XCS221 Assignment 2 — Text Reconstruction

Due Sunday, November 29 at 11:59pm PT.

Guidelines

1. These questions require thought, but do not require long answers. Please be as concise as possible.
2. If you have a question about this homework, we encourage you to post your question on our Slack channel, at `http://xcs221-scpd.slack.com/`
3. Familiarize yourself with the collaboration and honor code policy before starting work.
4. For the coding problems, you may not use any libraries except those defined in the provided started code. In particular, ML-specific libraries such as `scikit-learn` are not permitted.

Submission Instructions

Written Submission: All students must submit an electronic PDF containing solutions to the written questions. As long as the submission is legible and well-organized, the course staff has no preference between a handwritten and a typeset LaTeXsubmission. Students wishing to typeset their documents should follow these recommendations:

- Type responses only in the *-sol.tex files.
- Use the commented recommendations within the Makefile to get started.

Coding Submission: All assignment code is in the `src/` subirectory. You will submit only the `src/submission.py` file. Please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` will be used to autograde your submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
python grader.py
```

There are two types of unit tests used by our autograders:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases.
- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. In the student version of `src/grader.py`, only the setup and inputs to these unit tests are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder.

For debugging purposes, a single unit test can be run locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

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

Before beginning this course, we highly recommend you walk through our Anaconda tutorial NO LINK YET to familiarize yourself with our coding environment. Please use the env defined in `src/environment.yml` to run your code. This is the same environment used by our autograder.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

Introduction

In this assignment, 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 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.[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, $c(\textsf{big}, \textsf{fish})$ would be low, but $c(\textsf{fish}, \textsf{fish})$ 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

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

Similarly, 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 that part.

A note on low-level efficiency and expectations: This assignment was designed considering input sequences of length no greater than roughly 200, where these sequences can be sequences of characters or of list items, depending on the task. Of course, it's great if programs can tractably manage larger inputs, but it's okay if such inputs can lead to inefficiency due to overwhelming state space growth.

For convenience, you can run the terminal command `python submission.py` to enter a console for testing and debugging your code. It should look like this:

```
$ python submission.py
Training language cost functions [corpus: leo-will.txt]... Done!

>>
```

Console commands like `seg`, `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.

[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*.

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) **[1 point (Written)]**

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.

(b) **[7 points (Coding)]** Implement an algorithm that, unlike the greedy algorithm, 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. [5]

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.)

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)`.

To request a segmentation, type `seg mystring` into the prompt. For example:

```
$ python submission.py
Training language cost functions [corpus: leo-will.txt]... Done!

>> seg thisisnotmybeautifulhouse

Query (seg): thisisnotmybeautifulhouse

this is not my beautiful house
```

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`.

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

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) **[1 point (Written)]** Consider the following greedy-algorithm: from left to right, repeatedly pick the immediate-best vowel insertion for the 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 problem 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) **[10 points (Coding)]** 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.

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
```

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

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-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.

(a) **[1 point (Written)]**

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.

(b) **[14 points (Coding)]** 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`; 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) **[3 points (Written)]**

Let us define a new cost on the original search problem: $\text{Cost}_{rel}((w', i, j), \text{END}) = f_b(x_{i:j})$. This is clearly at most the original cost $\text{Cost}((w', i, j), \text{END}) = b(w', x_{i:j})$ by construction of f_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.

(d) **[1 point (Written)]** 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. 1 or 2 sentences will suffice.