NLP Assignment

Due: 10/4/2017

Introduction

The purpose of this assignment is to gain more familiarity with some of the models seen during the course, including firs-order HMMs, PCFGs and IBM word alignment models. The required parts should be doable in two weeks and there are several optional parts if you want to go further.

1 HMMs and PCFGs

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
```

Deliverable: 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
```

Deliverable: submit your code and the output of your parser using input files `pcfg`, `test_sents`.

1.3 (Optional) Dual decomposition

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

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

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 ... 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.

Deliverable: 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 (Optional) Word alignment

Word alignment is a fundamental task in statistical machine translation. This part of the assignment will give you an opportunity to explore solutions to this challenging and interesting problem.

The necessary files for performing the tasks described below are provided in the zip bundle: `word-align-files.zip`.

2.1 Heuristic alignment

In the word-align-files directory you will find a word-aligner written in Python, along with 100,000 German–English parallel sentences from the Europarl corpus (version 7). This aligner uses set similarity to determine which words are aligned to each other in a corpus of parallel sentences. The set similarity measure we use is Dice's coefficient, defined in terms of sets $\mathbf{X}$ and $\mathbf{Y}$ as follows:

$$D(\mathbf{X}, \mathbf{Y}) = \frac{2 \times |\mathbf{X} \cap \mathbf{Y}|}{|\mathbf{X}| + |\mathbf{Y}|}$$

Dice's coefficient ranges in value from 0 to 1.

How do we use set similarity to align words? The intuition is that if you look at the set of sentence pairs from a parallel corpus that contain an English word e, and that set is similar to the set of sentence pairs that contain a German word g, then these words are likely to be translations of each other.

Formally, every pair of word types (e, g) in the parallel corpus receives a Dice "score" $\sigma(e, g)$. The alignment algorithm then goes through all pairs of sentences $(\mathbf{e}, \mathbf{g})$ and predicts that English word e_i is aligned to German word g_j if $\sigma(e_i, g_j) > \tau$. By making τ closer to 1, fewer points are aligned but with higher precision; by making it closer to 0, more points are aligned, probably improving recall. By default, the aligner code we have provided you uses $\tau = 0.5$ as its threshold.

Run the baseline heuristic model 1,000 sentences using the command:

```
./align -n 1000  ./check > dice.al
```

This runs the aligner and stores the output in `dice.al`. To display the alignments visually and score the alignments, run this command (use the `-n N` option to display verbose output for only the first `N` sentences):

```
./grade < dice.al
```

This command scores the alignment quality by comparing the output alignments against a set of human alignment annotations using a metric called the alignment error rate (AER), which balances precision and recall of the guessed alignments (see Section 6 of `http://aclweb.org/anthology/P/P00/P00-1056.pdf`). Look at the terrible output of this heuristic model – it's better than chance, but not any good. Try training on 10,000 sentences instead of 1,000, by specifying the change on the command line:

```
./align -n 10000  ./check  ./grade
```

Performance should improve since the degree of association between words will be estimated from more sentence pairs. Another experiment that you can do is change the threshold criterion τ used by the aligner using the `-t X` option. How does this affect alignment error rate?

2.2 IBM Model 1

Your task for this assignment is to improve the alignment error rate as much as possible (lower is better). It shouldn't be hard to improve it at least some: you've probably noticed that thresholding a Dice coefficient is a bad idea because alignments don't compete against one another. A good way to correct this is with a model where alignments compete against each other, such as IBM Model 1. It forces all of the words you are conditioning on to compete to explain each word that is generated.

IBM Model 1 is a simple probabilistic translation model we talked about in class. A source sentence $\mathbf{g} = [\epsilon, g_1, g_2, \ldots, g_n]$ (where ϵ represents a null token present in every sentence) and a desired target sentence length m are given, and conditioned on these, Model 1 defines a distribution over translations of length m into the target language using the following process:

for each $i \in [1, 2, \ldots, m]$
  $a_i \sim \text{Uniform}(0, 1, 2, \ldots, n)$
  $e_i \sim \text{Categorical}(\theta_{g_{a_i}})$

The random variables $\mathbf{a} = [a_1, a_2, \ldots, a_m]$ are the alignments that pick out a source word to translate at each position in the target sentence. A source word g_j may be translated any number of times (0, 1, 2, etc.), but each word in the target language e_i that is generated is generated exactly one time by exactly one source word.

The marginal (marginalizing over all possible alignments) likelihood of a sentence $\mathbf{e} = [e_1, e_2, \ldots, e_m]$ given $\mathbf{g}$ and m is:

$$p(\mathbf{e}|\mathbf{g}, m) = \prod_{i=1}^{m} \sum_{j=0}^{n} p(a_i = j) \times p(e_i|g_i)$$

The iterative EM update for this model is straightforward. At each iteration, for every pair of an English word type e and a German word type g, you count up the expected (fractional) number of times token g is aligned to token e and divide by the expected number of times that g was chosen as a translation source. That will give you a new estimate of the translation probabilities $p(g|e)$, which leads to new alignment expectations, and so on. We recommend developing on a small data set (1,000 sentences) and a few iterations of EM (in practice, Model 1 needs only 4 or 5 iterations to give good results). When you are finished, you should see both qualitative and quantitative improvements in the alignments.

For more details on IBM Model 1, check `http://people.cs.umass.edu/~brenocon/inlp2014/notes/model1_em.html`, `http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/ibm12.pdf`, `http://mt-class.org/jhu/assets/papers/alopez-model1-tutorial.pdf` and `http://www.isi.edu/natural-language/mt/wkbk.pdf`.

Deliverable: submit your code and your alignment on the full set of parallel data (dev/test/training). The output will be evaluated on the test data (which you do not have gold alignments for), but the grade program will give you a good idea how you are doing on dev data.

2.3 Improving the aligner

Developing an aligner using the simple IBM Model 1 is good enough to get a reasonable alignment error rate (AER). However, getting closer to the best known accuracy on this task is a more interesting challenge. To improve over your aligner, you may experiment with some of the following ideas to name a few:

- Add an alignment model that favors alignment points close to the diagonal
  `http://aclweb.org/anthology/N/N13/N13-1073.pdf`
- Use a hidden Markov model (HMM) to model dependencies between adjacent alignment points
  `http://aclweb.org/anthology/C/C96/C96-2141.pdf`
- Add POS information
  `https://www.aclweb.org/anthology/W/W02/W02-1012.pdf`
- Train a discriminative aligner using the example alignments we've provided using CRFs
  `http://aclweb.org/anthology/P/P06/P06-1009.pdf`
- Use a neural network
  independent classification decisions `http://www.aclweb.org/anthology/P13-1017.pdf`
  sequential classification decisions `http://www.aclweb.org/anthology/P14-1138.pdf`

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.

Deliverables

Each team should deliver:

- Output files on test data for each of the specified tasks, see **"Deliverables"** in previous sections;
- Code used to compute the output files and short README with instructions on how to run them to reproduce the output files;
- A short report in .pdf format describing the work.

Optional parts will give you bonus points.

Acknowledgments

This assignment is adapted from work by Matt Post, Adam Lopez, Philipp Koehn, John DeNero and Chris Dyer.