NLP Assignment

Due: 1/4/2018

Introduction

The purpose of first part of this assignment is to gain more familiarity with some of the models seen during the course, including firs-order HMMs, PCFGs, as well as an optimization technique, called dual decomposition, to integrate them. The second part of the assignment will give you an idea on how shared tasks in NLP are organized by working on a CoNLL 2002 Named Entity Recognition (NER) shared task.

1 HMMs, PCFGs and Dual Decomposition

In this part of the assignment, you will implement a first-order HMM decoder for part-of-speech tagging and a CYK parser for context-free parsing. Then, you will use the dual decomposition method to integrate the tagger and the parser in an efficient decoder that maximizes the sum of the two models' log-probabilities. A similar integration has been shown to improve the performance of both the tagging and the parsing subproblems (Rush et al., 2010, `http://cs.nyu.edu/~dsontag/papers/RusSonColJaa_emnlp10.pdf`).

The necessary input files for performing the tasks described below are provided in the zip bundle: `hmm-pcfg-files.zip`.

1.1 POS tagger

Your first task is to implement a decoder for a first-order hidden Markov model for Arabic part-of-speech tagging. The decoder expects three input files which specify HMM transition probabilities, HMM emission probabilities, and input sentences. Given input sentence $\curvearrowleft$, the job of the tagger is to find the highest scoring tag sequence $\hat{\mathbf{z}} = \arg\max_{\mathbf{z}} \log p_{hmm}(\mathbf{z}|\mathbf{x})$. `sentence_boundary` is a special POS tag which is used to mark the beginning and end of a sentence.

You can evaluate the output of your tagger (e.g. `candidate-postags`) against gold standard POS tags of the development set `dev_sents` as follows:

```
./eval.py --reference_postags_filename=dev_postags \
                  --candidate_postags_filename=candidate-postags
```

1.1.1 Deliverables

Submit your code and the output of your tagger using input files `hmm_trans`, `hmm_emits`, `test_sents`.

1.2 Context free grammar parser

Your second task is to implement a parser for a Chomsky normal form probabilistic context-free grammar (PCFG) of Arabic syntax. The parser expects two input files which specify the PCFG, and input sentences. Given input sentence $\mathbf{x}$, the job of the tagger is to find the highest scoring derivation $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log p_{pcfg}(\mathbf{y}|\mathbf{x})$.

You can evaluate the output of your parser (e.g. `candidate-parses`) against gold standard parses of the development set `dev_sents` as shown below. The evaluation script provided `eval.py` reports the precision and recall on the binary trees, contrary to the common practice of reporting precision and recall with the original grammar.

```
./eval.py --reference_parses_filename=dev_parses \
          --candidate_parses_filename=candidate-parses
```

1.2.1 Deliverables

Submit your code and the output of your parser using input files `pcfg`, `test_sents`.

1.3 Dual decomposition

Given input sentence $\mathbf{x}$, it is required to find the highest scoring derivation:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log p_{pcfg}(\mathbf{y}|\mathbf{x}) + \log p_{hmm}(l(\mathbf{y})|\mathbf{x})) \tag{1}$$

where $l(\mathbf{y})$ maps a derivation $\mathbf{y}$ to the sequence of POS tags in $\mathbf{y}$. Since each of the PCFG and the HMM captures different types of information, combining both models may improve both the accuracy of parsing as well as POS tagging, compared to solving the two problems in isolation.

We use the same notation and definitions used in (Rush and Collins, 2012, `http://www.cs.columbia.edu/~mcollins/acltutorial.pdf`): $\mathcal{T}$ is the set of POS tags, $y(i,t) = 1$ iff parse tree $\mathbf{y}$ has a tag $t \in \mathcal{T}$ at position i in a sentence, $y(i,t) = 0$ otherwise. $z(i,t) = 1$ iff POS tagging sequence $\mathbf{z}$ has a tag $t \in \mathcal{T}$ at position i, $z(i,t) = 0$ otherwise. $u(i,t)$ is a Lagrange multiplier enforcing the constraint $y(i,t) = z(i,t)$.

The dual decomposition algorithm for integrating parsing and tagging, adapted from (Rush and Collins, 2012), is as follows:

```
initialization: $\mathbf{u} \leftarrow 0$ ;
for $k=1 \dots K$ do
    $\hat{\mathbf{y}} \leftarrow \arg\max_{\mathbf{y}} \log p_{pcfg}(\mathbf{y} \mid \mathbf{x}) + \sum_{i,t} u(i,t)y(i,t)$ ;
    $\hat{\mathbf{z}} \leftarrow \arg\max_{\mathbf{z}} \log p_{hmm}(\mathbf{z} \mid \mathbf{x}) - \sum_{i,t} u(i,t)z(i,t)$ ;
    if $\hat{\mathbf{y}}(i,t) = \hat{\mathbf{z}}(i,t) \forall i,t$ then
        print $\hat{\mathbf{y}}$ ;
    else
        $u(i,t) \leftarrow u(i,t) - \delta_k(y(i,t) - z(i,t))$ ;
    end
end
```

Algorithm 1: The dual decomposition algorithm.

Modify your implementation of the POS tagger and the CFG parser in order to account for the extra Lagrange multiplier terms. Then, implement the dual decomposition algorithm as described in Algorithm 1. The decoder expects four input files which specify HMM transitions, HMM emissions, a PCFG, and input sentences. Given an input sentence $\mathbf{x}$, solve the optimization problem 1, finding $\hat{\mathbf{y}}$. The decoder outputs two files: one for parse trees and another for the POS tag sequences. You can evaluate the two output files of your tagger using the development set as shown earlier.

1.3.1 Deliverables

Submit your code and the output files of your dual decomposition decoder with input files `hmm_trans`, `hmm_emits`, `pcfg`, `test_sents`.

1.4 Data formats

Parameter files

Three parameter files are provided: `hmm_trans`, `hmm_emits`, `pcfg`. Each file specifies a number of conditional distribution $p(decision|context)$. Each line consists of three tab-separated columns:

```
context     decision     log p(decision | context)
```

In `pcfg`, the start non-terminal is `S` and the *decision* consists of either one terminal symbol (e.g. 'dog') or two space-separated nonterminal symbols (e.g. 'ADJ N').

Plain text files

Two plain text files are provided: `dev_sents`, `test_sents`. Each of the two files consist of one tokenized sentence per line. Tokens are space-separated.

POS tagging files

We describe the format of the provided gold standard POS tags file `dev_postags` as well as your output for the HMM tagging task and the dual decomposition task. Each line consists of a space-separated POS tag sequence. The number of tags must be equal to the number of tokens in the corresponding sentence (i.e. use `sentence_boundary` tags at the beginning and the end while decoding to find the Viterbi POS tag sequence, but do not write `sentence_boundary` tags to the output file).

Parse files

We describe the format of the provided gold standard parse trees file `dev_parses` as well as your output for the CFG parser and dual decomposition task. Each line consists of a complete syntactic derivation for the corresponding sentence in a plain text file. For example, in the simple parse below, `S` is the root with two children: `NP` and `V`. `NP` has two children: `ADJ` and `N`. `ADJ`, `N` and `V` each has a single child: `bad`, `tornado` and `coming`, respectively.

```
(S (NP (ADJ bad) (N tornado)) (V coming))
```

2 Named Entity Recognition

Start by reading Chapter 21 of Jurafsky and Martin's book: *Speech and Language Processing*[1]. You will learn that Named Entity Recognition is the task of finding and classifying named entities in text. This task is often considered a sequence tagging task, like part-of-speech tagging, where words form a sequence through time, and each word is given a tag. Unlike part of speech tagging however, NER usually uses a relatively small number of tags, where the vast majority of words are tagged with the 'non-entity' tag, or O tag.

Your task is to implement your own named entity recognizer. There will be two versions of this task: the first, the constrained version, is the required entity tagger that you implement using Python's library scikit learn[2], filling out the stub that we give you[3]. The second is an unconstrained version where you use whatever tool, technique, or feature you can get your hands on to get the best possible score on the dataset.

As with nearly all NLP tasks, you will find that the two big points of variability in NER are (a) the features, and (b) the learning algorithm, with the features arguably being the more important of the two. The point of this assignment is for you to think about and experiment with both of these. Are

[1]Available on Moodle as well as on the following location `http://web.stanford.edu/~jurafsky/slp3/`
[2]`http://scikit-learn.org/`
[3]Available on Moodle.

there interesting features you can use? What latent signal might be important for NER? Can deep learning help with automatic feature learning?

Get a headstart on common NER features by looking at Figure 21.5 (Features commonly used in training named entity recognition systems) in the textbook.

2.1 The Data

The data we use comes from the Conference on Natural Language Learning (CoNLL) 2002 shared task of named entity recognition for Spanish and Dutch. Read the introductory paper to the shared task[4] to get a full understanding. We will use the Spanish corpus (although you are welcome to try out Dutch too).

The tagset is:

- PER: for Person
- LOC: for Location
- ORG: for Organization
- MISC: for miscellaneous named entities

The data uses `BIO` encoding (called IOB in the textbook), which means that each named entity tag is prefixed with a `B-`, which means beginning, or an `I-`, which means inside. So, for a multiword entity, like "`James Earle Jones`", the first token "`James`" would be tagged with "`B-PER`", and each subsequent token is "`I-PER`". The `O` tag is for non-entities.

We strongly recommend that you study the training and dev data (no one's going to stop you from examining the test data, but for the integrity of your model, it's best to not look at it). Are there idiosyncracies in the data? Are there patterns you can exploit with features? Are there obvious signals that identify names? For example, in some Turkish writing, there is a tradition of putting an apostrophe between a named entity and the morphology attached to it. A feature of `isApostrophePresent()` goes a long way. Of course, in English and several other languages, capitalization is a hugely important feature. In some African languages, there are certain words that always precede city names.

The data is packaged nicely from NLTK[5]. Get installation instructions from the installing NLTK page[6]. See the NLTK data page for for download options, but one way to get the conll2002 data is:

```
$ python -m nltk.downloader conll2002
```

2.2 Evaluation

There are two common ways of evaluating NER systems: phrase-based, and token-based. In phrase-based, the more common of the two, a system must predict the entire span correctly for each name. For example, say we have text containing "`James Earle Jones`", and our system predicts "`[PER James Earle] Jones`". Phrase-based gives no credit for this because it missed "`Jones`", whereas token-based would give partial credit for correctly identifying "`James`" and "`Earle`" as `B-PER` and `I-PER` respectively. We will use phrase-based to report scores.

The output of your code must be `word gold pred`, as in:

```
La B-LOC B-LOC
Coruna I-LOC I-LOC
, O O
23 O O
may O O
( O O
EFECOM B-ORG B-ORG
) O O
. O O
```

[4] http://www.aclweb.org/anthology/W02-2024
[5] http://www.nltk.org/
[6] http://www.nltk.org/install.html

Here's how to get scores (assuming the above format is in a file called `results.txt`):[7]

```
# Phrase-based score
$ python conlleval.py results.txt
```

2.3 Other resources

Here are some other NER frameworks which you are welcome to run in the unconstrained version:

- CogComp NER (`https://github.com/CogComp/cogcomp-nlp/tree/master/ner`), one of the best taggers
- LSTM-CRF (`https://github.com/glample/tagger`), recent neural network tagger
- Stanford NER (`https://nlp.stanford.edu/software/CRF-NER.shtml`), Stanford's tried and true tagger
- spaCy (`https://spacy.io/usage/training`)
- Brown clustering software (`https://github.com/percyliang/brown-cluster`). You might find it useful.
- Spanish text and vectors (`http://crscardellino.me/SBWCE/`)

Note: you are not allowed to use pre-trained NER models even in the unconstrained version. Please train your own. You are allowed to use pre-trained embeddings.

2.4 Baselines

The version we have given you gets about 49% F1 right out of the box. We made some very simple modifications, and got it to 60%. This is a generous baseline that any thoughtful model should be able to beat. The state of the art on the Spanish dataset is about 85%. If you manage to beat that, then look for conference deadlines and start writing, because you can publish it.

2.5 Recommended readings

In addition to Jurafsky and Marting's Chapter 21, we recommend the following resources:

- Design Challenges and Misconeptions in Named Entity Recognition (`http://cogcomp.org/papers/RatinovRo09.pdf`)
- Entity Extraction is a Boring Solved Problem – or is it? (`https://aclanthology.info/pdf/N/N07/N07-2046.pdf`)
- Neural Architectures for Named Entity Recognition (`https://arxiv.org/abs/1603.01360`)
- Introductory paper to CoNLL 2002 shared task (`http://www.aclweb.org/anthology/W02-2024`)

2.6 Deliverables

Here are the deliverables that you will need to submit:

- Code used to compute the results files and short README with instructions on how to run it;
- Constrained results (in a file called `constrained_results.txt`);
- Unconstrained results (in a file called `unconstrained_results.txt`);

Collaboration Policy

You are allowed to work in teams consisting of up to 4 students. You are also allowed to discuss the assignment with other teams and collaborate on developing algorithms at a high level. However, writeup and code must be done separately.

[7]The python version of conlleval doesn't calculate the token-based score, but if you really want it, you can use the original perl version `https://www.clips.uantwerpen.be/conll2000/chunking/output.html`. You would use the -r flag.

Deliverables

Every team should submit on Moodle an archive (e.g. `.tar.gz`) containing:

- The deliverables files and code listed in the *deliverables* paragraphs above;
- Short PDF report explaining your work and commenting on your results.

Acknowledgments

This assignment is adapted from Chris Dyer and Chris Callison-Burch.