CS221 Fall 2015 Homework [number]

SUNet ID: [waba]
Name: [Alexander Carlisle]
Collaborators: [Roger Song, Morgan Tsanijivich]

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

Problem 1

(a) This algorithm is sub optimal because it always chooses the best choice without looking ahead. This could be bad if it makes a choice that looks good early, but looking at the rest of the string shows us the first choice was actually very bad.
For example: The correct solution might be
staring into space
On input staringintospace, a greedy algorithm would also put a space at the next most likely spot to create a word and thus might incorrectly find :
star in gin to space.
Here the ngram function could have been a unigram, and for any word the function would return 1/(count of word in corpus) if count was > 0, and it could return 100 if the count was 0. That way words that showed up more often would be given a lower cost. In this case, "star" could have had a cost of .3 and "staring" could have had a cost of .8 resulting in the greedy algorithm choosing "star". Then "in" would have had the next lowest cost and so forth. Furthermore, returning a constant high cost if the possible word was never in the corpus makes sense because for any word not seen we want to give it an equally high cost to keep it from getting dequeued.

(b) Coded in submission.py

Problem 2

(a) Similar to problem one, greedily choosing the next best word is suboptimal because perhaps the word that best fits the previous word will not fit well with the following word. This is best shown by example:
For the sentence:
the lion tamer, the input would be:
th ln tmr.:
For th, the most likely possible word would be "the".
Next it would find all possible words for "ln" possibly returning a list of some of these words:
line, lane, loan, lion ...
Let's say the top results are the following: Bigram(the,line) = .5 and Bigram(the, lion)

= .9
Here the greedy algorithm would choose "line" as the next word.
Now it would find all possible words for tmr + vowels possibly returning:
timer, tumor, tamer ...
Let's say top two results are Bigram(line, timer) = 1.5, and Bigram(line, tamer) = 2.0. Now it would chose timer and the result would be "the line timer". If it had not been greedy, and had explored both "the line"... and "the lion", it could have seen that Bigram(lion,tamer) = .01 and so Bigram(the,lion)+Bigram(lion,tamer) < Bigram(the,line) + Bigram(line,timer). However, it chose the best available option and therefore didn't find the correct solution.

(b) Coded in submission.py

Problem 3

(a) Defining different sections of search problem: State: Previous word, and the remaining letters. You need the previous word to calculate the cost of your next choice and you need to keep track of all remaining letters because to calculate the next states you need to calculate putting a space at any point from the start of the remaining letters to the end. Initial State: (Begin Token, query) because nothing came before the start and you need to know the full query to try putting a space at every point from i=1 to i = $len + 1$(in that case you would just choose the entire word). Goal State: Remaining letters = "". Once all letters have been placed into words and nothing is remaining then we have reached the goal state. Action: Choosing where to insert the next space. Costs:For a given split, there will be a set of possible words. Each of those words and the remaining letters will become a state. The bigram cost of that word and the previous word will become the cost of that new state.

(b) (Coded in submission.py)

(c) For A* I would first do some preprocessing. For each pair of words I would compute the bigram cost. Then I would map each word to its minimum bigram cost(for instance if bigram("the", "dog") = .9 and bigram("good", "dog") = .1, "dog" $->$ .1). Therefore, for any word w, u(w) would be the minimum bigram cost it had for any preceding words. Note that u(w) will never be negative. To prove that this unigram cost is a consistent heuristic for the original problem I must show two things. First, that the hueristic at the goal state is 0. Second that in the original problem Cost'(s,a) = Cost(s,a) + h(succ(s,a)) - h(s) >0. The future cost of the goal state is 0, because the goal state will have the query string be empty. The empty string won't have a bigram cost associated with it so the minimum cost across all (empty string, word in corpus) pairs will be 0. Second I need to show that the new cost with the heuristic is always positive. This can be

more easily represented as Cost(s,a) + h(succ(s,a)) - h(s) > 0. Well the Cost(s,a) is some bigram cost between the actual (w',w). From that we subtract h(s) which is the minimum bigram cost associated between ANY previous word seen in the corpus with that specific w. This means that bigram(w',w) at state s must be >= h(w) = min(allPairsOfSeen previousWord, w). Thus, Cost(s,a) - h(s) >= 0, and if we add h(succ(s,a)) to the left the inequality still holds because h(succ(s,a)) is guaranteed to be positive(as earlier mentioned.) Therefore, we have shown the triangle inequality to hold and we have show that the futureCost(goalState) is 0 and so we know our heuristic is consistent.