Midterm for CS378: Natural Language Processing (Fall 2020)

Instructions:

- The exam is due at **5pm CST on Friday, October 16. Leave yourself ample time to upload it.**
- **The first thing you do should be to read and sign the honor code. You will upload this with your exam.** This exam is to be completed individually be each student. Again, **you may not collaborate with other students**! If we find out that you have done so, that will be considered a violation of the course Academic Honesty policy.
- This exam is an **open book take-home exam**. You are allowed to consult any resources that are helpful, with the exception of other people.
- Partial credit will be given for short-answer and long-answer questions, so please show work in your answers, but avoid writing essays. **You might be penalized for writing too much if it's incorrect.**
- For short-answer and long-answer questions, **please box or circle your final answer** unless it is an explanation.
- You will scan and upload your exam into Gradescope. You may type your responses, handwrite responses on a printed exam, or a mix of both. If you type your responses, please try to keep the layout of your pages matching the exam to make our grading easier (e.g., put the answers to the first four multiple choice questions on a single page).
- If you have questions during the exam, please use a private Piazza post or directly email the course staff. Important clarifications will be posted as Canvas announcements.

Grading Sheet (for instructor use only)

Question	Points	Score
1	10	
2	18	
3	14	
4	14	
5	14	
6	14	
7	17	
Total:	101	

Name: ____

Name: ________________________________

Honor Code (adapted from Dr. Elaine Rich)

The University and the Department are committed to preserving the reputation of your degree. In order to guarantee that every degree means what it says it means, we must enforce a strict policy that guarantees that the work that you turn in is your own and that the grades you receive measure your personal achievements in your classes:

By turning in this exam with your name on it, you are certifying that this is yours and yours alone. You are responsible for complying with this policy in two ways:

1. You must not turn in work that is not yours or work which constitutes any sort of collaborative effort with other students.
2. You must take all reasonable precautions to prevent your work from being stolen. It is important that you do nothing that would enable someone else to turn in work that is not theirs.

The penalty for academic dishonesty will be a course grade of F and a referral of the case to the Dean of Students Office. Further penalties, including suspension or expulsion from the University may be imposed by that office.

Please sign below to indicate that you have read and understood this honor code. If submitting electronically, you can submit any page with a text version of the honor code and a scanned or drawn signature.

Signature: ____

Part 1: Multiple Choice (10 points)

1. (10 points) Answer these questions by writing the answer in the provided blank (2 points each).

_____ C (1) What is the big-O runtime of a single forward pass through a Deep Averaging Network? The size of the hidden layer is D, the number of words is S, the size of the embedding space is E, and the number of classes is K.

A. $O(S + ED + K)$
B. $O(SEDK)$
C. $O(SE + ED + DK)$
D. $O(D + S)$
E. None of the above

_____ I (2) Suppose you remove the nonlinearity from our standard FFNN and replace it with the identity function. **Circle or list all that apply (there may be multiple correct choices):**
I. Any model represented by this neural network can be represented with a multiclass logistic regression model.
II. The model is no longer mathematically well-defined (i.e., your code would crash if you ran it).
III. The model will need a higher learning rate to train effectively.

_____ E (3) Assume you are implementing a simplified DAN with a single weight matrix W = $\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$ and no hidden layer. That is, your model is $P(y|\mathbf{x}) = \text{softmax}(W(\sum_{i=1}^{n} E(x_i)))$, where $E(x_i)$ is the word embedding of the ith word. Suppose $y = 0$ means negative and $y = 1$ means positive.

You have two training sentences: *cats and dogs* with **negative** sentiment and *dogs and dogs and dogs* with **positive** sentiment. Find the values of a and b such that this training set is fit *nearly* perfectly by the DAN model (i.e., it fits the training data with negative log likelihood less than 0.01). Assume the network has a single layer and a softmax layer at the end.

$$cats = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} \quad and = \begin{bmatrix} -0.4 \\ 0.1 \end{bmatrix} \quad dogs = \begin{bmatrix} 0.1 \\ -0.7 \end{bmatrix}$$

A. $[a = -100 , b = 100]$
B. $[a = 100 , b = -100]$
C. $[a = -100 , b = -100]$
D. $[a = 100 , b = 100]$
E. None of the above

_____ III, IV, V (4) Which of the following statements about dependency parses is correct? **Circle or list all that apply.** Furthermore, do not count ROOT as a word.
I. Every word has at least one child
II. Every word has at most one child
III. Every word has at least one parent
IV. Every word has exactly one parent
V. ROOT has exactly one child

______ I, III (5) Consider the skip-gram model

$$P(\text{context} = y|\text{word} = x) = \frac{\exp(\mathbf{v}_x \cdot \mathbf{c}_y)}{\sum_{y'} \exp(\mathbf{v}_x \cdot \mathbf{c}_{y'})}$$

Recall that the choice of which (context, word) pairs to train on is governed by a window size parameter k, which controls how far from a word we look to gather context. **Circle or list all that apply:**

I. If we're building a neural network classifier for POS tagging, a smaller window size parameter ($k < 4$) will work better.
II. If we're building a neural network classifier for POS tagging, a larger window size parameter ($k \geq 4$) will work better.
III. Skip-gram takes longer to train with a larger window size parameter.
IV. Using a larger window size will cause our vocabulary to be larger.
V. A larger window size will usually work better if the dimension of the skip-gram vectors is larger.

Part 2: Short Answer (18 points)

2. (18 points) Answer the following short-answer questions.

a. (4 points) Suppose you are doing authorship attribution with a multi-class bag-of-words classifier: you have a training set of documents written by several authors, and a test set consisting of a few documents whose authorship you're unsure of. Authorship attribution relies on distributions of function words, particularly the relative frequencies of common function words in text.

Given this application, say for each type of preprocessing whether this should **improve** performance, **worsen** performance, or make **no difference**, and write a one-sentence justification of your answer.

(1) lowercasing
Although the intended answer was that lowercasing would help, we ended up accepting any well-justified answer for this. It may help because it helps canonicalize across function words. It may hurt if capitalization is a major part of an author's style, or if it erases information about specific ways of starting a sentence. It likely makes little difference in general.

(2) removing stopwords
Worsen, stylometry relies on these words

(3) applying tf-idf weighting to the words
Worsen, stylometry relies on common words and this will downweight them

(4) applying tokenization

We would typically expect this to improve performance. However, if your justification for another answer was good, we accepted that answer too. You could argue that this is unlikely to make much difference, or that certain word-punctuation pairs may actually be useful features for this task.

b. (3 points) For part-of-speech tagging in Assignment 3, you were given smoothed emission probabilities that incorporated pseudo-counts: assume we saw every word with every POS tag α times for some very small fractional value of α. But smoothing all of the tags' emission distributions might not be the best assumption. If you were to only apply smoothing to some of the tags, which type of tag is more important to smooth: open-class or closed-class? Justify your answer.

Open-class tags are best to smooth. Closed-class tags should have enough observations of the words they occur with to learn the distribution without smoothing, and allocating probability mass to rare words having closed-class tags is not a good assumption.

c. (2 points) Name one of the ways in which the Adam optimizer improves over SGD, and state in your own words why this helps optimization of deep networks.

The per-feature learning rate helps balance across layers. Momentum smooths gradient updates and prevents "zigzagging" during optimization. Other answers were accepted as well.

d. (3 points) List three applications you can think of where you could use the bag-of-words classification model from Assignment 1, assuming you retrain this model on new data. **Do not include sentiment analysis.**

Many great and creative answers! Authorship attribution, spam classification, classifying political party alignment of tweets, subjectivity classification, and others were accepted. Note that this should be restricted to binary classifiers only. However, we gave you the benefit of the doubt if it was unclear, as most multi-class tasks could have binary formulations as well (e.g., authorship attribution where you're trying to decide if text was written by an author or not written by that author).

e. (3 points) Your pet ferret walks across your keyboard while you're working on Assignment 1 and adds a call to `shuffle` the words in each input sentence *before* its features are extracted, but *after* tokenization and other preprocessing. This applies to both training and test time. Recall the three question parts: Q1 (unigram perceptron), Q5 (unigram LR), and Q7 (bigrams). Which of these question parts will have their results impacted by the code change?

Q7 changes, no others do. Shuffling doesn't change the set of unigrams, but it does change bigrams.

f. (3 points) Your ferret is at it again and somehow deleted a random 50% of the words in the Assignment 1 *training* data; the test data is unchanged. Do you expect the model's performance to increase, stay the same, decrease slightly, or decrease significantly? Justify your answer.

Decrease slightly. For one thing, this is what happens if you actually do it (we tried). This is similar to dropout in neural networks, and losing half of each training sentence just isn't that impactful. A word like *bad* will still appear in enough sentences to learn a good weight for it. Another way of thinking about it is that typically in machine learning, removing half of the training examples usually only has a slight negative impact on performance (a few percent).

Part 3: Long Answer (72 points)

3. (14 points) Consider the problem of trying to automatically take words in a tweet and merge them into a hashtag: for example, instead of tweeting *going to SXSW2019*, you might tweet *going to #SXSW2019* to use the event's hashtag. We are going to approach this as a BIO tagging problem.

   Let $\mathbf{x} = (x_1, \ldots, x_n)$ be the sequence of words in the tweet. Let $\mathbf{y} = (y_1, \ldots, y_n)$ be the sequence of BIO labels. Here are some examples of what this looks like:

```
  O   O    O      B       I    O    O   O      O
gonna go watch Infinity War with the crew tonight
 O    O   O    O       B     O    O    O
Just got back from SXSW2019 and loved it
  O      O      O      O      O   O  O   O    O   O     B   I   I   I
After watching that football game I can tell you that Texas is not back
```

   A multi-word hashtag would be merged for the final tweet (e.g., *#InfinityWar*). Also, this task is only about detecting which existing words could be merged into a hashtag, *not* adding new hashtags to a tweet or changing the wording to conform to existing hashtags.

a. (6 points) First consider a simplified form of the task where we are only trying to detect *single-word* hashtags like SXSW2020. Imagine that the data is postprocessed so that *only* the single-B labels remain, and all B-I... sequences are just set to O (e.g., *Infinity War* is no longer detected).

Suppose we try to do this with a binary classifier, and we are running the classifier over the jth word in the input sentence. Write a piece of pseudocode that extracts indicator or count-based features from the input using *three feature templates* (that is, do *not* use skip-gram or neural networks for this question part). For example, the Assignment 1 features could be written as:

```
for i=1 to n:
    add_feature("Unigram=" + x[i])
```

and this would count as one feature template. (Reminder that n is the length of the sentence.)

Give your three feature templates and give a one-sentence justification for why each is useful.

What's needed here are *position-sensitive features*. For example,
`add_feature("CurrWord=" + x[j]),`
`add_feature("PrevWord=" + x[j-1]),`
`add_feature("NextWord=" + x[j+1]),`
`add_feature("CurrWordCapitalized=" + is_cap(x[j])),`
`add_feature("CurrWordContainsDigits=" + contains_digits(x[j])")`, and other such features.

The question was a bit tricky in that for looping over the input is actually the wrong thing to do. You are classifying *just* the jth token. Bag-of-words features as in A1 are not appropriate here and you should not be looping over all words and extracting CurrWord for each one or anything like that. You also should not be referencing BIO tags. You're trying to predict these tags, so using tags in the context is generally not something that's possible (perhaps if you specified that you're running the classifier sequentially in the forward direction or something, this could've been okay). The model in this question part is a classifier, not a CRF.

b. (4 points) Now consider the full version of this task. Do you think a standard HMM (using categorical distributions for transitions and emissions, like in Assignment 3) will work well here? Give **two reasons** to justify your response.

No. Too many Os make transitions unreliable, and rare entities are hard to model with HMM emissions. See the CRF lecture discussion for why HMMs don't work well for tasks like this.

c. (4 points) Suppose you are given a test set of new tweets all from a single day. Your job is to return predictions across all of them. First, you run your model on each tweet in isolation, but you're not that happy with the consistency of your results. Can you think of a way to postprocess your model's output to do something smarter and get better results? Put another way, what information could you use across test data points to do better at this problem, and how would you use it? Justify your answer.

You could find hashtags that are predicted in some places in the test set and then assume that those words should be hashtagged elsewhere in the test set as well. This could be done by looking at posterior distributions or classifier scores, or just in a hard way.

4. (14 points) In this question, we are going to consider doing multiclass logistic regression with a very large number of classes $K > 10000$. Here's the multiclass logistic regression update with "different features" as discussed in lecture:

Algorithm 1 Multiclass logistic regression update

```
1: Input: labeled data $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^D$, a feature extractor $\mathbf{f}$.
2: Initialize $\mathbf{w} = \mathbf{0}$, a $nK$-dimensional weight vector.
3: for $t = 1$ to $T$ do                                                   ▷ Loop over epochs
4:     for $i = 1$ to $D$ do                                               ▷ Loop over data
5:         Define local variables $\mathbf{x} = \mathbf{x}^{(i)}$, $y^* = y^{(i)}$ to denote the current example
6:         Compute $P(y = y_j|\mathbf{x}) = \frac{\exp(\mathbf{w}^\top \mathbf{f}(\mathbf{x}, y_j))}{\sum_{y' \in \mathcal{Y}} \exp(\mathbf{w}^\top \mathbf{f}(\mathbf{x}, y'))}$ for all $j$ in $1 \ldots K$
7:         for $j = 1$ to $K$ do                                           ▷ Loop over classes
8:             $\mathbf{w} \leftarrow \mathbf{w} - P(y = y_j|\mathbf{x})\mathbf{f}(\mathbf{x}, y_j)$
9:             if $y^* = y_j$ then
10:                $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(\mathbf{x}, y_j)$
11:            end if
12:        end for
13:    end for
14: end for
```

a. (3 points) Suppose that your "base" feature vector (before conjoining with classes) consists of n features, but at most d of these base features are nonzero on any give example. What is the big-O runtime of one iteration of the update (i.e., one pass through the i loop), assuming an efficient implementation?

$O(dK)$. Weights times features is $O(d)$ for each class, so you can compute all scores in $O(dK)$ time. Updates take K passes through with $O(d)$ for each pass.

b. (7 points) Let's try to make this update faster. There are two parts that are slow: computing the probabilities for all the classes and then applying the update to all the classes. One way to make this faster is by approximating the denominator and making fewer updates to the weight vector.

Suppose you've completed three full passes through the data (three epochs) with the "full" update, and then on the fourth pass you know that the weight vector isn't changing that dramatically anymore. Describe an approximation to the update you can do on this pass. Feel free to reference the algorithm by line numbers if you'd like. You do not need to provide working code, but you should be clear and specific enough that someone else could implement it given your description.

Hint: record information about the first three passes in some kind of data structure, and try to avoid making very small updates.

For each example, you should keep a list containing the gold class and the top predicted classes (by score) over each of the first three epochs (say, 10 values per epoch). Then, for the fourth epoch, you should only compute the distribution in line 6 over these $\leq$ 30 values and only make the update for these values. This involves both approximating the denominator by a sum over these values alone as well as approximating the gradient by ignoring most of the other classes.

This is related to the idea of negative sampling in skip-gram models (discussed in the video) and to approximating the normalizing constant in any log-linear models. Such techniques are broadly useful when dealing with settings like language modeling or skip-gram when the number of output classes is high (e.g., the number of words in a vocabulary).

c. (4 points) For multiclass perceptron, the algorithm is faster since you don't need to compute a normalization over K classes; however, you still need to compute y_{pred}, which requires computing scores for each class. Describe how you might modify multiclass perceptron in the same setting as part (b) (speeding up the fourth iteration).

This follows part (b): compute y_{pred} over the cached set of top predicted classes. In this case, y_{pred} is approximate but the gradient update is "exact" for this choice of y_{pred}.

5. (14 points) Consider the beam search algorithm as described in lecture. The algorithm we described

maintained a beam of k hypotheses at each timestep, with others being disregarded as possible predecessors for the next state in search. However, another way to implement beam search is to have the beam instead store **all the hypotheses which are no more than α worse than the optimal in terms of score**. Put another way, if the current best hypothesis in the beam has score $s_{\max}$, then any hypothesis with score at least $s_{\max} - \alpha$ will get added to the beam, even if this means *every* hypothesis gets added. When a hypothesis with score $s' > s_{\max}$ gets added to the beam, hypotheses that are now too low-ranking must be evicted.

a. (7 points) Assume you have the HMM parameters E and T as described in the lecture notes (you won't need the start parameters S). Using pseudocode, write the loop of the algorithm to take a previous beam b_p for timestep $i-1$ in the sentence and *efficiently* produce the new beam b_n. for timestep i. You may use a beam data structure (call it Beam) implemented with a min-heap: insertion is logarithmic in the number of elements, changing a value for a key is logarithmic, and removal of the "worst" element is constant time.

It's actually *not* most efficient to use the min-heap data structure. If there are $|T|$ tags, during inference, you might end up with up to $O(|T|)$ elements in it if you happen to add the best one at the end: basically consider the case when every tag other than the max has the same score, and the max tag gets checked at the end. At that point, the log term causes you incur $O(|T|\log(|T|))$ cost for using it. This, however, was not our original intent in the question, but something we noticed later.

```
initialize b_n as a list
for each tag_curr in the tag set:
  track tag_prev_best with tag_prev_best_score
  for each tag_prev in b_p:
    compute score = transition(tag_prev, tag_curr) + emission(tag_curr)
    update tag_prev_best = tag_prev if score > tag_prev_best_score
  add tag_curr to b_n with score tag_prev_best_score
compute s_max in b_n
drop all elements in b_n less than s_max - alpha
```

This beam search variant does occasionally show up in practice, though the fact that it might take longer to run is usually a bit of a disadvantage.

b. (5 points) Suppose over the n steps of inference each beam ends up being size B_i for $i = 1$ to n. Let B_{max} be the largest of these. What is the worst-case runtime in big-O notation in terms of the sentence length n, the number of tags $|T|$, and these beam sizes? Note that the worst case ranges over both possible HMMs (maximally confusing/tricky ones) as well as orders of visiting the states.

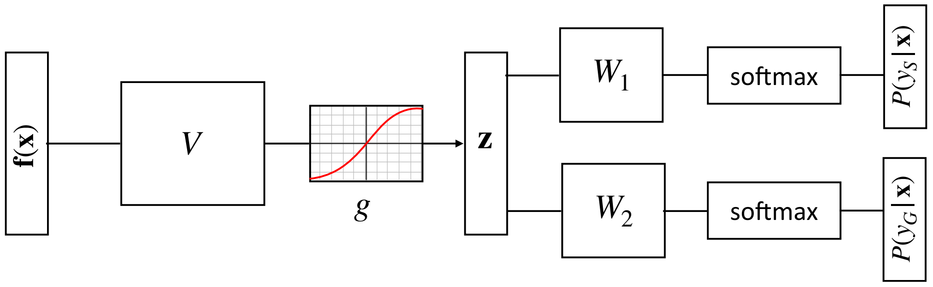
Write a brief (2-3 sentence) justification of your response, referencing your algorithm from part (a) if you'd like.

$O(n|T|B_{\text{max}})$ following the algorithm above. Again, note that you can't really save time using the min heap here. Note that if you did use the min heap, you should typically have an extract $O(\log |T|)$ term to correctly match your part (a).

c. (2 points) Suppose you have a sentence with mostly unambiguous words, where the emission probability of the word given one tag is substantially higher than those from all other tags (by at least α in terms of log probability). 10% of the words are highly ambiguous and these are spread uniformly at random throughout the sentence. Which do you think is more efficient: normal beam search or this modified version? Justify your answer.

Modified beam search. In this case, you will save time at most positions by not maintaining several low probability states; normal beam search will do this and maintain a fixed number of states everywhere. For settings of the beam cutoff designed to maintain equivalent levels of uncertainty at the ambiguous points, you'll come out ahead of standard beam search.

6. (14 points) Suppose you have a dataset containing movie reviews. Each movie review is annotated with two different labels, y_S and y_G, one corresponding to the sentiment (+ or -) and the other to the genre (action, comedy, or romance) of the movie. Consider the following neural network, which models both tasks at the same time (a multi-task model). The two classification tasks share the parameters at the lower layers of the network and have independent classifiers at the last layer. The network is then trained **jointly** to minimize the negative log likelihood of **both** tasks (these terms are added together). Such networks are useful when you have two related tasks and shared representations **z** might be helpful.



a. (7 points) Given below is the `nn.Module` initialization and forward function of a single-task network. Modify this appropriately for the multi-task case. Your forward function should return a tuple with log probabilities from task S and task G.

```
class FFNN(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
    self.V = nn.Linear(input_size, hidden_size)
    self.nonlin = nn.Tanh()
    self.W = nn.Linear(hidden_size, output_size)
    self.log_sm = nn.LogSoftmax(dim=0)

  # Assume f_x is the averaged embeddings.
  def forward(self, f_x, y_gold):
    return self.log_sm(self.W(self.nonlin(self.V(x))))
```

Need two output size parameters for the different classification tasks

self.W1 = nn.Linear(hidden_size, output_size1)
self.W2 = nn.Linear(hidden_size, output_size2)
self.log_sm = nn.LogSoftmax(dim=0)

forward:
intermediate = self.nonlin(self.V(x))
return (self.log_sm(self.W1(intermediate)), self.log_sm(self.W2(intermediate)))

b. (3 points) Suppose you take these logits and use them to compute `lossS`, the loss for the sentiment analysis task, and `lossG` for the genre classification task. What parameters does the following code update? List all parameters that get updated.

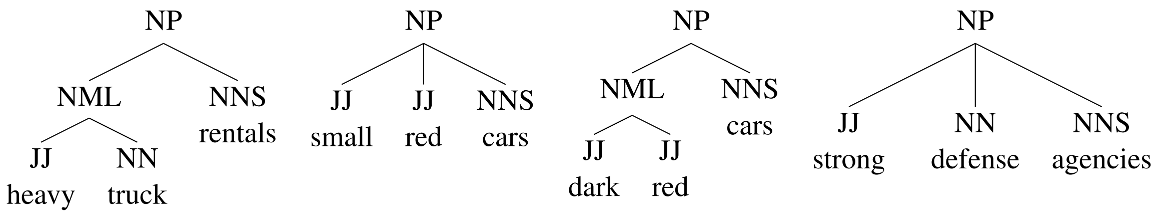
```
lossS.backward()
optimizer.step()
```

W1, V

c. (4 points) You are given the model trained using the multi-task objective, but **do not have access to the training data anymore**. Suppose you get 1000 **additional** reviews, but annotated only with the sentiment (+/-). What is the best strategy for using this data to further improve your sentiment classification without harming the performance on the genre prediction task?

You should freeze the rest of the network and only update W1. Another solution that got nearly full credit was to make predictions with the genre classifier, then use these as labels on this dataset, thereby encouraging the model to keep the same behavior as the initial trained model. However, this can still do weird things unless you use the posterior distribution as soft labels, and if you're training on examples drawn from a new distribution, it could still change the behavior of the underlying classifier on the original training examples.

7. (17 points) Suppose you have the following NPs provided to your as your treebank:



You will construct a PCFG from these trees with NP as the start symbol in this grammar.

To binarize the trees, you will introduce a $\overline{\text{NP}}$ symbol to turn the ternary rules into binary rules (you can write NP-B for "NP bar" if you're typing your answer). This will be described in the following questions.

For all question parts, assume that Greg comes along with a great part-of-speech tagger and tags these sentences. So: (a) **do not write down any grammar rules in the lexicon; only include the rules above the part-of-speech tag layer**; (b) **when doing CKY, assume the gold tag has a score of 0.0 and all other tags have a score of** $-\infty$. In this case, each word's tag is unambiguous anyway, so this does not actually change your answer, just reduces the amount of writing.

a. (5 points) Write the grammar (**rules and probabilities**) you get if you use right-binarization: that is, $\overline{\text{NP}}$ is introduced as a right child under the NP symbol for ternary-branching rules. (So NP $\rightarrow$ JJ $\overline{\text{NP}}$ is introduced for the *small red cars* example).

NP $\rightarrow$ NML NNS 0.5
NP $\rightarrow$ JJ NP-B 0.5
NML $\rightarrow$ JJ NN 0.5
NML $\rightarrow$ JJ JJ 0.5
NP-B $\rightarrow$ JJ NNS 0.5
NP-B $\rightarrow$ NN NNS 0.5

b. (4 points) Now parse the phrase *large red cars* with the tags JJ JJ NNS. Show your work and write out **every parse with nonzero probability** and its probability.

There are two parses:
1/4 for the parse with NML: (NP (NML (JJ large) (JJ red)) (NNS cars))
1/4 for the NP-B parse: (NP (JJ large) (NP-B (JJ red) (NNS cars))

c. (4 points) Write the grammar you get if you use right-binarization on *strong defense agencies* but left-binarization on *small red cars*. That is, for that one tree, make the $\overline{\text{NP}}$ the left child of the root NP.

NP $\rightarrow$ NML NNS 0.5
NP $\rightarrow$ NP-B NNS 0.25
NP $\rightarrow$ JJ NP-B 0.25
NP-B $\rightarrow$ JJ JJ 0.5
NP-B $\rightarrow$ NN NNS 0.5
NML $\rightarrow$ JJ NN 0.5
NML $\rightarrow$ JJ JJ 0.5

d. (4 points) Now parse *large red cars* with tags JJ JJ NNS using this new grammar. Show your work, write out **every parse with nonzero probability** and its probability.

1/4 for the parse with NML: (NP (NML (JJ large) (JJ red)) (NNS cars))
1/8 for the parse with NP-B, which is now left-branching: (NP (NP-B (JJ large) (JJ red)) (NNS cars))