CS 188 Spring 2021 — Introduction to Artificial Intelligence — Written HW 4

Due: Wednesday 4/28/2021 at 10:59pm (submit via Gradescope).

Policy: Can be solved in groups (acknowledge collaborators) but must be written up individually

Submission: It is recommended that your submission be a PDF that matches this template. You may also fill out this template digitally (e.g. using a tablet). **However, if you do not use this template, you will still need to write down the below four fields on the first page of your submission.**

First name	
Last name	
SID	
Collaborators	

For staff use only:

Q1.	Probabilistic Language Modeling	/35
Q2.	Machine Learning	/30
Q3.	MDPs and RL	/20
Q4.	Text Generation	/15
	Total	/100

Q1. [35 pts] Probabilistic Language Modeling

In lecture, you saw an example of supervised learning where we used Naive Bayes for a binary classification problem: to predict whether an email was ham or spam. To do so, we needed a labeled (i.e., ham or spam) dataset of emails. To avoid this requirement for labeled datasets, let's instead explore the area of unsupervised learning, where we don't need a labeled dataset. In this problem, let's consider the setting of language modeling.

Language modeling is a field of Natural Language Processing (NLP) that tries to model the probability of the next word, given the previous words. Here, instead of predicting a binary label of "yes" or "no," we instead need to predict a multiclass label, where the label is the word (from all possible words of the vocabulary) that is the correct word for the blank that we want to fill in.

One possible way to model this problem is with Naive Bayes. Recall that in Naive Bayes, the features $X_1, ..., X_m$ are assumed to be pairwise independent when given the label Y. For this problem, let Y be the word we are trying to predict, and our features be X_i for $i = -n, ..., -1, 1, ..., n$, where X_i is the word i places from Y. For example, in the sequence `Neural networks ____ a lot`, $X_{-2} =$ `Neural`, $X_{-1} =$ `networks`, $Y =$ the blank word (our label), $X_1 =$ `a`, and $X_2 =$ `lot`.

(a) First, let's examine the problem of language modeling with Naive Bayes.

(i) [1 pt] Draw the Bayes net structure for the Naive Bayes formulation of modeling the middle word of a sequence given two preceding words and two succeeding words. You may think of the example sequence listed above:
`Neural networks ____ a lot.`

(ii) [1 pt] Write the joint probability $P(X_{-2}, X_{-1}, Y, X_1, X_2)$ in terms of the relevant conditional probability tables (CPTs) from the Bayes net.

(iii) [1 pt] What is the size of the largest CPT involved in calculating the joint probability? Assume a vocabulary size of V, so each variable can take on one of V possible values.

(iv) [1 pt] Write an expression for the most probable value $y*$ of Y given observed values x_{-2}, x_{-1}, x_1, x_2, in terms of CPTs from the Bayes net. (Hint: Your answer should involve some kind of $\arg\max$.)

(v) [2 pts] Describe the main issue of Naive Bayes assumptions for language modeling.

(vi) [1 pt] Provide an example 5 word sequence (where the Naive Bayes model tries to predict the third word) where the posterior probability of that third word $P(Y|X_i)$ would likely be very erroneous due to the problem you described with Naive Bayes assumptions in the previous subpart. Justify why this sequence would be problematic to Naive Bayes.

Now, let's change our setting a bit. Instead of trying to fill in a blank given surrounding words, we are now only given the preceding words. Say that we have a sequence of words: $X_1, ..., X_{m-1}, X_m$. We know $X_{1:m-1}$ but we don't know X_m.

(b) For this part, assume that every word is conditioned on all previous words. We will call this the **Sequence Model**.

(i) [1 pt] Draw the Bayes net (of only X_1, X_2, X_3, X_4, X_5) for a 5-word sequence, where we want to predict the fifth word in a sequence X_5 given the previous four words X_1, X_2, X_3, X_4. Again, we are assuming here that each word depends on all previous words.

(ii) [1 pt] Write an expression for the joint distribution of a general sequence of length m: $P(X_1, ..., X_m)$.

(iii) [1 pt] What is the size of the largest CPT involved in calculating the joint probability? Assume a vocabulary size of V, so each variable can take on one of possible V values.

(c) The CPT size in this model grows without bound (and very fast) as m increases, which shows how infeasible the sequence model is. Instead of the model above, let's now examine another modeling option: N-grams. In N-gram language modeling, we add back some conditional independence assumptions to bound the size of the CPTs that we consider. Instead of taking into account all previous words we use instead the previous $N-1$ words. This creates the assumption that, given the previous $N-1$ words, the current word is conditionally independent of any word before the previous $N-1$ words. For example, for $N=3$, if we are trying to predict the 100th word, then given the previous $N-1=2$ words (98th and 99th words), then the 100th word is independent of words $1,\ldots,97$ of the sequence.

(i) [1 pt] Making these additional conditional independence assumption changes our Bayes net. Redraw the Bayes net from part (b)(i) to represent this new N-gram modeling of our 5-word sequence: X_1, X_2, X_3, X_4, X_5. Use $N=3$.

(ii) [2 pts] Write an expression for the N-gram representation of the joint distribution of a general sequence of length m: $P(X_1,...,X_m)$. Please use set notation (for example: For tokens $X_i,...,X_j$, please write something of the form $X_{i:j}$). Your answer should express the joint distribution $P(X_{1:m})$, in terms of m and N.

Hint: If you find it helpful, try it for the 5 word graph above first before going to a general m length sequence.

(iii) [1 pt] What is the size of the largest CPT involved in calculating the joint probability above? Again, assume a vocabulary size of V, and $m > N$.

(iv) [2 pts] Describe one disadvantage of using N-gram over Naive Bayes.

(v) [4 pts] Describe an advantage and disadvantage of using N-gram over the Sequence Model above.

(d) In this question, we see a real-world application of smoothing in the context of language modeling.

Say we have the following training corpus from Ted Geisel:

`i am sam . sam i am . i do not like green eggs and ham .`

Consider the counts given in the tables below, as calculated from the sentence above.

1-gram	
Token	Count
i	3
am	2
sam	2
.	3
do	1
not	1
like	1
green	1
eggs	1
and	1
ham	1
TOTAL	17

2-gram	phrases	starting with i
Token1	Token2	Count
i	am	2
i	do	1
TOTAL		3

2-gram	phrases	starting with am
Token1	Token2	Count
am	sam	1
am	.	1
TOTAL		2

(i) [1 pt] Based on the above dataset and counts, what is the N-gram estimate for $N = 1$, for the sequence of 3 tokens `i am ham`? In other words, what is $P(i, am, ham)$ for $N = 1$?

(ii) [1 pt] Based on the above dataset and counts, what is the N-gram estimate for $N = 2$, for the sequence of 3 tokens `i am ham`? In other words, what is $P(i, am, ham)$ for $N = 2$?

(iii) [1 pt] What is the importance of smoothing in the context of language modeling?
Hint: see your answer for the previous subquestion.

(iv) [2 pts] Perform Laplace k-smoothing on the above problem and re-compute $P(i, am, ham)$ with the smoothed distribution, for $N = 2$. In order to calculate this, complete the pseudocount column for each entry in the probability tables. Note we add a new `<unk>` entry, which represents any token not in the table.

Hint: the count for the new `<unk>` row in each table would be 0.

1-gram		
Token	Count	Pseudocount
i	3	
am	2	
sam	2	
.	3	
do	1	
not	1	
like	1	
green	1	
eggs	1	
and	1	
ham	1	
<unk>	0	
TOTAL	17	

2-gram	phrases	starting	with "i"
Token1	Token2	Count	Pseudocount
i	am	2	
i	do	1	
i	<unk>	0	
TOTAL		3	

2-gram	phrases	starting	with "am"
Token1	Token2	Count	Pseudocount
am	sam	1	
am	.	1	
am	<unk>	0	
TOTAL		2	

(v) [4 pts] What is a potential problem with Laplace smoothing? Propose a solution. (Assume that you have chosen the best k, so finding the best k is not a problem.)
Hint: Consider the effect of smoothing on a small CPT.

(vi) [2 pts] Let the likelihood $\mathcal{L}(k) = P(i, am, sam)$, give an expression for the log likelihood $\ln \mathcal{L}(k)$ of this sequence after k-smoothing. Continue to assume $N = 2$.

(vii) [4 pts] Describe a procedure we could do to find a reasonable value of k. No mathematical computations needed.
Hint: you might want to maximize the log likelihood $\ln \mathcal{L}(k)$ on something.

Q2. [30 pts] Machine Learning

In this question, we will attempt to develop more intuition about how neural networks work. In parts (a) and (b), we will discuss gradient descent, and in part (c) we look at backprop.

(a) Gradient descent is a procedure that allows you to minimize any loss function. As an example, let's consider a simple function $Loss(w) = w^2$ and let's assume that we want to minimize this function. Perform gradient descent on this loss function by using the update rule $w \leftarrow w - \alpha\frac{dLoss}{dw}$, where α is the learning rate.

(i) [1 pt] What is $\frac{dLoss}{dw}$? Write your answer in terms of w.

(ii) [1 pt] What is the optimal w that minimizes this loss function? We denote this value of w as w^*.

(iii) [2 pts] Carry out one iteration of gradient descent (i.e., weight update). What are the resulting weight and corresponding post-update loss for the scenarios below? Plot the loss function (w^2) by hand and, for each of the two scenarios below, draw the direction in which w is updated (an arrow on the w axis from w_{old} to w_{new}).

1. $\alpha = 0.1$, $w = 2$

2. $\alpha = 1$, $w = -2$

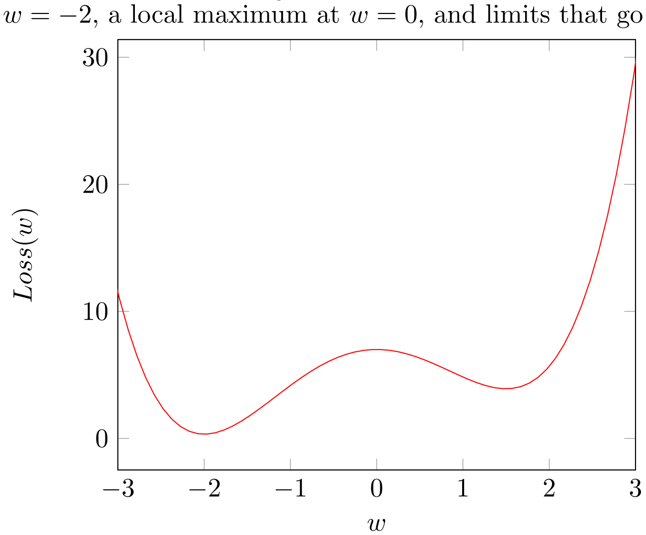
(iv) [4 pts] Assume w is initialized to some nonzero value. Assume we are still working with $Loss(w) = w^2$

1. For which value(s) of α does gradient descent cause w converge to w^* in the least amount of steps?

2. For what range of α does w never converge?
   Hint: for what values of γ does the iteration $w_t = \gamma w_{t-1}$ fail to converge?

(v) [2 pts] Why must α always be positive when performing gradient descent?

(b) It is unlikely that we have a loss function as nice as w^2. Say we instead want to minimize some more complex loss $Loss(w) = \frac{w^4}{2} + \frac{w^3}{3} - 3w^2 + 7$, a polynomial with local minima at $w = -2, 1.5$, a global minimum at $w = -2$, a local maximum at $w = 0$, and limits that go to infinity for both $w \to \infty$ and $w \to -\infty$.



(i) [1 pt] Why do neural networks use gradient descent for optimization instead of just taking the derivative and setting it equal to 0? Explain in 1-2 sentences. You may use the example error function from above to explain your reasoning.

(ii) [1 pt] What is the optimal w^*, given the loss above?

(iii) [2 pts] Let α and w take on the values below. For each case, perform some update steps and report whether or not gradient descent will converge to the optimum w^* after an infinite number of steps. If not, report whether it converges to some other value, or does not converge to any value.

1. $\alpha = 1$, $w = 0$

2. $\alpha = 1$, $w = -2$

3. $\alpha = 1$, $w = 1$

4. $\alpha = 0.1, w = 3$

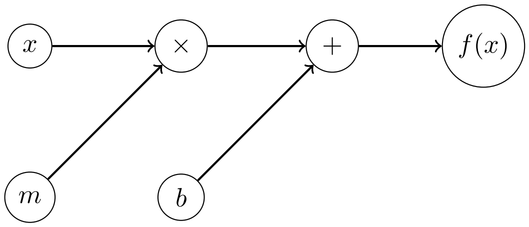
5. $\alpha = 0.1, w = 2$

6. $\alpha = 0.1, w = -10$

(iv) [1 pt] From the subquestion above, explain in 1-2 sentences the effect of learning rate being (a) too high and (b) too low.

(c) Let's now look at some basic neural networks drawn as computation graphs.

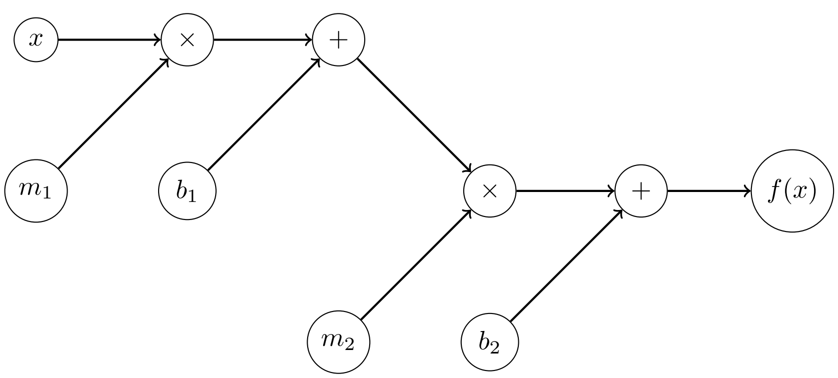
(i) [1 pt] Consider the following computation graph, which represents a 1-layer neural network.



1. Write the equation for the network's output ($y = f(x)$) in terms of m, x, b.

2. Describe the types of functions that can be encoded by such a network, given that the parameters that it can vary are m and b.

(ii) [1 pt] Let's stack two of the above graphs together to represent a very simple 2-layer neural network.

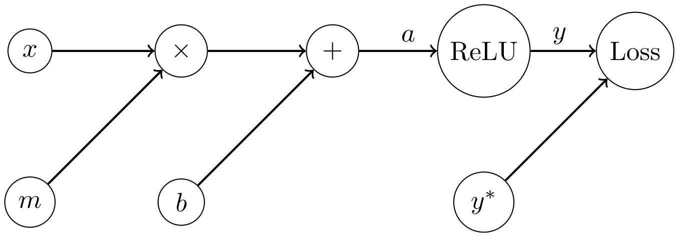


1. Write the equation for $y = f(x)$ in terms of m_1, m_2, b_1, b_2, x.

2. Describe the types of functions that can be encoded by such a network, given that the parameters that it can vary are m_1, m_2, b_1, b_2). Compare this with the previous neural network's expressive power.

3. Is this actually a 2-layer network? If it is, explain in 1-2 sentences. If not, rewrite it (algebraically) as a 1-layer network with only 2 learnable weights. Why do neural networks need nonlinear nodes in the computation graph?

(iii) [2 pts] Now, let's go back to the first NN and add a nonlinear node. Recall $ReLU(x) = \max(0, x)$. Also consider a loss function $Loss(y, y^*)$ which represents the error between our network's output (y) and the true label (y^*) from the dataset. We will perform an abbreviated version of backpropagation on this network.



1. Compute $\frac{\partial Loss}{\partial a}$ using the chain rule. Use the mean squared error (MSE) as the loss function, which is defined as $MSE(y, y^*) = (y - y^*)^2$ where y^* = true label and y is the predicted output from the neural network.

2. Find $\frac{\partial Loss}{\partial b}$. Note that since we are doing backprop, we can reuse calculations from part 1.

3. Find $\frac{\partial Loss}{\partial m}$. Note that since we are doing backprop, we can reuse calculations from part 1.

4. What is the gradient descent update rule for updating m? What is the update rule for b?

For the next few parts, we analyze the perceptron algorithm. In the perceptron algorithm, we predict $+1$ if $\vec{w}^T\vec{f}(x) \geq 0$, and predict -1 else, where $\vec{f}(x)$ is a feature vector.

(d) [1 pt] When implementing the perceptron algorithm with a neural network, the following function might be of use: $sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$. If we added this $sign(x)$ node to our neural network drawings, what would happen during backpropagation through this node?
Hint: what does the gradient look like for various x values?

(e) [2 pts] Draw the binary perceptron prediction function as a "neural-network"-styled computation graph. Assume 3 dimensional weight and feature vectors: that is, $[w_0, w_1, w_2]$ is the weight vector and $[f_0(x), f_1(x), f_2(x)]$ is the feature vector, with x being an arbitrary and potentially high-dimensional object. Recall that in the perceptron algorithm, we take the dot product of the weight vector and the feature vector. In addition to the addition and multiplication nodes, add a loss node at the end, to represent the prediction error that we would like to minimize. Label the edge which represents the perceptron model's output as y.
Hint: $y = sign(w_0 * f_0(x) + w_1 * f_1(x) + w_2 * f_2(x))$

(f) [2 pts] Using the MSE $(y - y^*)^2$ as the loss function, compute $\frac{\partial Loss}{\partial w_i}$. Because of the problem you noticed in the previous part with including the $sign$ node, as we are doing chain rule below, use the custom gradient $\frac{\partial sign(x)}{\partial x} = \left[\frac{\partial sign(x)}{\partial x}\right]_{custom} = 1$.

(g) In this part, we will derive the gradient update rule for the perceptron using our graph above.

(i) [1 pt] The loss gradient is defined as $\nabla_w Loss = \begin{bmatrix} \frac{\partial Loss}{\partial w_0} \\ \frac{\partial Loss}{\partial w_1} \\ \frac{\partial Loss}{\partial w_2} \end{bmatrix}$. Using your answer from the previous question, write out the loss gradient.

(ii) [4 pts] What is the gradient update rule ($\vec{w} \leftarrow \vec{w} - \alpha \nabla_w Loss$) for the cases below?
Hint: your answers will be in terms of $\vec{f}(x)$ and α.

1. $y = -1, y^* = 1$

2. $y = 1, y^* = -1$

3. $y = y^*$

(iii) [1 pt] For $\alpha = \frac{1}{4}$, compare the update rules you derived for the 3 cases above with the perceptron update formula in the notes and lecture. Briefly describe your observations.

Q3. [20 pts] MDPs and RL

The agent is in a 2×4 gridworld as shown in the figure. We start from square 1 and finish in square 8. When square 8 is reached, we receive a reward of $+10$ at the game end. For anything else, we receive a constant reward of -1 (you can think of this as a time penalty).

1	2	3	4
5	6	7	8

The actions in this MDP include: up, down, left and right. The agent cannot take actions that take them off the board. In the table below, we provide initial non-zero estimates of Q values (Q values for invalid actions are left as blanks):

Table 1

	action=up	action=down	action=left	action=right
state=1		Q(1, down)=4		Q(1, right)=3
state=2		Q(2, down)=6	Q(2, left)=4	Q(2, right)=5
state=3		Q(3, down)=8	Q(3, left)=5	Q(3, right)=7
state=4		Q(4, down)=9	Q(4, left)=6	
state=5	Q(5, up)=5			Q(5, right)=6
state=6	Q(6, up)=4		Q(6, left)=5	Q(6, right)=7
state=7	Q(7, up)=6		Q(7, left)=6	Q(7, right)=8

(a) Your friend Adam guesses that the actions in this MDP are fully deterministic (e.g. taking down from 2 will land you in 6 with probability 1 and everywhere else with probability 0). Since we have full knowledge of T and R, we can thus use the Bellman equation to improve (i.e., further update) the initial Q estimates.

Adam tells you to use the following update rule for Q values, where he assumes that your policy is greedy and thus does $\max_a Q(s, a)$. The update rule he prescribes is as follows:

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

(i) [1 pt] Perform one update of $Q(3, \text{left})$ using the equation above, where $\gamma = 0.9$. You may break ties in any way.

(ii) [1 pt] Perform one update of $Q(3, \text{down})$ using the equation above, where $\gamma = 0.9$.

(iii) [1 pt] For the Q update rule prescribed above, how is it different from the Q learning update that we saw in lecture, which is $Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha * \text{sample}$?

(b) After observing the agent for a while, Adam realized that his assumption of T being deterministic is wrong in one specific way: **when the agent tries to legally move down, it occasionally ends up moving left instead** (except from grid 1 where moving left results in out-of-bound). All other movements are still deterministic.

Suppose we have run the Q updates outlined in the equation above until convergence, to get $Q^*_{wrong}(s,a)$ under the original assumption of the wrong (deterministic) T. Suppose $Q^*_{correct}(s,a)$ denotes the Q values under the new correct T. Note that you don't explicitly know the exact probabilities associated with this new T, but you know that it qualitatively differs in the way described above. As prompted below, list the set of (s,a) pairs where $Q^*_{wrong}(s,a)$ is either an over-estimate or under-estimate of $Q^*_{correct}(s,a)$.

(i) [2 pts] List of (s,a) where $Q^*_{wrong}(s,a)$ is an over-estimate. Explain why.

(ii) [2 pts] List of (s,a) where $Q^*_{wrong}(s,a)$ is an under-estimate (and why):

(c) [2 pts] Suppose that we have a mysterious oracle that can give us either all the correct Q-values $Q(s, a)$ or all the correct state values $V(s)$. Which one do you prefer to be given if you want to use it to find the optimal policy, and why?

(d) [2 pts] Suppose that you perform actions in this grid and observe the following episode: 3, right, 4, down, 8 (terminal).

With learning rate $\alpha = 0.2$, discount $\gamma = 0.9$, perform an update of $Q(3, right)$ and $Q(4, down)$. Note that here, we update Q values based on the sampled actions as in TD learning, rather than the greedy actions.

(e) [2 pts] One way to encourage an agent to perform more exploration in the world is known as the "ϵ-greedy" algorithm. For any given policy $\pi(s)$, this algorithm says to take the original action $a = \pi(s)$ with probability $(1 - \epsilon)$, and to take a random action (drawn from a uniform distribution over all legal actions) with probability ϵ. If ϵ can be tuned, would you assign it to be a high or low value at the beginning of training? What about at the end of the training? Please answer both questions and justify your choices.

(f) Instead of using the "ϵ-greedy" algorithm, we will now do some interesting exploration with softmax. We first introduce a new type of policy: A stochastic policy $\pi(a|s)$ represents the probability of action a being prescribed, conditioned on the current state. In other words, the policy is a now a distribution over possible actions, rather than a function that outputs a deterministic action.

Let's define a new policy as follows:

$$\pi(a|s) = \frac{e^{Q(s,a)}}{\sum_{a'} e^{Q(s,a')}}$$

(i) [2 pts] Suppose we are at square 3 in the grid and we want to use the originally provided Q values from the table. What is the probability that this policy will tell us to go right? What is the probability that this policy will tell us to go left? Note that the sum over actions prescribed above refers to a sum over legal actions.

(ii) [2 pts] How is this exploration strategy qualitatively different from "ϵ-greedy"?

(g) Your friend Cody argues that we could still explicitly calculate Q updates (like Adam's approach in part (a)) even if we don't know the true underlying transition function $T(s, a, s')$, because he believes that our T can be roughly approximated from samples.

(i) [2 pts] Suppose you collect 1,000 transitions from $s = 3, a = Down$, in the form of (s_{start}, a, s_{end}). Describe how you can use these samples to compute $T_{approx}(s = 3, a = Down, s')$, which is an approximation of the true underlying (unknown) $T(s, a, s')$.

(s =3, a = Down, s'= 6)	(s = 3, a= Down, s'=7)
99	901

(ii) [1 pt] Now perform one step of q-value iteration based on your transition model computed above.

Q4. [15 pts] Text Generation

Now we will implement some of the ideas from Q1 and Q2 in code (in a Google Colab Notebook, link posted on piazza) to perform language modeling, and then use the language model to generate some novel text!

(a) [10 pts] You will implement some of the math you computed earlier to complete the following functions in the provided N-gram class. Note that if you follow our hints, this should not require more than 15 total lines of code for all the functions below.

First, follow the instructions in the instruction PDF to set up your Google Colab Notebook.

You will need to implement the following functions:

- `count_words`
  1. This function returns a dictionary with the count of each word in `self.text_list`.
  2. HINT: You can do this in one line by using `collections.Counter`.
- `calc_word_probs`
  1. This function converts a dictionary of counts from `count_words` and normalizes the counts into probabilities.
  2. HINT: You can do this in 1-2 lines by using `self.normalize_dict(...)`
- `probs_to_neg_log_probs`
  1. This function converts an inputted dictionary of probabilities `probs_dict` into a dictionary of negative log probabilities.
  2. HINT: Use `np.log`.
- `filter_adj_counter`
  1. This function is a little more complicated. Given a length $N-1$ tuple of tokens and their associated counts (frequencies), this function searches through all the length N phrases it has stored in `self.adj_counter` (or is passed in via the argument `adj_counter`) and returns a dictionary with only the length N phrases with the same first $N-1$ words as `word_tuple`, plus their associated counts (frequencies). See the docstring for a concrete example.
  2. HINT1: Use `phrase[:len(word_tuple)]` to get a tuple of the first $N-1$ words of each N-length `phrase` in the `adj_counter` to compare with `word_tuple`.
  3. HINT2: We are returning the filtered dictionary which is stored in the variable `subset_word_adj_counter`, so you need to modify this dictionary in some way.
- `p_naive`
  1. This function calculates the non-smoothed empirical probability of a length n phrase occurring given length $n-1$ tuple of tokens `prev_phrase`. In other words, it calculates $P(\text{current token}|\text{previous N - 1 tokens})$. The probability is based on counts, exactly like how we calculated probabilities in the green eggs and ham example earlier in this problem without smoothing.
  2. HINT1: You need to define `prob` because it is being returned.
  3. HINT2: You need to normalize `filtered_adj_counter` which is already defined for you.
- `calc_neg_log_prob_of_sentence`
  1. This function calculates and returns the negative log probability of the entire sequence of tokens `sentence_list` given a probability function `p_func` (which is either the smoothed or the non-smoothed probabilities).
  2. HINT1: `curr_word_prob` is defined for you, and is $P(\text{currToken}|\text{previous N - 1 tokens})$.
  3. HINT2: `cum_neg_log_prob` is what the function returns. For each iteration of the for-loop, what must we do to update `cum_neg_log_prob`?
  4. HINT3: Think about how we combine log probabilities for each word.
- `calc_prob_of_sentence`
  1. This function calculates and returns the probability of a sequence of tokens.
  2. HINT1: Use the function you just wrote, `calc_neg_log_prob_of_sentence`.

3. HINT2: Use `np.exp`.

(b) [3 pts] After writing the above functions and mounting the corpus on your google drive, you should be able to run the text generation algorithm. This algorithm works by first using the N-gram model to construct CPTs (as we saw earlier in this homework). Then, it uses the CPTs to generate a sequence of words that our model thinks can occur with relatively "high probability." Our hope is that the "high probability" sequences are sequences of words that make some kind of sense.

Run the text generation algorithm and record (in the spaces below) some of your N-gram model-generated sentences with the following parameters. Please do these in order or else you will be very disappointed by the mediocrity of the text generated. What are the effects of increasing N and k on the quality of the generated text? Modify the N, k variables in the "play with params in code here" section.

1. $N = 1$, $k = 1$

2. $N = 2$, $k = 1$

3. $N = 2$, $k = 5$

4. $N = 3$, $k = 1$

5. $N = 3$, $k = 5$

(c) [1 pt] Now, perform language modeling on a dataset/corpus of your choosing. Select a corpus, put it in a text file, and upload it to the google drive folder with all the other .txt files you uploaded earlier. Then redefine `training_corpora_path_list` under the comment "REDEFINE `training_corpora_path_list` here if you wish to use your own corpus" and use the model to perform text generation (as you did above). For good results, select a corpus at least 50,000 words long. If you are not feeling creative, feel free to use the other files in the cs188whw4public folder.

Below, write a sentence that your N-gram model generated on your custom corpus.

(d) [0 pts] So far, we have used N-gram to do language modeling and text generation. We can also use N-gram, with some modifications (that staff has already coded for you) to capitalize a sequence of words correctly. This is done using probability maximization; in other words, which options of capitalization look most like things we have seen in the training dataset. The current implementation is slow, but using the Viterbi Algorithm can help speed it up.

Run the capitalization script on the strings provided (you do not need to make any changes for this part). If you wrote your code correctly, you should get capitalizations of the inputted sentences that make sense. In other words, your model is smart enough to know when to capitalize tricky words like "united!"

(e) [1 pt] In Question 2, you used Chain Rule to derive the backpropagation equations for a simple loss function, small network, and low-dimensional problem. As these various aspects of the problem get harder, it quickly becomes impractical to derive every equation by hand. Luckily, there are great libraries available which automate the backpropagation process for the neural networks that you create. On your Colab Notebook, run the NN program that we have provided.

This code revisits the text generation problem from earlier. However, instead of using Naive Bayes or N-grams for modeling, it uses a neural network. Briefly describe the improvements in text generation from using this neural network (the Transformer model) over what you were able to accomplish with N-grams.

(f) [0 pts] **Submission Directions**: On your colab, please go to `File` → `Download .py`, and submit this python file to gradescope. We will set up an autograder for the coding portion after the due date. Note that the autograder will not run if you do not submit a python file!