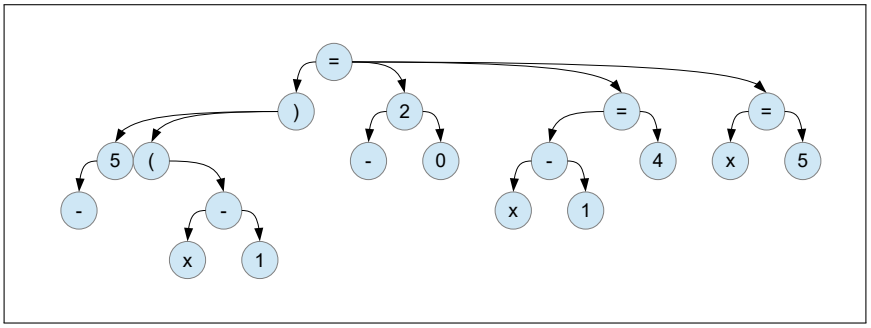

Figure 4: This figure shows the dependency tree corresponding to the equations in Figure 2a.

5.2.2 Transition sequences

Transition sequences for both arc-eager and arc-hybrid parsing were constructed from the dependency trees. Since both parsing methods require that the all left dependents be evaluated before all right dependents, we can recursively convert subtrees into transition sequences. For example, the arc-eager sequence for Figure 2a is `slssslrrddllslsslrrddslsrdlrrddslrrdd`, where `s,r,l,d` represent SHIFT, RIGHT$_{lb}$, LEFT$_{lb}$, REDUCE respectively. The arc-hybrid sequence for Figure 2a is `slssslssrrllslsslssrrslssrlssrrslssrr`.

5.2.3 Feature extraction

Features may refer to two concepts. The first is the features of symbols. These are used to construct the second, the features of configurations, which are the features that are used to train classifiers and to predict transitions.

Stroke/symbol features Data is stored in the PDF as a series of strokes. Each stroke is a list of coordinates representing the location of the pen from the time it touched the tablet until the time it was lifted. For example, the character + is typically written with two strokes. The first step in preprocessing the handwriting is to associate strokes that are part of the same symbol. This was done with a combination of manual and automatic processing: we first automatically produced a grouping by assuming that two subsequent strokes are part of the same symbol if their horizontal overlap is at least 40% of the width of one or the other of the strokes and the strokes are on the same line. Then we manually examined the results and corrected any errors.

Once the strokes are associated into symbols, the maximum dimension of the bounding box and the center of the bounding box are saved as features. Additionally, we used the vector between the center of the previous and current symbols as a feature.

The symbols are then normalized to have a maximum dimension of 200. The symbols are then drawn to an image in MATLAB with line weight 6, and histogram of oriented gradients (HOG) features are calculated using MATLAB's `extractHOGFeatures` function and cell size 50×50.

Configuration features As described in subsection 3.1.1, a configuration consists of a stack, a buffer, and a set of arcs. For the configuration features, we used the features of the top 5 symbols in the stack, the most recent 5 pairs of symbols in the set of arcs, and the first symbol in the buffer. If any of these were not present the relevant features were set to 0.

5.3 Tuning

All imitation learning methods implemented here use LIBSVM support vector machines as black box classifiers. In particular, we use multi-class C-support vector classifiers (C-SVCs). We use a radial basis function (Gaussian) kernel. There are two variables to tune, C the regularization cost and γ a parameter of the kernel. We only tune one C and γ, though some methods use multiple classifiers.

Define $f_{A,D}(C, \gamma)$ to be a function that returns the result of performing 2-fold cross validation of algorithm A on data D with parameters C and γ. Assume a higher score is better (see subsection 5.5 for details). We then tune C and γ using the Algorithm 5, based on a grid search method described by Hsu, Chang, and Lin in [20].

Tuning was performed using 1/4 of the dataset used in testing (100 pages). Tuning was performed separately on arc-eager and arc-hybrid ground truth transitions. Tuning results are given in table 1.

The SMILe algorithm was tuned with parameters $\alpha = 0.0025$ and $m = 50$ sample sequences per SVM. The DAgger algorithm was tuned with parameters $\alpha = 0.5$ and $N = 5$ iterations.

5.4 Testing

Testing was performed on 1/4 of the dataset (100 pages). The testing data did not overlap with the tuning data. We tested the following algorithms, with parameters as described in subsection 3.2.

- Greedy transition-based parsing (GTP)
- Forward training (FT)
- SMILe with parameters $m = 50$, $\alpha = 0.0025$ (SMILe)
- DAgger using downsampling, with $N = 5$ iterations and $\beta_n = 0.5^{n-1}$ (DAgger)

All algorithms were tested with both arc-eager and arc-hybrid transition systems.

Initialize $l_C^* = 0$,$l_\gamma^* = 0$, $d = 16$;
while $d > \epsilon = 1.0$ **do**
  $l_C^{**} = l_C^*$;
  $l_\gamma^{**} = l_\gamma^*$;
  for $(l_C, l_\gamma) \in \{l_C^* - d, l_C^* - \frac{d}{2}, l_C^*, l_C^* + \frac{d}{2}, l_C^* + d\} \times \{l_\gamma^* - d, l_\gamma^* - \frac{d}{2}, l_\gamma^*, l_\gamma^* + \frac{d}{2}, l_\gamma^* + d\}$ **do**
    if $f(2^{l_C}, 2^{l_\gamma}) > f(2^{l_C^{**}}, 2^{l_\gamma^{**}})$ **then**
      $l_C^{**} = l_C$;
      $l_\gamma^{**} = l_\gamma$;
    end
  end
  if $(l_C^{**}, l_\gamma^{**}) \neq (l_C^{**}, l_\gamma^{**})$ **then**
    $l_C^* = l_C^{**}$;
    $l_\gamma^* = l_\gamma^{**}$;
  else
    $d = d/2$;
  end
end
return $2^{l_C^*}, 2^{l_\gamma^*}$

Algorithm 5: This algorithm is a simple decreasing-step-size grid search in the logs of parameters C and γ. $f_{A,D}(C, \gamma)$ evaluates 2-fold cross validation of algorithm A on data D with parameters C, γ. The minimum step size $\epsilon = 1.0$ was chosen as it was observed that the recall did not noticeably improve for smaller values of d.

Algorithm	**Transition type**	$\log_2 C$	$\log_2 \gamma$	**Other parameters**
GTP	arc-eager	11	-20	
GTP	arc-hybrid	10	-19	
FT	arc-eager	14	-20	
FT	arc-hybrid	12	-20	
SMILe	arc-eager	11.5	-20.5	$\alpha = 0.0025, m = 50$
SMILe	arc-hybrid	11	-17.5	$\alpha = 0.0025, m = 50$
DAgger	arc-eager	16	-20	$\alpha = 0.5, N = 5$
DAgger	arc-hybrid	11	-17.5	$\alpha = 0.5, N = 5$

Table 1: This table lists the results of tuning C and γ. Algorithm abbreviations are defined in subsection 5.4.

5.5 Evaluation metrics

The metric used for evaluation is the average fraction of correctly identified arcs (recall) over test pages. Note that since the number of arcs in a complete tree is determined by the number of vertices, the number of correctly identified arcs equals the number of true arcs so recall and precision are equal. We calculate recall using 10-fold cross validation over the testing data set. Of the data that is held out, half (5 pages) is either used for model selection or is discarded. The other half is used for testing.

We also report σ^* the standard deviation of the mean of recall. This is estimated by $\sigma/\sqrt{n_t}$, where σ is the standard deviation of the arc recalls over test pages and n_t is the number of test pages (50).

6 Results

The results (mean recall, recall standard deviation of the mean, and mean execution time) for arc-eager and arc-hybrid transition prediction are given in Tables 2 and 3 respectively.

Algorithm	Transition type	Mean recall	σ^*	Mean execution time
GTP	arc-eager	**0.9222**	0.0220	$2.3 \cdot 10^1 s$
FT	arc-eager	0.8243	0.0141	$1.2 \cdot 10^1 s$
SMILe	arc-eager	0.8180	0.0223	$8.4 \cdot 10^1 s$
DAgger	arc-eager	0.8317	0.0220	$2.3 \cdot 10^2 s$

Table 2: This table lists the mean recall, σ^* the standard deviation of the mean recall, and the mean execution time for arc-eager transitions.

Algorithm	Transition type	Mean recall	σ^*	Mean execution time
GTP	arc-hybrid	0.9260	0.0167	$2.8 \cdot 10^1 s$
FT	arc-hybrid	0.9319	0.0169	$8.5 \cdot 10^0 s$
SMILe	arc-hybrid	0.8933	0.0216	$1.2 \cdot 10^2 s$
DAgger	arc-hybrid	**0.9430**	0.0125	$2.4 \cdot 10^2 s$

Table 3: This table lists the mean recall, σ^* the standard deviation of the mean recall, and the mean execution time for arc-hybrid transitions.

7 Discussion

7.1 Findings

We found that greedy transition-based parsing (GTP) produced better recall on this data set than any of the no-regret imitation learning algorithms for the arc-eager transition system. For the arc-hybrid transition system, all methods except for SMILe were within two standard deviations of the mean of each other. This is surprising, because the no-regret algorithms have guarantees that GTP does not.

This may be because the other algorithms did not use enough training points to train classifiers. GTP used all of the data to train a single SVM. FT and SMILe only used part of the data on each SVM.

Forward training likely had problems for later time steps, because not all sequences of transitions were the same length, so fewer training examples were available at later time steps. Since an SVM was trained at each time step, later SVMs would be less accurate. Additionally, the fact that transition sequences were not aligned by structure indicates that classifiers in the middle time steps would need to be able to predict transitions from various different configurations, requiring more training examples.

For SMILe, we trained using a larger value of α (a smaller number of iterations) than there are theoretical guarantees for, which may explain its poor performance.

Of note is that all methods had worse performance on the arc-eager transition system, which has 4 transitions, while the arc-hybrid system has 3. The SVMs for the arc-eager system must perform multi-class classification on 4 classes, requiring 6 pairwise decision boundaries, while the arc-eager system only requires 3. The arc-eager system makes the problem more difficult. Additionally, GTP more clearly outperformed the other methods on the arc-eager system. This suggests that GTP would generalize more easily to more complex problems. Alternatively, it may indicate that with more data, the other methods would outperform GTP on the arc-eager system. It may be the case that the non-optimal configurations which the no-regret methods are trained on are very sparse within the space of non-optimal configurations, requiring large amounts of training data to do more good than harm in providing an accurate model of the configurations likely to be encountered by the parser.

As an interesting aside, we noticed during implementation that ambiguity in the choice of transition has a large effect on performance. There are some configurations which have multiple optimal transitions. If the dynamic oracle chooses a transition at random, we are trying to train a classifier to predict that randomness, which SVMs are not intended to do. On the other hand, when we set a deterministic rule for choosing transitions (in particular, we specified a priority order), recall increased by up to 5%. We also considered choosing the transition which the classifier currently ranks the highest out of the optimal transitions. However, this did not further affect recall.

7.2 Limitations

We did not tune the α parameter or number of iterations in DAgger, or the α or m parameters in SMILe. For DAgger we used an α parameter used in [16], and the number of iterations was determined by computational inefficiency. For SMILe, several values of α and m were tried to find acceptable values, but we did not perform explicit tuning on those variables.

7.3 Conclusion

It is possible to predict the high level structure of handwritten algebra data without predicting the symbols or using an explicit grammar outside of ground truth generation. Although it is not a no-regret method, greedy transition-based parsing gave similar results to the no-regret methods for the arc-hybrid transition system and better results for the arc-eager system.

7.4 Future

We have not explored using different features. We may be able to improve recall by adding features.

We currently use the same tuned parameters for all classifiers in the methods that train multiple classifiers. It is possible that we can see improvements by tuning parameters as sequences of values.

The algorithms we have implemented do not maintain state. Some state information is available from the features of configurations, but we may see improvements by using an algorithm that maintains internal state as it predicts transitions, such as kernel PSR [21].

We currently only use 'first-k' features of the configurations. In particular, the stack only contains the heads of merged dependent subtrees. We could include more long-term information by using features of subtrees. This would provide a natural way to use stroke level features instead of character level features, by viewing characters as subtrees of strokes. This approaches our goal of lowering the amount of required supervision.

References

[1] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, accessed 2015-10-24.

[2] CoNLL-2012 shared task: Data. `http://conll.cemantix.org/2012/data.html`, accessed 2015-10-24.

[3] Mitchell Marcus et al. The Penn Treebank Project. `https://www.cis.upenn.edu/~treebank/`, accessed 2015-10-24.

[4] Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. Greed is good if randomized: New inference for dependency parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[5] Abdelwaheb Belaid and Jean-Paul Haton. A syntactic approach for handwritten mathematical formula recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 6(1):105–111, 1984.

[6] Utpal Garain and Bidyut Baran Chaudhuri. Recognition of online handwritten mathematical expressions. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2366–2376, 2004.

[7] Kenichi Toyozumi, Naoya Yamada, Takayuki Kitasaka, Kensaku Mori, Yasuhito Suenaga, Kenji Mase, and Tomoichi Takahashi. A study of symbol segmentation method for handwritten mathematical formula recognition using mathematical structure information. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 630–633. IEEE, 2004.

[8] Yu Shi, HaiYang Li, and Frank K Soong. A unified framework for symbol segmentation and recognition of handwritten mathematical expressions. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 854–858. IEEE, 2007.

[9] Zhen Xuan Luo, Yu Shi, and Frank K. Soong. Symbol graph based discriminative training and rescoring for improved math symbol recognition. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 1953–1956. IEEE, 2008.

[10] Ryo Yamamoto, Shinji Sako, Takuya Nishimoto, and Shigeki Sagayama. On-line recognition of handwritten mathematical expressions based on stroke-based stochastic context-free grammar. In *tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[11] Ahmad-Montaser Awal, Harold Mouchere, and Christian Viard-Gaudin. Towards handwritten mathematical expression recognition. In *Document Analysis and Recognition, 2009. ICDAR'09. 10th International Conference on*, pages 1046–1050. IEEE, 2009.

[12] Ahmad-Montaser Awal, Harold Mouchère, and Christian Viard-Gaudin. A global learning approach for an online handwritten mathematical expression recognition system. *Pattern Recognition Letters*, 35:68–77, 2014.

[13] Francisco Álvaro, Joan-Andreu Sánchez, and José-Miguel Benedí. Recognition of on-line handwritten mathematical expressions using 2d stochastic context-free grammars and hidden markov models. *Pattern Recognition Letters*, 35:58–67, 2014.

[14] Harold Mouchere, Christian Viard-Gaudin, Dae Hwan Kim, Jin Hyung Kim, and Utpal Garain. Crohme2011: Competition on recognition of online handwritten mathematical expressions. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1497–1500. IEEE, 2011.

[15] Yoav Goldberg and Joakim Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1:403–414, 2013.

[16] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. In *In AISTATS*. Citeseer, 2011.

[17] Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics*, pages 661–668, 2010.

[18] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[19] MATLAB. *Version 8.4.0.150421 (R2014b)*. The MathWorks Inc., Natick, Massachusetts, 2014.

[20] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification, 2003.

[21] Byron Boots, Geoffrey Gordon, and Arthur Gretton. Hilbert space embeddings of predictive state representations. *arXiv preprint arXiv:1309.6819*, 2013.