Citations:
- https://towardsdatascience.com/probability-concepts-explained-maximum-likelihood-estimation-c7b4342fdbb1
- https://towardsdatascience.com/decomposing-eigendecomposition-f68f48470830
- ChatGPT

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

Homework 2: Visual Data Science

Matthew Lawson (mtl41232)

DUE: Tuesday, September 26 by 11:59:59pm

Out September 7, 2023

Questions

This homework assignment incorporates our topics on computer vision, as well as covering linear models.

1 Linear Regression [20pts]

Assume we are given n training examples $(\sim x_1, y_1), (\sim x_2, y_2), \ldots, (\sim x_n, y_n)$, where each data point $\sim x_i$ has m real-valued features (i.e., $\sim x_i$ is m-dimensional). The goal of regression is to learn to predict y from $\sim x$, where each y_i is also real-valued (i.e. continuous).

The linear regression model assumes that the output Y is a linear combination of input features X plus noise terms from a given distribution, with weights on the input features given by β.

We can write this in matrix form by stacking the data points $\sim x_i$ as rows of a matrix X, such that x_{ij} is the j-th feature of the i-th data point. We can also write Y , β, and as column vectors, so that the matrix form of the linear regression model is:

$$Y = X\beta + \epsilon$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix}, \text{and } X = \begin{bmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_n \end{bmatrix},$$

and where $\vec{x}_i = \begin{bmatrix} x_1, x_2, ..., x_n \end{bmatrix}.$

Linear regression seeks to find the parameter vector β that provides the best fit of the above regression model. There are lots of ways to measure the goodness of fit; one criteria is to find the β that minimizes the squared-error loss function:

$$J(\beta) = \sum_{i=1}^{n} (y_i - \vec{x}_i^T \beta)^2,$$

or more simply in matrix form:

$$J(\beta) = (X\beta - Y)^T(X\beta - Y), \text{ which} \quad (1)$$

can be solved directly under certain circumstances:

$$\hat{\beta} = (X^T X)^{-1} X^T Y \quad (2)$$

(recall that the "hat" notation $\hat{\beta}$ is used to denote an *estimate* of a true but unknown–and possibly, unknow*able*–value)

When we throw in the error term, assuming it is drawn from independent and identically distributed ("i.i.d.") Gaussians (i.e.,$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$), then the above solution is also the MLE estimate for $P(Y|X;\beta)$.

All told, then, we can make predictions $\hat{Y}$ using $\hat{\beta}$ (X could be the training set, or new data altogether):

$$\hat{Y} = X\hat{\beta} + \epsilon$$

Now, when we perform least squares regression, we make certain idealized assumptions about the vector of error terms , namely that each $_i$ is i.i.d. according to $N(0,\sigma^2)$ for some value of σ. In practice, these idealized assumptions often don't hold, and when they fail, they can outright implode. An easy example and inherent drawback of Gaussians is that they are sensitive to outliers; as a result, noise with a "heavy tail" (more weight at the ends of the distribution than your usual Gaussian) will pull your regression weights toward it and away from its optimal solution.

In cases where the noise term $_i$ can be arbitrarily large, you have a situation where your linear regression needs to be *robust* to outliers. Robust methods start by weighting each observation *unequally*: specifically, observations that produce large residuals are downweighted.

[15pts] In this problem, you will assume $\epsilon_1, \ldots, \epsilon_n$ are i.i.d. drawn from a Laplace distribution (rather than N(0,σ^2)); that is, each $\epsilon_i \sim \mathrm{Lap}(0, b)$, where Lap$(0, b) = \frac{1}{2b}\exp(-\frac{|\epsilon_i|}{b})$.

Derive the loss function $J_{\mathrm{Lap}}(\beta)$ whose minimization is equivalent to finding the MLE of β under the above noise model.

Hint #1: Recall the critical point above about the form of the MLE; start by writing out $P(Y_i|X_i;\beta)$.

Hint #2: Logarithms nuke pesky terms with exponents without changing linear relationships.

Hint #3: Multiplying an equation by -1 will switch from "argmin" to "argmax" and vice versa.

We need to find the equation of the likelihood function of observing our data given β as our parameter, which we will eventually maximize. So starting with P($Y_i|X_i$;β), we need to use the Laplace distribution to find the probability of our data Y_i for X_i & β parameters. We are given the Laplace distribution with a mean of 0 formula as:

$$\mathrm{Lap}(0, b) = \frac{1}{2b}\exp(-\frac{|\epsilon_i|}{b})$$

Since we know the mean to be the product of vectors X_i and β we can basically plug into the formula knowing that the error term e_i is the difference between Y_i and our mean. Thus:

$$\mathrm{P}(Y_i|X_i;\beta,\mathrm{b}) = \frac{1}{2b}\exp\left(-\frac{|Y_i - X_i * \beta|}{b}\right)$$

Now we must find the likelihood function for our entire dataset as the product of all of the individual probabilities within our dataset. Therefore, the likelihood function should look like:

$$\mathrm{P}(Y_i|X_i;\beta,\mathrm{b}) = argmax \prod_{i=1}^{n} \frac{1}{2b}\exp\left(-\frac{|Y_i - X_i * \beta|}{b}\right)$$

Where we want to solve for the maximum probability of Y given our data X.

From here we can take the natural log of both sides to reduce the product on the right hand side to a summation using the Product Rule of logarithms (log(AB) = log(A) + log(B)):

$$\ln(\mathrm{P}(Y_i|X_i;\beta,\mathrm{b})) = \ln\left(argmax \prod_{i=1}^{n} \frac{1}{2b}\exp\left(-\frac{|Y_i - X_i * \beta|}{b}\right)\right)$$

Which turns into the sum of the term for each i:

$$\ln\left(\mathrm{P}(Y_i|X_i;\beta,\mathrm{b})\right) = argmax\sum_{i=1}^{n}\ln\left(\left(\frac{1}{2b}\right)\exp\left(-\frac{|Y_i - X_i * \beta|}{b}\right)\right)$$

However, within each term we can use the Power Rule (ln(A^B) = B * ln(A)) to simplify each term within the equation. The result is the following equation:

$$\ln\left(\mathrm{P}(Y|\mathrm{X};\beta,\mathrm{b})\right) = argmax\sum_{i=1}^{n}\ln\left(\frac{1}{2b}\right) * \left(-\frac{|Y_i - X_i * \beta|}{b}\right))$$

This is effectively the loss function "J" for this model given β parameter and b scale factor for the distribution which we can rewrite as:

$$\mathrm{J}(\beta,\mathrm{b}) = argmax\sum_{i=1}^{n}\ln\left(\frac{1}{2b}\right) * \left(-\frac{|Y_i - X_i * \beta|}{b}\right))$$

This again can be simplified as we know we want to *minimize* the loss function so as hinted, we can multiply argmax by -1. As well, because we are minimizing the function, the $\ln\left(\frac{1}{2b}\right)$ and and $\left(\frac{-1}{b}\right)$constants can be "nuked" as they have no impact on the relative minimum of the function. So:

$$\mathrm{J}(\beta,\mathrm{b}) = (-1) * argmax\sum_{i=1}^{n}\ln\left(\frac{1}{2b}\right) * \left(-\frac{|Y_i - X_i * \beta|}{b}\right))$$

Turns into:

$$\mathrm{J}(\beta) = argmin\sum_{i=1}^{n}(|Y_i - X_i * \beta|)$$

You can square the errors to get the mean squared error used in linear regression described above:

$$\mathrm{J}(\hat{\beta}) = argmin\sum_{i=1}^{n}(Y_i - X_i * \beta)^2$$

Where $\widehat{\beta}$ is the model parameter that minimizes the mean-squared error loss function.

[5pts] Why do you think the above model provides a more robust fit to data compared to the standard model assuming the noise terms are distributed as Gaussians? Be specific!

Using a Laplace distribution for errors can provide a more robust fit than a Gaussian distribution due to its larger tails. A gaussian distribution assigns rather low probabilities to either tail (as evident by very thin tails when viewing a graphed distribution). So when outliers are observed, the mean of the distribution is affected more strongly. In comparison, the larger tails on the Laplace distribution help negate this effect to some degree as increased probabilities assigned to the tails constitute more of the mean of the distribution. Thus, the Laplace distribution accounts for more values to occur at the tails, so in calculating a mean, an extreme value has less of an effect.

2 Linear Dynamical Systems [40pts]

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":

$$\sim y_t = C \sim x_t + \sim u_t$$

where $\sim y_t$ is the current frame, $\sim x_t$ is the corresponding "state," $\sim u_t$ 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:

$$\sim x_t = A \sim x_{t-1} + W \sim v_t$$

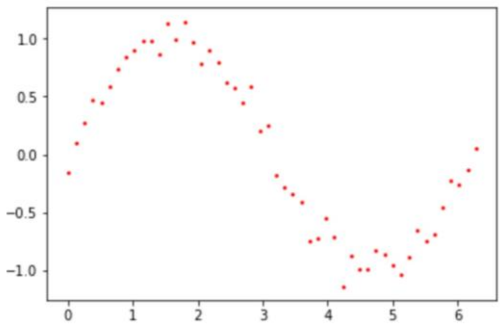
where $\sim x_t$ and $\sim x_{t-1}$ are the positions of the model in the state space at times t and $t-1$ respectively, $W \sim v_t$ is the *driving noise* at time t, and A is the transition between states that defines how the model moves.

[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 $\sim y_t$ is also our state space data $\sim x_t$?

To formalize "ignoring" the appearance component you could just set it equal to some constant value. In that case you would essentially just be ignoring any possible difference in appearance across the available time series data (frames if multiple images).

To set the original data equal to our state space data within our appearance component, you could set q equal to the h x w value of the image so that flattened image vector and state space vectors are the same length. Then the two vectors would be equivalent.

[15pts] To simplify, let's ignore the appearance component and focus on a toy example in two dimensions.



Suppose each $\sim x_t$ 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. $W\sim v_t = 0$)).

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

Each x_t is a vector containing two values corresponding to the points on an (x,y) graph.

$x_1 = A[x_0, \quad y_0]$

$x_2 = A[x_1, \quad y_1]$

$x_3 = A[x_2, \quad y_2]$

$x_t = A\sim x_{t-1}$

Therefore if Xt is a matrix of t rows of 2D vectors, we can solve for the :

A = (x_t) * $(x_{t-1})^{-1}$

[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 $\sim v$ such that $\sim v_{t+1} = M\sim v_t$, and each additional iteration will bring $\sim v$ 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 $[\sim u_1,\ldots,\sim u_n]$, and Σ is the diagonal matrix of eigenvalues $\lambda_1,\ldots,\lambda_n$ sorted in descending order $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$.

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

We know that $V_{t+2} = M * V_{t+1}$ so to get V_{t+2} in terms of M and V_t we can plug in V_{t+1} in those terms. Which would give us:

$V_{t+2} = M * (M * V_t)$

One more iteration and it would give us V_{t+3}:

$V_{t+3} = M(M * (M * V_t))$

So after a few iterations we can tell that we are essentially raising the power of M for each iteration, using the first V value in the time series as the reference point for the iterations. Thus you could generalize to n steps/iterations with the following:

$V_{t+n} = M^n * (V_t)$

Each step we are effectively raising M to one higher power.

[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 $\sim u_i$ 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.

Using the $\mathrm{M} = U\Sigma U^T$ formula, we can substitute for M:

$V_{t+2} = U\Sigma U^T * (U\Sigma U^T * V_t)$ **OR** $V_{t+2} = U\Sigma U^T U\Sigma U^T * V_t$

$V_{t+3} = U\Sigma U^T * U\Sigma U^T * (U\Sigma U^T * V_t)$ **OR** $V_{t+3} = U\Sigma U^T U\Sigma U^T U\Sigma U^T * V_t$

$V_{t+n} = (U\Sigma U^T)^n * (V_t)$

Considering the property described above, we can cancel out the U^TU terms given that they produce the I identity matrix, which when multiplied by the first U term, just produces U. Then we are left with:

$$V_{t+2} = U\Sigma\Sigma U^T * V_t$$

$$V_{t+3} = U\Sigma\Sigma\Sigma U^T * V_t$$

Which can again be reduced to:

$$V_{t+2} = U\Sigma^2 U^T * V_t$$

$$V_{t+3} = U\Sigma^3 U^T * V_t$$

Lastly, using these two examples, we can rewrite the generalized equation as:

$$V_{t+n} = \mathrm{U}\Sigma^n U^T * \ V_t$$

As $n \to \infty$, the vector being updated changes according to the nature of Σ. If you consider the eigenvalue as the multiple of the eigenvector equal to the product of your matrix and the eigenvector, then you can understand the above equation to represent the compounding of the diagonal matrix of eigenvalues an infinite number of times. So, we can then rephrase the following conditions for eigenvalues as the infinite compounding of a normal scalar value.

If $\lambda_i < 1$, then as you infinitely compound any value less than 1, you approach zero. As such, the vector V_t in the above equation would also approach zero.

If $\lambda_i > 1$, then the vector V_t would approach infinity as n approaches infinity.

If $\lambda_i = 1$, then the vector V_t should stay the same.

3 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 × height × 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 $\vec{v}_t \sim \mathrm{N}(0, I)$. 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 secondorder 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 Homework 2 assessment that is open. When you do, you'll submit two files:

1. homework2.py: the Python script that implements your algorithms, and
2. homework2.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 homework2.tar homework2.py homework2.pdf
```

This will create a new file, homework2.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 delayed; AutoLab is programmed to implement late penalties *promptly* at midnight after the deadline, 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 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 (e.g. scikit-learn, OpenCV) are NOT allowed! You have to do the coding yourself!
- If you collaborate with anyone or anybot, just mention their names in a code comment and/or at the top of your homework writeup.