CSCI-GA 2590: Natural Language Processing

Predicting Sequences

Name
NYU ID

Collaborators:
By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Before you get started, please read the Submission section thoroughly.

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. You can either directly type your solution between the `shaded` environments in the released `.tex` file, or write your solution using pen or stylus. A `.pdf` file must be submitted.
Programming: Questions marked with "coding" next to the assigned to the points require a coding part in `submission.py`. Submit `submission.py` and we will run an autograder on Gradescope. You can use functions in `util.py`. However, please do not import additional libraries (e.g. `numpy`, `sklearn`) that aren't mentioned in the assignment, otherwise the grader may crash and no credit will be given. You can run `test.py` to test your code but you don't need to submit it.

Problem 1: N-gram Language Models

In this problem, we will derive the MLE solution of n-gram language models. Recall that in n-gram language models, we assume that a token only depends on $n-1$ previous tokens, namely:

$$p(x_{1:m}) = \prod_{i=1}^{m} p(x_i \mid x_{i-n+1:i-1}) ,$$

where $x_i \in \mathcal{V}$ and $x_{1:i}$ denotes a sequence of i tokens $x_1, x_2, \dots, x_i$. Note that we assume all sequences are prepended with a special start token $*$ and appended with the stop token $\texttt{STOP}$, thus $x_i = *$ if $i < 1$ and $x_m = \texttt{STOP}$. We model the conditional distribution $p(x_i \mid x_{i-n+1:i-1})$ by a categorical distribution with parameters α:

$$p(w \mid c) = \alpha[w, c] \quad \text{for } w \in \mathcal{V}, c \in \mathcal{V}^{n-1} .$$

Let $D = \{x^i_{1:m_i}\}_{i=1}^{N}$ be our training set of N sequences, each of length m_i.

1. [2 points] Write down the MLE objective for the n-gram model defined above. Note that we need to add the constraint that the conditional probabilities sum to one given each context.

2. [2 points] Recall that the method of Lagrange multipliers allows us to solve an optimization problem with equality constraints by forming a Lagrangian function, which can be optimized without explicitly parameterizing in terms of the constraints.

   Given an optimization problem to maximize $f(x)$ subject to the constraint $g(x) = 0$, we can express it in the form of the Langrangian, which can be written as $f(x) - \lambda g(x)$.

   Write down the Langrangian $\mathcal{L}(\alpha, \lambda)$ for the MLE objective using the method of Lagrange multipliers.

3. [4 points] Find the solution for α. Define count$(\cdot)$ to be a function which maps a sequence to its frequency in D. You can assume count$(c) > 0$ for $c \in \mathcal{V}^{n-1}$. [**HINT:** The solution for α should be a function of w and c. You can start by setting the partial derivative of $\mathcal{L}$ w.r.t. $\alpha[w, c]$; and w.r.t. λ_c to 0 and combining the two equations.]

Problem 2: Noise Contrastive Estimation

In this problem, we will explore efficient training of neural language models using noise-contrastive estimation. Recall that in neural language modeling, the conditional probability $p(w \mid c)$ is modeled by

$$p_\theta(w \mid c) = \frac{\exp(f_\theta(w, c))}{\sum_{w' \in \mathcal{V}} \exp(f_\theta(w', c))} , \tag{1}$$

where $w \in \mathcal{V}$ is a token in the vocabulary, $c \in \mathcal{C}$ is some context, and $f_\theta \colon \mathcal{V} \times \mathcal{C} \to \mathbb{R}$ is a scoring function indicating how compatible w and c are, e.g. a recurrent neural network.

1. [2 points] As usual, we use MLE to learn the parameters $\theta \in \mathbb{R}^d$. Show that the gradient of the log likelihood for a single observation $\ell(\theta, w)$ is

$$\nabla_\theta f_\theta(w, c) - \mathbb{E}_{w \sim p_\theta} [\nabla_\theta f_\theta(w, c)] .$$

2. [2 points] Note that computing the gradient can be expensive due to summing over the vocabulary when computing the expectation term, which arises from the normalizer (or the partition function) in (1). One idea is to treat the normalizer as another parameter to estimate, i.e.

$$p_\theta(w \mid c) = \frac{\exp(f_\theta(w, c))}{\exp(z_c)} ,$$

where $z_c \in \mathbb{R}$ for each context c. Explain why the MLE solution for z_c doesn't exist.

3. [3 points] The key idea in noise contrastive estimation is to reduce the density estimation problem to a binary classification problem, i.e. deciding whether a word comes from the "true" distribution $p(w \mid c)$ or a "noise" distribution $p_n(w)$. Note that the noise distribution is context-independent. (This should remind you of negative sampling in HW1.) Now consider a new data-generating process: Given context c, with probability $\frac{1}{k+1}$ we sample a word from $p(w \mid c)$; with probability $\frac{k}{k+1}$ we sample a word from $p_n(w)$ ($k \in \mathbb{N}$). In other words, for each "true" sample, we generate k "fake" samples. Let Y be a binary random variable indicating whether w is a true sample or a fake sample. Show that

$$p(Y = 1 \mid w, c) = \frac{p(w \mid c)}{p(w \mid c) + kp_n(w)} .$$

[**HINT:** Use Bayes' rule.]

4. [4 points] **[Optional]** We have reduced the problem of estimating $p(w \mid c)$ to predicting whether a sample (w, c) is true or fake. To learn a classifier, let's parametrize $p(Y = 1 \mid w, c)$. Note that p_n is known since it's chosen by us, so we just need to parametrize $p(w \mid c)$. Recall that we do not want to compute the normalizer, so (1) is not an option. Instead, let's model the normalizer as another parameter. We can either explicitly model it as in (2), or directly learn a self-normalizing function (i.e. $z_c = 0$):

$$\tilde{p}_\theta(w \mid c) = \exp(g_\theta(w, c)) \ .$$

Here we will proceed with the latter.[1]

For each word, we sample k fake words w^n from $p_n(w)$. Thus the log likelihood for a word and its noise samples is

$$\ell_{\text{NCE}}(\theta, w, k) = \log p_\theta(Y = 1 \mid w, c) + k\mathbb{E}_{w' \sim p_n} \log p_\theta(Y = 0 \mid w', c) \ .$$

In practice, expectation over p_n is approxmiated by k Monte Carlo samples.

Next, let's analyze how this objective connects to the MLE objective. Let $p_D(w \mid c)$ be the true distribution of words and consider the expected log likelihood. Let θ^* be the solution of

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{MLE}}(\theta, w)\right]$$

and θ_n^* be the solution of

$$\max_{\theta \in \mathbb{R}^d} \mathbb{E}_{w \sim p_D} \left[\ell_{\text{NCE}}(\theta, w, k)\right] \ .$$

Assuming f_θ and g_θ have the same parametrization, show that when $k \to \infty$, θ^* and θ_n^* satisfy the same first order condition, i.e.

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] = \mathbb{E}_{w \sim p_{\theta^*}} \left[\nabla_\theta f_\theta(w, c)|_{\theta=\theta^*}\right] \ ,$$

$$\mathbb{E}_{w \sim p_D} \left[\nabla_\theta g_\theta(w, c)|_{\theta=\theta_n^*}\right] = \mathbb{E}_{w \sim \tilde{p}_{\theta_n^*}} \left[\nabla_\theta g_\theta(w, c)|_{\theta=\theta_n^*}\right] \ .$$

[1] Empirically, it has been observed that setting z_c to be a constant works just fine when g_θ is a neural network.

Problem 3: Conditional Random Fields

In this problem, you will implement inference algorithms for the CRF model and compare different sequence prediction models on synthetic data. You may want to go over the `mxnet_tutorial.ipynb` first before you start.

Environment setup: Follow instructions in `README.md` to set up the environment for running the code.

(a) [2 points] To get started, take a look at the function `generate_dataset_identity` in `util.py` and the class `UnigramModel` in `model.py`. Given $x = (x_1, \ldots x_n)$ where $x_i \in \mathcal{V}$, the model makes an independent prediction at each step using only input at that step, i.e. $p(y_i \mid x_i)$. Run `python test.py unigram` to train a `UnigramModel`. It outputs the average hamming loss in the end. Let $y = (y_1, \ldots, y_n)$ be the gold labels and $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_n)$ be the predicted labels, take a look at `hamming_loss` in `submission.py` and write down the loss function.

(b) [2 points] Take a look at the `RNNModel` in `model.py`. It uses a bi-directional LSTM to encode x and makes independent predictions for each y_i. This time let's use the dataset generated by `generate_dataset_rnn`. Compare the result by running `python test.py unigram --data rnn` and `python test.py rnn --data rnn`. Which model has a lower error rate? Explain your findings.

(c) [4 points, coding] Next, we are going to add a CRF layer on top of the RNN model (see `CRFRNNModel` in `model.py`). Here we use the autograd function in MXNet to compute gradient for us, so we only need to implement the forward pass (the counterpart of the forward algorithm). Take a look at `crf_loss`. The main challenge here is to compute the normalizer which sums over all possible sequences:

$$\begin{aligned} \text{normalizer} &= \sum_{y \in \mathcal{Y}^n} \exp\left[s(y)\right] \\ &= \sum_{y \in \mathcal{Y}^n} \exp\left[\sum_{i=1}^{n} u(y_i) + \sum_{i=2}^{n} b(y_i, y_{i-1})\right] \end{aligned}$$

where u and b are scores from the `CRFRNNModel`. Note that here we assume $y_1 = *$ (the start symbol). Implement `compute_normalizer` using the `logsumexp` function in `util.py`. Your result must match `bruteforce_normalizer`. [**HINT:** You can compute all sums using array operations. `np.expand_dims` is very helpful here.]

See `submission.py`. No written submission.

(d) [4 points, coding] During inference, we will use Viterbi decoding to find

$$\arg\max_{y \in \mathcal{Y}^n} s(y)$$

where $s(y) = \sum_{i=1}^{n} u(y_i) + \sum_{i=2}^{n} b(y_i, y_{i-1})$. Implement `viterbi_decode`. Your result must match `bruteforce_decode`. [**HINT:** You can compute all sums using array operations. `np.expand_dims` is very helpful here.]

See `submission.py`. No written submission.

(e) [3 points] We are ready to test the CRFRNN model now. Use the HMM data (take a look at `generate_dataset_hmm` in util.py) and compare it with the RNN model by running `python test.py rnn --data hmm` and `python test.py crfrnn --data hmm`. Compare the results. [**NOTE:** This is an open-ended question. Discuss any findings you have is fine, e.g. runtime, error rate, convergence rate etc.]