CPSC 440/550 Advanced Machine Learning (Jan-Apr 2024)
Bonus Assignment – due Saturday April 27th at **11:59pm**

The assignment instructions are the same as for the previous assignments. If you do the assignment with a partner, please only hand in one copy for the group (using the appropriate Gradescope feature).

1 Discrete Markov Chain Inference [40 points]

The function `grad_chain()` in `main.py` makes a Markov chain model corresponding to the "CS graduate Markov chain" from class.

Note that, slightly contrary to class, this question will use the notation that the first step is index 0; this will help avoid confusion between the math and zero-indexed Python code:

$$p(x_0, x_1, \dots, x_{d-1}) = p(x_0) \prod_{j=1}^{d-1} p(x_j \mid x_{j-1}).$$

[1.1] [6 points] Implement the function `MarkovChain.sample`, in `markov_chain.py`, that uses ancestral sampling to sample sequences of a given length d. Hand in this code, and report the univariate marginal probabilities for time 30 using a Monte Carlo estimate based on 10,000 samples (from `main.py mc-sample`).

Hint: `rng.choice` might be helpful.

Answer: TODO

[1.2] [6 points] Write a function, `MarkovChain.marginals`, that uses the CK equations to compute the exact univariate marginals up to a given time d. Use `main.py mc-marginals` to find the exact marginals at time 30. Hand in this code, report the exact univariate marginals at time 30 (to three decimal places), and report how this differs from the marginals in the previous question.

Answer: TODO

[1.3] [2 points] What is the state c with highest marginal probability, $p(x_j = c)$, for each time j?

Answer: TODO

[1.4] [9 points] Write a function, `MarkovChain.mode`, that uses the Viterbi decoding algorithm for Markov chains to find the optimal decoding up to a time d. Hand in this code and report the optimal decoding of the Markov chain up to time 50, and up to 100.

Answer: TODO

[1.5] [2 points] Report all the univariate conditional probabilities at time 30 if the student starts in grad school, $p(x_{30} = c \mid x_0 = 2)$, for all c.

Hint: you can do this by changing the input to the CK equations.

Answer: TODO

[1.6] [4 points] Report for all c the univariate conditional probabilities $p(x_4 = c \mid x_9 = 5)$ ("where you were likely to be 4 years after graduation, if you ended up in academia after 9 years") obtained using a Monte Carlo estimate based on 10,000 samples and rejection sampling. Also report the number of samples accepted from the 10,000 samples.

Answer: TODO

[1.7] [9 points] Give code implementing a dynamic programming approach to exactly compute $p(x_3 = c \mid x_6 = 5)$, and report the exact values for all c.

Answer: TODO

[1.8] [2 points] Why is $p(x_j = 6 \mid x_{20} = 5)$ equal to zero for all j less than 20?

Answer: TODO

2 MCMC for Bayesian Logistic Regression [20 points]

`main.py mcmc-blogreg` loads a set of images of '2' and '3' digits. It then runs the Metropolis MCMC algorithm to try to generate samples from the posterior over w, in a logistic regression model with a Gaussian prior. Once the samples are generated, it makes a histogram of the samples for several of the variables. The plots also show the MAP estimate, as a vertical line.[1]

[2.1] [4 points] Why might the samples coming from the Metropolis algorithm as run here not give a good approximation to the posterior?

Answer: TODO

[2.2] [4 points] Modify the proposal used by the demo to $\hat{w} \sim \mathcal{N}(w, (1/10)^2\mathbf{I})$ instead of $\hat{w} \sim \mathcal{N}(w, \mathbf{I})$. Hand in your modifications to the code and the updated histogram plot.

Answer: TODO

[2.3] [4 points] Modify the proposal to use $\hat{w} \sim \mathcal{N}(w, (1/100)^2\mathbf{I})$. Do you think this is probably sampling from the posterior distribution more or less effectively than the previous choice? (Briefly explain.) (Don't base your answer on the test likelihoods, those are for the next part!)

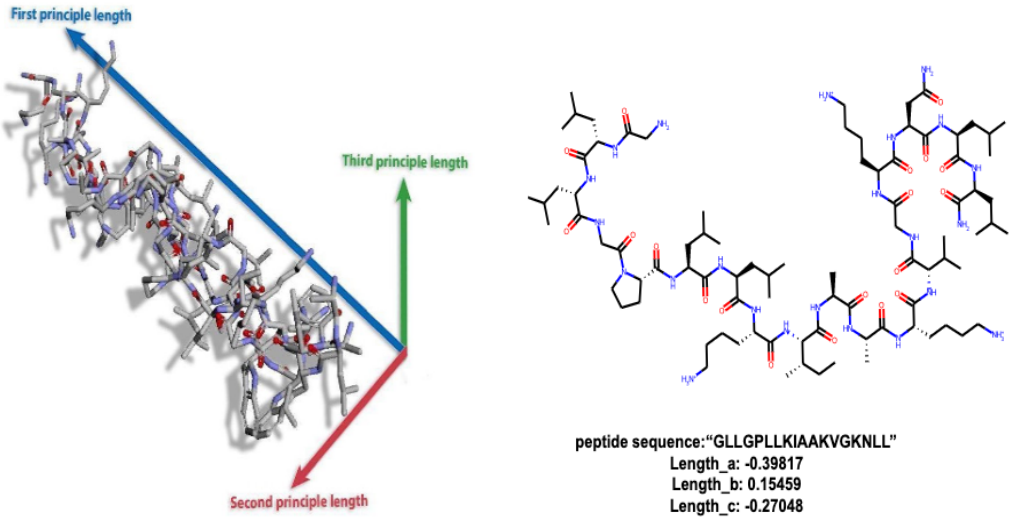
Answer: TODO

[2.4] [8 points] The `BayesianLogisticRegression.different_log_likelihoods` method implements four different attempts at evaluating the predictive log-likelihood of this method on test data. Each one has a different interpretation. Explain each computation; don't just directly translate the code into English, but say what it's computing in words. Which is the correct log-likelihood for the fully-Bayesian prediction?

[1] The division into "positive," "negative," and "neutral" variables is based on the sign of that MAP estimate.

3 Graph Neural Networks [40 points]

While not nearly as common as tabular data, images, or natural language, another major modality of data that's seen a lot of interest particularly in the past five years or so is *graph* data. There are multiple related problem setups, but here we wish to learn a *function on graphs*: similar to how in image classification we might want to learn a function whose input is an image and output is a class label, here we want to learn a function whose input is the molecular graph of a peptide, and whose output gives some aggregate 3D properties of that molecule's structure, such as its length, "sphericity," and inertia.



This is the `Peptides-struct` dataset from the Long-Range Graph Benchmark. Each node represents a heavy atom in the peptide, and the (undirected) edges a molecular bond; nodes have pre-computed features associated with them. (There are also edge features in this dataset, but we're ignoring them.)

We've pre-processed the dataset to include eigendecompositions of the graph Laplacian, which will be useful for constructing Laplacian positional encodings in the `LapPENodeEncoder` class in `gnns.py`. This gives some notion of "location" within the graph to each node, similar to the trigonometric position features we discussed for sequence Transformers. (We won't use this for the GCN, though you could, and it would help.) You don't have to worry about this, but you can check out that class (and the code inside the `if False` block in `gnn_utils.py`) if you're curious.

To work with graph data, you'll want to install the PyTorch Geometric library, with `pip install torch_geometric`. You won't have to worry about any internals here, but some of its helpers will handle some of the grunt work of working with this kind of data. The first time you run `main.py gcn`, it'll download the dataset into the `data` folder; it's about 400 MB, so I didn't put it in the zip.

This dataset is bigger than the datasets we've used before (ie MNIST); it's still runnable on a decent laptop, but it might be a little annoying, especially for the Transformer later. You might prefer to use a GPU on Google Colab (which is free), or some other machine with a CUDA-capable GPU. To use Colab, go to `https://colab.research.google.com`, and request a T4 GPU under `Runtime` → `Change runtime type`. Then run `pip install torch_geometric`, open the Files tab on the left (the folder icon) and upload the contents of the code directory. Then, after `import main`, you can use either `main.train_gcn()` or `main.run("gcn")`. Be careful with editing files on Colab, though; unless you save them to your Google Drive or similar, they'll just vanish on you! One thing you could do is write your code locally, make sure it runs on a batch or two, then upload to Colab for a final run to make sure it trains okay.

If you have CUDA set up for your GPU in pytorch, e.g. on Colab, it should use that automatically. You

can use `main.py use-cpu gcn`, `main.py use-mps gcn`, or `main.py use-cuda gcn` to force a device. MPS is the GPU on recent Macs; on my machine, I get pretty mixed results with CPU usually faster on this workload, but your mileage may vary. If you run into memory issues, try decreasing the batch size given to the data loader; if you have plenty of free memory, you could try increasing it to see if that's faster.

`main.py gcn` will run a Graph Convolutional Network on this data, using the `torch_geometric` library's implementation. This model is, while not state of the art or anything, "pretty okay" on this dataset. This dataset is pretty accessible by the standards of modern datasets; the code as-is runs an epoch in about ten seconds on my laptop's CPU (four on a Colab GPU). Here we only run three epochs out of laziness, but if you train longer it'll do better.

[3.1] [15 points] Implement your own graph convolution operation, rather than using PyTorch Geometric's; there's scaffolding for you in `MyGCNConv`. (`main.py my-gcn` runs that for you; you shouldn't need to change anything in the `GCN` class.) A graph convolution is given by

$$\text{GCN}(x)_v = \sum_{u \in \text{neighbours}(v) \cup \{v\}} \frac{1}{\sqrt{\deg(v)\deg(u)}} W x_u + b,$$

where v and u are nodes in the graph with corresponding node feature vectors x_v and x_u. (Recall that a neighbour of a node is any node with an edge the degree of a node is its number of neighbours.) The parameters of the matrix are the matrix W of shape `[dim_out, dim_in]` and the bias vector b of shape `[dim_out]`.

While they implement it with a relatively complex message passing framework, do not use this in your code. You may want to use a matrix multiplication framing (you probably don't want to do explicit looping). Your implementation will likely be slower than theirs, but run it for a few epochs to make sure the performance is about the same; my straightforward matrix-multiplication implementation takes about 90 seconds per epoch (instead of 10) on my laptop's CPU, or about 7 seconds (instead of 4) on a Colab GPU. The accuracy might not be exactly the same (there's a bunch of randomness, and a few things will be slightly different); as long as it's going down and not a huge amount higher than the other implementation, it should be fine. Hand in your code.

Answer: TODO

[3.2] [25 points] While there are many variants of Transformers for graph data, the class `gnns.GraphTransformer` implements a simple variant that does pairwise attention over all nodes in the graph. Similarly to how the sequence order of a sentence is only available to a standard Transformer via positional encodings, in this version of a graph Transformer, the structure of the graph is only available via the Laplacian positional encoding. While this can cause issues on some datasets, in this dataset it turns out to be pretty okay. `main.py graph-transformer` runs this model using torch's implementation of multi-headed self-attention.

Because of the additional pairwise attention, this method is slower than GCNs, so if you don't have a decent GPU it'll really be nicer to train on Colab (where an epoch takes 15 seconds, instead of about 10 minutes on my CPU!). But you can develop locally, make sure your code runs for a couple of batches, and then try training on Colab. (If you want, try training for longer to get a much better predictive model!)

Finish the implementation of the `gnns.MultiHeadSelfAttention.forward()` method; you can run it with `main.py my-graph-transformer`. My implementation is again slower than the pytorch implementation (which has had a lot of optimization effort put into it), but regression performance is the same. Make sure it runs okay, and hand in your code.

Answer: TODO