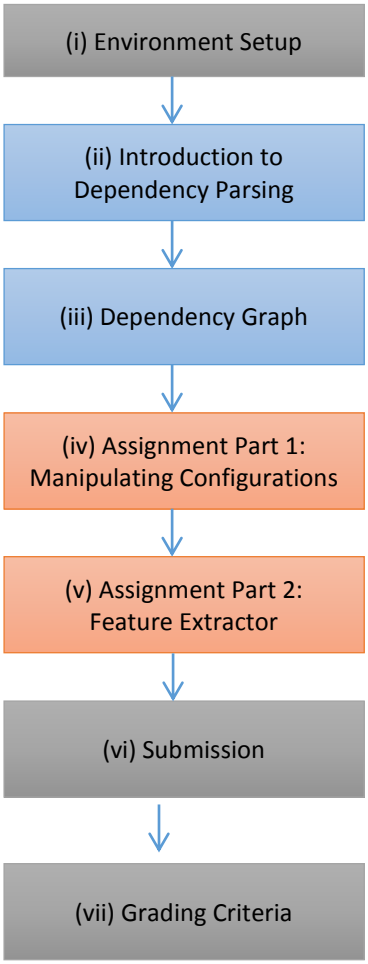
Introduction

In this assignment, we'll be implementing a Dependency Parsing algorithm. This document includes the following sections:

Section A: Background
- a) Introduction to dependency parsing
- b) Understanding and generating dependency graphs

Section B: Implementation of Dependency Parser
- c) Implementing transition operations
- d) Improving performance by experimenting with feature extractors

Structure of this document:



Setup everything to get you ready for this assignment

A Quick introduction to Dependency Parsing, a well-known problem in Natural Language Processing.

Introduction to and tutorial on generating dependency graphs.

Complete the implementation of transition operations, the key part of the Nivre dependency parser, and run test.

Improve the performance of your parser by experimenting with different features in feature extractor.

Follow the instructions to submit. Be sure to submit everything needed and set the right permissions.

Grading criteria for assignment 2

BACKGROUND INFORMATION

(ii) Introduction to Dependency Parsing

Dependency parsing is a well-known problem in Natural Language Processing. It was the shared task of CoNLL for two consecutive years, and provides a fun introduction to more complex parsing algorithms. For this assignment, we will be implementing a form[1] of Joakim Nivre's arc-eager transition-based dependency parser. You are encouraged to refer to his paper "A Dynamic Oracle for Arc-Eager Dependency Parsing"[2] for details of the algorithm.

Parsing is a general problem in computer science wherein we take in an input of a sequence of symbols, and analyze their structure based on some formal grammar. An example of this problem is in the study of compilers, where the compiler parses the source code and transforms it into an abstract syntax tree.

The current state-of-the-art compilers almost universally use variants of shift-reduce parsing algorithms. In a shift-reduce algorithm, the parse tree is constructed bottom-up by either *shift*ing data onto the stack, or by *reduc*ing the data using a rule in the formal grammar. The parser continues until all of the input has been consumed, and all parse trees have been reduced to a single parse tree that encompasses all of the input.

In natural language processing, dependency parsing is the problem of taking sentences and determining which parts depend on others, and in what way. For example, in the phrase "the dog", the word "the" is dependent on "dog", and furthermore the dependency relation implies that "the" is the determiner for "dog". As humans reading English, we naturally determine the dependency relations of the sentences we read so as to infer their intrinsic meaning.

Unfortunately, unlike in compilers, we do not know the formal grammar that describes which parts of a sentence are dependent on which other parts. This is especially difficult because we would like to be able to parse sentences in languages that we are not personally experienced in. As a result, we need to infer the grammar from data.

Dependency parsing as a supervised learning problem
While in theory we could describe the creation of a dependency parser directly as a supervised machine learning problem, it turns out that this is more complex and less effective than a slightly

[1] This implementation is based in part on work by Long Duong (University of Melbourne), Steven Bird (University of Melbourne) and Jason Narad (University College London).
[2] Yoav Goldberg and Joakim Nivre, "A Dynamic Oracle for Arc-Eager Dependency Parsing" Proceedings of COLING 2012: Technical Papers, pages 959–976, COLING 2012, Mumbai, December 2012

modified form of the problem.

Thus, we change the shift-reduce parser as follows: instead of allowing only shift and reduce operations based on a formal grammar, we have four classes of "transitions" – shift, reduce, arc left (label), and arc right (label), where arc left and arc right represent arcs in the dependency graph with a given label.

The supervised learning problem is therefore:

Input:

A list of sentences with their dependency relations and part of speech tags

Output:

A function $f: C \rightarrow T$, where C is the set of parser configurations, T is the set of transitions, and f returns the best transition at the given parser configuration.

(iii) Dependency Graphs

In brief, we are trying to take sentences in various languages and construct their dependency graphs. To help build an intuition for what exactly a dependency graph is, we have included a java jar file that visualizes a dependency graph for data in the CoNLL format; however, it depends on having an X11 server available i.e.

```
ssh -XC UNI@clic.cs.columbia.edu
```

On Windows, MobaXTerm (http://mobaxterm.mobatek.net/) includes X tunneling. On OSX, you may need to install XQuartz (http://xquartz.macosforge.org/landing/) to get the display to show up. If neither of these options works, you can also go to the physical CLIC lab and run the relevant commands. You can also run this Java code on a local machine.

Visualize gold dependency parse (test data)

```
java -jar MaltEval.jar -g PATH/TO/TEST/DATA.conll -v 1
```

Evaluate corpus given gold dependency

```
java -jar MaltEval.jar -g PATH/TO/TEST/DATA.conll -s PATH/TO/RESULT/DATA.conll
```

Visual debugging for dependencies (green is good, red is bad):

```
java -jar MaltEval.jar -g PATH/TO/TEST/DATA.conll -s PATH/TO/RESULT/DATA.conll -v 1
```

An example image is included at the end of this section.

You can save the image by clicking *File -> format* and setting it to *PNG*, and then clicking *File -> export -> gold sentence*.

A dependency graph for a sentence $S = w_1, w_2, \ldots, w_n$ is a directed graph:

$G = (V, A)$,

Where:

$V = \{1,\ldots,n\}$ is the set of nodes (representing tokens),

$A \subseteq V \times L \times V$ is the set of labeled arcs, representing dependencies.

The arc $i \rightarrow_l j$ is a dependency with a head w_i and a dependent w_j, and is labeled with dependency l.

In this assignment, we will work only with *projective* dependency graphs; that is, dependency graphs that can be represented as a tree with a single root, and where all subtrees have a contiguous yield.

The sentence:

"The non-callable issue, which can be put back to the company in 1999, was priced at 99 basis points above the Treasury's 10-year note."

is projective, whereas the sentence:

"John saw a dog yesterday which was a Yorkshire Terrier"

is not projective, as there is no way to draw the dependency graph without a crossing edge – the subsentence "which was a Yorkshire Terrier" is connected to "a dog", but "yesterday" is connected to "saw".

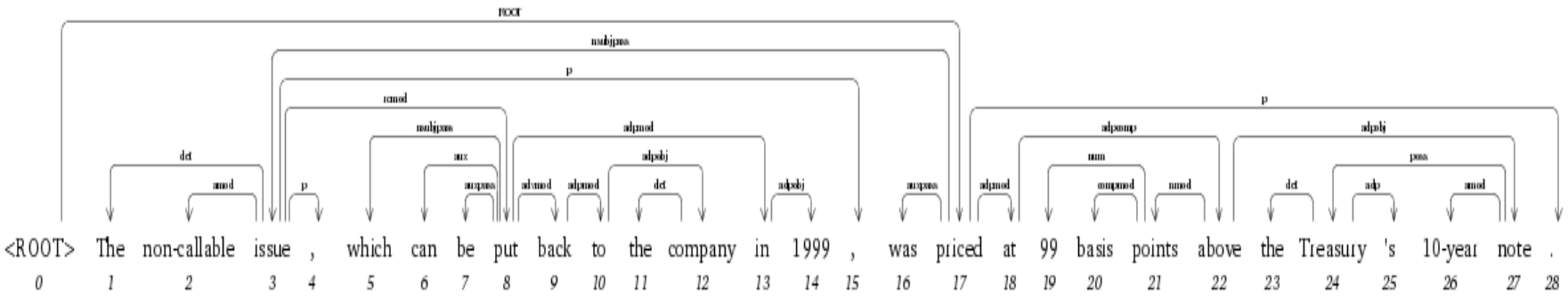


Figure 1: A sentence with its projective dependency parse

IMPLEMENTATION OF A DEPENDENCY PARSER

Overview

Since we have provided code to read input and represent it as a DependencyGraph object, the implementation of this parser breaks down into two main steps.

Part (a): Firstly, we need to implement the transition operations, which allow us to move from one parser configuration to another parser configuration.

A parser configuration is the tuple $C = (\Sigma, B, A)$, where Σ is the *stack*, B is the *buffer*, and A is the set of *arcs* that represent the *dependency relations*. You can think of the arc-eager algorithm as defining when and how to transition between parser configurations C -> C', eventually reaching a terminal configuration $C_T = (\Sigma_T, [], A_T)$.

We then learn the correct sequence of transitions using an "oracle", implemented in this assignment as a trained support vector machine (SVM).

Transitions

Let s be the next node in Σ, b be the next node in B.

`left_arc`$_L$

Add the arc (b, L, s) to *A*, and pop Σ. That is, draw an arc between the next node on the buffer and the next node on the stack, with the label L.

`right_arc`$_L$

Add the arc (s, L, b) to *A*, and push b onto Σ.

`shift`

Remove b from *B* and add it to Σ.

`reduce`

pop Σ

These operations have some preconditions – not all configurations can make all four transitions. You can read about these in greater detail in Nivre's paper.

Support Vector Machine

For this assignment, you can treat the SVM almost (to help you understand the whole general process including transformation of some NLP features used in some models, you will be asked to implement small chunks of code) as a black box which performs the classification operation

```
oracle: f^d -> {left_arc_label, right_arc_label, shift_label, reduce_label}
                for all label
```

where f^d is the feature space that you define, and the output is the correct transition. For simplicity, we can assume each feature to be a binary feature, i.e. present or not present.[3]

Your choice of features will heavily determine the performance of your parser. Experiment with a variety of options – a couple of them are implemented already in the scaffolding we have provided.

Provided data

We will be using data from the CoNLL-X shared task on multilingual dependency parsing, specifically the English and Swedish data sets.

[3] For a fun, relatively intuitive explanation of how a support vector machine works, check out Reddit. For a more rigorous treatment, here are some lecture notes.

The CoNLL data format is a tab-separated text file, where the ten fields are:

1) **ID** - a token counter, which restarts at 1 for each new sentence
2) **FORM** - the word form, or a punctuation symbol
3) **LEMMA** - the lemma or the stem of the word form, or an underscore if this is not available
4) **CPOSTAG** - course-grained part-of-speech tag
5) **POSTAG** - fine-grained part-of-speech tag
6) **FEATS** - unordered set of additional syntactic features, separated by |
7) **HEAD** - the head of the current token, either an ID or 0 if the token links to the root node. The data is not guaranteed to be projective, so multiple HEADs may be 0.
8) **DEPREL** - the type of dependency relation to the HEAD. The set of dependency relations depends on the treebank.
9) **PHEAD** - the projective head of the current token. PHEAD/PDEPREL are available for some data sets, and are guaranteed to be projective. If not available, they will be underscores.
10) **PDEPREL** - the dependency relationship to the PHEAD, if available.

Dependency parsing systems were evaluated by computing the **labeled attachment score** (LAS), the percentage of scoring tokens for which the parsing system has predicted the correct head and dependency label. We have provided an evaluator and a corpus reader for you already, so you should not need to re-implement either of these. To convert `DependencyGraph` objects into the CoNLL format, call the function `to_conll(10)`.

Datasets:
English: `~coms4705/Homework2/data/English/`
Danish: `~coms4705/Homework2/data/danish/ (you don't need it this time)`
Korean: `~coms4705/Homework2/data/Korean/ (you don't need it this time)`
Swedish: `~coms4705/Homework2/data/swedish/`

Each of these directories sets has a file `train`, which is the training data set, dev, which is the test set and dev_blind, which is the same as dev dataset but missing the gold-standard dependency tags.

Provided code

There are a number of topics in this assignment that you may not have encountered before. Since COMS W4771 Machine Learning is not a prerequisite for this course, we have provided a working implementation of the support vector machine used in the algorithm. *However, in order to give you a big picture of the entire parsing process, you need to take a look at some scripts and implement a small part of code in* `transitionparser.py`**.** Additionally, we provide some scaffolding for reading and working with the data sets.

`dataset.py`

This file contains helper functions that can retrieve the relevant code for various data sets. Note that not all functions within this file will be used; in particular, `get_english_test_corpus()` will not be made available to you (though `get_english_dev_corpus()` will be available instead). These functions all return `DependencyCorpusReader` objects, which include their parsed sentences as `DependencyGraph` objects in an accessor method. Take a look at `test.py` for usage details.

Useful functions to look at:

- ➔ `get_swedish_train_corpus`
- ➔ `get_swedish_dev_corpus`
- ➔ `...`

`transitionparser.py`

This file contains the actual transition parser implementation. You should **not** need to edit anything in this file *expects a small chunk of code*, but you can take a look at it to see what arguments get passed to the functions you have to write. If you're curious about how we work with the SVM, that code is also included here.

Useful methods to look at:

- ➔ `__init__`
- ➔ `_is_projective`
- ➔ `train`
- ➔ `parse`
- ➔ `save`
- ➔ `load`

`dependencygraph.py`
This file implements an abstract dependency graph. Not all instances of these objects have valid dependency parses. There are some helper methods for manipulating and reading this data.

Useful methods to look at:

- ➔ `__init__`
- ➔ `to_conll`
- ➔ `add_node`
- ➔ `add_arc`
- ➔ `left_children`
- ➔ `right_children`
- ➔ `from_sentence`

In order to generate files in the correct CoNLL format for the `MaltEval.jar` program, you will need to call `to_conll(10)`.

Example:

```
def depgraph_list_to_file(depgraphs, filename):
	with open(filename, 'w') as f:
		for dg in depgraphs:
			f.write(dg.to_conll(10).encode('utf-8'))
			f.write('\n')
```

`test.py`
This file is really just an example of how to correctly call the various helper functions and objects we have provided. It's short, so we hope that you understand it fully.

ASSIGNMENT INSTRUCTIONS

For this assignment, you will be dependency parsing a number of datasets. We have provided the following files:

`envsetup.sh`
`featureextractor.py`
`providedcode/dataset.py`
`providedcode/dependencycorpusreader.py`
`providedcode/dependencygraph.py`
`providedcode/evaluate.py`
`providedcode/transitionparser.py`
`providedcode/__init__.py`
`MaltEval.jar`
`englishfile`
`test.py`
`transition.py`

1) Dependency graphs

   a. Generate a visualized dependency graph of a sentence from each of the English and Swedish training data sets.
   b. Some of these training sentences do not have projective dependency graphs (about 10% of the Swedish data set, for example). In your `README.txt`, please discuss how to determine if a dependency graph is projective. You may find it educational to read the source code available in `providedcode/transitionparser.py`, which has an implementation of a function which checks for this property.

2) (iv) Manipulating configurations (Assignment part 1)

   a. A key part of the Nivre dependency parser is manipulating the parser configuration. In `transition.py`, we have provided an incomplete implementation of the four operations `left_arc`, `right_arc`, `shift`, and `reduce` that are used by the parser. Complete this implementation, and run python `test.py` to see the results. In the next step, we will be significantly improving the performance of your parser.

      Note: You should check out the Configuration object in providedcode/transitionparser.py to see what you can do with conf in `transition.py`.

      You may find this video particularly helpful in explaining how to implement the transition-based dependency parsing.

   b. In your README.txt, examine the performance of your parser using the provided `badfeatures.model`.

3) (v) Dependency parsing (Assignment part 2)

   a. The extract_features function in featureextractor.py takes nodes in the dependency graph and partially processed words etc. to get a list of feature strings/names (still have to be processed as the input of the SVM model), e.g. *'STK_0_LDEP_<left most dependency>'* (These are just names, you can call them whatever you want. But each feature name should be unique).

      Once you have the list of feature strings you should convert it to binary features such as (1,0,0,0,0,1,0,0,0,0,...). This is what _convert_to_binary_features function does in providedcode/transitionparser.py. After having the feature vector, you need to get the label (we name it 'key' in the code. The label can be one of $\texttt{left\_arc}_{\text{label}}$, $\texttt{right\_arc}_{\text{label}}$, $\texttt{shift}_{\text{label}}$, and $\texttt{reduce}_{\text{label}}$) and write the feature vector and the label to a svmlight file (basically, for each line, the first number is the label following by '\n' and the feature vector. _write_to_file function does this step).

      Here, what you need to do is to implement right arc and shift operation parts in the `_create_training_examples_arc_eager` function of `providedcode/transitionparser.py`. We provide an example of how to implement the left-arc and reduce operations.

   b. Edit `featureextractor.py` and try to improve the performance of your feature extractor! The features that we have provided are not particularly effective, but adding a few more can go a long way. Add at least three feature types and describe

their implementation, complexity, and performance in your `README.txt`. It may be helpful to take a look at the book Dependency Parsing[4] by Kuebler, McDonald, and Nivre for some ideas.

Note: There are several features in the gold standard, which should not be used in your implementation since you will be tested on blind datasets (same as dev_blind). Specifically, you should not use the following in your implementation: HEAD, DEPREL, PHEAD, PDEPREL.

c. Generate trained models for English and Swedish data sets, and save the trained model for later evaluation. **All of your two models should come** from **training only the train set.**
d. Score your models against the test data sets for English and Swedish. You should see dramatically improved results compared to before, around 70% or better for both LAS and UAS with 200 training sentences.
e. In your `README.txt` file, discuss in a few sentences the complexity of the arc-eager shift-reduce parser, and what tradeoffs it makes.

4) Parser executable

a. Create a file parse.py which can be called as follows:
```
cat englishfile | python parse.py english.model > englishfile.conll
```
b. The standard input of the program will be the sentences to be parsed, separated by line. The expected standard output is a valid CoNLL-format file which can be viewed by the MaltEval evaluator, complete with HEAD and REL columns. The first argument should be the path to the trained model file created in the previous step.
c. Sample output for `englishfile` is not directly provided, as your model may have been trained on different features. However, we do expect that the CoNLL output is valid and contains a projective dependency graph for each sentence.