HW3: Decoding

Submit Assignment

Due Friday by 11:59pm **Points** 100 **Submitting** a file upload

Homework 3: Decoding

Due March 2nd, 2018 11:59pm Eastern time

Please submit your decoding code and a report answering the questions below.

Decoding is process of taking input in French:

honorables sénateurs , que se est - il passé ici , mardi dernier ?

…And finding its best English translation under your model:

honourable senators , what happened here last Tuesday ?

To decode, we need a model of English sentences conditioned on the French sentence. You did most of the work of creating such a model in **Homework 2**. In this homework, we will give you some French sentences and a probabilistic model consisting of a phrase-based translation model $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$ and an n-gram language model $p_{\text{LM}}(\mathbf{e})$. We will assume a noisy channel decomposition and our goal will be to solve the following *decision function*.

$$\begin{aligned}
\mathbf{e}^* &= \arg\max_{\mathbf{e}} p(\mathbf{e} \mid \mathbf{f}) \\
&= \arg\max_{\mathbf{e}} \frac{p_{\text{TM}}(\mathbf{f}|\mathbf{e}) \times p_{\text{LM}}(\mathbf{e})}{p(\mathbf{f})} \\
&= \arg\max_{\mathbf{e}} p_{\text{TM}}(\mathbf{f} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \\
&= \arg\max_{\mathbf{e}} \textstyle\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e})
\end{aligned}$$

Since multiplying many small probabilities together can lead to numerical underflow, we'll work in logspace using the equivalent decision function:

$$\mathbf{e}^* = \arg\max_{\mathbf{e}} \log \left(\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \right)$$

We will keep the model fixed for this homework. Your goal will be to write a program that, given an input French sentence, attempts to find its most probable English translation under this model using this decision function.

Getting Started

If you don't already have the code, get a fresh copy:

```
git clone https://github.com/achimr/dreamt-gt.git
```

Under the new `decoder` directory, you now have simple decoder and some data files. Take a look at the input French, from a speech in the Canadian parliament:

```
cat data/input
```

Let's translate them!

```
python decode | tee output.default
```

The `decode` program translates the contents of `data/input` and writes the result to the terminal and to the file `output`. It translates using a *phrase-based* translation model: a model that replaces sequences of French words with sequences of English words.[1] The replacement must be one-to-one and every input word must be accounted for by a single phrase. Our model makes the simplifying assumption that segmentation and ordering probabilities are uniform across all sentences, hence constant. This means that $p(\mathbf{e}, \mathbf{a} \mid \mathbf{f})$ is proportional to the product of the n-gram probabilities in $p_{\text{LM}}(\mathbf{e})$ and the phrase translation probabilities in $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$. In phrase-based translation, the output sequence can be permuted, but the default decoder does not do this.[2] We'll discuss the model and the decoding algorithm in more detail below.

You can probably get the gist of the Canadian senator's complaint from this translation, but it isn't very readable. There are a couple of possible explanations for this:

1. Our model of $p(\mathbf{e} \mid \mathbf{f})$ gives high probability to bad translations. This is called *model error*.
2. Our model is good but our decoder fails to find $\arg\max_{\mathbf{e}} p(\mathbf{e} \mid \mathbf{f})$. This is called *search error*.

How can we tell which problem is the culprit?[3] One way would be to solve problem 2, a purely computational problem. If we do and the resulting translations are good, then our work is finished. If the resulting translations are still poor, then we at least know that we should focus on creating better models. For this homework, we will focus only on problem 2. (But don't worry, we'll revisit problem 1 in the next homework.)

If we're going to improve search so that we can find translations with higher probability, we need to measure the probability of the translations that our decoder finds. You can compute a value that is proportional to $p(\mathbf{e} \mid \mathbf{f})$ using `compute-model-score`.

```
python compute-model-score < output.default
```

It will take a few minutes. Make a cup of tea.

The `compute-model-score` program computes the probability of the decoder's output according to the model. It does this by summing the probabilities of all possible alignments that the model could have used to generate the English from the French, including translations that permute the phrases. That is, for each input $\mathbf{f}$ and output $\mathbf{e}$ it exactly computes:

$$\log \left(\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \right)$$

This value is proportional to $p(\mathbf{e} \mid \mathbf{f})$ up to a constant $\frac{1}{p(\mathbf{f})}$. In general it is intractable to compute this sum, and if you inspect the code you'll find that the implementation uses an exponential-time algorithm. But exponential worst case is a loose upper bound for this computation, and for these particular instances the sum will only take a few minutes. It is easier to do this than it is to find the optimal translation.

I highly recommend that you look at **the code** (https://github.com/achimr/dreamt-gt/blob/master/decoder/compute-model-score) for `compute-model-score`. It uses an unpruned exact dynamic program to compute the sum, so it may give you some hints about how to do the homework! It also contains some useful utility functions to **add probabilities in logarithmic space** (https://github.com/achimr/dreamt-gt/blob/master/decoder/compute-model-score#L16) and **manipulate bitmaps** (https://github.com/achimr/dreamt-gt/blob/master/decoder/compute-model-score#L8).

Now let's turn our attention to the decoder you've been given. It generates the most probable translations that it can find, but it uses three common approximations that can cause search error.

The first approximation of our decoder is the *Viterbi approximation*. Instead of computing the intractable sum over all alignments for each sentence, it seeks the best single alignment and uses the corresponding translation.

$$\begin{aligned} \mathbf{e}^* &= \arg\max_{\mathbf{e}} \log \left(\textstyle\sum_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \right) \\ &\approx \arg\max_{\mathbf{e}} \log \left(\max_{\mathbf{a}} p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e}) \times p_{\text{LM}}(\mathbf{e}) \right) \end{aligned}$$

This approximation vastly simplifies the search problem, since as we've already discussed, computing the sum for even a single $\mathbf{e}$ can take exponential time. Computing sums for an exponential number of outputs is doubly exponential (that's bad).

The second approximation of our decoder is that it translates French phrases into English without changing their order. So, it only reorders words if the reordering has been memorized as a phrase pair. For example, in the first sentence, we see that *mardi dernier* is correctly translated as *last Tuesday*.

```
head -1 output.default
```

If we consult `data/tm`, we will find that this happens because the model has memorized the phrase translation `mardi dernier ||| last Tuesday`.

```
grep "mardi dernier" data/tm
```

But in the second sentence, we see that *Comité de sélection* is translated as *committee selection*, rather than the more fluent *selection committee*. To show that this is a search problem rather than a modeling problem, we can generate the latter output by hand and check that the model really prefers it.

```
head -2 data/input | tail -1 > example
```

```
python decode -i example | python compute-model-score -i example
echo a selection committee was achievement . | python compute-model-score -i example
```

The scores are reported as log-probabilities, and higher scores (with lower absolute value) are better. We see that the model prefers *selection committee*, but the decoder does not consider this word order since it has never memorized this phrase pair.

Not searching over permutations (reorderings) of the translated phrases is harmful because the decoder cannot find translations with higher model score, but it is beneficial in one way, because admits a straightforward dynamic program, which we will now define under the simplifying assumption of a bigram language model (the default decoder uses a trigram language model). Let $\mathbf{f} = f_1 ... f_I$ and, for each $f_i ... f_j$ let $t(f_i ... f_j)$ denote its set of possible phrase translations. The question our decoder must answer is: what is the most probable translation under our model, under the constraint that phrase translations are one-to-one, in the same order as the source, covering all source words exactly once?

We called this translation $\mathbf{e}^*$ above, so let's continue to use that notation. Although $\mathbf{e}^*$ must be chosen from a large set of translations, we'd like to decompose this choice into a smaller set of decisions that factor with $p_{\text{TM}}(\mathbf{f}, \mathbf{a} \mid \mathbf{e})$ and $p_{\text{LM}}(\mathbf{e})$. Since our language model must define a stop probability, we know that the best translation must contain a bigram probability of the $STOP$ symbol, conditioned on its final word. Let's use $h(j, e)$ to denote the highest probability sequence that translates j words of the input and ends in word e from English vocabulary V_E, and $p(h(j, e))$ to denote the product of the translation model probability for all phrases used and the language model probability for the entire sequence up to word e. For $h(j, e)$, this is the probability of the full translation divided by the probability of the transition from e to $STOP$, so we can define $\mathbf{e}^*$ this way:

$$\mathbf{e}^* = \arg \max_{h(I,e): e \in V_E} \log p(h(I, e)) + \log p_{\text{LM}}(STOP \mid e)$$

This is a good start, because now we've defined $\mathbf{e}^*$ as a choice from among V_E objects rather than an exponential number of objects (as a practical matter, we'll only need to inspect $h(I, e)$ for those e that can actually appear in a valid translation of $\mathbf{f}$, a much smaller set than V_E). But now we have a new problem: how do we define $h(j, e)$? Since it is constructed from a sequence of phrase translations, let's break it into two parts: a prefix that translates the words $f_1 ... f_i$ and the final English phrase, which must be a translation of the French words $f_{i+1} ... f_j$ for some position $0 \leq i < j$. There are many possible choices for i, the translation of $f_1 ... f_i$ may end in many possible English words, and there may be many translations of $f_{i+1} ... f_j$. We must maximize over all combinations of these choices:

$$h(j, e) = \arg \max_{h(i,e')e_1 ... e_k e: 0 \leq i < j, e' \in V_E, e_1 ... e_k e \in t(f_{i+1} ... f_j)} \log p(h(i, e')) + \log p_{\text{TM}}(f_{i+1} ... f_j \mid e_1 ... e_k e) + \log p_{\text{LM}}(e_1 \mid e') + \sum_{k'=1}^{k-1} \log p_{\text{LM}}(e_{k'+1} \mid e_{k'}) + \log p_{\text{LM}}(e \mid e_k)$$

All that is left is to define a base case:

$$h(0, e) = \begin{cases} 1 & \text{if } e = START \\ 0 & \text{otherwise} \end{cases}$$

Using induction, convince yourself that this recursion defines every highest-probability *hypothesis* (or partial translation) of the sentence, because it recursively considers all possible ways of reaching that hypothesis. By extension, it must also define the highest-probability translation $\mathbf{e}^*$.

To implement this dynamic program, we actually compute the recursion left-to-right, from smaller to larger values of j. In this way, we always have all the information we need when computing $h(j, e)$. As a practical matter, the way we do this is to ask, for each $h(i, e')$, which larger hypothesis $h(j, e)$ it might be a maximizing prefix to, and then compute the probability of $h(j, e)$ as if this were true. If the newly computed probability is indeed higher than any probability we previously computed for $h(j, e)$, we store it with $h(j, e)$ and make $h(i, e')$ its predecessor. The MT jargon term for this is *recombination*.

The form of the recursion gives us a strong hint about the upper bound complexity of the dynamic program, since if we consider all possible assignments of i, j, e, e', and $e_1, ..., e_k$ in the central recursion, we can deduce that complexity is $\mathcal{O}(I^2)$ if phrases can be arbitrarily long, and $\mathcal{O}(IK)$ if phrases have a maximum length K (which is true in the default decoder and most practical implementations). However there is a large factor (constant in the input length) due to language model computations.

The third approximation of our decoder is pruning: as it constructs the dynamic program from left to right, for each source position j it remembers only the highest probability values for $h(j, e)$ (over all $e \in V_E$) and discards the rest. The decoder uses *histogram pruning*, in which at most s hypotheses are retained at each position j. By default it only keeps 1 hypothesis. Pruning introduces approximation into the search, because a hypothesis leading to the overall best translation may be pruned during search. At each position j, we keep the s best hypotheses in a *stack* (the unfortunate MT jargon term for a priority queue).

The default decoder also implements another common form of pruning, which is that it only considers the k most probable translations for each phrase. By default, $k = 1$.

To see all of this in code, start at **line 33** (https://github.com/achimr/dreamt-gt/blob/master/decoder/decode#L33) of the default decoder, which closely follows the pseudocode in your textbook (Figure 6.6 on p. 165).

- Lines 33-35 define data structures: we have a hypothesis data structure that summarizes a partial English translation of the French sentence, and a set of n+1 stacks. `stack[i]` will eventually hold different hypotheses about ways to translate exactly i words of the French. The hypotheses in a stack are indexed by their language model state, which is the only information needed to correctly score their extensions. In particular, notice that the hypothesis *does not* store j, the number of words translated, because in the default decoder we can determine that from the stack on which we found the hypothesis. This is not true in more advanced dynamic programs, so you may need to revisit this implementation decision when you get to later parts of the homework.
- Line 36 places an initial hypothesis ($h(0, START)$) in the 0th stack. Its LM state is simply the $START$ token. (Fig 6.6 line 1).
- At line 37, we iterate over the stacks, processing each in turn. In other words, we process the hypotheses in order of the number of French words they have translated. (Fig 6.6 line 2).
- At line 38, we iterate over all hypotheses in the stack (Fig 6.6 line 3). Prior to this, we sort the hypotheses according the LM score and then prune the stack, leaving only the top s hypotheses (Fig 6.6 line 9).
- Lines 39-41 iterate over all possible phrases that we might choose to extend the current hypothesis (Fig

6.6 line 4-5). Since the default decoder is monotonic, we know that all hypotheses in stack i represent translations of the first i words. So we only look at translations covering the sequence of words starting at position $i+1$ and going to any position j. But note that the strings in our discussion have been 1-indexed while those in python are 0-indexed.

- Lines 42-48 create a new hypothesis that extend the current hypothesis by appending the selected phrase translation (Fig 6.6 line 6). This requires incorporating the phrase translation probability (line 42) and the language model score of the new words, including possibly the end-of-sentence token if the new hypothesis will represent a translation of the entire sentence (lines 43-47).
- Lines 49-50 add the new hypothesis to a stack (Fig 6.6 line 7). Again, since we're translating monotonically, we can just put it in stack j. Notice that in the second condition of line 49 we also do recombination: if there was already a hypothesis with the same LM state but lower probability, we replace it with the newly created one, which represents a higher-probability path with identical completions (Fig 6.6 line 8).
- Lines 51-54 print the best hypothesis. First, we choose the best hypothesis in the final stack, which is simply the one with the highest probability (line 51). Then we trace back through the sequence of hypotheses that lead to it, and print out the corresponding English words in order using the recursive function `extract_english` (this is not a closure. I just included it at this point to keep all of the logic in one place, in the interest of clarity). You probably won't need to modify this bit of code.

Now that we've seen how all of this works, it's time to experiment with some of the pruning parameters, and see how they trade search accuracy for speed.

```
time python decode > output.default
python compute-model-score < output.default
time python decode -s 100 -k 10 > output.s=100.k=10
python compute-model-score < output.s=100.k=10
vimdiff output.default output.s=100.k=10
```

(Type `:q:q` to exit vimdiff; on Windows use `gvim -d` instead of `vimdiff`).

Question 1 [20 points]: Experiment with different combinations of values for the stack size parameter `-s` and the maximum number of translations `-k`, until you can no longer obtain any improvements. How do changes in these parameters affect the resulting log-probabilities? How do they affect speed? How do they affect the translations?

Baseline: Local Reordering

Your task is to **find the most probable English translation**. Our model assumes that any segmentation of the French sentence into phrases followed by a one-for-one substitution and permutation of those phrases is a valid translation.

The baseline approach that I want you to explore is one with a very limited amount of reordering: the ability to swap translations of adjacent phrases. More precisely, if $\vec{e}_1$ is the translation of French words $f_i...f_k$ and $\vec{e}_2$ is the translation of French words $f_{k+1}...f_j$, then your output can contain them in the order $\vec{e}_2\vec{e}_1$. Notice that since phrases can be swapped only if they are translations of adjacent phrases, further swaps involving $\vec{e}_1$ and $\vec{e}_2$ are no longer possible, because the right neighbour of $\vec{e}_1$ was not adjacent to it in the original order,

likewise the left neighbour of $\vec{e}_2$. In other words, we cannot permute sequences of three or more phrases under this definition of reordering.

Your new decoder will now contain the correct translation for the *selection committee* example in its search space, though it may discover some other, even more probable translation, so you may not get exactly this output.

Question 2 [20 points]: Define a new dynamic program for the search space described above. You may find it helpful to use the notation for the default search decoder, but you are free to use a different notation as long as you describe its meaning.

HINT: You can start as above, by assuming $h(I, e)$ is the highest-probability translation of $f_1...f_I$. Think carefully about all possible ways of reaching this translation. You may (recursively) discover some new cases that you'll also have to reason about until you have a complete system.

HINT: The more precise your description, the more easily you'll be able to translate it into code.

Question 3 [10 points]: What is the computational complexity of your dynamic program?

Question 4 [5 points]: Define a mapping from hypothesis objects of your new dynamic program to stacks. In other words: which stack should a hypothesis be placed on?

Question 5 [15 points]: Implement your new dynamic program.

HINT: If you adapt the default decoder, this requires a relatively small amount of code; my implementation is about 15 lines longer than the default decoder. You will need to change the hypothesis object according to the new dynamic program that you've written, and you may even need multiple types of hypothesis objects. You will also need to think carefully about which stack to place new hypothesis objects in. If you've carefully answered the questions above, you should have a good idea about how to do this.

Question 6 [10 points]: Using your new decoder, experiment with different combinations of values for the stack size parameter `-s` and the maximum number of translations `-k`, until you can no longer obtain any improvements. How do changes in these parameters affect the resulting log-probabilities? How do they affect speed? How do they affect the translations?

The Challenge

Question 7 [20 points]: Implementing a decoder that can swap adjacent phrases and answering the accompanying questions is a good start to understanding decoding. But swapping adjacent phrases will not get you anywhere close to the most probable translation according to this model. To get fully understand decoding, you **must** additionally experiment with another decoding algorithm. Any permutation of phrases is a valid translation, so you might attempt to search over all or some part of this larger space. (Correctly described dynamic programs without an implementation will get you part of the way there). This search is NP-Hard, so it will not be easy. You can trade efficiency for search effectiveness by implementing histogram pruning or threshold pruning, or by using reordering limits as described in the textbook (Chapter 6). You might widen the scope of the search by considering local permutations of three phrases, rather than two. Or, you might consider implementing other approaches to solving the combinatorial optimization problem implied by the Viterbi approximation:

- **Implement a greedy decoder** (http://www.iro.umontreal.ca/~felipe/bib2webV0.81/cv/papers/paper-tmi-2007.pdf).
- **Use chart parsing to search over many permutations in polynomial time** (http://aclweb.org/anthology/C/C04/C04-1030.pdf).
- **Use a traveling salesman problem (TSP) solver** (http://aclweb.org/anthology/P/P09/P09-1038.pdf).
- **Use finite-state algorithms** (http://mi.eng.cam.ac.uk/~wjb31/ppubs/ttmjnle.pdf).
- **Use Lagrangian relaxation** (http://aclweb.org/anthology/D/D13/D13-1022.pdf).
- **Use integer linear programming** (http://aclweb.org/anthology/N/N09/N09-2002.pdf).
- **Use A* search** (http://aclweb.org/anthology/W/W01/W01-1408.pdf).

These methods all attempt to approximate or solve the Viterbi approximation to decoding. You can also try to approximate $p(\mathbf{e} \mid \mathbf{f})$ directly.

- **Change the decision function** (http://anthology.aclweb.org/N/N04/N04-1022.pdf) to minimize Bayes risk, which explicitly sums over translations.
- **Use variational algorithms** (http://aclweb.org/anthology//P/P09/P09-1067.pdf).
- **Use Markov chain Monte Carlo algorithms** (http://aclweb.org/anthology//W/W09/W09-1114.pdf).

But the sky's the limit! There are many ways to decode. You can try anything you want.

Acknowledgements

This homework was developed by **Adam Lopez** (https://alopez.github.io/) in collaboration with **Chris Callison-Burch** (http://www.cis.upenn.edu/~ccb/), **Chris Dyer** (http://www.cs.cmu.edu/~cdyer), and **Matt Post** (http://cs.jhu.edu/~post/).

Footnotes

---

1. the machine translation jargon for a sequence of words is *phrase*, but these aren't phrases in any linguistic theory—they can be any sequence of words in the sentence.↩
2. Technically, this makes the default decoder a *hidden semi-Markov model*↩
3. Sometimes these problem combine to create a ***fortuitous search error*** (https://www.aclweb.org/anthology/P/P01/P01-1030.pdf), where inexact search finds a better translation than the one preferred by a poor model. We will not try to cause fortuitous search errors!↩