Reconstruct - Stanford CS 221 Fall 2017-2018

Rohit Apte

2017-10-13

1 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 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.

Lets take a simple cost function (similar to the programming exercise)
Lets assume `vocabulary` is the list of words `[this,is,hard]`

```
def unigramCost(x):
     if x in vocabulary:
          return 1.0
     else:
          return 1000
```

Then, the input string `thisisveryhard` should have the optimal cost of `1000` (output is `thisisveryhard`) but the greedy algorithm would split it as `this`, `is`, `veryhard` = 1002.
Therefore greedy algorithm is suboptimal (I guess greed isn't always good!)

2 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 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, 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.

Lets say we have the following words `rnt,s,p` with the following bigram cost function
`cost(BEGIN,rant)=1`
`cost(BEGIN,rent)=100`

`cost(rant,is)=1000`
`cost(rent,is)=1`
`cost(is,up)=1`
The greedy algorithm would return `rant is up` for cost `1002` while the optimal path is `rent is up` for cost `102` (see figure).

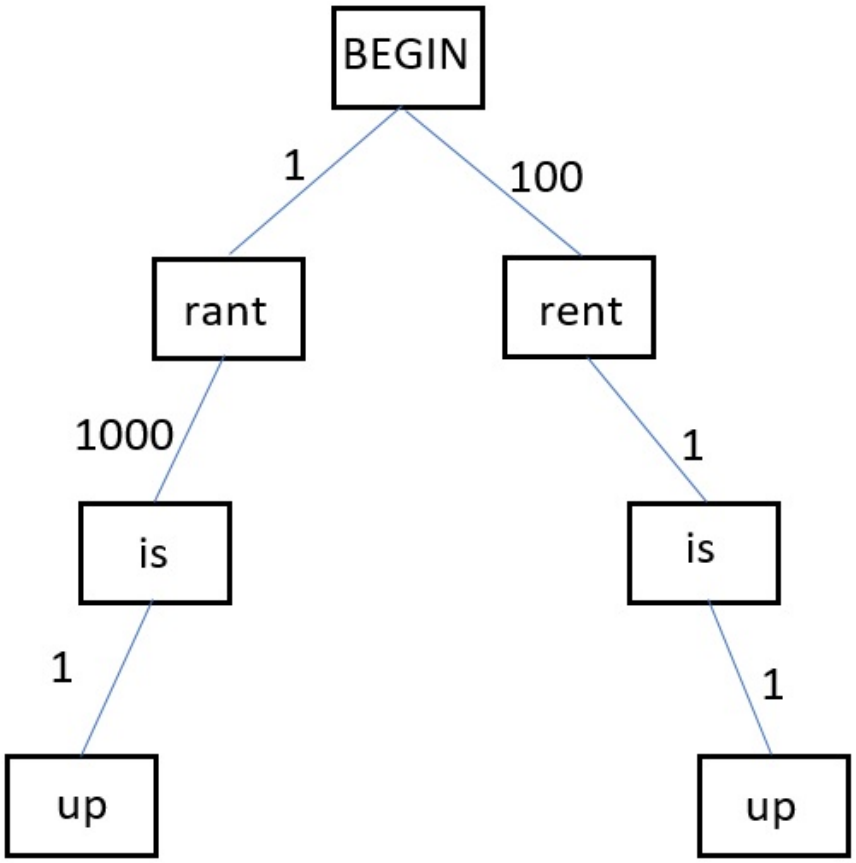

Figure 1: Greed isn't good

3 Problem 3: putting it all 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 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.

The problem involves 2 steps - segmentation and vowel insertion.
A state in our problem is a tuple of `(remainingText,previousWord)`
Our starting state would be `queryWord,SENTENCE-BEGIN`
The end state is when `remainingText` is empty.

The actions are as follows
Apply segmentation to remainingText to generate all possible words
For each word generated above, insert vowels. If a vowel inserted word exists, compute `bigramCost(previousWord,vowelInsertedWord)`
Our new state is `(remainingTextAfterSegmentation,vowelInsertedWord)`
Do this for all vowel inserted words.

b See grader.py

c Let's find a way to speed up joint space and vowel insertion with A*. Recall that having to score an output using a bigram model $b(w\prime, w)$ is more expensive than using a unigram model $u(w)$ because we have to remember the previous word $w\prime$ in the state. Given the bigram model b (a function that takes any $(w\prime, w)$ and returns a number), define a unigram model u_b(a function that takes any w and returns a number) based on b. Now define a heuristic h based on solving a simpler minimum cost path problem with u_b, and prove that $h$is consistent.

To show that h is consistent, construct a relaxed search problem. Recall that a search problem $P\prime$ with cost function $Cost\prime(s, a)$ is a relaxation of a search problem P with cost function $Cost(s, a)$ if they have the same states and actions and $Cost\prime(s, a) \leq Cost(s, a)$ for all states s and actions a. Explicitly define the states, actions, cost, start state, and end test of the relaxation.

Let us define our relaxation P_{rel} of our search problem P to have the cost as the smallest cost of w and any previous word $w\prime$
$u_b(w) = \min_{w\prime} b(w\prime, w)$
For example if we had the following bigram cost function

this	is	0.5
that	is	1.0
who	is	0.75
which	is	1.0
that	was	0.1
which	was	9.0

then $u_b(is) = min((this, is), (that, is), (who, is), (which, is)) = 0.5$
Similarly $u_b(was) = 0.1$

Our states, actions, cost, start state, and end test are the same as $3a$
It is obvious that $Cost_{relaxed}(s, a) \leq Cost(s, a)$ for all states s and actions a.

We then define the relaxed heuristic to be $h(s) = FutureCost_{relaxed}(s)$
Since $h(s)$ is a relaxed heuristic, as per the theorem it is a consistent heuristic.