Text Reconstruction

Stanford CS221 Spring 2018-2019

Owner CA: Anna Zhu

Version: 1

General Instructions

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

The full assignment with our supporting code and scripts can be downloaded as reconstruct.zip.

a.  This icon means a written answer is expected in `reconstruct.pdf`.

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

All written answers must be **in order** and **clearly and correctly labeled** to receive credit.

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`.

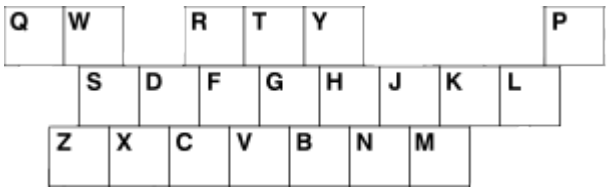
Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden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

```
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`.



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.[1] 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.[2] More generally, this is an instance of a reconstruction problem with a 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 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, gives a cost based on to the negative log 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. Note that any words not in the corpus are automatically assigned a high cost, 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.  [2 points]

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. Note that the cost function doesn't need to be explicitly defined. You can just point out the relative cost of different word sequences that are relevant to the example you provide. And your example should be based on a realistic English word sequence — don't simply use abstract symbols with designated costs.

> **Using $n = 1$:** We have a unigram cost function u. Consider the input `fifthsonata`. It's reasonable for u to favor `fifth` as the first greedily-obtained segment, and then to favor `son` over `sonata` as the former is more common. However, the greedy approach then ends up with a segmentation such as `fifth son a ta`, incurring high cost for the non-word `ta` (sorry, course TAs), whereas `fifth sonata` incurs moderate net cost.
>
> **Using $n = 2$:** We have a bigram cost function b. Consider the input `aftershavesmellsbad`. It's reasonable for b to favor `after` to `aftershave` as the former occurs more commonly in natural text at the front of a sentence. However, the sentence `after shave smells bad` is not nearly as fluent as `aftershave smells bad`.
>
> Counter-examples using $n > 2$ may be technically correct but penalized for unnecessary complexity.

b.  [10 points]

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. 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.

> **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 final actions that `ucs`(a `UniformCostSearch` object) takes can be accessed through `ucs.actions`.

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.  [2 points]

Consider the following greedy-algorithm: from left to right, repeatedly pick the immediate-best vowel insertion for current vowel-free word given the insertion that was chosen for the previous vowel-free word. This algorithm does not take into account future insertions beyond the current word.

Show, as in question 1-a, 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.

> Consider the input `ct ntrlly`. Assuming $b($ `-BEGIN- , cat` $) <$ $b($ `-BEGIN- , act` $)$, greedy gives `cat naturally` where $b(\cdot)$ would reasonably favor `act naturally`.

b.  [10 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 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 string and returns a `set` of reconstructions.

Since we use a limited corpus, some seemingly obvious strings may have no filling, e.g. chclt -> {}, where chocolate is actually a valid filling. 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.

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 as fluent as possible. As in the previous task, costs are based on a bigram cost function.

a.  [2 points]

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

Let $x_1, \ldots, x_L$ be the characters of the input.

Solution 1

- Each state is as follows: $s = (w', i, j)$ where w' is the previous word, i is the starting position of the current segment, and j is the current position.
- There are two possible actions: (i) ending the word, thus creating a new word from i to j (END), or (ii) if $j < L$, incrementing j and moving on (CONT).
- The cost from $\text{Cost}((w', i, j), \text{END}) = b(w', x_{i:j})$. The cost from $\text{Cost}((w', i, j), \text{CONT}) = 0$.
- The initial state is $s =$ (`-BEGIN-`$, 0, 0)$ and the end test is $s = (w', L, L)$

Solution 2

- Each state is as follows: $s = (w', i)$ where w' is the previous word, i is the starting position of the current segment.
- The actions of a state (w', i) are the possible fills for all substrings of the query starting from index i $(x_{i:i+1}, x_{i:i+2}, \ldots, x_{i:L})$.
- $\text{Cost}((w', i), a) = b(w', a)$.
- The initial state is $s =$ (`-BEGIN-`$, 0)$ and the end test is $s = (w', L)$

b. [20 points]

Implement an algorithm that finds the optimal space and vowel insertions. Use the methods of the `UniformCostSearch` class from the `ucs` object.

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. For example:

```
>> both mgnllthppl

  Query (both): mgnllthppl

  imagine all the people
```

c. [4 points]

Let's find a way to speed up joint space and vowel insertion with A*. Recall that one way to find the heuristic function $h(s)$ for A* is to define a relaxed search problem P_{rel} where $\text{Cost}_{rel}(s,a) \leq \text{Cost}(s,a)$ and letting $h(s) = \text{FutureCost}_{rel}(s)$.

Given a bigram model b (a function that takes any (w', w) and returns a number), define a unigram model u_b (a function that takes any w and returns a number) based on b.

Use this function u_b to help define P_{rel}.

One example of a u_b is $u_b(w) = b(w, w)$. However this will not lead to a consistent heuristic because $\text{Cost}_{rel}(s,a)$ is not guaranteed to be less than or equal to $\text{Cost}(s,a)$ with this scheme.

Explicitly define the states, actions, cost, start state, and end state of the **relaxed problem** and explain why $h(s)$ is consistent.

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

Hint: If u_b only accepts a single w, do we need to keep track of the previous word in our state?

First, define $u_b(w) = \min_{w'} b(w', w)$.

- Explanation Corresponding to **Solution 1** in 3a

  Recall that our search state in the original problem is (w', i, j), where w' is the previous word, i is the starting position of the current segment, and j is the current position.

  Let us define a new cost on the original search problem: $\text{Cost}_{rel}((w', i, j), \text{END}) = u_b(x_{i:j})$. This is clearly at most the original cost $\text{Cost}((w', i, j), \text{END}) = b(w', x_{i:j})$ by construction of u_b. Therefore $h(s) = \text{FutureCost}_{rel}(s)$ is a consistent heuristic for the original search problem, as shown in class.

  Now, notice that $\text{Cost}_{rel}((w', i, j), a)$ does not depend on the previous word w'. Therefore, we can also simplify the state of our relaxed search problem to only include (i, j) such that computing the future costs in that problem also yields h.

- Explanation Corresponding to **Solution 2** in 3a

Recall that our search state in the original problem is (w', i), where w' is the previous word, i is the starting position of the current segment.

Let us define a new cost on the original search problem: $\text{Cost}_{rel}((w', i), a) = u_b(a)$. This is clearly at most the original cost $\text{Cost}((w', i), a) = b(w', a)$ by construction of u_b. Therefore $h(s) = \text{FutureCost}_{rel}(s)$ is a consistent heuristic for the original search problem, as shown in class.

Now, notice that $\text{Cost}_{rel}((w', i), a)$ does not depend on the previous word w'. Therefore, we can also simplify the state of our relaxed search problem to only include (i) such that computing the future costs in that problem also yields h.

d. [2 points]

We defined many different search techniques in class, so let's see how they relate to one another.

Is UCS a special case of A*? Explain why or why not.

Is BFS a special case of UCS? Explain why or why not.

Your explanations can be short.

UCS is a special case of A* where h(s) = 0 for every s.

BFS is a special case of UCS where all the edge weights are 1.

---

[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.