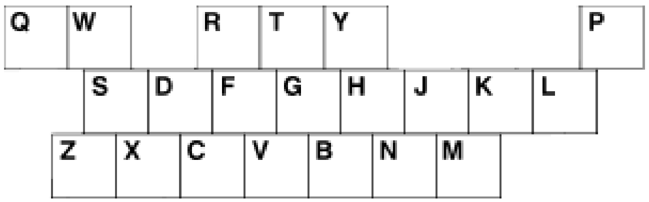
Text reconstruction

Stanford CS221 Fall 2014-2015

Assignment by Roy Frostig
Owner TA: Will Harvey
Note: grader.py only provides basic tests. Passing grader.py does not by any means guarantee full points.



Version 2
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 min-cost state-space search problem with graph search algorithms such as uniform cost search or A*. Our goal here is modeling, 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.

A very common language model in NLP is an n-gram sequence model: a function that, given n consecutive words, gives a cost based on to the negative likelihood that the n-th word appears just after the first $n-1$.[3] 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, $c(\texttt{big}, \texttt{fish})$ would be low, but $c(\texttt{fish}, \texttt{fish})$ 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

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

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, so you do not have to worry about this part.

A note on low-level efficiency and expectations: this assignment was designed considering input sequences of length no greater than roughly 200 (characters, or list items, depending on the task). Of course, it's great if programs tractably manage larger inputs, but it isn't expected that such inputs not lead to inefficiency due to overwhelming state space growth.

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.

a. Consider the following greedy algorithm: Begin at the front of the string. Find the ending position for the next word that minimizes the language model cost. Repeat, beginning at the end of this chosen segment.

   Show that this greedy search is suboptimal. In particular, provide an example input string on which the greedy approach would fail to find the lowest-cost segmentation of the input.

   In creating this example, you are free to design the n-gram cost function (both the choice of n and the cost of any n-gram sequences) but costs must be positive and lower cost should indicate better fluency. Your example should be based on a realistic English word sequence — don't simply use abstract symbols with designated costs.

b. Implement an algorithm that, unlike greedy, 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? (You don't need to answer these questions in your writeup.)

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

   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. 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.

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'])`.

a. Consider the following greedy-algorithm: from left to right, repeatedly pick the immediate-best (according to bigram cost) vowel insertion for the current vowel-free word given the insertion for the previous (i.e., not taking into account future insertions).

   Show, as in question 1, that this greedy algorithm is suboptimal, by providing a realistic counter-example using English text. Make any assumptions you'd like about possibleFills and the bigram cost function, but bigram costs must remain positive.

b. 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)`.

   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.

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

   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: 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.

a. Consider a search problem for finding the optimal space and vowel insertions. Formalize the problem as a search problem, that is, what are the states, actions, costs, initial state, and goal test? Try to find a minimal representation of the states.

b. 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 never include in your output a word that is not the reconstruction of some vowel-free word according to `possibleFills`. Additionally, you should not include all vowel words 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. For example:

```
>> both mgnllthppl

  Query (both): mgnllthppl

  imagine all the people
```

c. Let's find a way to speed up joint space and vowel insertion with A*. Recall that having to 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. Given $b(w', w)$, define $u(w)$ based on $b(w', w)$, define a heuristic h_u based on solving a simpler minimum cost path problem with u, and prove that $h_u(s)$ is consistent. In your description, you must be explicit about which search problem (the original or the simpler one) you are referring to and what the states are.

---

[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] Modulo edge cases, the n-gram model score in this assignment is given by $\ell(w_1, \ldots, w_n) = -\log(p(w_n \mid w_1, \ldots, w_{n-1}))$. Here, $p(\cdot)$ is an estimate of the conditional probability distribution over words given the sequence of previous $n-1$ words. This estimate is gathered from frequency counts taken by reading Leo Tolstoy's *War and Peace* and William Shakespeare's *Romeo and Juliet*.

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

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