Problem Set 3

Artificial Intelligence
Fall 2021 CS47100-AI

Student name: ____________________ Student PUID: ____________

Note: You are free to use your intuition to find the steps in the proof. But, make sure you do not use your intuition to justify steps in your proofs.[1]

In this homework, we consider two tasks: *word segmentation* and *vowel insertion*. Word segmentation often comes up when processing many non-English languages, in which words might not be flanked by spaces on either end, such as written Chinese or long compound German words. Vowel insertion is relevant for languages like Arabic or Hebrew, where modern script eschews notations for vowel sounds and the human reader infers them from context. More generally, this is an instance of a reconstruction problem with a lossy encoding and some context.

Our algorithm will base its segmentation and insertion decisions on the cost of processed text according to a language model. A language model is some function of the processed text that captures its fluency.

A very common language model in NLP is an *n-gram sequence model.* This is a function that, given n consecutive words, provides a cost based on the negative log likelihood that the n-th word appears just after the first $n-1$ words. The cost will always be positive, and lower costs indicate better fluency. As a simple example: In a case where $n = 2$ and c is our n-gram cost function, $c(\text{big, fish})$ would be low, but $c(\text{fish, fish})$ would be fairly high.

Furthermore, these costs are additive: For a unigram model $u(n = 1)$, the cost assigned to $[w1, w2, w3, w4]$ is,

$$u(w1) + u(w2) + u(w3) + u(w4)$$

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

$$b(w0, w1) + b(w1, w2) + b(w2, w3) + b(w3, w4)$$

where $w0$ 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. Note that any words not in the corpus are automatically assigned a high cost, so you do not have to worry about that part.

Now, answer the following 3 questions based on this information.

Problem 1. 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.

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. `UniformCostSearch (UCS)` is implemented for you in util.py, and you should make use of it here.

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? (You don't need to answer these questions anywhere.)

[1] *The contents of this problem set is based on the AI course CS221 taught at Stanford University.*

Fill in the member functions of the `SegmentationProblem` 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 of 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 in the upcoming parts of the assignment. Other commands that might help with debugging can be found by typing help at the prompt.

Hint: 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 for what your code should look like.

Hint: The actions that are valid for the `ucs` object can be accessed through `ucs.actions`.

Problem 2. Now you are given a sequence of English words with their vowels missing (A, E, I, O, and U; never any consonant). 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). For example, `possibleFills('fg')` returns `set(['fugue', 'fog'])`.

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 the empty string is a valid element of the list.

The argument `queryWords` is the input sequence of vowel-free words. Note 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 that takes a word as a string and returns a set of reconstructions.

Since we use a limited corpus, some seemingly obvious strings may have no filling, such as `chclt -> {}`, where chocolate is actually a valid filling. Don't 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. Otherwise you should always use one of its possible completions according to `possibleFills`.

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

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 that takes a word as a string and returns a set of reconstructions.

Note: In problem 2, a vowel-free word could, under certain circumstances, be considered a valid reconstruction of itself. However, for this problem, in your output, you should only include 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* or out of vocabulary words; all words should include at least one consonant from the input string and a solution is guaranteed. Additionally, aim to use a minimal state representation for full credit.

Use the command `both` in the program console to try your implementation. For example:

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