CPSC 440/540 Machine Learning (Jan-Apr 2023)
Assignment 2 – due Friday March 1 at **11:59pm**

The assignment instructions are the same as for the previous assignment, but for this assignment you can work in groups of 1 or 2. Please only hand in one assignment for the group.

1 Bernoulli Inference [25 points]

Consider a Bernoulli distribution with $\theta = 0.26$.

[1.1] [2 points] Suppose we generate ten iid samples $\{x^{(1)}, x^{(2)}, ..., x^{(10)}\}$ according to this model. What is the probability that all 10 samples are equal to 0?

Answer: TODO

[1.2] [2 points] Write a Python function that generates t iid examples from this distribution, based on numpy's `rng.random()` function (which returns uniformly pseudo-random numbers in $[0, 1]$) as the source of randomness. Hand in your code.

Answer: TODO

[1.3] [2 points] Consider a game where you get \$3 if a sample from this distribution is 0, and lose \$7 if the sample is 1. What is the expected \$ value you get from playing this game?

Answer: TODO

[1.4] [9 points] Supposing you weren't able to do that math, one way to approximate the answer of the previous question is to take

$$\frac{1}{t}\sum_{i=1}^{t} f(x^{(i)}),$$

where each $x^{(i)}$ is an iid sample, and f gives the \$ value for that sample (either 3 or -7).

Implement this approximation, and use the code in `bernoulli_mc` (inside `main.py`) to plot the following: draw a line of the "running mean" of the approximation as you see more samples, going from 1 to 100,000 samples. Show three independent runs on the same plot, in different colours (just calling `ax.plot` more than once and using its default colour cycle is fine). Also use `ax.axhline` to show the analytical expected value from the previous part. Hand in this plot.

Answer: TODO

[1.5] [10 points] Now, suppose that you start playing the game with \$20. You'd really like to turn that into \$200 to have a night on the town, but if you ever can't pay your debt (that is, your balance goes below \$0) then you need to go on the lam. What's the probability that you'll be able to have a fun night? Answer to within 1%, i.e. if the answer were exactly 21.34% (it's not) then it's okay if your answer is anywhere between 20.34% and 22.34%. Hand in your code and a justification that your answer is accurate enough.

It's possible to answer this analytically or in various other ways, and you're welcome to do that if you want. But remember not to look up specific homework problems online; you should be able to design a simple Monte Carlo algorithm to do this!

Answer: TODO

2 Learning with Weights [5 points]

We often use categorical (or Bernoulli) distributions as parts of more complicated distributions. In these cases we often to need to maximize a weighted log-likelihood of the form

$$f(\theta) = \sum_{i=1}^{n} v^{(i)} \log p(x^{(i)} \mid \theta),$$

where each $v^{(i)}$ is a non-negative weight for example i. Derive the MLE for this model when we use a categorical likelihood with k categories for $p(x^{(i)} \mid \theta)$. You can take as given that f is concave (so that any stationary point is a global optimum).

Answer: TODO

3 Naive Bayes [35 points]

If you run `main.py mnist-naive-nb`, it will load training and test data for MNIST, discretized into binary values. It will then fit a "naive" naive Bayes model, which is a generative classifier that assumes $p(x_1, x_2, \ldots, x_d, y)$ is product of Bernoullis. This model has a very high test error (88.6%), since the features do not affect the predictions.

[3.1] [3 points] The regular naive Bayes model discussed in class writes the joint probability of a dataset $(\mathbf{X}, \mathbf{y})$ as

$$\begin{aligned} p(\mathbf{X}, \mathbf{y}) &= \prod_{i=1}^{n} \left[p(y^{(i)}) \prod_{j=1}^{d} p(x_j^{(i)} \mid y^{(i)}) \right] \\ &= \prod_{i=1}^{n} \left[\theta_y^{y^{(i)}} (1-\theta_y)^{1-y^{(i)}} \prod_{j=1}^{d} \left[\theta_{j|y^{(i)}}^{x_j^{(i)}} (1-\theta_{j|y^{(i)}})^{1-x_j^{(i)}} \right] \right], \end{aligned}$$

using Bernoullis to parameterize $p(y^i)$ and each conditionals $p(x_j \mid y)$. Show how to derive the MLE for any particular parameter $\theta_{j|c}$. You can assume here that the log-likelihood is concave (so that any stationary point is a global optimum).

Answer: TODO

[3.2] [2 points] Even though the naive Bayes likelihood has many parameters, the MLE for a particular parameter $\theta_{j|c}$ does not depend on θ or any any $\theta_{j'|c'}$ with $j \neq j'$ and $c' \neq c$. Explain why these terms do not appear in the MLE. (Don't use the probabilistic concept of "independence" in your answer. Instead explain how the form of the log-likelihood makes them not depend on each other.)

Answer: TODO

[3.3] [8 points] Implement a standard naive Bayes classifier with this parameterization in `naive_bayes.py`. Include the same Beta prior for each $X_j \mid Y$, since otherwise you'll get some NaNs (but no need for a prior on Y, since we have lots of data there). Hand in your code, and the output of `python main.py mnist-nb`, which tests with several different choices for the amount of Laplace smoothing.

Answer: TODO

[3.4] [5 points] Naive Naive Bayes was *way* too naive; Naive Bayes still doesn't do *that* well. What if we removed the independence assumption and just went for a full tabular parameterization in a "Galaxy Brain Bayes" model

$$p(\mathbf{X}, \mathbf{y}) = \prod_{i=1}^{n} \left[p(y^{(i)}) p(x^{(i)} \mid y^{(i)}) \right],$$

where $p(y)$ is Bernoulli and $p(x \mid y)$ is a full tabular distribution, with Laplace smoothing (adding one pseudocount per possible x value). Without actually implementing it (unless you want to), what would the test and training error of this model be on the current problem? Give your reasoning; be explicit about what the predictions will be.

Answer: TODO

How much better can we expect to get? `python main.py mnist-logreg` will fit a multiclass logistic regression (aka softmax regression) model from scikit-learn, which gets about 7.6% error, less than half the error rate of naive Bayes. (Various nonlinear models can get almost zero error on this problem.) So, let's try a different way to make naive Bayes less naive, which will substantially improve its performance.

It seems reasonable that there would be clusters in the classes. For example, there might are several different "general ways" to draw the digit 7. Instead of assuming that the features are independent given the class label, we might instead assume they're independent given the *cluster* that they are in. Consider a *vector-quantized naive Bayes* (VQNB) that implements this idea:

- It clusters the examples associated with *each digit* into k clusters, using k-means clustering (k clusters for each class, $10k$ clusters total). We'll use $z^{(i)}$ as the cluster number of example i. The value $z^{(i)}$ will be from 1 to k, with $y^{(i)}$ determining whether we are considering the k "0" clusters, the k "1" clusters, or so on.
- Since we don't know $z^{(i)}$ (it is a "latent variable"), we can marginalize over its possible values. Using the marginalization and then the product rule, we can write the joint probability as

$$\begin{aligned}
p(x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}, y^{(i)}) &= \sum_{z=1}^{k} p(x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)}, y^{(i)}, z^{(i)} = z) \\
&= \sum_{z=1}^{k} p(x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)} \mid y^{(i)}, z^{(i)} = z)\, p(y^{(i)}, z^{(i)} = z) \\
&= \sum_{z=1}^{k} p(x_1^{(i)}, x_2^{(i)}, \ldots, x_d^{(i)} \mid y^{(i)}, z^{(i)} = z)\, p(z^{(i)} = z \mid y^{(i)})\, p(y^{(i)}) \\
&= p(y^{(i)}) \sum_{z=1}^{k} p(z \mid y^{(i)}) \left[\prod_{j=1}^{d} p(x_j^{(i)} \mid y^{(i)}, z) \right];
\end{aligned}$$

  the last line uses independence of the features given the cluster.

[3.5] [12 points] Implement the VQNB method (there's a stub in `naive_bayes.py`). Hand in your code, and the test error you obtain with this model for $k = 2$ through $k = 5$.

Hint: The same KMeans class as last time is in the handout code for you to use, or feel free to use `scikit-learn`'s.

Hint: $p(y)$ you can handle just as before. $p(z \mid y)$ is a categorical variable. $p(x_j \mid y, z)$ is Bernoulli; you'll have one Bernoulli parameter for each (y, z) pair, which it might be convenient to organize in a $10 \times k$ array.

Hint: If you're working with log-probabilities (which is a good idea), `scipy.special.logsumexp` might be helpful for the sum operation.

Hint: To help with debugging, note that you should get the naive Bayes model in the special case of $k = 1$. Further, with this type of model you usually see the biggest performance gain when going from $k = 1$ to $k = 2$.

Hint: You may not be able to exactly match the performance of logistic regression.

Answer: TODO

[3.6] [3 points] For a run of the method with $k = 5$, show the images obtained by plotting the estimates for $p(x_j = 1 \mid z, y)$ for all j as a 28 by 28 image, for each value of z and y (so there should be 50 images). There's a code stub in there for plotting it, just fill that out.

Answer: TODO

[3.7] [2 points] What is the big-O computational cost of making a prediction with this model?

Answer: TODO

4 Neural Networks [35 points]

[4.1] [8 points] `main.py nn-regression` runs a stochastic gradient method to train a neural network on the `basis_data` dataset from a previous assignment. However, in its current form it doesn't fit the data very well. Modify the training procedure and model to improve the performance of the neural network. Hand in your plot after changing the code to have better performance, and list the changes you made.

Hint: There are many possible strategies you could take to improve performance. Below are some suggestions, but note that the some will be more effective than others:

- *Changing the network structure (`hidden_layer_sizes` gives the number of hidden units per layer).*
- *Changing the training procedure (you can change the stochastic gradient step-size, use decreasing step sizes, use mini-batches, run it for more iterations, add momentum, switch to `findMin`, use Adam, and so on).*
- *Transform the data by standardizing the features, standardizing the targets, and so on.*
- *Add regularization (L2-regularization, L1-regularization, dropout, and so on).*
- *Change the initialization.*
- *Add bias variables.*
- *Change the loss function or the non-linearities (right now it uses squared error and* tanh *to introduce non-linearity).*
- *Use mini-batches of data, possibly with batch normalization.*

Answer: TODO

Okay, that's enough of our hand-coded neural net. Time for some autodiff!

We're going to use PyTorch; see `https://pytorch.org` for installation instructions (the CPU version is fine; if you're having trouble, you could also use Google Colab). This tutorial page is a decent starting place for reference.

[4.2] [3 points] For the provided code in `TorchNeuralNetClassifier(layer_sizes=[3])`, how many parameters are there, and what do they represent?

Hint: `list(thing.parameters())` will get you the parameters of either a layer or a `torch.nn.Module`.

Answer: TODO

[4.3] [4 points] Modify the network to use a more typical classification loss: maximizing the likelihood of a categorical likelihood, aka softmax loss, aka cross-entropy. Hand in the modifications to your code.

You don't need to implement it yourself from scratch – you can use anything from PyTorch here.

Answer: TODO

[4.4] [3 points] PyTorch doesn't have an easy built-in way to compute the loss after going through a `torch.nn.Softmax` layer. Why not? Why does it insist on log inputs? Describe an example where using log-probabilities would give meaningfully different results from "plain" probabilities.

Answer: TODO

[4.5] [7 points] Change `TorchNeuralNetClassifier` to work well on MNIST dataset – that is, at least match logistic regression (error rate 7.5%), but you can probably do better. You'll probably do similar stuff to Question 4. Describe the changes you made and your best test performance.

Answer: TODO

[4.6] [7 points] Change the model in `Convnet.build` to use convolutional layers, while getting test error at least as good as the MLP got. (Feel free to make it less generic, e.g. not supporting arbitrary `layer_sizes` anymore.) Submit your `build()` method, and any other relevant changes, as well as your final test error. (We're probably overfitting a bit at this point, though. . . .)

Hint: To make it a little faster by using your GPU, you can pass `device="cuda"` if you have an appropriate version of PyTorch installed, or `device="mps"` if you're a recent Mac.

[4.7] [3 points] If we use a layer `torch.nn.Conv2d(in_channels=1, out_channels=128, kernel_size=(4, 4))`, what are the resulting parameter shapes, and why? Why don't we need to pass in the shape of the input image?

Answer: TODO