XCS236 Problem Set 1

Due Sunday, September 8 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs236-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Instructor's Note

PS1 is weighted more on the written side whereas subsequent assigments will be more coding intensive. You will submit the entire `submission` directory. To do so run the command below from the `src` directory

```
zip -r submission.zip submission
```

or you can choose to zip the folder up manually.

Your zipped `submission` directory should look as follows:

1. `submission/__init__.py`
2. `submission/sample.py`
3. `submission/classifier.py`
4. `submission/likelihood.py`
5. `submission/samples.txt`
6. `submission/random_raw.pkl`
7. `submission/shakespeare_raw.pkl`
8. `submission/neurips_raw.pkl`
9. `submission/classification.pkl`
10. `submission/samples_temperature0-95.txt`
11. `submission/samples_temperature0-95_horizon2.txt` as Extra Credit

1. **[3 points (Written)] Maximum Likelihood Estimation and KL Divergence**

   Let $\hat{p}(x, y)$ denote the empirical data distribution over a space of inputs $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$. For example, in an image recognition task, x can be an image and y can be whether the image contains a cat or not. Let $p_\theta(y \mid x)$ be a probabilistic classifier parameterized by θ, e.g., a logistic regression classifier with coefficients θ. Show that the following equivalence holds:

   $$\arg\max_{\theta \in \Theta} \mathbb{E}_{\hat{p}(x,y)}[\log p_\theta(y \mid x)] = \arg\min_{\theta \in \Theta} \mathbb{E}_{\hat{p}(x)}[D_{\mathrm{KL}}(\hat{p}(y \mid x) \,||\, p_\theta(y \mid x))] \tag{1}$$

   where D_{KL} denotes the KL-divergence:

   $$D_{\mathrm{KL}}(p(x) \,||\, q(x)) = \mathbb{E}_{x \sim p(x)}[\log p(x) - \log q(x)] \tag{2}$$

   To help you get started consider this: We rely on the known property that if ψ is a strictly monotonically decreasing function, then the following two problems are equivalent:

   $$\max_\theta f(\theta) = \min_\theta \psi(f(\theta))$$

2. **[4 points (Written)] Logistic Regression and Naive Bayes**

A mixture of k Gaussians specifies a joint distribution given by $p_\theta(\boldsymbol{x}, y)$ where $y \in \{1, ..., k\}$ signifies the mixture id and $\boldsymbol{x} \in \mathbb{R}^n$ denotes n-dimensional real valued points. The generative process for this mixture can be specified as:

$$p_\theta(y) = \pi_y, \text{where} \sum_{y=1}^{k} \pi_y = 1 \tag{3}$$

$$p_\theta(\boldsymbol{x} \mid y) = \mathcal{N}(\boldsymbol{x} \mid \mu_y, \sigma^2 I) \tag{4}$$

where we assume a diagonal covariance structure for modeling each of the Gaussians in the mixture. Such a model is parameterized by $\theta = (\pi_1, \pi_2, ..., \pi_k, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_k, \sigma)$, where $\pi_i \in \mathbb{R}_{++}$, $\boldsymbol{\mu}_i \in \mathbb{R}^n$, and $\sigma \in \mathbb{R}_{++}$. Now consider the multi-class logistic regression model for directly predicting y from x as:

$$p_\gamma(y \mid \boldsymbol{x}) = \frac{\exp(\boldsymbol{x}^\top \boldsymbol{w}_y + b_y)}{\sum_{i=1}^{k} \exp(\boldsymbol{x}^\top \boldsymbol{w}_i + b_i)} \tag{5}$$

parameterized by vectors $\gamma = \{\boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_k, b_1, b_2, ..., b_k\}$, where $\boldsymbol{w}_i \in \mathbb{R}^n$ and $b_i \in \mathbb{R}$. Show that for any choice of θ, there exists γ such that:

$$p_\theta(y \mid \boldsymbol{x}) = p_\gamma(y \mid \boldsymbol{x}) \tag{6}$$

To help you get start consider that:

$$p_\theta(y \mid x) = \frac{p_\theta(x,y)}{p_\theta(x)} = \frac{\pi_y \cdot \exp\left(-\frac{1}{2\sigma^2}(x-\mu_y)^\top (x-\mu_y)\right) \cdot Z^{-1}(\sigma)}{\sum_i \pi_i \cdot \exp\left(-\frac{1}{2\sigma^2}(x-\mu_i)^\top (x-\mu_i)\right) \cdot Z^{-1}(\sigma)}$$

where $Z(\sigma)$ is the Gaussian partition function (which is a function of σ).

3. **Conditional Independence and Parameterization**

Consider a collection of n discrete random variables $\{X_i\}_{i=1}^n$, where the number of outcomes for X_i is $|\text{ val}(X_i)| = k_i$.

(a) **[1.50 points (Written)]** Without any conditional independence assumptions, what is the total number of independent parameters needed to describe the joint distribution over $(X_1, ..., X_n)$?

(b) **[1.50 points (Written)]** Under what independence assumptions is it possible to represent the joint distribution $(X_1, ..., X_n)$ with $\sum_{i=1}^n (k_i - 1)$ total number of independent parameters?

(c) **[2 points (Written)]**

Let $1, 2, ..., n$ denote the topological sort for a Bayesian network for the random variables $X_1, X_2, ..., X_n$. Let m be a positive integer in $\{1, 2, ..., n-1\}$. Suppose, for every $i > m$, the random variable X_i is conditionally independent of all ancestors given the previous m ancestors in the topological ordering. Mathematically, we impose the independence assumptions

$$p(X_i \mid X_{i-1}, X_{i-2}, ..., X_2, X_1) = p(X_i \mid X_{i-1}, X_{i-2}, ..., X_{i-m}) \tag{7}$$

for $i > m$. For $i \leq m$, we impose no conditional independence of X_i with respect to its ancestors. Derive the total number of independent parameters to specify the joint distribution over $(X_1, ..., X_n)$. You can express the answer using summation and product symbols.

4. **[5 points (Written)] Autoregressive Models**

Consider a set of n univariate *continuous* real-valued random variables $(X_1, ..., X_n)$. You have access to powerful neural networks $\{\mu_i\}_{i=1}^n$ and $\{\sigma_i\}_{i=1}^n$ that can represent any function $\mu_i : \mathbb{R}^{i-1} \to \mathbb{R}$ and $\sigma_i : \mathbb{R}^{i-1} \to \mathbb{R}_{++}$. We shall, for notational simplicity, define $\mathbb{R}^0 = \{0\}$. You choose to build the following Gaussian autoregressive model in the *forward* direction:

$$p_f(x_1, ..., x_n) = \prod_{i=1}^{n} p_f(x_i \mid x_{<i}) = \prod_{i=1}^{n} \mathcal{N}(x_i \mid \mu_i(x_{<i}), \sigma_i^2(x_{<i})) \tag{8}$$

where $x_{<i}$ denotes:

$$x_{<i} = \begin{cases} (x_1, ..., x_{i-1})^\top & \text{if } i > 1 \\ 0 & \text{if } i = 1 \end{cases} \tag{9}$$

Your friend chooses to factor the model in the *reverse* order using equally powerful neural networks $\{\hat{\mu}_i\}_{i=1}^n$ and $\{\hat{\sigma}_i\}_{i=1}^n$ that can represent any function $\hat{\mu}_i : \mathbb{R}^{n-i} \to \mathbb{R}$ and $\hat{\sigma}_i : R^{n-i} \to R_{++}$:

$$p_r(x_1, ..., x_n) = \prod_{i=1}^{n} p_r(x_i \mid x_{>i}) = \prod_{i=1}^{n} \mathcal{N}(x_i \mid \hat{\mu}_i(x_{>i}), \hat{\sigma}_i^2(x_{>i})) \tag{10}$$

where $x_{>i}$ denotes:

$$x_{>i} = \begin{cases} (x_{i+1}, ..., x_n)^\top & \text{if } i < n \\ 0 & \text{if } i = n \end{cases} \tag{11}$$

Do these models cover the same hypothesis space of distributions? In other words, given any choice of $\{\mu_i, \sigma_i\}_{i=1}^n$, does there always exist a choice of $\{\hat{\mu}_i, \hat{\sigma}_i\}_{i=1}^n$ such that $p_f = p_r$? If yes, provide a proof. Else, provide a concrete counterexample, including mathematical definitions of the modeled functions, and explain why.

Hint: Consider the case where $n = 2$.

$$p_f(x_1 \mid x_2) = \frac{p_f(x_1, x_2)}{p_f x_2}$$

is a mixture of truncated Gaussians whose mixture weights depend on ϵ.

To provide more direction consider proving (*) or (**) below: Consider the simple case of describing a joint distribution over (X_1, X_2) using the forward versus reverse factorizations. Consider the forward factorization where

$$p_f(x_1) = \mathcal{N}(x_1 \mid 0, 1)$$
$$p_f(x_2 \mid x_1) = \mathcal{N}(x_2 \mid \mu_2(x_1), \epsilon)$$

for which

$$\mu_2(x_1) = \begin{cases} 0 & \text{if } x_1 \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

(*) This construction makes $p_f(x_2)$ a mixture of two distinct Gaussians, which $p_r(x_2)$ cannot match, since $p_r(x_2)$ is strictly Gaussian. Any counterexample of this form, which makes $p_f(x_2)$ non-Gaussian, suffices for full-credit.

(**) Interestingly, we can also intuit about the distribution $p_f(x_1 \mid x_2)$. If one chooses a very small positive ϵ, then the corresponding $p_f(x_1 \mid x_2)$ will approach a truncated Gaussian distribution, which cannot be approximated by the Gaussian $p_r(x_1 \mid x_2)$ [1].

Optionally, we can prove (*) and a variant of (**) which states that, any $\epsilon > 0$, the distribution:

[1] This observation will be useful when we move on to variational autoencoders $p(z, x)$ (where z is a latent variable) and discuss the importance of having good variational approximations of the true posterior $p(z \mid x)$

$$p_f(x_1 \mid x_2) = \frac{p_f(x_1, x_2)}{p_f x_2}$$

is a mixture of truncated Gaussians whose mixture weights depend on ϵ.

5. **Monte Carlo Integration**

A latent variable generative model specifies a joint probability distribution $p(x, z)$ between a set of observed variables $x \in \mathcal{X}$ and a set of latent variables $z \in \mathcal{Z}$. From the definition of conditional probability, we can express the joint distribution as $p(x, z) = p(z)p(x \mid z)$. Here, $p(z)$ is referred to as the prior distribution over z and $p(x \mid z)$ is the likelihood of the observed data given the latent variables. One natural objective for learning a latent variable model is to maximize the marginal likelihood of the observed data given by:

$$p(x) = \int_z p(x, z)dz \tag{12}$$

When z is high dimensional, evaluation of the marginal likelihood is computationally intractable even if we can tractably evaluate the prior and the conditional likelihood for any given x and z. We can however use Monte Carlo to estimate the above integral. To do so, we sample k samples from the prior $p(z)$ and our estimate is given as:

$$A(z^{(1)}, ..., z^{(k)}) = \frac{1}{k} \sum_{i=1}^{k} p(x \mid z^{(i)}), \text{ where } z^{(i)} \sim p(z) \tag{13}$$

(a) **[1.50 points (Written)]** An estimator $\hat{\theta}$ is an unbiased estimator of θ if and only if $\mathbb{E}[\hat{\theta}] = \theta$. Show that A is an unbiased estimator of $p(x)$.

(b) **[1.50 points (Written)]** Is $\log A$ an unbiased estimator of $\log p(x)$? Prove why or why not.

Hint: The proof is short, estimator of $p(x)$ using the definition of an unbiased estimator and Jensen's Inequality

6. **Programming assignment**

In this programming assignment, we will use an autoregressive generative model to create text. We will generate samples from the recent OpenAI GPT-2 model. It is a model based on the Transformer, which is a special kind of autoregressive model built on self-attention blocks, and has become the backbone of many recent state-of-the-art sequence models in Natural Language Processing. See this blog post for a friendly introduction. You will be asked to implement the sampling procedure for this model and compute likelihoods.

In Figure 1, we show an illustration on how this model (roughly) works. Consider a sequence of tokens $x_0, x_1, ..., x_T$. For each token x_i, it has 50257 possible values. First, for each possible value of a token, we use a 768-dimensional trainable vector as its embedding, which results in a total of 50257 different embedding vectors. Next, we feed the embeddings into a GPT-2 network. The output vectors of the GPT-2 network are finally passed through a fully-connected layer to form a 50257-way softmax representing the probability distribution of the next token p_{i+1}. See Figure 1 for an illustration.

Training such models can be computationally expensive, requiring specialized GPU hardware. In this particular assignment, we provide a smaller pretrained model. It should be feasible to run this model without using any GPUs. After loading this pretrained model into `PyTorch`, you are expected to implement and answer the following questions.

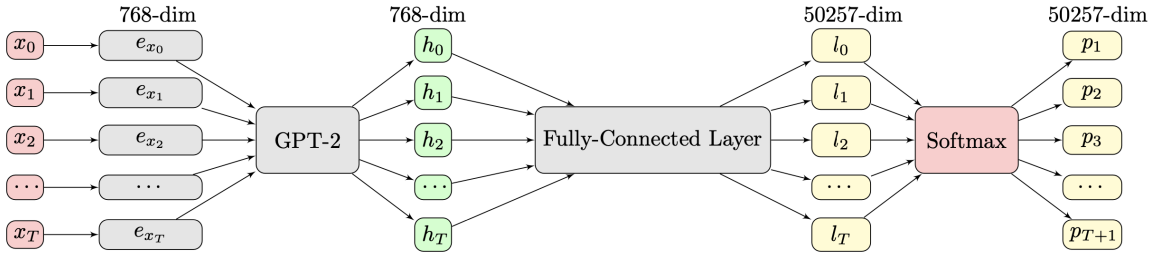

Figure 1: The architecture of our model. T is the sequence length of a given input. x_i is the index token. e_{x_i} is the trainable embedding of token x_i. h_i is the output of GPT-2. l_i is the logit and p_i is the probability. Nodes in gray contain trainable parameters.

To run the models for 6.c through 6.f, please run

```
python main.py
```

For GPU acceleration run run the command below. **Note:** we are not supporting MPS GPUs as it trains slower than CPU-enabled training on Apple Silicon devices.

```
python main.py --device gpu
```

(a) **[1 point (Written)]**

Suppose we wish to find an efficient bit representation for the 50257 tokens. That is, every token is represented as $(a_1, a_2, ..., a_n)$, where $a_i \in \{0, 1\}, \forall i = 1, 2, ..., n$. What is the minimal n that we can use?

(b) **[1 point (Written)]**

If the number of possible tokens increases from 50257 to 60000, what is the increase in the number of parameters? Give an exact number and explain your answer.

Hint: The number of parameters in the GPT-2 module in Fig. 1 does not change.

(c) **[5 points (Coding)]**

In this question, we will try to generate paper abstracts using the GPT-2 model. We will first implement the sampling procedure for GPT-2. Then, we will choose 5 sentences from the abstracts of some NeurIPS[2]

[2]Neural Information Processing Systems (NeurIPS) is a machine learning conference.

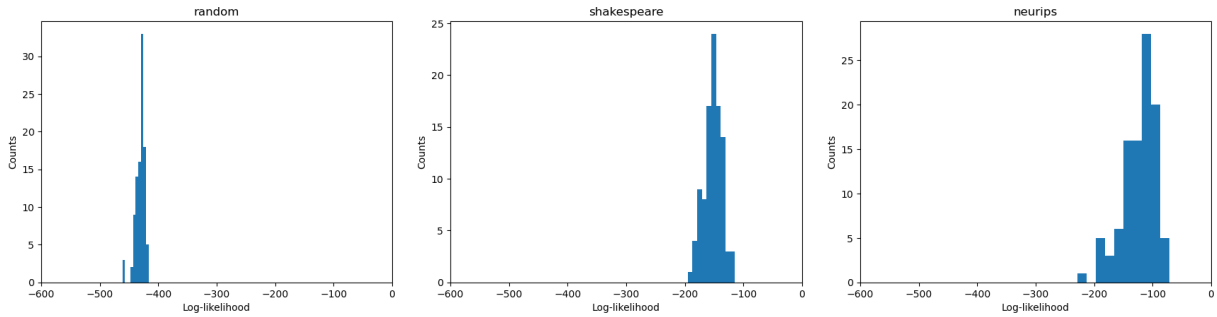

Figure 2: Expected log likelihood histograms for each text type

2015 papers and see how GPT-2 generates the rest of the abstract. You will need to complete the method sample in `src/submission/sample.py` in the starter code.

Hint: The text generated should look like a technical paper.

(d) **[5 points (Coding)]**

Complete the function `log_likelihood` in `src/submission/likelihood.py` to compute the log-likelihoods for each string. The histograms of the log-likelihoods of strings for each file should look like those in Fig. 2.

(e) **[3 points (Coding)]**

We now provide new texts in `snippets.pkl`. The texts are either taken from NeurIPS 2015 papers or Shakespeare's work, or generated randomly. Try to infer **whether the text is random**. You will need to complete the function `classification` in `src/submission/classifier.py`.

Hint: Look carefully at the plots above in Fig. 2. Is there a simple representation space in which the random data is separable from the non-random data?

(f) **[5 points (Coding)]**

Temperature scaling is a commonly used technique for adjusting the likelihood of the next token. We pick a scalar temperature $T > 0$ and divide the next token logits by T:

$$p_T(x_i \mid x_{<i}) \propto e^{\log p(x_i|x_{<i})/T} \tag{14}$$

The model p is the GPT-2 model used in question (6.c) and the model p_T is the temperature-scaled model. For $T < 1$, we can see that p_T induces a *sharper* distribution than p, since it makes likely tokens even more likely.

Complete the function `temperature_scale` in `src/submission/sample.py` only for

the case when `temperature_horizon=1` to perform temperature scaling during sampling.

(g) **[2 points (Written, Extra Credit)]**

In the previous question, we performed temperature scaling only over the next token, i.e., with $T < 1$ we made likely next-tokens even more likely. What if we want to make likely *sentences* even more likely? In this case, we should consider scaling the *joint temperature*.

$$p_T^{\text{joint}}(x_0, x_1, ..., x_M) \propto e^{\log p(x_0,x_1,...,x_M)/T} \tag{15}$$

Does applying chain rule with single-token temperature scaling recover joint temperature scaling? In other words, determine if the following equation holds for arbitrary T:

$$\prod_{i=0}^{M} p_T(x_i \mid x_{<i}) \stackrel{?}{=} p_T^{\text{joint}}(x_0, x_1, ..., x_M) \tag{16}$$

Hint: Suppose we have an autoregressive model $p(x, y)$ with the following parameterization:

$$p(x) \propto f(x)$$
$$p(y) \propto g(x, y)$$

with non-negative functions f, g. What can we say about $p(x, y)$? We can NOT assume that

$$p(x, y) \propto f(x)g(x, y)$$

because we must write $p(y \mid x) = \frac{g(x,y)}{\int_y g(x,y)dy'}$, where the normalizing constant in the denominator depends on x. If we write the joint probability this way, we incorrectly assume that the normalizing constant in the denominator will be the same for all x's.

(h) **[4 points (Coding, Extra Credit)]**

Next, we will implement temperature scaling over more than than one token (for simplicity, we will do temperature scaling over two tokens).

$$p_T^{\texttt{joint-2}}(x_i, x_{i+1} \mid x_{<i}) \propto e^{\log p(x_i, x_{i+1} \mid x_{<i})/T} \tag{17}$$

$$p_T^{\texttt{joint-2}}(x_i \mid x_{<i}) = \sum_j p_T^{\text{joint-2}}(x_i, x_{i+1} = a_j \mid x_{<i}) \tag{18}$$

Complete the function `temperature_scale` in `src/submission/sample.py` for the case when `temperature_horizon=2` to perform temperature scaling over a horizon of two tokens.