CSCI 4360/6360 Data Science II
Department of Computer Science
University of Georgia

Homework 5: Embeddings All The Way Down

DUE: Tuesday, November 21 by 11:59:59pm

Out November 2, 2023

Questions

This homework assignment wraps up embedding strategies and introduces the fundamentals of neural networks, kernel regression, and a randomized embedding strategy.

This homework requires more coding than previous assignments, so plan accordingly!

1 Neural Networks [25pts]

In this question we'll look at some of the basic properties of feed-forward neural networks.

First, consider the myriad activation functions available to neural networks. In this problem, we'll only look at very simple ones, starting with a combination of linear activation functions and the hard threshold; specifically, where the output of a node y is either 1 or 0:

$$y = \begin{cases} 1 & \text{if } w_0 + \sum_i w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Which of the following functions can be exactly represented by a neural network with one hidden layer, and which uses linear and/or hard threshold activation functions? For each case, *briefly* justify your answer (sketching an example is fine).

[3pts] Polynomials of degree one.

[3pts] Hinge loss $h(x) = \max(1 - x, 0)$.

[3pts] Polynomials of degree two.

[3pts] Piecewise constant functions.

Consider the following XOR-like function in two-dimensional space:

$$f(x_1, x_2) = \begin{cases} 1 & x_1, x_2 \geq 0 \text{ or } x_1, x_2 < 0 \\ -1 & \text{otherwise} \end{cases}$$

We want to represent this function with a neural network. For some reason, we decide we only want to use the threshold activation function for the hidden units and output unit:

$$h_\theta(v) = \begin{cases} 1 & v \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

[13pts] Show that the smallest number of hidden layers needed to represent this XOR function is two. Give a neural network with two hidden layers of threshold functions that represent f, the XOR function. Again, you are welcome to provide a drawing, but that drawing must include values being propagated from each neuron. Alternatively, you could draw a table showing the values at each layer.

2 Kernel Smoothing [35pts]

In this problem, we'll look at nonparametric kernel smoothing for approximating a function from noisy data. We'll also throw in leave-one-out cross-validation to observe its effects on the learned function. For the sake of simplicity, we'll stick with one-dimensional data.

We have a "dataset" $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, as follows:

$$y_i = f(x_i) + \epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$.

The goal of any regression problem is to estimate the true $f(x)$ with an empirical estimate $\hat{f}(x)$. The Nadaraya-Watson estimator is given by:

$$\hat{f}(x_k) = \frac{\sum_{i=1}^n y_i K\left(\frac{|x_i - x_k|}{h}\right)}{\sum_{i=1}^n K\left(\frac{|x_i - x_k|}{h}\right)}$$

where $K(\cdot)$ is the kernel, and h is the bandwidth. In this example, we'll use the Gaussian kernel:

$$K(a) = \frac{1}{\sqrt{2\pi}} \exp\left\{\frac{-a^2}{2}\right\}$$

In this equation, h takes the place of the standard deviation, and the data point x will take the place of the mean.

(yes: you're going to be writing some code!)

[5pts] Write a basic Python program that generates our dataset.

- Sample $x_i \sim U(-5, 5)$ (that's a uniform distribution from -5 to 5)
- Sample $\epsilon_i \sim \mathcal{N}(0, 0.1)$
- Set $y_i = \sin(x_i) + \epsilon_i$

[3pts] Implement a squared loss function $\ell(\cdot)$ (you can use vectorized NumPy arrays for this):

$$\ell(y, \hat{f}(x)) = (y - \hat{f}(x))^2$$

[2pts] Sample a dataset of size $n = 100$ and plot it; you can use `matplotlib.pyplot.scatter`). Overlay the scatter plot with the true regression function (meaning compute the $\sin(\cdot)$ of each x_i and plot that in addition to the y_i you computed); you can use `matplotlib.pyplot.plot`.

[20pts] Now, write a program which performs the following:

- Sample a "training set" of size $n = 100$, and a "testing set" of size $m = 100$.
- Compute the kernel smoother for a particular choice of h, along with the empirical error (average loss between all y and $\hat{f}(x)$), leave-one-out cross-validation error (average loss), and testing error (average loss).
- Compute those measurements for the following values of $h \in \{1.0, 0.75, 0.5, 0.25, 0.1, 0.05, 0.01, 0.005, 0.001\}$.
- Construct scatter plots of test error versus empirical error, and test error versus leave-one-out cross-validation error. Test error should always be on the y-axis.
- Choose the function $\hat{f}$ which minimizes leave-one-out cross-validation error, and plot the training data sample along with the value of this function evaluated on the training data x values.

[5pts] Explain why it is a bad idea to merely minimize the empirical risk in problems like this (*HINT:* refer to the last two plots).

Include your code in a file named `homework5_q2.py` when you submit to AutoLab, as this will be manually inspected. Include the plots in your write-up.

3 Stochastic SVD **[40pts]**

In this question, you'll implement Stochastic SVD (SSVD) and compare its performance in certain applications. You are free to use the `scikit-learn` and `scipy.linalg` libraries.

The strength of SSVD lies in its reliance on randomization to generate an initial basis. In doing so, the rest of the algorithm becomes highly parallelizable; a 2011 PhD thesis proposed this method for computing the SVD of extremely large datasets in a single pass.

Fundamentally, SSVD has two main phases. In the first phase, you are computing a *preconditioner matrix* Q that, when applied to the data matrix A, "conditions" the system such that it is quantitatively better-behaved. Formally, it reduces the *condition number* κ of the linear system, where $\kappa = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}$ ($\lambda_{\max}$ is the largest eigenvalue of A, while $\lambda_{\min}$ is the smallest). In general, this quantity is a strong proxy for "stability" of the corresponding system: if the condition number of a system is small, then it tends to be a smooth, continuous system: small changes in input result in small changes in output. Accordingly, reducing this quantity has all kinds of benefits, chief in particular that makes the system easier to solve (in terms of finding the eigenvalues and eigenvectors, which as we know, SVD is related to that task).

In the second phase, we use our preconditioner Q to compute a small matrix whose singular vectors *approximate* the basis of A, our data matrix. Then, by projecting the vectors back into the space of A using our preconditioner Q, we arrive at an estimate of the true singular vectors of A, and therefore, an estimate of the eigen-decomposition of A.

3.1 1A **[20pts]**

Start by implementing a Python function that computes the preconditioner matrix Q from the data matrix $A \in \mathbb{R}^{n \times m}$ for some number of basis vectors k, where $1 \leq k \leq m$:

1. Create a matrix $\Omega \in \mathbb{R}^{m \times k}$, where $\Omega \sim \mathcal{N}(0, 1)$ (*HINT*: look into the `numpy.random.standard_normal` function).
2. Form the matrix $Y \in \mathbb{R}^{n \times k}$ from the product $A\Omega$.
3. Perform a QR decomposition of Y. This creates two matrices: Q, which is orthogonal and unitary (and our $n \times k$ preconditioner!), and R, and upper-triangular matrix that we actually don't need (*HINT*: look into the `scipy.linalg.qr` function).

Next, implement the SSVD itself using our preconditioner Q and our data matrix A:

1. Precondition the system by forming the matrix $B \in \mathbb{R}^{k \times m}$ from the product $Q^T A$.
2. Perform the SVD (can be "truncated") of $BB^T = \hat{U}\Sigma^2\hat{U}^T$.
3. We can then extract the left singular vectors of A as $U = Q\hat{U}$.

Use the provided functions in the handout script to load the data, as well as to write out the U singular vectors and Σ singular values (the latter as a 1D array of numbers).

3.2 1B **[10pts]**

A potentially fatal flaw in this SSVD formulation is that, by relying on randomization in $\Omega \in \mathbb{R}^{m \times k}$, we are creating a matrix with rank *at most* k, our desired number of dimensions. However, thanks to the inherent stochasticity, it's very likely that on any given draw our Ω may actually have a rank that is smaller, ultimately culminating in estimated singular vectors of A that are pure noise.

To mitigate this, we can *oversample* in creating Ω. Return to the code you wrote in the first part and, for whatever k is provided as the target dimensionality, include an oversampling parameter p, where the dimensionality of Ω is $\mathbb{R}^{m \times (k+p)}$. We are still targeting a rank-k decomposition, but we are simply oversampling in this one step to greatly enhance our chances that Ω is, in fact, k-rank.

You can implement this using the `-p` command-line option that is already implemented in the sample script.

3.3 1C **[10pts]**

Another way of stabilizing SSVD beyond oversampling is to perform *power iterations* during the orthogonalization step of the preconditioner computations. After computing the initial Q matrix from the QR decomposition, but before applying it to compute B, some number of power iterations q are applied to the system to "refine" the preconditioner Q.

Each power iteration i consists of two discrete steps:

1. Form the product $Y = AA^T Q_{i-1}$
2. Re-run the QR decomposition to find Q_i using Y from the first step

BONUS [5pts] How does the spectrum of singular values deviate from those of, say, the built-in `scipy` SVD solver? Is there a pattern in the deviations–something systemic–or are they random?

BONUS [15pts] Prove your assertion in the previous bonus question.

Administration

1 Script Design

Your code should be able to process: an input file containing the $N \times M$ matrix, the number of SVD components k, oversampling parameter p and number of power iterations

q, random seed r, and an output directory to write the stochastic singular vectors and values. Most of these parameters are optional, except for the input and output flags.

You'll also be provided the boilerplate to read in the necessary command-line parameters:

1. `-i`: a file path to a text file containing the data
2. `-k`: number of SVD components to take in the decomposition
3. `-p`: oversampling rate for generating the random basis Ω
4. `-q`: number of power iterations to perform on the preconditioner Q
5. `-r`: random seed to use (for debugging)
6. `-o`: a filesystem path to an output directory, where the singular values and vectors will be written

The format of the input file will be whitespace-delimited, where a single row of the input matrix will be on one line, and individual values are separated by whitespace. You can use the provided utility functions in the boilerplate script, `_save_data` and `_load_data`, to save outputs and load inputs respectively. These functions are already written in the `homework5_q3.py-TEMPLATE` file that will handle reading in data and parsing command-line arguments.

The format of the output file should be two separate files: one containing the k singular vectors (in identical format to the input, with one row of the matrix $\hat{U}$ per line), and one containing the k singular values (one number per line). You can very easily test how well your SSVD is doing–you can use the built-in SciPy SVD solver. The autograder on AutoLab has been scaled so that SSVD with the properties mentioned in each subproblem should receive full credit, if implemented correctly.

2 Submitting

All submissions will go to **AutoLab**. You can access AutoLab at:

- https://autolab.cs.uga.edu

You can submit deliverables to the **Homework 5** assessment that is open. When you do, you'll submit **three** files:

1. `homework5_q2.py`: the Python script that implements kernel smoothing
2. `homework5_q3.py`: the Python script that implements SSVD
3. `homework5.pdf`: the PDF write-up with any questions that were asked (figures from Q2 can be embedded here)

These should be packaged together in a tarball; the archive can be named whatever you want when you upload it to AutoLab, but the files in the archive should be named **exactly** what is above. Deviating from this convention could result in my annoyance (and autograder failures, but let's be honest the former is the more important)!

To create the tarball archive to submit, run the following command (on a *nix machine):

```
> tar cvf homework5.tar homework5_q2.py homework5_q3.py homework5.pdf
```

This will create a new file, `homework5.tar`, which is basically a zip file containing your Python scripts and PDF write-up. Upload the archive to AutoLab. There's no penalty for submitting as many times as you need to, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on November 21, so give yourself plenty of time! A late submission because the server got hammered at the deadline will *not* be acceptable (there is a *small* grace period to account for unusually high load at deadline, but I strongly recommend you avoid the problem altogether and start early).

Also, to save time while you're working on the coding portion, you are welcome to create a tarball archive of just the Python script and upload that to AutoLab. Once you get the autograder score you're looking for, you can then include the PDF in the folder, tarball everything, and upload it. AutoLab stores the entire submission history of every student on every assignment, so your autograder (code) score will be maintained and I can just use your most recent submission to get the PDF.

3 Reminders

- If you run into problems, ping the `#questions` room of the Discord server. If you still run into problems, ask me. But please please please, **do NOT** ask Google to give you the code you seek! I will be on the lookout for this (and already know some of the most popular venues that might have solutions or partial solutions to the questions here).
- Prefabricated solutions are NOT allowed, unless of course they are specifically allowed!
- If you collaborate with anyone or anybot, just mention their names in a code comment and/or at the top of your homework writeup.
- Cite any external and/or non-course materials you referenced in working on this assignment.