Text Reconstruction

Stanford CS221 Autumn 2021-2022

Owner CA: Xiaoyuan Ni

Version: 04/18/2022

Ed Release Post

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 you should write responses in `reconstruct.pdf`.

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

All written answers must be **typeset (preferably in LaTeX)**. We strongly recommend using Overleaf. A link to a tex file with prompts can be found on Ed and a link to a starter guide and a generic LaTeX written answer template is provided on the main course page.

Also note that your answers should be **in order** and **clearly and correctly labeled** to receive credit. Be sure to submit your final answers as a PDF and tag all pages correctly when submitting to Gradescope.

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`.

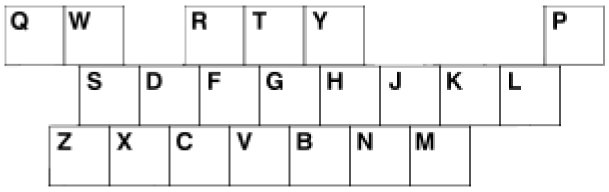
Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden 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.

We've created a LaTeX template here for you to use that contains the prompts for each question.

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, $c(\texttt{big}, \texttt{fish})$ 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

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

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.

Note: We have estimated u and b based on the statistics of n-grams in text (leo-will.txt). Note that any words not in the corpus (like "hello") are automatically assigned a high cost. This might lead to some unexpected sentences but you do not have to worry about that.

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.

You are encouraged to look over the given codebase and how functions like cleanLine() (in wordsegUtil.py) are called by grader.py and shell.py to preprocess lines.

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. Suppose that we have a unigram model u and we are given the string `breakfastservedinside`. The unigram costs of words are given as $u(\texttt{break})=3$, $u(\texttt{fast})=6$, $u(\texttt{breakfast})=8$, $u(\texttt{served})=8$, $u(\texttt{in})=3$, $u(\texttt{side})=5$,
$u(\texttt{inside})=2$. Assume $u(s)=100$ for any other substring s of our string.

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.

What is the total model cost from running this greedy algorithm on `breakfastservedinside`? Is this greedy search optimal for general inputs? In other words, does it find the lowest-cost segmentation of any input? Explain why or why not in 1-2 sentences.

What we expect: The value of the total model cost and an explanation of why the greedy algorithm is or is not optimal.

The greedy algorithm selects words in this order: `break`, `fast`, `served`, `inside`. The total unigram cost is $3+6+8+2=19$.

However, note that the segmentation of words that minimizes the cost is `breakfast served inside`, with cost $8+8+2=18$. As a result, the greedy algorithm is not optimal, as it does not find the lowest-cost segmentation for this example.

b. 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. [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)`.

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`.

What we expect: An implementation of the member functions of the `SegmentationProblem` class and the `segmentWords` function.

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 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 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 be positive.

In creating this example, 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.

What we expect: A specific (realistic) example explained within 4 sentences.

Consider the input `ct ntrlly`. Assuming $b($ `-BEGIN-` $,$ `cat` $) < b($ `-BEGIN-` $,$ `act`$)$, the greedy algorithm deduces `cat naturally`, even though a more reasonable solution would be `act naturally`.

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)`. 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: Only for Problem 2, 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`. This is NOT the case for Problem 3.

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
```

What we expect: An implementation of the member functions of the `VowelInsertionProblem` class and the `insertVowels` function.

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-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. 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.

What we expect: A formal definition of the search problem with definitions for the states, actions, costs, initial state, and end test.

Let $x_1, \dots, x_L$ 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.
- The actions of a state (w', i, j) are: (i) each of possible vowel fills for substring from i to j (END, fill), or (ii) if $j < L$, incrementing j and moving on (CONT).
- The cost for $\text{Cost}((w', i, j), (\text{END}, \text{fill})) = b(w', \text{fill})$; the cost for $\text{Cost}((w', i, j), \text{CONT}) = 0$.
- 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 ($x_{i:i+1}, x_{i:i+2}, \ldots, x_{i:L}$).
- $\text{Cost}((w', i), \text{fill}) = b(w', \text{fill})$.
- The initial state is $s =($ `-BEGIN-`$, 0)$ and the end test is $s = (w', L)$.

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 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
```

What we expect: An implementation of the member functions of the `JointSegmentationInsertionProblem` class and the `segmentAndInsert` function.

Problem 4: Failure Modes and Transparency

Now that you have a working reconstruction algorithm, let's try reconstructing a few examples. Take each of the below phrases and pass them to the `both` command from the program console.

- Example 1: "yrhnrshwllgrcslyccptthffr" (original: "your honor she will graciously accept the affair")
- Example 2: "wlcmtthhttkzn" (original: "welcome to the hot take zone")
- Example 3: "grlwllnrflprrghtnw" (original: "girl we all in our flop era right now")

a. First, indicate which examples were reconstructed correctly versus incorrectly. Recall that the system chooses outputs based on a bigram cost function [4], which is roughly low if a bigram occurred in Leo Tolstoy's *War and Peace* and William Shakespeare's *Romeo and Juliet*, and high if it didn't (the details don't matter for this problem). Then, explain what about the training data may have led to this behavior.

What we expect: 1-2 sentences listing whether each example was correctly or incorrectly reconstructed and a brief explanation with justification as to what about the training data may have led to this result.

Example: "girl we all in our flop era right now" → "grlwllnrflprrghtnw"→ "girl will near of leper right now" incorrectly reconstructed
Example: "welcome to the hot take zone" → "wlcmtthhttkzn" → "welcome to the hut to kazan" incorrectly reconstructed
Example: "your honor she will graciously accept the affair" → "yrhnrshwllgrcslyccptthffr" → "your honor she will graciously accept the affair" correctly reconstructed
Students might put that the bigrams from the last two examples aren't present in the training data. Students might justify this by explaining how the use of English has changed since the text in the corpus was written,

formal vs. informal speech, dialects, or they may talk about how some of the words in the examples have low frequency and thus low likelihood of occurrence. The student must include justification.

b. Your system, like all systems, has limitations and potential points of failure. As a responsible AI practitioner, it's important for you to recognize and communicate these limitations to users of the systems you build. Imagine that you are deploying your search algorithm from this assignment to real users on mobile phones. Write a **transparency statement** for your system, which communicates to users the conditions under which the system should be expected to work and when it might not work.

What we expect: 2-4 sentences explaining the potential failure modes of your system. Be sure to acknowledge the limitations that your system has and who should know about these limitations (i.e., who are the affected parties?).

Any reasonable explanation works. An example is "This product is able to reconstruct standard English, but is best at reconstructing older/literary/formal English and may fail for more modern English or dialects. Both senders and receivers should be aware that their messages may be incorrectly reconstructed."

c. Given the limitations found in part (a) and described in your transparency statement from (b), how could you improve your system?

What we expect: 2-4 sentences proposing a change to the datasets, how you would implement it, and why it would address the limitations you identified above.

There are many reasonable strategies. You could expand the dataset used to train the n-gram model backing the cost function to include more modern and diverse sources, such as corpora of informal chat messages, modern news articles, conversations in different dialects, or even the user's own past messages (for a more personalized experience). For the last option, retraining the model on the user's phone itself gives the added benefit of protecting their privacy.

---

[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 based on word frequencies in 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 was also obtained by reading Tolstoy and Shakespeare and removing vowels.