Assignment 3. Text Reconstruction

Hwanjo Yu
CSED342 - Artificial Intelligence

Contact: TA WooJoo Kim (kimuj0103@postech.ac.kr)
Deadline: Apr 4th 2023 at 14:00 (50% penalty for every 1 day)

General Instructions

This (and every) assignment has a written part and a programming part.

✏This icon means a written answer is expected in `writeup.pdf`. Refer to `writeup.tex` for pdf file generation.

⌨This icon means you should write code in `submission.py`.

You should modify the code in `submission.py` between

```
# BEGIN_YOUR_CODE
```

and

```
# END_YOUR_CODE
```

but you can add other helper functions outside this block if you want. Do not make changes to files other than `submission.py`.

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `grader.py`, but the correct outputs are not. To run all the tests, type

```
python grader.py
```

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., `3a-0-basic`) by typing

```
python grader.py 3a-0-basic
```

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

Problems

In this homework, we consider two tasks: word segmentation and vowel insertion. **Word segmentation** often comes up in processing many non-English languages, in which words might not be flanked by spaces on either end, such as in written Chinese or in long compound German words.[1]

Vowel insertion is relevant in languages such as Arabic or Hebrew, for example, where modern script eschews notations for vowel sounds and the human reader infers them from context.[2] More generally, it is an instance of a reconstruction problem given lossy encoding and some context.

We already know how to optimally solve any particular search problem with graph search algorithms such as uniform cost search or A*. Our goal here is modeling — that is, converting real-world tasks into state-space search problems.

Setup: n-gram language models and uniform-cost search

Our algorithm will base segmentation and insertion decisions based on the cost of produced text according to a language model. A language model is some function of the processed text that captures its fluency by estimating the likelihood of text $p(w_1, w_2, \ldots, w_{N-1}, w_N) = \sum_{i=1}^{N} p(w_i | w_1, \ldots, w_{i-1})$.

A very common language model in NLP is an n-gram sequence model, which assumes $p(w_i | w_1, \ldots, w_{i-1}) = p(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$. [3] We'll use the n-gram model's negative log-likelihood $-\log p(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$ as a cost function $c(w_{i-(n-1)}, w_{i-1}, \ldots, w_i)$. The cost will always be positive, and lower costs indicate better fluency.[4] As a simple example: in a case where $n = 2$ and c is our n-gram cost function, $c(\mathsf{big}, \mathsf{fish})$ would be low, but $c(\mathsf{fish}, \mathsf{fish})$ would be fairly high.

Furthermore, these costs are additive: for a unigram model u ($n = 1$), the cost assigned to $[w_1, w_2, w_3, w_4]$ is

$$u(w_1) + u(w_2) + u(w_3) + u(w_4).$$

---

[1]In German, Windschutzscheibenwischer is "windshield wiper". Broken into parts: wind wind; schutz block / protection; scheiben panes; wischer wiper.

[2]See https://en.wikipedia.org/wiki/Abjad.

[3]This model works under the assumption that text roughly satisfies the Markov property.

[4]This estimate $p(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$ is gathered from frequency counts taken by reading Leo Tolstoy's *War and Peace* and William Shakespeare's *Romeo and Juliet*. But, the estimation is applied with 1) *parameter sharing* that assume $p(w_i | w_{i-(n-1)}, \ldots, w_{i-1}) = p(w_j | w_{j-(n-1)}, \ldots, w_{j-1})$ when two n-gram word sequences are identical, and 2) *laplace smoothing* that prevents $p(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$ to become zero. We'll cover these two techniques in the chapter 7.

For a bigram model b ($n = 2$), the cost is

$$b(w_0, w_1) + b(w_1, w_2) + b(w_2, w_3) + b(w_3, w_4),$$

where w_0 is `-BEGIN-`, a special token that denotes the beginning of the sentence. We have estimated u and b based on the statistics of n-grams in text. All the costs returned by the cost functions are non zero. Also, any words not in the corpus are automatically assigned a high cost, so you do not have to worry about this part.

Problem 1. Word Segmentation

In word segmentation, you are given as input a string of alphabetical characters (`[a-z]`) without whitespace, and your goal is to insert spaces into this string such that the result is the most fluent according to the language model.

Problem 1a [4 points] ⌨

Implement an algorithm that finds the optimal word segmentation of an input character sequence. Your algorithm will consider costs based simply on a unigram cost function.

Before jumping into code, you should think about how to frame this problem as a state-space search problem. How would you represent a state? What are the successors of a state? What are the state transition costs?

Uniform cost search (UCS) is implemented for you, and you should make use of it here.[5]

Fill in the member functions of the `WordSegmentationProblem` class and the `segmentWords` function. The argument `unigramCost` is a function that takes in a single string representing a word and outputs its unigram cost. You can assume that all the inputs would be in lower case. The function `segmentWords` should return the segmented sentence with spaces as delimiters, i.e. ' '.join(words).

For convenience, you can actually run `python submission.py` to enter a console in which you can type character sequences that will be segmented by your implementation of `segmentWords`. To request a segmentation, type `seg mystring` into the prompt. For example:

```
>> seg thisisnotmybeautifulhouse
    Query (seg): thisisnotmybeautifulhouse
    this is not my beautiful house
```

Console commands other than `seg` – namely `ins` and `both` – will be used for the upcoming parts of the assignment. Other commands that might help with debugging can be found by typing `help` at the prompt.

You are encouraged to refer to `NumberLineSearchProblem` and `GridSearchProblem` implemented in util.py for reference. They don't contribute to testing your submitted code but only serve as a guideline to how your code should look like.

---

[5]Solutions that use UCS ought to exhibit fairly fast execution time for this problem, so using A* here is unnecessary.

Problem 1b [6 points] ⌨

Implement an algorithm that finds the optimal word segmentation for a given input character sequence, with the constraint that it can only have at most k words. This is called k-word segmentation. For example, if the input sequence is 'pepperonimage' and k is 2, it should be segmented into 'pepperoni mage' instead of 'pepper on image', even though the latter is more fluent.

When you've completed your implementation, the function segmentKWords should return the k-segmented sentence with spaces as delimiters, i.e., ' '.join(words). You can assume that k ranges from 1 to the length of the input sequence (or the number of characters in the sequence). The argument unigramCost is the same as in problem 1a.

To test your implementation, use the k-seg command in the program console followed by the value of k. For example:

```
>> k-seg 4 thisisnotmybeautifulhouse
   Query (k-seg) (k = 4): thisisnotmybeautifulhouse
   this isnotmy beautiful house
```

Problem 2. Vowel Insertion

Now you are given a sequence of English words with their vowels missing (A, E, I, O, and U; never Y). Your task is to place vowels back into these words in a way that maximizes sentence fluency (i.e., that minimizes sentence cost). For this task, you will use a bigram cost function.

You are also given a mapping possibleFills that maps any vowel-free word to a set of possible reconstructions (complete words).[6] For example, possibleFills('fg') returns set(['fugue', 'fog']).

Problem 2a [4 points] ⌨

Implement an algorithm that finds optimal vowel insertions. Use the UCS subroutines.

When you've completed your implementation, the function insertVowels should return the reconstructed word sequence as a string with space delimiters, i.e. ' '.join(filledWords). Assume that you have a list of strings as the input, i.e. the sentence has already been split into words for you. Note that empty string is a valid element of the list.

The argument queryWords is the input sequence of vowel-free words. Note well that the empty string is a valid such word. The argument bigramCost is a function that takes two strings representing two sequential words and provides their bigram score. The special out-of-vocabulary beginning-of-sentence word -BEGIN- is given by wordsegUtil.SENTENCE_BEGIN. The argument possibleFills is a function; it takes a word as string and returns a set of reconstructions.

Since we use a limited corpus, some seemingly obvious strings may have no fills, eg chclt → {}, where chocolate is actually a valid fills. Dont worry about these cases.

Note: If some vowel-free word w has no reconstructions according to possibleFills, your implementation should consider w itself as the sole possible reconstruction.

---

[6]This mapping, too, was obtained by reading Tolstoy and Shakespeare and removing vowels.

Use the `ins` command in the program console to try your implementation. For example:
```
>> ins thts m n th crnr
    Query (ins): thts m n th crnr
    thats me in the corner
```
The console strips away any vowels you do insert, so you can actually type in plain English and the vowel-free query will be issued to your program. This also means that you can use a single vowel letter as a means to place an empty string in the sequence. For example:
```
>> ins its a beautiful day in the neighborhood
    Query (ins): ts  btfl dy n th nghbrhd
    its a beautiful day in the neighborhood
```

Problem 2b [6 points] ⌨

This time, you are given a sequence of English words with missing vowels and a set of specific vowels. Implement an algorithm for the limited vowel insertion problem that inserts vowels into the words without using the provided set of restricted vowels. Use the UCS subroutines.

When you've completed your implementation, the function `insertLimitedVowels` should return the reconstructed word sequence containing only the allowed vowels as a string with space delimiters, i.e., `' '.join(filledWords)`. The input set of restricted vowels is assumed to be given as a string (e.g., 'a', 'iou'). The other assumptions and arguments `queryWords`, `bigramCost`, and `possibleFills` are the same as in problem 2a.

To test your implementation, use the `limited-ins` command in the program console, followed by the string of restricted vowels. The query is preprocessed in the same way as the `ins` command. For example:
```
>> limited-ins i thats me in the corner
    Query (limited-ins) (limited_vowels = 'i'): thts m n th crnr
    thats me on the corner
```

Problem 3: Putting It Together

We'll now see that it's possible to solve both of these tasks at once. This time, you are given a whitespace- and vowel-free string of alphabetical characters. Your goal is to insert spaces and vowels into this string such that the result is the most fluent possible one. As in the previous task, costs are based on a bigram cost function.

Problem 3a [6 points] ⌨

Implement an algorithm that finds the optimal space and vowel insertions. Use the UCS subroutines.

When you've completed your implementation, the function `segmentAndInsert` should return a segmented and reconstructed word sequence as a string with space delimiters, i.e. `' '.join(filledWords)`.

The argument `query` is the input string of space- and vowel-free words. The argument `bigramCost` is a function that takes two strings representing two sequential words and provides their bigram score. The special out-of-vocabulary beginning-of-sentence word `-BEGIN-`

is given by `wordsegUtil.SENTENCE_BEGIN`. The argument `possibleFills` is a function; it takes a word as string and returns a `set` of reconstructions.

Note: Unlike in problem 2, where a vowel-free word could (under certain circumstances) be considered a valid reconstruction of itself, here you should only include in your output words that are the reconstruction of some vowel-free word according to `possibleFills`. Additionally, you should not include words containing only vowels such as "a" or "I"; all words should include at least one consonant from the input string.

Use the command `both` in the program console to try your implementation. Similar to `ins` command, vowels are striped and spaces are also ignored. For example:

```
>> both imagine all the people
   Query (both): mgnllthppl
   imagine all the people
```

Problem 4: A* search

Now, we'll apply A* search to accelerate search speed. First, you exercise by making a simple problem and a heuristic function to be familiar with A* search. Then, you make a heuristic function for the text reconstruction task.

Problem 4a [4 points] ⌨

In this problem, you should define <u>your own</u> simple search problem `SimpleProblem` and a heuristic function `admissibleButInconsistentHeuristic`. As the name suggests, the heuristic function should be admissible but not consistent, so A* <u>cannot find the minimum cost path</u> with the heuristic function. Also, we assume the heuristic returns 0 when given an end state. Before implementing them, check `UniformCostSearch` and its parameter `heuristic` to examine how A* works.

Problem 4b [6 points] ⌨

We're going to speed up the joint space and vowel insertion problem with A*. Recall that score an output using a bigram model $b(w', w)$ is more expensive than using a unigram model $u(w)$ because we have to remember the previous word w' in the state. Now let's tackle the task by following the guideline below:

1. Implement `makeWordCost` that returns a unigram cost function `wordCost` (which takes any w and returns a number) when given the bigram cost function `bigramCost` (which takes any (w', w) and returns a number). Your can exploit $u_b(w) = \min_{w'} b(w', w)$, where u_b and b corresponds to `wordCost` and `bigramCost` respectively. Also, you may need `wordsegUtil.SENTENCE_UNK` to indicate any non-existing word in training corpus.

   Note: Don't confuse u_b defined here with the unigram cost function u used in Problem 1.

2. Implement `RelaxedProblem` which is a relaxed problem of `JointSegmentationInsertionProblem`. The relaxed problem calculate action's cost based on `wordCost`.

3. Implement `makeHeuristic` which returns a consistent heuristic function for the given query. You can exploit `RelaxedProblem` and `util.DynamicProgramming`.

4. Finally implement `fastSegmentAndInsert` which should be faster than `segmentAndInsert`. You should use `UniformCostSearch.solve` with a proper `heuristic` argument.