XCS221 Assignment 2 — Text Reconstruction

Due Sunday, November 21 at 11:59pm PT.

Guidelines

1. 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/`
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

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. More information regarding the Stanford honor code can be foudn at `https://communitystandards.stanford.edu/policies-and-guidance/honor-code`.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, 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` (the autograder) will be used to verify a correct 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 the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test 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, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

This error caused the test to fail.

In this case, start your debugging in line 23 of grader.py.

Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Introduction

In this assignment, you will 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.

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

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

Your 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 your 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.

Estimate 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:

```
(XCS221)$ 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 $\ell(w_1, \dots, w_n) = -\log(p(w_n \mid w_1, \dots, w_{n-1}))$. 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) **[3 points (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) **[11 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) **[3 points (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) **[13 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) [**3 points (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) [**16 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's find a way to speed up the joint insertion of space and vowel 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 $\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a)$, and then $h(s)$ is defined to be $\text{FutureCost}_{\text{rel}}(s)$.

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

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

One example of a f_b is $f_b(w) = b(w, w)$. However, this will not lead to a consistent heuristic because $\text{Cost}_{\text{rel}}(s, a)$ 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.

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

(d) [**3 points (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.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.a

2.a

3.a

3.c

3.d