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

Assignment 2: Visual Data Science

DUE: Tuesday, September 24 by 11:59:59pm

Out September 10, 2019

Questions

This homework assignment incorporates our topics on computer vision, as well as more theory of machine learning.

1 Law of Large Numbers [15pts]

As explained in class, the Hoeffding inequality has the form:

$$P[|\mu_{emp} - \mu| > \epsilon] \leq 2e^{-2\epsilon^2 N} \tag{1}$$

and provides a probabilistic bound on how often the empirical mean μ_{emp} of some sample of size N deviates form the true mean μ of some random process by more than a certain tolerance ϵ.

The importance of this inequality comes from its uniform (non-asymptotic) bound (i.e., N does not need to approach ∞ for it to be true).

[10pts] Design and implement an experiment showing this inequality holds in practice. Show that for means of samples from a random variable $X \sim Bernoulli(p = 0.6)$ abide by the Hoeffding inequality over a large range of sizes ($N = 1$ to 1000), with a fixed tolerance of $\epsilon = 0.1$. You can provide pseudocode in your write-up as long as it is *provably* correct. One way to do that would be to provide full code (e.g., Python) embedded in your write-up.

[5pts] Plot your trials against the bound provided by the inequality. You will need to run the same trial for each N multiple times (e.g., 10) and take the fraction of times of deviation as your $P[|\mu_{emp} - \mu| > \epsilon]$. Include this plot in your write-up. Make sure you are plotting both the fraction of deviations AND the bound for each N, as a function of the current iteration.

2 Generalization Bounds [20pts]

In Probably Approximately Correct (PAC) Learning, concentration bounds like Eq(1) play a pivotal role for providing guarantees of generalising to unseen data-points. The inequality provides bounds on how a certain candidate solution $\mathbf{w}$ will behave over the entire domain of data-points compared to how many you tested on (N).

$$P[|\mathcal{R}_{emp}(\mathbf{w}) - \mathcal{R}(\mathbf{w})| > \epsilon] \leq 2e^{-2\epsilon^2 N} \tag{2}$$

$$\mathcal{R}_{emp}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} loss(f_{\mathbf{w}}(x), y_i)$$

$$\mathcal{R}(\mathbf{w}) = \int loss(f_{\mathbf{w}}(x), y) dP(x, y)$$

where $\mathcal{R}_{emp}(\mathbf{w})$ represents the error on the finite set of points N available to you, and $\mathcal{R}(\mathbf{w})$ is the error on all possible data-points in the domain (unavailable to you). $f_{\mathbf{w}}$ is your classifier/regression model parametrized by $\mathbf{w}$.

[15pts] Inequality 2 provides bounds for a specific $\mathbf{w}$. However, in data science techniques, we will be searching for a candidate solution over a whole family of solutions. If the family of solutions has 3 possible candidate solutions $\mathbf{w}_1$,$\mathbf{w}_2$,$\mathbf{w}_3$, what will be a bound over ANY of $\mathcal{R}_{emp}(\mathbf{w}_1)$, $\mathcal{R}_{emp}(\mathbf{w}_2)$, $\mathcal{R}_{emp}(\mathbf{w}_3)$ deviating from $\mathcal{R}(\mathbf{w}_1)$, $\mathcal{R}(\mathbf{w}_2)$, $\mathcal{R}(\mathbf{w}_3)$ respectively?

In other words, provide a tight bound for:

$$P[|\mathcal{R}_{emp}(\mathbf{w}_1) - \mathcal{R}(\mathbf{w}_1)| > \epsilon \cup |\mathcal{R}_{emp}(\mathbf{w}_2) - \mathcal{R}(\mathbf{w}_2)| > \epsilon \cup |\mathcal{R}_{emp}(\mathbf{w}_3) - \mathcal{R}(\mathbf{w}_3)| > \epsilon]$$

[5pts] What does that mean for a family that has infinite candidate solutions (e.g., all possible weights and biases for a logistic regression classifier)?

3 Linear Dynamical Systems [25pts]

Linear dynamical systems (LDS) are multidimensional time series models with two components: an appearance component, and state component, that are used to model and identify dynamic textures. Dynamic textures are video sequences that display spatial regularities over time, regularities we want to capture while simultaneously retaining their temporal coherence.

The appearance component is a straightforward application of dimensionality reduction, projecting the original temporal state into a low-dimensional "state space":

$$\vec{y}_t = C\vec{x}_t + \vec{u}_t$$

where $\vec{y}_t$ is the current frame, $\vec{x}_t$ is the corresponding "state," $\vec{u}_t$ is white noise, and C is the matrix that maps the appearance space to the state space (and vice versa), sometimes referred to as the *output matrix*: it defines the appearance of the model.

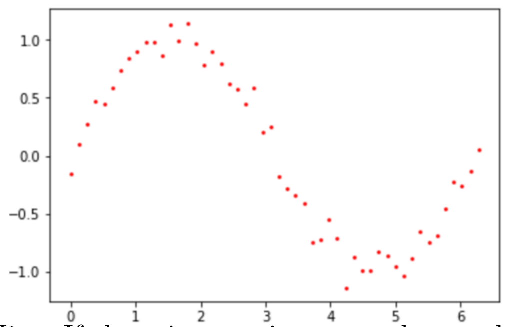
The state component is nothing more than an autoregressive model, a linear combination of Markov assumptions that model the movement of the system in the low-dimensional state space:

$$\vec{x}_t = A\vec{x}_{t-1} + W\vec{v}_t$$

where $\vec{x}_t$ and $\vec{x}_{t-1}$ are the positions of the model in the state space at times t and $t-1$ respectively, $W\vec{v}_t$ is the *driving noise* at time t, and A is the transition between states that defines how the model moves.

[5pts] In the following questions, we're going to use only the state component of the LDS (i.e., we'll only use the second equation to model motion). How could we formalize "ignoring" the appearance component? What values could we use in the appearance component so that the original data $\vec{y}_t$ is also our state space data $\vec{x}_t$?

[10pts] To simplify, let's ignore the appearance component and focus on a toy example in two dimensions.



Suppose each $\vec{x}_t$ is an (x, y) pair from the plot. Set up the equations to solve for A (Note: your solution should generalize to n 2D points. Also, you can assume there is no noise term (i.e. $W\vec{v}_t = 0$)).

Hint: If there is no noise term, then each $\vec{x}_t$ can be written as $A\vec{x}_{t-1}$ for all t. Write a few of these out, then organize them into systems of equations.

[10pts] An interesting property of any model is its behavior in the limit. Those familiar with certain dimensionality reduction strategies will notice the simplified autoregressive model from the previous step looks an awful lot like a power iteration for finding approximate eigenvalues and eigenvectors of a matrix: if M is your matrix of interest, you can iteratively update a vector $\vec{v}$ such that $\vec{v}_{n+1} = M\vec{v}_n$, and each additional iteration will bring $\vec{v}$ closer to the true leading eigenvector of M.

Assuming M is invertible, we have the full eigen-decomposition of $M = U\Sigma U^T$, where U is the matrix of eigenvectors $[\vec{u}_1, ..., \vec{u}_n]$, and Σ is the diagonal matrix of eigenvalues $\lambda_1, ..., \lambda_n$ sorted in descending order $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$.

Write out the equation for $\vec{v}_{n+2}$ using **only** M and $\vec{v}_t$. Do the same for $\vec{v}_{t+3}$. Describe how this generalizes for n steps. What is happening in terms of M?

[10pts] Now, rewrite those same equations, but instead of M, use its eigen-decomposition form. What happen as the number of iterations $n \to \infty$? What does this mean if there are eigenvalues $\lambda_i < 1$? What if $\lambda_i = 1$? What if $\lambda_i > 1$? What is therefore happening to the corresponding eigenvectors $\vec{u}_i$ of λ_i? (Note: $n \to \infty$ is known as the *steady state*)

Hint: The eigenvector matrix U has the property $U^TU = UU^T = I$, where I is the identity matrix.

4 Coding [40pts]

In this question, you'll implement a basic LDS to model some example dynamic textures.

You'll be allowed the following external Python functions: `scipy.linalg.svd` for computing the appearance model (output matrix) C, and `scipy.linalg.pinv` for computing the pseudo-inverse of the state-space observations for deriving the transition matrices. **No other external packages or SciPy functions will be allowed** (besides NumPy of course).

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

1. `-f`: a file path to a NumPy array file (the dynamic texture)
2. `-q`: an integer number of dimensions for the state-space
3. `-o`: a file path to an output file, where the prediction will be written

Your code will read in the NumPy array representing a dynamic texture video; it will have dimensions `frames` $\times$ `height` $\times$ `width`. We'll call this M, and say it has shape $f \times h \times w$. From there, you'll need to derive the parameters of the LDS: the appearance model C and the state space data X (both can be derived by performing a singular value decomposition on Y, which is formed by stacking all the pixel trajectories of M as rows of Y). Once you've learned C and X, you can learn the transition matrix A using the pseudo-inverse:

$$A = X_2^f (X_1^{f-1})^\diamond$$

where X_1^{f-1} is a matrix of $\vec{x}_1$ through $\vec{x}_{f-1}$ stacked as rows, X_2^f is a matrix of $\vec{x}_2$ through $\vec{x}_f$ stacked as rows, and $D^\diamond = D^T(DD^T)^{-1}$ is the pseudo-inverse of D.

Once you've learned C, X, and A, use these parameters to **simulate one time step**, generating $\vec{x}_{f+1}$. Use C to project this simulated point into the appearance space, generating $\vec{y}_{f+1}$. Reshape it to be the same size as the original input sequence (i.e., $h \times w$), and then **write the array to the output file**. You can use the `numpy.save` function for this. **Any other program output will be ignored.**

[BONUS: 10pts] Re-formulate your LDS implementation so that it also learns $W\vec{v}_t$, the driving noise parameter in the state space model. Recall that this first relies on the one-step prediction error:

$$\vec{p}_t = \vec{x}_{t+1} - A\vec{x}_t$$

which is used to compute the covariance matrix of the driving noise:

$$Q = \frac{1}{f-1} \sum_{1}^{f-1} \vec{p}_t \vec{p}_t^T$$

Perform a singular value decomposition of $Q = U\Sigma V^T$, set $W = U\Sigma^{1/2}$, and $\vec{v}_t \sim \mathcal{N}(0, I)$.

Implement the same 1-step prediction as before, this time with the noise term, so the state-space prediction is done as $\vec{x}_{t+1} = A\vec{x}_t + W\vec{v}_t$. Does your accuracy improve? Why do you think this is the case?

[BONUS: 30pts] Re-formulate your LDS so that your state space model is a second-order autoregressive process. That is, your model should now be:

$$\vec{x}_{t+1} = A_1\vec{x}_t + A_2\vec{x}_{t-1}$$

Learning the transition matrices A_1 and A_2 is conceptually the same as before, but implementation-wise requires considerably more ingenuity to implement, so if you need help, please come to office hours!

Implement the same 1-step prediction as before, this time with the second-order model. Does your accuracy improve? Why do you think this is the case?

(it doesn't matter if you include the driving noise from the first bonus question here or not; as in, you don't have to implement the first bonus question to also get this one)

Administration

1 Submitting

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

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

You can submit deliverables to the **Assignment 2** assessment that is open. When you do, you'll submit two files:

1. `assignment2.py`: the Python script that implements your algorithms, and
2. `assignment2.pdf`: the PDF write-up with any questions that were asked

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 the autograder failing!

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

```
> tar cvf assignment2.tar assignment2.py assignment2.pdf
```

This will create a new file, `assignment2.tar`, which is basically a zip file containing your Python script 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 September 24, 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.

2 Reminders

- If you run into problems, ping the `#questions` room of the Slack chat. 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 (e.g. `scikit-learn`, OpenCV) are NOT allowed! You have to do the coding yourself!
- If you collaborate with anyone, just mention their names in a code comment and/or at the top of your homework writeup.