CS 295: Statistical NLP: Winter 2017

Homework 4: Phrase-Based Translation

Sameer Singh
http://sameersingh.org/courses/statnlp/wi17/

One of the most widespread and public-facing applications of natural language processing is machine translation. It has gained a lot of attention in recent years, both infamously for its lack of ability to understand the nuance in human communications, and for near human-level performance achieved using neural models. In this homework, we will be looking at phrase-based translation from French-English, and implementing stack-based decoders of various complexity to achieve this. The submissions are due by midnight on **March 13, 2017**.

1 Task: Phrase-Based Machine Translation

I will briefly describe the phrase-based translation model and the decoding algorithm here, however you should refer to the notes on phrase-based translation available here: http://www.cs.columbia.edu/~mcollins/pb.pdf. In particular, focus on Sections 1 (for notation), 3, and 4 (for the algorithm).

We are initially given a phrase translation table consisting of tuples of translation $(\mathbf{f}_p, \mathbf{e}_p, g_p)$ where $\mathbf{f}_p$ is the french phrase, $\mathbf{e}_p$ the english phrase, and g_p the score of the translation. Given the sequence of source sentence tokens $f_1, \ldots, f_n$, we obtain all possible phrases from the translation table that apply to this sentence $P = \{(s_p, t_p, \mathbf{e}_p, g_p)\}$, where s_p, t_p is the start/end of the foreign tokens covered by this phrase, $\mathbf{e}_p$ the translated tokens in english, and g_p the score of the translation.

1.1 Stack-based Decoding

As we covered in the lecture, stack-based decoding provides a family of search-based decoding algorithms. Give the decoding problem, the stack-based decoder works by maintaining a set of partial translations (each of which is known as a state/hypothesis) of the source sentence, selecting the next state to *expand* by one more phrase from the translation table, and then adding the expanded states (one for each possible phrase) back to the stack of current translations.

Each state in the algorithm consists of a sequence of phrases that partially cover the source sentence. Formally, a state $q \equiv (\mathbf{e}_q, b_q, r_q, \alpha_q)$ consists of $\mathbf{e}_q$ as the last two tokens of the English translation so far (e.g. last two tokens of the last phrase p if $|\mathbf{e}_p| \geq 2$), b_q as the bit vector of length n representing which tokens of $\mathbf{f}$ have been covered, r_q is the end point of the last phrase (e.g. t_p if p is the last phrase of q), and α_q as the score of the partial translation so far. The partial score of the translation consists of the sum of the translation scores of each phrase, the language model score of the tokens, and any distortion penalty that applies.

```
Require f (foreign sentence)
  P ← PHRASES(f)                          ▶ Phrases that apply to f
  Q_0 ← {q_0}, Q_i ← {}                   ▶ Initial state, no translation
  for i ← (0, ..., n−1) do
      for q ← TOPBEAM(Q_i); p ← COMPATIBLE(q) do
          q' ← NEXT(q, p)                 ▶ Augment q by p to get new state
          j ← |b_q'|                      ▶ Number of 1s in b_q'
          Q_j ← Q_j ∪ {q'}                ▶ Add q' to stack Q_j
      end for
  end for
  q* ← argmax_{q∈Q_n} α_q
  return e(q*)      ▶ Follow backpointers to construct the English sentence
```

One variation from the regular stack decoding that we study here is the multi-stack decoding. Single stack decoding suffers from two major problems, (1) by comparing partial translation scores of states that cover different parts of the sentence, they are essentially comparing pommes to oranges (so to say), and (2) it is difficult to perform hypothesis recombination (i.e. keeping the high scoring of the partial translations that cover the same

source tokens). Multi-stack decoding consists of a stack for each "number of source words translated", thus scores are *more* comparable, and hypotheses can be combined within each stack.

Much of the *smarts* of this algorithm are in `Next`, which computes the scores $\alpha_{q'}$ based on the partial translation and the next phrase. As we will see next, `Compatible` is also a very important function that defines the candidates phrases to be added, thus defining monotonicity of the decoder.

1.2 Monotonicity and Non-Monotonicity

The stack decoder always builds the English sentence left to right, i.e. each phrase is added to the end of the English token sequence. However, by default, there is no requirement for the tokens that the phrases cover in the foreign sentence to be left to right. This property defines the monotonicity of the decoder, i.e. whether the predicted translation $p_1, \ldots, p_L$ meet the requirement of $\forall i \in (2, \ldots, L), t_{p_{i-1}} = s_{p_i}$, and $s_{p_1} = 1$.

Although for many languages such a constraint may be too harsh, such as the ones that do not follow the SVO structure of English, it is often not a horrible assumption for languages such as French and Spanish. Constraining the decoder to be monotonic for such languages, therefore, provides significant computational benefits, at the cost of accuracy of these translations. However, there are still many situations that benefit from arbitrary word ordering that non-monotonic decoders would provide.

Non-monotonic decoders only vary in the phrases that they consider compatible to a given state q: any phrase p that does not cover any of the existing tokens covered by q is used to extend q. Thus the branching factor explodes with the number of total phrases for the sentence, as opposed to the number of phrases for each token for monotonic decoders. In order to be computationally efficient, the notion of distortion limit was described in the class, that describes the *amount of non-monotonicity* that is tolerated. It is worth noting that incorporating language model is extremely important for non-monotonic decoders, since the phrase ordering is not really constrained.

1.3 Data and Source Code

I have released the initial source code, available at `https://github.com/sameersingh/uci-statnlp/tree/master/hw4`, and the data archive available on Canvas. You will need to uncompress the archive, and put it in the `data/` folder for the code to work. The available code and source code contains the following:

- **`lm.py`** and `lm.gz`: Similar to the assignment in HW2, this code provides an implementation of a Trigram language model with Kneser-Ney smoothing, along with the parameters of such a model trained on a really large corpus of English documents. Note, since we are computing $P(f|e)P(e)$, we do not require a language model of French in order to perform the decoding. The format of the language model file, known as the ARPA model, consists of 1-, 2-, and 3-grams, with their log probabilities and backoff scores. You can load and query the language model using the main function of this file.
- **`phrase.py`** and `phrasetable.txt.gz`: Code and data for the French to English phrase table. Each line in the file contains a pair of these phrases, along with a number of scores for different *features* of the pair. The code reads this file and computes the single score g_p for each pair of phrases. This code also provides a handy method to get all the possible phrase translations for a given sentence, i.e. `phrases()` corresponds to PHRASES in the above pseudocode. You can investigate the translation table as shown in the main function.
- **`decoder.py`**: Implementation of the multiple stack-based decoding algorithm. This implementation attempts to follow the above notation of the pseudocode (and Collins' notes) as much as possible, deviating as needed for an optimized implementation. The code implements a working monotonic decoder that does not take the language model into account. This is especially important when you are looking at the code for finding compatible phrases (`Compatible`), computing the language model score (`lm_score`), and the distortion score (`dist_score`). Some code that differs from the pseudocode includes precomputing the set of phrases that should be considered for position r in `index_phrases` and extra fields in the state to make equality comparisons efficient (`key` in `State`). You will need to develop a reasonable understanding of this code, so please post privately or publicly on Piazza if you are not able to understand something.
- **`submission.py`**: Skeleton code for the submission. It contains the three types of decoders, outof which only the first one, `MonotonicDecoder`, works as intended. You have to implement the other functions in this skeleton, as we will describe next.
- **`data.py`** and `test.en/test.fr`: This is the code that reads in the files related to the translation model, reads French sentences from `test.fr`, corresponding English translations from `test.en`, runs the model on the French sentences, and computes the Bleu score on the predictions. It also contains some simple words and phrases to translate into French, just to test your decoder.

- `bleu_score.py`: Code for computing the Bleu score for each translation/prediction pair (this code was ported from NLTK by Zhengli Zhao).

By default, running `python data.py` will run the monotonic decoder without any language model on the french sentences, showing the translations and computing the Bleu scores. Details about what you need to implement is in the sections below.

2 What to Submit?

Prepare and submit a single write-up (**PDF, maximum 5 pages**) and your `submission.py` and any other modified files (compressed in a single `zip` or `tar.gz` file; we will not be compiling or executing it, nor will we be evaluating the quality of the code) to Canvas. The write-up and code should address the following.

2.1 Incorporating Language Models (20 points)

The current monotonic decoder translates the words from left to right, without any regard for fluency of the resulting translation. This often results in translations that are hard to understand, without a lot of effort on the reader's part. Incorporating a language model into monotonic decoder goes a long way in addressing this. In particular, although the phrases are still translated from left to right, the model with language model ends up preferring phrases that flow better, and thus supports word reordering as much as allowed via phrases in the phrase table.

Most of the code remains the same for this decoder. The only difference is that this decoder now computes a valid score from the language model in `lm_score`, instead of returning 0.0. In particular, you have to compute the log probability of the rest of the sequence of tokens in `words`, conditioned on the first two tokens in `words`. You will need the language model in the translation model (`self.model.lm`) to query for the conditional probabilities. You do not have to include anything in the write up about the implementation, unless you think there was an interesting hiccup.

2.2 Incorporating Non-Monotonicity (30 points)

Although incorporating the language model goes a long way in improving the fluency of the sentences, often you need long range reorderings in order to get the correct translation. Non-monotonic decoders, by not translating the foreign sentence in a left to right manner, allow arbitrary reorderings of words. Due to the explosion in the search space, these models often have constraints on how long-range can the reorderings be, for example the distortion limit as a hard constraint, and a distortion penalty as a soft constraint. Now we will extend our stack-based decoder to non-monotonic decoder.

In this part of the assignment, you have to implement two functions that make the decoder non-monotonic. First, the space of possible phrases to consider at any time are not constrained to translate the *next* tokens, but instead can be any non-overlapping tokens. You will have to implement the `CheckOverlap` function, which is effectively redundant in monotonic decoders, but now have to check whether the tokens in the phrase p overlap with the bits already set in the state q. Second, we have to make sure that the set of possible phrases are within the distortion limit, and that any phrase outside the distortion limit should be excluded. To this end, you have to relax the condition in `CheckDistLimit` function, which by default only checks whether the translation is monotonic, i.e. assumes that the distortion limit is 0, instead of the value set in `self.model.dist_limit`. As in the previous part, do you not have to include anything in the report unless you faced an interesting obstacle.

2.3 Comparing Decoders (50 points)

The primary goal of the write up is to compare the three decoders that you will have at your disposal: monotonic decoder without a language model, monotonic decoder with a language model, and a non-monotonic decoder. Since the decoding algorithms are quite slow, and the language models and the phrase tables involved are quite big, I encourage you to brainstorm about what the differences between the decoders is, and how to best demonstrate the differences between them.

Examples of the sort of the questions you should consider addressing are: which decoder performs best for short versus long sentences, which decoder is most sensitive to beam width for accuracy, how is the running time affected by the beam width of the decoders, which decoder is most sensitive to random reorderings of words in the foreign language, and how does the performance (time or accuracy) of the non-monotonic decoder depend on

the distortion limit. You should pick a few such questions that you find most interesting, and focus your analysis on them. Your analysis should contain a quantitative and a qualitative section:

- *Quantitative*: As described above, you are given the code to compute the Bleu score of a translation. Use this score to present your analysis and compare the above approaches. For multiple sentences, it is fair to average the scores of the individual sentence, or optionally, use the `corpus_bleu` method to compute the score for multiple sentences. Due to the running time, do not be too ambitious about the experiments, do pilot studies with simpler sentences to see what the trends look like, before jumping into larger scale evaluations.
- *Qualitative*: For this assignment, qualitative analysis is quite crucial, as not all changes in the decoders is obvious from the Bleu score. You can use either subsets of the provided sentences and translations, or construct your own. They do not need to be correct French (I wouldn't be able to tell anyway) as long as they use the French phrases, the objective is to highlight how different the translations are.

As before, generously use tables, plots, graphs, and figures to describe your experiments and results.

3 Suggestions/Tips

Being the last homework, I really do not want you to be struggling with it. Please post on Piazza if you have any concerns, and come to my office hours. That said, here are some suggestions to consider.

- Start by creating *very* simple sentences in French, with only a few words, and see the translations by your decoders. These sentences do not even have to be grammatical in French. If you identify short sentences that give you different translations by the three models, some of your qualitative analysis for Section 2.3 is already done.
- For debugging, you might find it helpful to not load the language model, in which case send `None` to the `load_model` in `data.py`. Of course, once you are close to completion, you should load the language model, since the translations without the language model do not look very good for this model.
- Since the non-monotonic decoder can be incredibly slow, be quite restrictive in terms of distortion limits, maximum lengths of the sentences, and the beam width during debugging. Once you have a final version of the decoder, you can run it on longer sentences, with larger beam widths and distortion limits.
- When writing your own sentences in French, use words that appear in the language model and the phrase table files, to get predictable results.
- Since the language model and the phrase table are stored in `.gz` files, the best way to look at them from the console is something like `zcat lm.gz | less`, or `zcat lm.gz | grep president | less`.

4 Statement of Collaboration

It is **mandatory** to include a *Statement of Collaboration* in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Piazza) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content *before* they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to Piazza, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Piazza as much as possible, so that there is no doubt as to the extent of your collaboration.

Since we do not have a leaderboard for this assignment, you are free to discuss the numbers you are getting with others, and again, I encourage you to use Piazza to post your translations and compare them with others.

Acknowledgements

This homework was made possible with the help (and generosity) of Prof. Dan Klein of the University of California, Berkeley and Prof. Yejin Choi of the University of Washington. Thanks, Dan and Yejin! I'd also like to thank Zhengli Zhao with the help in implementing the Bleu score.