COMS 4705 – Natural Language Processing – Spring 2015
Assignment 2
Multilingual Dependency Parsing

(version 1.1, February 18, 2015)

Due: Friday, March 6, 23:59:59

Background Information

Introduction

Dependency parsing is a well-known problem in Natural Language Processing. It was the shared task of the 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.

Dependency Parsing?

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 which 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 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.

---

[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] Goldberg, Nivre, "A Dynamic Oracle for Arc-Eager Dependency Parsing" Proceedings of COLING 2012: Technical Papers, pages 959–976, COLING 2012, Mumbai, December 2012

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 -> 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.

Dependency Graphs
Succinctly, 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 the assignment.

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), and $A \subseteq V \times L \times V$ 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 in the next page:

“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”.

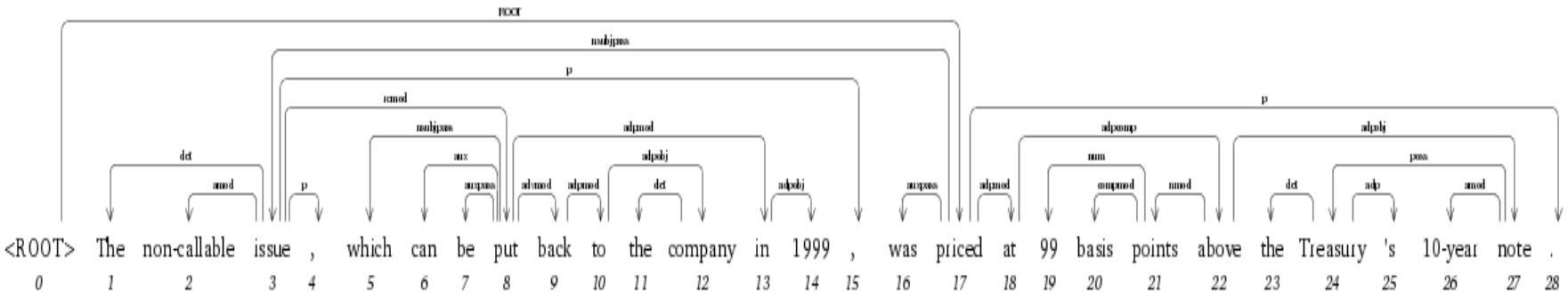

Figure 1: A sentence with its projective dependency parse

Implementation
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. 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 = (Σ, 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.

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 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. You are not expected to know how the SVM training or classification work. 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.

Environment setup
You can acquire the provided code by running the following commands:

```
cd ~/hidden/$HIDDENNUMBER/
mkdir Homework2
cd Homework2
cp -r ~coms4705/Documents/Homework2/public/* .
```

[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.

Before running code for this assignment, make sure to run this command:

```
source envsetup.sh
```

to set your PYTHONPATH correctly.

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

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 reimplement either of these. To convert `DependencyGraph` objects into the CoNLL format, call the function `to_conll(10)`.

Datasets:
English: `~coms4705/Documents/Homework2/data/english`
Danish: `~coms4705/Documents/Homework2/data/danish/ddt`
Korean: `~coms4705/Documents/Homework2/data/korean`
Swedish: `~coms4705/Documents/Homework2/data/swedish/talbanken05`

Each of these data sets has a subdirectory `train`, which contains the training data set. This data set is **much larger than you can feasibly train with**, so please select a random subset of the sentences in each language for training. We have achieved good performance with about 200 sentences in the sample. For Danish, Korean, and Swedish, we have also provided the test data set in `test`. In the English dataset, we have provided you with a development dataset in `dev`; this is the same as the test dataset but missing the gold-standard dependency tags.

If you would like to evaluate your parser on English, you can manually inspect the output, or alternatively generate your own test dataset from a random sample of the training dataset that is disjoint with the sentences that you used for training.

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. 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_test_corpus`
- ➔ `...`

`transitionparser.py`

This file contains the actual transition parser implementation. You should not need to edit anything in this file, 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.

Note that your copy of this file will be replaced by the copy from the starter code, so you should not edit its contents.

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

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
```

NOTE: There is the distinct possibility that running the training and parsing programs will incur a large memory overhead, especially if you forget to select only a small number of sentences to train on. As such, it would not be particularly surprising if we managed to run CLIC out of memory when everybody is running their code. This has happened before. Start early and beat the rush!

1) Dependency graphs
    a. Generate a visualized dependency graph of a sentence from each of the English, Danish, Korean, 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.
    c. Write an example of a sentence in English which has a projective dependency graph, as well as an example of a sentence in English which does not have a projective dependency graph, in your `README.txt`. You may not use either of the sentences given as examples above.
2) Manipulating configurations
    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 `test.py` to see the results. In the next step, we will be significantly improving the performance of your parser.
    b. In your README.txt, examine the performance of your parser using the provided `badfeatures.model`.
3) Dependency parsing
    a. 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 Kubler, McDonald, and Nivre for some ideas.
    b. Generate trained models for English, Danish, Swedish, and Korean data sets, and save the trained model for later evaluation.
    c. Score your models against the test data sets for Danish, Swedish, and Korean. You should see dramatically improved results as compared to before, around 70% or better for both LAS and UAS with 200 training sentences.
    d. 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:

[4] Available here: http://books.google.com/books/about/Dependency_Parsing.html?id=k3iiup7HB9UC Particularly useful is Table 3.2 on page 31.

```
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.

Deliverables

You should have a copy of each of these files in your directory when you finish your assignment, e.g. in

```
HW2_ROOT=/home/$USER/hidden/$HIDDENNUMBER/Homework2/
```

where `$USER` is your UNI and `$HIDDENNUMBER` is the number that Prof. Radev emailed you.

Please follow this folder structure exactly, otherwise you may lose points for otherwise correct submissions!

1. Dependency graph plots
    a. `$HW2_ROOT/figure_en.png` - image of a sentence from English training data
    b. `$HW2_ROOT/figure_da.png` - image of a sentence from Danish training data
    c. `$HW2_ROOT/figure_ko.png` - image of a sentence from Korean training data
    d. `$HW2_ROOT/figure_sw.png` - image of a sentence from Swedish training data
2. Transitions between configurations
    a. `$HW2_ROOT/transition.py`
3. Feature extractor
    a. `$HW2_ROOT/featureextractor.py`
4. Trained model files
    a. `$HW2_ROOT/english.model` - trained TransitionParser for English
    b. `$HW2_ROOT/danish.model` - trained TransitionParser for Danish
    c. `$HW2_ROOT/korean.model` - trained TransitionParser for Korean
    d. `$HW2_ROOT/swedish.model` - trained TransitionParser for Swedish
5. Standard parser
    a. `$HW2_ROOT/parse.py` - parser program
6. README file containing results and discussions
    a. `$HW_ROOT/README.txt`

You may note that we have not provided sample lines or output. This is because you are only being asked to turn in the trained model files, the images, the edited code, and a readme file. For evaluation, we will be running your code on the English test dataset and examining your performance, as well as looking at how your parser performs on the other languages. Please ensure that your model can be loaded with `TransitionParser.load`