Multilingual Dependency Parsing: A Pipeline Approach

MING-WEI CHANG, QUANG DO & DAN ROTH
University of Illinois at Urbana-Champaign

Abstract

This paper develops a general framework for machine learning based dependency parsing based on a pipeline approach, where a task is decomposed into several sequential stages. To overcome the error accumulation problem of pipeline models, we propose two natural principles for pipeline frameworks: (i) make local decisions as reliable as possible, and (ii) reduce the number of sequential decisions made. We develop an algorithm that provably satisfies these principles and show that the proposed principles support several algorithmic choices that improve the dependency parsing accuracy significantly. We present state of the art experimental results for English and several other languages.[1]

1 Introduction

A pipeline process over the decisions of learned classifiers is a common computational strategy in natural language processing. In this model a task is decomposed into several stages that are solved sequentially, where the computation in the ith stage typically depends on the outcome of computations done in previous stages. For example, a semantic role labeling program (Punyakanok et al. 2005) may start by using a part-of-speech tagger, then apply a shallow parser to chunk the sentence into phrases, identify predicates and arguments and then classify them to types. In fact, any left to right processing of an English sentence may be viewed as a pipeline computation as it processes a token and, potentially, makes use of this result when processing the token to the right.

The pipeline model is a standard model of computation in natural language processing for good reasons. It is based on the assumption that some decisions may be easier to make and that the outcome of earlier decision may sometimes be useful when making further decisions. Nevertheless, it is clear that it results in error accumulation and suffers from its inability to correct mistakes made in previous stages. Researchers have recently started to address some of the disadvantages of this model. Roth & Yih (2004) suggests a model in which global constraints are taken into account in a

[1] This paper extends and unifies our previous works (Chang et al. 2006a, 2006b).

later stage to fix mistakes due to the pipeline. Punyakanok et al. (2005) and Marciniak & Strube (2005) also address some aspects of this problem. However, these solutions rely on the fact that all decisions are made with respect to the same input; specifically, all classifiers considered use the same examples as their input. In addition, the pipelines they study are shallow.

This paper develops a general framework for decisions in pipeline models which addresses these difficulties. Specifically, we are interested in *deep pipelines*—a large number of predictions that are being chained.

A pipeline process is one in which decisions made in the ith stage (1) depend on earlier decisions and (2) feed on input that depends on earlier decisions. The latter issue is especially important at evaluation time since, at training time, a gold standard data set might be used to avoid this issue. We develop and study the framework in the context of a *bottom up approach to dependency parsing*. We suggest that two principles should guide the pipeline algorithm development:

(i) Make local decisions as reliable as possible.

(ii) Reduce the number of decisions made.

Using these as guidelines we devise an algorithm for dependency parsing, prove that it satisfies these principles, and show experimentally that this improves the accuracy of the resulting tree.

Specifically, our approach is based on a shift-reduced parsing as in (Yamada & Matsumoto 2003). Our general framework provides insights that allow us to improve their algorithm, and to principally justify some of the algorithmic decisions. Specifically, the first principle suggests to improve the reliability of the local predictions, which we do by improving the set of actions taken by the parsing algorithm, and by using a look-ahead search. The second principle is used to justify the control policy of the parsing algorithm, namely, which edges to consider at different stages in the algorithm. We prove that our control policy is optimal in some sense and, overall, that the decisions we made guided by these principles lead to a significant improvement in the accuracy of the resulting parse tree.

1.1 *Dependency parsing and pipeline models*

Dependency trees provide a syntactic representation that encodes functional relationships between words; it is relatively independent of the grammar theory and can be used to represent the structure of sentences in different languages. Dependency structures are more efficient to parse (Eisner 1996) and are believed to be easier to learn, yet they still capture much of the predicate-argument information needed in applications (Haghighi et al. 2005), which is one reason for the recent interest in learning these structures (Lin 1994, Eisner 1996, Yamada & Matsumoto 2003, Nivre & Scholz 2004, Mcdonald et al. 2005).

Eisner's work—$O(n^3)$ parsing time generative algorithm—embarked the interest in this area. His model, however, seems to be limited when dealing with complex and long sentences. Mcdonald et al. (2005) build on this work, and use a global discriminative training approach to improve the edges' scores, along with Eisner's algorithm, to yield an expected improvement.

A different approach was studied by Yamada & Matsumoto (2003), that develop a bottom-up approach and learn the parsing decisions between consecutive words in the sentence. Local actions are used to generate a dependency tree using a shift-reduce parsing approach (Aho et al. 1986). This is a true pipeline approach in that the classifiers are trained on individual decisions rather than on the overall quality of the parser, and chained to yield the global structure. Conceptually similar work was done in other successful parsers, e.g., (Ratnaparkhi 1997). Clearly, this approach suffers from the limitations of pipeline processing, such as accumulation of errors, but nevertheless, yields very competitive parsing results. A somewhat similar approach was used in (Nivre & Scholz 2004) to develop a hybrid bottom-up/top-down approach; there, the edges are also labeled with semantic types, yielding lower accuracy than the works mentioned above.

The overall goal of dependency parsing (DP) learning is to infer a tree structure. An example of a dependency tree is shown in Figure 1.

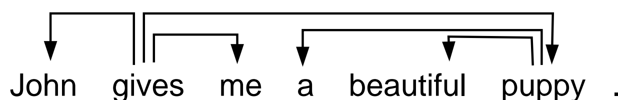

Fig. 1: *An example of a dependency tree*

A common way to do that is to predict with respect to each potential edge (i, j) in the tree, and then choose a global structure that (1) is a tree and that (2) maximizes some score. The first provides a set of structural constraints on the resulting structure, and the second provides a principled way to choose among (possibly many) resulting trees. In the context of DPs, this "edge based factorization method" was proposed by (Eisner 1996). In other contexts, this is similar to the approach of (Roth & Yih 2004) in that scoring each edge depends only on the raw data observed and not on the classifications of other edges, and that global considerations can be used to overwrite the local (edge-based) decisions.

The key in a pipeline model is that making a decision with respect to the edge (i, j) may gain from taking into account decisions already made with respect to neighboring edges. However, given that these decisions are noisy, there is a need to devise policies for reducing the number of predictions in order to make the parser more robust. This is exemplified in (Yamada &

Matsumoto 2003)—a bottom-up approach, that is most related to the work presented here. Their model is a "traditional" pipeline model—a classifier suggests a decision that, once taken, determines the next action to be taken (as well as the input the next action observes).

For many languages such as English, Chinese and Japanese (with a few exceptions), DPs without edge crossings, called *projective dependency trees*, are sufficient to analyze most sentences. Even for non-projective languages, usually the proportion of non-projective edges is low (Buchholz & Marsi 2006). As is common in most earlier work on DP, the work described here is also concerned mainly with projective trees. However, we also discuss an extension that deals with non-projective trees by converting them into projective trees (Nivre & Nilsson 2005).

In the rest of this paper, we propose and justify a framework for improving pipeline processing based on the principles mentioned above: (i) make local decisions as reliably as possible, and (ii) reduce the number of decisions made. We use the proposed principles to examine the (Yamada & Matsumoto 2003) parsing algorithm and show that the framework suggests modifying some of the decisions made there and, consequently, results in better overall dependency trees. After introducing the task and our general approach, Section 2 formally defines the model we consider, the pipeline dependency parsing approach and its properties. Our experimental results on English are presented in Section 3. In Section 4 we discuss a few extensions to the model, and experimental results on other languages. Section 5 concludes and discusses some future directions.

2 Dependency parsing as a pipeline model

This section describes our dependency parsing algorithm and justifies its advantages by viewing and analyzing it as a pipeline model.

Definition 1 *For words x, y in a sentence T, we introduce the following notation:*

$x \to y$: x is the *direct parent* of y.
$x \to^* y$: x is an *ancestor* of y.
$x \leftrightarrow y$: $x \to y$ or $y \to x$.
$x < y$: x is *to the left of* y in T.

We now introduce the concept of a projective language (Nivre 2003):

Definition 2 (Projective language)
$\forall a, b, c \in T, a \leftrightarrow b$ *and* $a < c < b$ *imply that* $a \to^* c$ *or* $b \to^* c$.

2.1 *A pipeline dependency parsing algorithm*

Our parsing algorithm is a modified shift-reduce parser (Aho et al. 1986) that applies a set of actions (described below) in a left to right manner on consecutive pairs of words (a, b) $(a < b)$ in a sentence. A machine learning algorithm is used to determine what actions (namely, parsing decisions) to take between two candidate words in the sentence, and a tree is thus constructed in a bottom up manner. The basic actions used in this model, as in (Yamada & Matsumoto 2003), are:

S*hift*: there is no relation between a and b, or the action is deferred because the child word of a and b still has some unlinked children. For example, if a is the parent of b and b has a child to the right of b, we must defer the action.

R*ight*: b is the parent of a (and a is thus eliminated from further consideration).

L*eft*: a is the parent of b (and b is thus eliminated from further consideration).

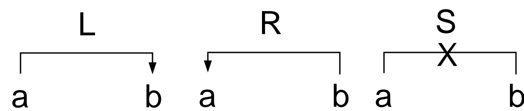

Fig. 2: *This figure demonstrates the basic parsing actions:* **L***eft,* **R***ight and* **S***hift*

These three actions are illustrated in Figure 2. It is important to note that one word (the child word) will be eliminated from T after performing **L***eft* or **R***ight*. We remove the child word because we already found its parent.

This is a true pipeline approach in that (1) the classifiers are trained on individual decisions rather than on the overall quality of the parse tree, and are chained to yield a global structure; and consequently, decisions made with respect to a pair of words affect what pair of words is considered next by the algorithm.

In order to complete the description of the algorithm we need to describe which edge to consider once an action is taken. We describe it via the notion of the *focus point*:

Definition 3 (Focus point) *When the algorithm considers the pair* (a, b)*,* $a < b$*, we call the word* a *the current 'focus point'.*

Next we describe several policies for determining the focus point of the algorithm following an action. We note that, with a few exceptions, determining

the focus point does not affect the *correctness* of the algorithm. It is easy to show that for (almost) any focus point chosen, if the correct action is selected for the corresponding word pair, the algorithm will eventually yield the correct tree (but may require multiple cycles through the sentence). In practice, the actions selected are noisy, and a wasteful focus point policy will result in a large number of actions, and thus in error accumulation. To minimize the number of actions taken, we want to find a good focus point placement policy.

We always let the focus point move one word to the right after **S**. After **L** or **R** there are several natural placement policies to consider for the focus point:

Start Over: Move focus point to the first word in T.

Stay: Move focus point to be the next word to the right. That is, for $T = (a, b, c)$, and the focus point is a, an **L** action results in a as the focus point (and the pair considers (a, c), while **R** action results in the focus being b (and the pair considered (b, c)).

Step Back: The focus point moves to the previous word (on the left). That is, for $T = (a, b, c)$, and the focus point is b, in both cases, a will be the focus point (and the pair considered are (a, c) in case of a **R** action, and (a, b) in case of a **L**).

In practice, different placement policies have a significant effect on the number of pairs considered by the algorithm and, therefore, on the final accuracy[2].

The following analysis justifies the **Step Back** policy. We claim that if **Step Back** is used, the algorithm will not waste any action. Thus, it achieves the goal of minimizing the number of actions taken in a pipeline algorithm. Notice that using this policy, when **L** is taken, the pair (a, b) is reconsidered, but with new information, since now it is known that c is the child of b. Although this seems wasteful, we will show in Lemma 1 that this movement is necessary in order to reduce the number of actions.

As mentioned above, each of these policies yields the correct tree. Table 1 provides an experimental comparison of the three policies in terms of the number of actions required to build a tree. As a function of the focus point placement policy, and using the correct action for each pair considered, it shows the number of word pairs considered in the process of generating all the trees for section 23 of Penn Treebank. It is clear from Table 1 that the policies result in very different number of actions and that **Step Back** is the best choice. Note that since the actions are the gold-standard actions,

[2] Note that Yamada & Matsumoto 2003) mention that they move the focus point back after **R**, but do not state what they do after executing **L** actions, and why. Yamada (p.c. 2006) indicates that they also moved the focus point back after **L**.

Policy	*#Shift*	*#Left*	*#Right*
Start over	156545	26351	27918
Stay	117819	26351	27918
Step back	43374	26351	27918

Table 1: *The number of actions required to build all the trees for the sentences in section* 23 *of Penn Treebank* (*Marcus et al. 1993*) *as a function of the focus point placement policy. The statistics are taken with the correct* (*gold-standard*) *actions*

the policy affects only the number of **S** actions used, and not the **L** and **R** actions, which are a direct function of the correct tree. The number of required actions in the test stage (when the actions taken are based on a learned classifier) shows the same trend and consequently the **Step Back** also gives the best dependency accuracy. The parsing algorithm with this policy is given Algorithm 1. Note that a good policy should always try to consider a child before considering its parent to save the number of decisions. As we will show later, the **Step Back** has several nice properties to be considered as a good policy.

2.2 *Correctness and pipeline properties*

We can prove several properties of our algorithm. First we show that the algorithm builds the dependency tree in only one pass over the sentence. Then, we show that the algorithm does not waste actions in the sense that it never considers a word pair twice *in the same situation*. Consequently, this shows that under the assumption of a perfect action predictor, our algorithm makes the smallest possible number of actions, among all algorithms that build a tree sequentially in one pass. Finally, we show that the algorithm only requires linear number of actions. Note that this may not be true if the action classifier is not perfect, and one can contrive examples in which an algorithm that makes several passes on a sentence can actually make fewer actions than a single pass algorithm. In practice, however, as our experimental data shows, this is unlikely.

The following analysis is made under the assumption that the action classifiers are perfect. While this assumption does not hold in practice, it makes the following analysis tractable, and still gives a very good insight into the practical properties of our algorithm, given that, as we show the action classifier is quite accurate, even if not perfect.

For example, in Lemma 1, we show that the algorithm requires only one round to parse a sentence under this assumption. In practice, in the evaluation stage, we allow the parsing algorithm to run multiple rounds. That is, when the focus point reaches the last word in T, we will set the

Algorithm 1 Dependency parsing with **Step Back** policy. *getFeatures* extracts the features describing the word pair currently considered; *getAction* determines the appropriate action for the pair; *assignParent* assigns a parent for the child word based on the action; and *deleteWord* deletes the child word in T at the *focus* once the action is taken. Note that if other policies are used, the algorithm needs to set $focus = 0$ after $focus = T$ if the tree is not completed.

```
t: a word token
focus: index into the sentence
T = {t_1, t_2, ..., t_|T|}: sentence
focus = 1
while focus < |T| do
  v = getFeatures(t_focus, t_focus+1)
  α = getAction(t_focus, t_focus+1, v)
  if α = L or α = R then
    assignParent(t_focus, t_focus+1, α)
    deleteWord(T, focus, α)
    // performing Step Back here
    if focus ≠ 1 then focus = focus − 1
  else
    if focus ≠ 1 then focus = focus − 1
  end if
end while
```

focus point at the beginning of the sentence if the tree is not complete. We found that over 99% of the sentences can be parsed in a single round and that, the average number of rounds needed to parse a sentence (average taken over all sentences in the test corpus) is 1.01. Therefore, we believe that the following analysis, done, under the assumption of golden actions can still be quite useful.

Lemma 1 *For projective languages, the dependency parsing algorithm that uses the* **Step Back** *policy completes the tree when it reaches the end of the sentence for the first time. That is, when the focus point is at the last word of T, the algorithm completes the tree.*

Before providing our argument, it is important to recall that one child word will be eliminated from T after performing **L***eft* or **R***ight*. The **Step Back** policy and this elimination procedure are the key points in this lemma.

Note that in Algorithm 1, we use the focus point to choose the word pair. Since we move the focus point in the word vector T, we can only consider "consecutive" pairs in T. Note that the elimination after **L***eft* or

R*ight* will generate new "consecutive" word pairs in T.
In order to prove the lemma, we need the following definition. We call a pair of words (a, b) a *free pair* if and only if there is a relation between a and b and the algorithm can perform either a **L** or a **R** actions on that pair when it is considered. There are two requirement so that we can say that the algorithm can perform either **L** or **R**. First, the word pair must be consecutive in the currently considered T. Second, the action can be perform right now (we do not need to defer it). Formally,

Definition 4 (Free pair) *A pair (a, b) considered by the algorithm is a free pair, if it satisfies all the following conditions:*

1. *$a \leftrightarrow b$*
2. *a, b are consecutive in T (but not necessarily in the original sentence).*
3. *If $a \rightarrow b$, no other word in T is the child of b. If $b \rightarrow a$, no other word in T is the child of a.*

Proof: It is easy to see that there is at least one *free pair* in T, with $|T| > 2$. The reason is that if no such pair exists, there must be three words $\{a, b, c\}$ s.t. $a \leftrightarrow b$, $a < c < b$ and $\neg(a \rightarrow c \vee b \rightarrow c)$. However, this violates the properties of a projective language.

Now assume $\{c, a, b\}$ are three consecutive words in T. We claim that when using **Step Back**, the focus point is always to the left of all free pairs in T. This is clearly true when the algorithm starts. Assume that (a, b) is the first *free pair* in T and let c be just to the left of a and b. Then, the algorithm will not make a **L** or **R** action before the focus point meets (a, b), and will make one of these actions then. It's possible that $(c, a \vee b)$ becomes a free pair after removing a or b in T so we need to move the focus point back. However, we also know that there is no *free pair* to the left of c. Therefore, during the algorithm, the focus point will always remain to the left of all free pairs. So, when we reach the end of the sentence, every free pair in the sentence has been taken care of, and the sentence has been completely parsed. □

Lemma 2 *All actions made by a dependency parsing algorithm that uses the Step Back policy are necessary. Specifically, a pair (a, b) will never be considered again unless there is an additional child linked to a or b.*

Proof: Note that if **R** or **L** is taken, either a or b will become a child word and be eliminated from further considerations by the algorithm. Therefore, if the action taken on (a, b) is **R** or **L**, this pair of words will never be considered again.

If **S** is taken, it is possible to consider the word pair (a, b) again. From Lemma 1, the algorithm will complete the tree in one round and a pair (a, b)

will be considered again only if the focus point "steps back". Therefore, we consider two cases here. First, if the next action followed **S** is **L**, b will get a new child and the algorithm will step back and consider (a, b) again. Since b gets new information, it is reasonable to check (a, b) again to see if the algorithm can recover the deferred action or not. Second, if the next action followed **S** is **R**, b will be eliminated from T and (a, b) will not be considered again. □

The next lemma claims a stronger property: the algorithm requires only $O(n)$ actions, where n is the length of the input sentence.

Lemma 3 *For a sentence T of size n, the number of actions used by a dependency parsing algorithm with the Step Back policy is bounded by $3n$.*

Proof: In order to calculate the number of actions, we consider the number of word pairs processed by the algorithms, taking into account word pairs that may be processed multiple times. At the beginning of the algorithm, the algorithm only has $n-1$ word pairs since it can only consider two consecutive words. Every time the algorithm performs an action other than **S**, some words will be eliminated and new word pairs will be generated. Furthermore, since a word gets new information (a new child), some word pairs may be considered again.

From Lemma 2, we know that a word pair will be considered again only if one of the words has a new child connected to it. Therefore, we only need to calculate the number of newly generated word pairs and the number of word pairs needed to be reconsidered.

Assume, w.l.o.g, that there are four words (a, b, c, d) and the focus point is at b, that is, the algorithm is considering (b, c). If the algorithm performs **R**, b becomes c's child and (a, c) becomes a new word pair. We also need to consider (c, d) (again, if we considered it before) since now c gets more information. Similarly, if **L** is performed, we need to reconsider (a, b) for the new information and examine the new word pair (b, d). In conclusion, after performing every **L** or **R** action we may need to (re)consider two additional word pairs.

Let m be the total number of word pairs which are needed to be considered. The previous discussion shows that we have:

$$m \leq n - 1 + 2l + 2r$$

where l and r are the number of **L** and **R** actions taken, respectively. Note that $l + r \leq n - 1$ since the sentence has only n words. Therefore, $m \leq 3n$ and the total number of actions is bounded by $3n$. □

2.3 *Improving the parsing action set*

So far we considered the "standard" set of three actions. While the **L** and **R** actions have a clear meaning, the **S** action covers several cases. When we use machine learning techniques to learn when to perform each action, this may result in an inaccurate decision with respect to this action. In order to improve the accuracy of the action predictors, we suggest a new (hierarchical) set of actions: *Shift, Left, Right, WaitLeft, WaitRight.* We believe that predicting these is easier due to finer granularity – the **S** action is broken into sub-actions in a natural way. The new actions have the following semantics:

W*ait***L***eft***:** $a < b$. a is the parent of b, but it's possible that b is a parent of other nodes. Action is deferred. Notice that if we mistakenly perform **L***eft* instead, the child of b can not find its parents later.

W*ait***R***ight:* $a < b$. b is the parent of a, but it's possible that a is a parent of other nodes. Similar to **WL**, action is deferred.

Consequently, we also change the meaning of the action **S** to take effect only if there is no relationship between a and b[3]. The new set of actions is shown to better support our parsing algorithm, when tested on different placement policies. When **W***ait***L***eft* or **W***ait***R***ight* is performed, the focus will move to the next word, as in **S**.

It is interesting to note that **W***ait***R***ight* is not needed in projective languages if **Step Back** is used. This gives another strong justification to use **Step Back**, since the action classification becomes more accurate—a more natural class of actions, with a smaller number of candidate actions. We show this property formally below.

Lemma 4 *If we apply the algorithm on projective languages, the* WaitRight *action is not necessary if the algorithm uses Step Back policy.*

Proof: Assume w.l.o.g. that the target sentence is (a, b, c) with b as the focus word, and that the algorithm needs to perform **W***ait***R***ight*. That it, c is the parent of b and b is the parent of other words (for example, a). This implies that there is a free pair to the left of the focus point, contradicting Lemma 1. Note that Lemma 1 still holds since **W***ait***R***ight* and **W***ait***L***eft* behave as **S***hift*. Therefore, the algorithm never needs to use the **W***ait***R***ight* action. □

2.4 *Improving the actions' accuracy: A pipeline model with look ahead*

Once the parsing algorithm, along with the focus point policy, is determined, it only remains to determine a way to choose an action. We do this using

[3] Interestingly, Yamada & Matsumoto (2003) mention the possibility of an additional *single* **W***ait* action, but it is not added it to the model considered there.

a machine learning algorithm. Given an annotated corpus we use of the parsing algorithm to determine the action needed for each consecutive pair; this is used to train a classifier to predict one of the possible actions. The details of the classifier and the feature used are given in Section 3.

When the learned model is evaluated on new data, the sentence is processed left to right and the parsing algorithm, along with the action classifier, are used to produce the dependency tree. The parsing process is somewhat more involved, since the action classifier is not used as is, but rather via a look ahead inference step described next.

The advantage of a pipeline model is that it can use more information, based on the outcomes of previous predictions. As discussed earlier, this may result in error accumulation. The importance of having a reliable action predictor in a pipeline model motivates the following approach. We devise a look ahead algorithm and use this policy as a way to determine the predicted action more accurately. This approach can be used in any pipeline model but we illustrate it below in the context of our dependency parser.

The following example illustrates a situation in which an early mistake in predicting an action causes a chain reaction and results in further mistakes. This stresses the importance of correct early decisions, and motivates our look ahead policy.

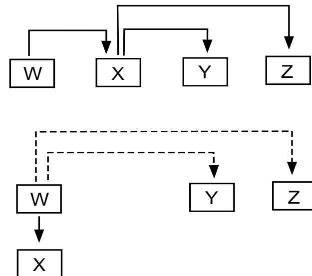

Fig. 3: *Top: correct dependency relations between w, x, y and z. Bottom: if the algorithm mistakenly decides that x is a child of w before deciding that y and z are x's children, we cannot find the correct parent for y and z*

Let (w, x, y, z) be a sentence of four words, and assume that the correct dependency relations are as shown in the top part of Figure 3. If the system mistakenly predicts that x is a child of w before y and z becomes x's children, we can only consider the relationship between w and y in the next stage. Consequently, we will never find the correct parent for y and z. The previous prediction error propagates and impacts future predictions. On the other hand, if the algorithm makes a correct prediction, in the next

Algorithm 2 Look ahead algorithm. $\mathbf{y}$ represents an action sequence. The function *search* considers all possible action sequences with $|depth|$ actions and returns the sequence with the highest score.

```
Algo predictAction(Model, Depth, State)
x = getNextFeature(State)
y = search(x, Depth, Model, State)
lab = y[1]
State = update(State, lab)
return lab

Algo search(x, Depth, Model, State)
maxScore = −∞
F = {y | ‖y‖ = Depth}
for y in F do
   s = 0, TmpState = State
   for  i = 1 ... Depth do
      x = getNextFeature(TmpState)
      s = s+ score(y[i], x, Model)
      TmpState = update(TmpState, y[i])
   end for
   if s > maxScore then
      ŷ = y
      maxScore = s
   end if
end for
return ŷ
```

stage, we do not need to consider w and y. As shown, getting useful rather than misleading information in a pipeline model, requires correct early predictions. Therefore, it is advised to utilize some inference framework that may help minimizing the error accumulation problem.

In order to improve the accuracy of the action prediction, we might want to examine all possible combinations of action sequences and choose the one that maximizes some score. It is clearly intractable to find the global optimal prediction sequences in a pipeline model of the depth we consider. Therefore, we use a look ahead strategy, implemented via a local search framework, which uses additional information but is still tractable.

The look-ahead search algorithm is presented in Algorithm 2. The algorithm accepts three parameters, *Model*, *Depth* and *State*:

Model: is our learning model—the classifier that is used to predict the action. We assume a classifier that can give a confidence in its prediction

(see Section 3).

Depth: The parameter *Depth* determines the depth of the search procedure.

State: encodes the configuration of the environment (in the context of the dependency parsing this includes the sentence, the focus point and the current parent and children for each word). Note that *State* changes when a prediction is made and that the features extracted for the action classifier also depend on *State*.

The search algorithm will perform a search of length *Depth*. Additive scoring is used to score the sequence, and the first action in this sequence is selected and performed. Then, the *State* is updated, the new features for the action classifiers are computed and *search* is called again.

One interesting property of this framework is that it allows the use of future information in addition to past information. The pipeline model naturally allows access to all the past information. Since the algorithm uses a look ahead policy, it also uses future predictions. The significance of this becomes clear in Section 3. There are several parameters, in addition to *Depth* that can be used to improve the efficiency of the framework. For example, given that the action predictor is a multi-class classifier, we do not need to consider all future possibilities in order to determine the current action. In our experiments, we only consider two actions with highest score at each level (which was shown to produce almost the same accuracy as considering all four actions). Note that our look ahead search is a local search so the depth is usually small. Furthermore, we can keep a cache of the search paths of the look-ahead search for the i-th action. These paths can be reused in the look-ahead search for the $(i+1)$-th action. Therefore, our look ahead search is not very expensive.

3 Experimental study

In this section, we describe a set of experiments done to investigate the properties of our algorithm and evaluate it. All the experiments here were done on English. Results on other languages are described in Section 4.

We use the standard corpus for this task, the Penn Treebank (Marcus et al. 1993). The training set consists of sections 02 to 21 and the test set is section 23. The POS tags for the evaluation data sets were provided by the tagger of (Toutanova et al. 2003) (which has an accuracy of 97.2% on section 23 of the Penn Treebank).

3.1 *Learning algorithm*

As our learning algorithm we use a regularized variation of the perceptron update rule, as incorporated in SNoW (Roth 1998, Carlson et al. 1999)[4], a multi-class classifier that is tailored for large scale learning tasks and has been used successfully in a large number of NLP tasks, e.g., (Punyakanok et al. 2005). SNoW uses softmax over the raw activation values as its confidence measure, which can be shown to produce a reliable approximation of the labels' conditional probabilities. The softmax score can be written as follows:

$$score(x, lab) = p_i = \frac{e^{act_{lab}}}{\sum_{1 \leq i \leq n} e^{act_i}}$$

where act_i means the normalized activation value for i-th class of feature vector x. The activation value is directly come from our machine learning model.

3.2 *Features*

For each word pair (w_1, w_2) we use the words, their POS tags and also these features of the children of w_1 and w_2. We also include the lexicon and POS tags of 2 words before w_1 and 4 words after w_2 (as in (Yamada & Matsumoto 2003)). The key additional feature we use, relative to (Yamada & Matsumoto 2003), is that we include the previous predicted action as a feature. We also add conjunctions of above features to ensure expressiveness of the model. Yamada & Matsumoto (2003) make use of polynomial kernels of degree 2 which is equivalent to using even more conjunctive features (Cumby & Roth 2003). Overall, the average number of active features in an example is about 50.

3.3 *Evaluation methodology*

We use the same evaluation metrics as in (Mcdonald et al. 2005). Dependency accuracy is the proportion of non-root words that are assigned the correct head. Sentence accuracy indicates the fraction of sentences that have a complete correct analysis. We also measure the root accuracy and leaf accuracy, as in (Yamada & Matsumoto 2003). When evaluating the result, we exclude the punctuation marks as done in (Mcdonald et al. 2005) and (Yamada & Matsumoto 2003). The punctuation include comma, colon, dot, quotation, and double quotation. All other symbols are scoring tokens.

---

[4] The package can be downloaded from http://L2R.cs.uiuc.edu/∼cogcomp/software.php

3.4 *Results*

We first present the results of several of the experiments that were intended to help us analyze and understand some of the design decisions in our pipeline algorithm.

	Action set	
Evaluation metric	w/o *WaitLeft*	w/ *WaitLeft*
Dependencies	90.27	90.53
Root	90.73	90.76
Sentence	39.28	39.74
Leaf	93.87	93.94

Table 2: *The significance of the action WaitLeft*

To see the effect of the additional action, we present in Table 2 a comparison between a system that does not have the *WaitLeft* action (similar to the Yamada & Matsumoto's (2003) approach) with one that does. For a fair comparison, in both cases, we do not use the look ahead procedure. Note that, as stated above, the action *WaitRight* is never needed for our parsing algorithm. It is clear that adding *WaitLeft* increases the accuracy significantly.

Evaluation metric	*Depth*=1	*Depth*=2	*Depth*=3	*Depth*=4
Dependencies	90.53	90.67	90.69	90.79
Root	90.76	91.51	92.05	92.26
Sentence	39.74	40.23	40.52	40.68
Leaf	93.94	93.96	93.94	93.95

Table 3: *The effect of different Depth settings. Increasing the Depth usually improves the accuracy*

Table 3 investigates the effect of the look ahead, and presents results with different *Depth* parameters (*Depth*= 1 means "no search"), showing a consistent trend of improvement.

Table 4 breaks down the results as a function of the sentence length; it is especially noticeable that the system also performs very well for long sentences, another indication for its global performance robustness.

	Sentence length				
Evaluation metric	<11	11-20	21-30	31-40	> 40
Dependencies	93.4	92.4	90.4	90.4	89.7
Root	96.7	93.7	91.8	89.8	87.9
Sentence	85.2	56.1	32.5	16.8	8.7
Leaf	94.6	94.7	93.4	94.0	93.3

Table 4: *The effect of sentence length* (*Depth* = 4). *Note that our algorithm performs very well on long sentences*

Table 5 shows the results with three settings of the POS tagger. The best result is, naturally, when we use the gold standard also in testing. However, it is worthwhile noticing that it is better to train with the same POS tagger available in testing, even if its performance is somewhat lower.

	Sources of tags (*train-test*)		
Evaluation metric	gold−pos	pos−pos	gold−gold
Dependencies	90.7	90.8	92.0
Root	92.0	92.3	93.9
Sentence	40.8	40.7	43.6
Leaf	93.8	94.0	95.0

Table 5: *Comparing different sources of POS tagging in a pipeline model. We use "gold" if true tags are applied and we use "pos" if tags are generated by a trained tagger (Depth= 4 for all experiments in this table). It is better to use the POS used in the evaluation also when training*

	Dependency parsing systems			
Evaluation metric	Y&M03	N&S04	M&C&P05	Current
Dependencies	90.3	87.3	90.9	90.8
Root	91.6	84.3	94.2	92.3
Sentence	38.4	30.4	37.5	40.7
Leaf	93.5	n/a	n/a	94.0

Table 6: *Comparison between the current work and other dependency parsers. Our system performs especially well at the sentence level*

Finally, Table 6 compares the performances of several of the state of the art dependency parsing systems with ours. When comparing with other dependency parsing systems it is especially worth noticing that our system gives significantly better accuracy on completely parsed sentences.

4 Extensions: Non-projective trees and edge labels

Some languages such as Czech are very different from English. In this section, we address two major difficulties one needs to deal with in order to extend the approach we described here to other languages. First, the dependency graph in some languages may have multiple roots. In this case, we need to refer to the dependency edges of a sentence as a dependency graph. Second, some languages are non-projective . We discuss below how to overcome these two difficulties within the approach described in this work. Fortunately, the multi-root problem is not difficult as it might seem. In fact, the algorithm described earlier can handle multiple roots problem if the edges are projective. Assume the dependency graph of a sentence has multiple roots and that the language is projective. Then, we can view

the sentence as a union of several small sentences with no dependencies among them, where each of them has a single-rooted dependency tree. The algorithm can generate multiple trees by performing **S**hift between root words.

In order to deal with crossing edges we can use one of two possible approaches. The first one is to introduce some additional actions to handle the long distance relationships. The new actions, along with **S**, **R** and **L**, could handle most of the non-projective trees efficiently (Attardi 2006). Another approach is to convert non-projective trees into projective ones (Nivre & Nilsson 2005). In this article, we adopt the second method, as described below. Any projective dependency tree can be mapped into a projective tree by using the *Lift* operation, which is defined as follows:

Lift$(w_j \rightarrow w_k) = \mathbf{parent}(w_j) \rightarrow w_k$, where $a \rightarrow b$ means that a is the parent of b, and **parent** is a function which returns the parent word of the given word.

The procedure is as follows. First, the mapping algorithm examines if there is a crossing edge in the current tree. If there is a crossing edge, it will perform repeated *Lift* operations and replace this edge until the tree becomes projective. Let E be the dependency graph of a sentence. The algorithm for converting E into a projective graph is as follows (Nivre & Nilsson 2005):

1. If E is projective, exit. Otherwise,
2. Choose the smallest non-projective edge e, and
3. Let e' be the new edge after performing *Lift* on e.
4. Let $E \leftarrow E - \{e\} \cup \{e'\}$. Goto 1.

After we convert every training non-projective sentence in the training data, we train the model as before, on the modified data. In the evaluation phase, we directly apply the trained model to predict the dependency, resulting in projective graphs. In this paper, we do not try to recover the information lost during the mapping procedure.

4.1 *Dealing with dependency labels*

It is often desirable to predict also the dependency label (type) corresponding to the edges in the parse tree. Dependency labels can be very informative and include, in English, labels such as the "object" of a verb or the "subject" of a verb. In this paper, we evaluated a simple two-stage framework to predict the dependency labels.

We view labeling the type of the dependencies as a post-task after the phase of predicting the head for each token in the sentence. Thus, it is a multi-class classification task. The number of the dependency types for each language can be found in (Buchholz & Marsi 2006). In the phase of

learning dependency types, the parent of each token, which was labeled in the first phase, will be used among the features. The predicted actions can therefore help us to make accurate predictions of the dependency types.

4.2 *Experimenting with multilingual dependency parsing*

In this section, we report the result of applying our system, along with the simple extensions described above in CoNLL-X shared task (Buchholz & Marsi 2006). The target dataset consists of data in 12 languages (English is not included), listed in Table 7. The resources provided for the 12 languages are described in (Hajič et al. 2004, Chen et al. 2003, Böhmová et al. 2003, Kromann 2003, van der Beek et al. 2002, Brants et al. 2002, Kawata & Bartels 2000, Afonso et al. 2002, Džeroski et al. 2006, Civit Torruella & Martí Antonín 2002, Nilsson et al. 2005, Oflazer et al. 2003, Atalay et al. 2003). We apply the techniques earlier in this section to extend our algorithm to on non-projective languages and to the prediction of both head words and dependency labels.

Language	UAS			LAS		
	Ours	AV	SD	Ours	AV	SD
Arabic	76.09	73.48	4.94	60.92	59.94	6.53
Chinese	89.60	84.85	5.99	85.05	78.32	8.82
Czech	81.78	77.01	6.70	72.88	67.17	8.93
Danish	86.85	84.52	8.97	80.60	78.31	11.34
Dutch	76.25	75.07	5.78	72.91	70.73	6.66
German	86.90	82.60	6.73	84.17	78.58	7.51
Japanese	90.77	89.05	5.20	89.07	85.86	7.09
Portuguese	88.60	86.46	4.17	83.99	80.63	5.83
Slovene	80.32	76.53	4.67	69.52	65.16	6.78
Spanish	83.09	77.76	7.81	79.72	73.52	8.41
Swedish	89.05	84.21	5.45	82.31	76.44	6.46
Turkish	73.15	69.35	5.51	60.51	55.95	7.71

Table 7: *Our results are compared with the average scores. UAS=Unlabeled Attachment Score, LAS=Labeled Attachment Score, AV=Average score, and SD=standard deviation*

We emphasize that no special language enhancement has been applied for any language in this experiment. We use the same features and the same parameters for all languages. In the dataset used, some languages have additional morphological information. The information is stored in the column FEAT. We incorporated the information of FEAT for the languages when it is available. The evaluation presented here is done using the script provided by CoNLL-X and is thus somewhat different from the evaluation

used in Section 3. Two kinds of evaluation metrics are used here:

$$LAS = \frac{\text{number of tokens with correct head and correct labels}}{\text{number of tokens}}$$

and

$$UAS = \frac{\text{number of tokens with correct head}}{\text{number of tokens}}$$

Note that UAS is the proportion of tokens with correct head. The only difference between UAS and dependency accuracy is that UAS does consider root words.

4.3 *Multilingual results*

Table 7 shows our results on UAS and LAS. Our results are compared with the average scores (AV) and the standard deviations (SD) of all the systems that took part in the shared task of CoNLL-X. Our average UAS for 12 languages is 83.54% with the standard deviation 6.01; and 76.80% with the standard deviation 9.43 for average LAS. Comparing our results to other systems that participated in the tasks (Buchholz & Marsi 2006), our system performs better than the average of systems participated in CoNLL-X for every language. Perhaps surprising given that no tuning to any specific langauge was made.

Method	UAS	LAS
w/o *WaitLeft*	87.95	81.48
w/ *WaitLeft*	88.47	81.82
w/ *WaitLeft* + search	89.07	82.31

Table 8: *An algorithmic analysis on a Swedish dataset* (*Depth*= 3)

To provide some more understanding in the context of other languages, we show next some specific cases. Table 8 shows the results on the Swedish dataset. We compare three systems. The first system used only the three standard actions, without look-ahead search. The second system used the improved action set. The best system used improved action set and look-ahead search. Thus, we observe the same trends as we observed when running these experiments on the English dataset.

5 Conclusions and further work

We have addressed the problem of using learned classifiers in a pipeline fashion, where a task is decomposed into several stages and stage classifiers are used sequentially, with each stage using the outcome of previous stages

as its input. This is a common computational strategy in natural language processing and is known to suffer from error accumulation and an inability to correct mistakes in previous stages.

We abstracted two natural principles, one which calls for making the local classifiers used in the computation more reliable and a second, which suggests to devise the pipeline algorithm in such a way that minimizes the number of decisions (actions) made.

In this work we studied this framework in the context of designing a *bottom up dependency parsing*. Not only we manage to use this framework to justify several design decisions, but we also showed experimentally that following these principles results in improving the accuracy of the inferred trees relative to existing models.
Interestingly, we can show that the trees produced by our algorithm are relatively good even for long sentences, and that our algorithm is doing especially well when evaluated globally, at a sentence level, where our results are significantly better than those of existing approaches—perhaps showing that the design goals were achieved.

We also present the primarily results for the multilingual datasets provided in the CoNLL-X shared task. This shows that dependency parsers based on a pipeline approach can handle many different languages. Recently, (McDonald & Pereira 2006) proposed using "second-order" feature information when applying the Einser's algorithm and a global reranking algorithm. The proposed method outperform (Mcdonald et al. 2005) and the approach proposed in this paper. Although there is more room for comparison, we believe the the key advantage there is simply using a better feature set. Overall, the bottom line from this work, as well as from the CoNLL-X shared task is that a pipeline approach is competitive with a global reranking approach (Buchholz & Marsi 2006).

Beyond enhancing our current results by introducing more and better features, we intend to consider incorporating more information sources (e.g., shallow parsing results) into the pipeline process, to investigate different search algorithms in the pipeline framework and to study approaches to predicts dependencies and their type as a single task.

Acknowledgements. We are grateful to Ryan McDonald and the organizers of CoNLL-X for providing the annotated data set and to Vasin Punyakanok for useful comments and suggestions. This research was supported by the Advanced Research and Development Activity (ARDA)'s Advanced Question Answering for Intelligence (AQUAINT) Program and a DOI grant under the Reflex program.

REFERENCES

Abeillé, A. ed. 2003. *Treebanks: Building and Using Parsed Corpora.* (= *Text,*

Speech and Language Technology, 20). Dordrecht: Kluwer Academic.

Afonso, S., E. Bick, R. Haber & D. Santos. 2002. "Floresta sintá(c)tica: A treebank for Portuguese". *3rd Int. Conf. on Language Resources and Evaluation* (*LREC*), 1698-1703. Las Palmas, Canary Islands, Spain.

Atalay, N. B., K. Oflazer & B.Say. 2003. "The Annotation Process in the Turkish Treebank". *4th Int. Workshop on Linguistically Interpreteted Corpora* (*LINC*). Budapest, Hungary.

Aho, A.V., R. Sethi & J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley.

Attardi, Giuseppe. 2006. "Experiments with a Multilanguage Non-projective Dependency Parser". *10th Conf. on Computational Natural Language Learning* (*CoNLL-X*), 166-170. New York.

Brants, S., S. Dipper, S. Hansen, W. Lezius & G. Smith. 2002. "The TIGER Treebank". *1st Workshop on Treebanks and Linguistic Theories* (*TLT*). Sozopol, Bulgaria.

Böhmová, A., J. Hajič, E. Hajičová & B. Hladká. "The PDT: A 3-level Annotation Scenario". In (Abeillé 2003), ch. 7.

Buchholz, Sabine & Erwin Marsi. 2006. "Conll-X Shared Task on Multilingual Dependency Parsing". *10th Conference on Computational Natural Language Learning* (*CoNLL-X*), 149-164. New York.

Carlson, A., C. Cumby, J. Rosen & D. Roth. 1999. "The SNoW learning architecture". Technical Report UIUCDCS-R-99-2101. Urbana, Illinois: University of Illinois at Urbana-Champaign (UIUC) Computer Science Department.

Chang, M.-W., Q. Do & D. Roth. 2006a. "A Pipeline Framework for Dependency Parsing". *Annual Meeting of the Association for Computational Linguistics* (*ACL*), 65-72. Sydney, Australia.

Chang, M.-W., Q. Do & D. Roth. 2006b. "A Pipeline Model for Bottom-up Dependency Parsing". *10th Conf. on Computational Natural Language Learning* (*CoNLL-X*), 186-190. New York City.

Chen, K., C. Luo, M. Chang, F. Chen, C. Chen, C. Huang & Z. Gao. 2003. "Sinica Treebank: Design Criteria, Representational Issues and Implementation". In (Abeillé 2003), ch. 13, 231-248.

M. Civit Torruella & M[a] A. Martí Antonín. 2002. "Design Principles for a Spanish Treebank". *1st Workshop on Treebanks and Linguistic Theories* (*TLT*). Sozopol, Bulgaria.

Cumby, C. & D.Roth. 2003. "On Kernel Methods for Relational Learning". *Int. Conf. on Machine Learning* (*ICML*), 107-114. Washington, DC, USA..

Džeroski, S., T. Erjavec, N. Ledinek, P. Pajas, Z. Žabokrtsky & A. Žele. 2006. "Towards a Slovene Dependency Treebank". *5th Int. Conf. on Language Resources and Evaluation* (*LREC*).Geneva, Switzerland.

Eisner, J. 1996. "Three New Probabilistic Models for Dependency Parsing: An Exploration". *Int. Conf. on Computational Linguistics* (*COLING*), 340-345. Copenhagen, Denmark.

Haghighi, A., A. Ng & C. Manning. 2006. "Robust Textual Inference via Graph Matching". *Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, 387-394. Vancouver, Canada.

Hajič, J., O. Smrž, P. Zemánek, J. Šnaidauf & E. Beška. 2004. "Prague Arabic Dependency Treebank: Development in Data and Tools". *NEMLAR Int. Conf. on Arabic Language Resources and Tools*, 110-117. Cairo, Egypt.

Kawata, Y. & J. Bartels. 2000. "Stylebook for the Japanese Treebank in VERBMOBIL". Verbmobil-Report 240. Tübingen, Germany: Seminar für Sprachwissenschaft, Universität Tübingen.

Kromann, Matthias Trautner. 2003. "The Danish Dependency Treebank and the DTAG Treebank Tool". *2nd Workshop on Treebanks and Linguistic Theories* (*TLT*) ed. by Joakim Nivre & Erhard Hinrichs, 217-220. Växjö, Sweden.

Lin, D. 1994. "Principar—An Efficient, Broad-coverage, Principle-based Parser". *Int. Conf. on Computational Linguistics* (*COLING*), 482-488. Kyoto, Japan.

McDonald, R., K. Crammer & F. Pereira. 2005. "Online Large-margin Training of Dependency Parsers". *Annual Meeting of the Association for Computational Linguistics* (*ACL'05*), 91-98. Ann Arbor, Michigan.

McDonald, R. & F. Pereira. 2006. "Online Learning of Approximate Dependency Parsing Algorithms". *11th Conf. of the European Chapter of the Association for Computational Linguistics* (*EACL'06*), 81-88. Trento, Italy.

Marciniak, T. & M. Strube. 2005. "Beyond the Pipeline: Discrete Optimization in NLP". *9th Conference on Computational Natural Language Learning* (*CoNLL-2005*), 136-143. Ann Arbor, Michigan.

Marcus, M. P., B. Santorini & M. Marcinkiewicz. 1993. "Building a Large Annotated Corpus of English: The Penn Treebank". *Computational Linguistics* 19:2.313-330.

Nilsson, J., J. Hall & J. Nivre. 2005. "MAMBA Meets TIGER: Reconstructing a Swedish Treebank from Antiquity". *15th Nordic Conference of Computational Linguistics* (*NODALIDA*) *Special Session on Treebanks.* Copenhagen Studies in Language 32, 119-132. Copenhagen, Denmark.

Nivre, Joakim. 2003. "An Efficient Algorithm for Projective Dependency Parsing". *Int. Workshop on Parsing Technology* (*IWPT*), 149-160. Nancy, France.

Nivre, J. & J. Nilsson. 2005. "Pseudo-projective Dependency Parsing". *43rd Annual Meeting of the Association for Computational Linguistics* (*ACL*), 99-106. Ann Arbor, Michigan.

Nivre, J. & M. Scholz. 2004. "Deterministic Dependency Parsing of English Text". *20th Int. Conference on Computational Linguistics* (*COLING-04*), 64-70. Geneva, Switzerland.

Oflazer, K., B. Say, D. Zeynep Hakkani-Tür & G. Tür. 2003. "Building a Turkish Treebank". In (Abeillé 2003), ch. 15.

Punyakanok, V., D. Roth & W. Yih. 2005. "The Necessity of Syntactic Parsing for Semantic Role Labeling". *Int. Joint Conference on Artificial Intelligence* (*IJCAI*), 1117-1123. Edinburgh, Scotland, UK.

Ratnaparkhi, Adwait. 1997. "A Linear Observed Time Statistical Parser Based on Maximum Entropy Models". *2nd Conf. on Empirical Methods in Natural Language Processing* (*EMNLP*), 1-10. Somerset, New Jersey.

Roth, Dan. 1998. "Learning to Resolve Natural Language Ambiguities: A Unified Approach". *National Conference on Artificial Intelligence* (*AAAI*), 806-813. Montréal, Québec, Canada.

Roth, D. & W. Yih. 2004. "A Linear Programming Formulation for Global Inference in Natural Language Tasks". *Annual Conf. on Computational Natural Language Learning* (*CoNLL*) ed. by Hwee Tou Ng & Ellen Riloff, 1-8. Boston, Mass.

Toutanova, K., D. Klein & C. Manning. 2003. "Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network". *Human Language Technology Conference* (*HLT-NAACL*), 173-180. Edmonton, Canada.

van der Beek, L., G. Bouma, R. Malouf & G. van Noord. 2002. "The Alpino Dependency Treebank". *Computational Linguistics in the Netherlands* (*CLIN*). Groningen, The Netherlands.

Yamada, H. & Y.Matsumoto. 2003. "Statistical Dependency Analysis with Support Vector Machines". *Int. Workshop on Parsing Technologies* (*IWPT*), 195-206. Nancy, France.

Index of Subjects and Terms

D.
dependency parsing 2

E.
Eisner's algorithm 3

L.
look-ahead search 13

N.
non-projective 17

P.
Penn Treebank 14
pipeline process 1
projective language 4

S.
shift-reduce parser 5