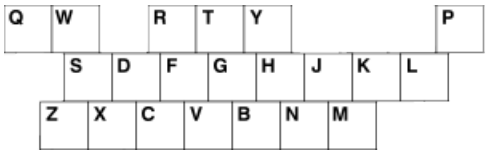
CS 221: Artificial Intelligence: Principles and Techniques

Assignment 3: Reconstruction

SUNet ID: jchan7
Name: Jason Chan
Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.



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. 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. 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. The cost will always be positive, and lower costs indicate better fluency. 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.
We have estimated u and b based on the statistics of n-grams in text (`leo-will.txt`). Note that any words (like "hello") not in the corpus 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.

Your Solution: The total model cost of the greedy algorithm is: break(3) + fast(6) + served (8) + in(3) + side(5) = 25. This greedy algorithm is not optimal for general inputs. The globally optimal cost would be breakfast(8) + served (8) + inside (2) = 18 but the greedy algorithm solves for local optimums in its iterations. For example, in the first iteration the greedy algorithm evaluates: break (3) < breakfast (8) < breakfastservedinside (100) and chooses break as the first word. Then in the next iteration it tries fast (6) < fastservedinside (100) and chooses fast, and so on.

b. 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.

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). 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.

Your Solution: Let the sequence be [th, ct, n, ht].

Assume possible fills

(a) ct is [city, coat]
(b) ht is [hat, haiti]
(c) n is [in, and]
(d) the is [the]

Assume bigramCosts are

(a) (the, city) = 7
(b) (the, coat) = 5
(c) (city, in) = 4
(d) (coat, in) = 10
(e) (in, haiti) = 12

The greedy algorithm will choose: the coat in haiti (5+10+12 = 27) rather than the city in haiti (7+4+12= 23). The greedy choice made at the start of the sentence locked the result onto a sequence with a higher mid-phase cost, and thus a suboptimal result.

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.

Your Solution:

- state: (currentIdx, previous action)
  - where currentIdx is the current character index of the string in *query* (zero based)
  - previous 'action' is the previously chosen word from the possibleFills function
- startState: (0, '-BEGIN-')
- action: a word from the possibleFills function for the current state
  - Note: for each state there is more than one action. That set of actions is the output of possibleFills(query[currentIdx:i]), possibleFills(query[currentIdx:i+1]), ..., possibleFills(query[currentIdx:n]), where n is the length of the query string
- successorState: (currentIdx+1, current action)
- cost: bigramCost(previous action, current action)
- endState: when currentIdx of current state equals n

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.

Your Solution:

Reconstructed correctly:

- Example 1

Reconstructed incorrectly:

- Example 2 became: welcome to the hut to kazan
- Example 3 became: girl will near of leper right now

The outputs of any machine learning model is based on its training. This particular language model is trained on War and Peace and Romeo and Juliet. Incorrect output translations implies that the input phrases don't occur at all or often enough in the training data.

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?).

Your Solution: The system is expected to work when the inputs to the translation model (strings with no vowels or spaces) represents commonly used English words and phrases.

All users are cautioned that there exists the potential for this system to produce incorrect translations even if the outputs are grammatically sensible. Users must check the outputs are satisfactory for their use case before using the model's outputs.

All users must be aware of the two major limitations in this model.

- The translations will bias the phrases found in the corpus. This system is trained on a corpus of two texts: the translated versions of the historic texts, War and Peace (1867) and Romeo and Juliet (1595). This version of War and Peace was originally translated from Russian to English by an anonymous Volunteer, and David Widger in 2009 and last updated in March 2013. This version of Romeo and Juliet was first published in 1997 by Project Gutenberg a 'volunteer effort to digitize cultural works' and last updated in May 2002.
- The system will fail to translate words and phrases invented after 1867 that can't be found in the corpus texts.

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.

Your Solution: I would add more recent texts to represent vocabulary and phrases used today. Wikipedia would be a powerful training set because it would help overcome both limitations of the limited corpus of two historical texts (3.3 MB total). Wikipedia is edited everyday by users on a huge variety of topics which will offer much better coverage than the fictional works of War and Peace and Romeo and Juliet. However, Wikipedia is 20.7Gb so the training would take a significant time. To implement this in practice, I would reduce the training volume of the Wikipedia data set by taking random sample of articles - starting with a 10MB dataset then 100MB then 1GB. I would split the dataset 80 per cent for training and 20 per cent for testing. Once I'm satisfied that the model performs better with the enhanced corpus, I could try fine tune the output of the model in an attempt to create more 'natural sounding sentences' by trying n=3 or n=4 in the n-gram cost model.

Submission

Submission is done on Gradescope.

Written: When submitting the written parts, make sure to select **all** the pages that contain part of your answer for that problem, or else you will not get credit. To double check after submission, you can click on each problem link on the right side and it should show the pages that are selected for that problem.

Programming: After you submit, the autograder will take a few minutes to run. Check back after it runs to make sure that your submission succeeded. If your autograder crashes, you will receive a 0 on the programming part of the assignment. Note: the only file to be submitted to Gradescope is `submission.py`.

More details can be found in the Submission section on the course website.

References

[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.