Citations:

- https://www.analyticsvidhya.com/blog/2021/11/study-of-regularization-techniques-of-linear-model-and-its-roles/
- https://towardsdatascience.com/understanding-regularization-in-machine-learning-d7dd0729dde5
- https://towardsdatascience.com/spectral-clustering-aba2640c0d5b
- ChatGPT

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

Homework 3: I Heard You Like Math

Matthew Lawson (mtl41232)

DUE: Tuesday, October 10 by 11:59:59pm

Out September 28, 2023

Questions

This homework assignment focuses on machine learning theory, linear and ensemble models, and graph theoretic methods.

1 Regularization in Linear Regression [35pts]

Recall our high-dimensional linear regression equation from the previous homework assignment, where we needed to find the β that minimized 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:

$$J(\beta) = (X\beta - Y)^T(X\beta - Y), \quad (1)$$

When the number of features m is much larger than the number of training examples n, or very few of the features are non-zero (as we saw in Homework 1), the matrix $X^T X$ is not full rank, and therefore cannot be inverted. This wasn't a problem for logistic regression which didn't have a closed-form solution anyway; for "vanilla" linear regression, however, this is a show-stopper.

Instead of minimizing our original loss function $J(\beta)$, we minimize a new loss function $J_R(\beta)$ (where the R is for "regularized" linear regression):

$$J_R(\beta) = \sum_{i=1}^{n} (Y_i - X_i^T \beta)^2 + \lambda \sum_{j=1}^{m} \beta_j^2 ,$$

which can be rewritten as:

$$J_R(\beta) = (X\beta - Y)^T (X\beta - Y) + \lambda ||\beta||^2 \qquad (2)$$

[5pts] Explain what happens as $\lambda \to 0$ and $\lambda \to \infty$ in terms of J, J_R, and β.

By changing the size of the lambda term you add a penalty of sorts to the error function that is proportional to the lambda ratio multiplied by the model parameters β. So, you're basically penalizing more complex models (with greater model parameters β) via a fixed ratio lambda, as a function of the complexity (size of β). By only optimizing J, the loss function is incentivized to find the smallest squared error, which may capture relationships that don't translate beyond the training dataset. The result of implementing the "$+\lambda \sum_{j=1}^{m} \beta_j^2$" should be that we should get an optimized model $J(\beta)$ that contains smaller parameters β, hopefully preventing the model from learning parameters that overfit the model to the training data.

***Long winded explanation to make sure I understand topic, answer below:**

So therefore:

As the lambda parameter approaches 0, the $J_R(\beta)$ model should approach the normal unregularized model $J(\beta)$. The $J_R(\beta)$ model is essentially just the $J(\beta)$ model with the added component " $+\lambda \sum_{j=1}^{m} \beta_j^2$ ", and as λ approaches 0, the "penalty" term approaches 0 as well, which when dropped leaves us with the original non-regularized model $J(\beta)$. The size of parameters/complexity of model, has no effect on $J_R(\beta)$.

As the lambda parameter approaches ∞, the penalty term would also approach ∞. This basically equates to an infinite value for our penalty term. Thus, in order to minimize the $J_R(\beta)$ loss function, you would have to make β approach 0. If β is zero,

you again get left with the original non-regularized model $J(\beta)$. The difference being that your model parameters are 0, and thus underfitting occurs as your model did not learn any relationships that can be mapped to non-training data.

[10pts] Rather than viewing β as a fixed but unknown parameter (i.e. something we need to solve for), we can consider β as a random variable. In this setting, we can specify a prior distribution $P(\beta)$ on β that expresses our prior beliefs about the types of values β should take. Then, we can estimate β as:

$$\beta_{MAP} = argmax_{\beta} \prod_{i=1}^{n} P(Yi|Xi;\beta)P(\beta), \qquad (3)$$

where MAP is the *maximum a posteriori* estimate.

(aside: this is different from the MLE, which is the frequentist strategy for solving for a parameter. think of MAP as the Bayesian version.)

Show that maximizing Equation 3 can be expressed as *minimizing* Equation 2 with the assumption of a Gaussian prior on β, i.e. $P(\beta) \sim N(0, I\sigma^2/\lambda)$. In other words, show that the L_2-norm regularization term in Equation 2 is effectively imposing a Gaussian prior assumption on the parameter β.

Hint #1: Start by writing out Equation 3 and filling in the probability terms.

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.

Starting with:

$$\boldsymbol{\beta_{MAP} = argmax_{\beta} \prod_{i=1}^{n} P(Yi|Xi;\beta)P(\beta),}$$

We know that $P(Y_i|X_i;\boldsymbol{\beta})$ is the probability of Y_i given X_i and $\boldsymbol{\beta}$, and that $P(\boldsymbol{\beta}) \sim N(0, I\sigma^2/\lambda)$, we can substitute and rewrite the individual components

$$\mathbf{P}(\boldsymbol{Y_i}|\boldsymbol{X_i};\boldsymbol{\beta}) = \boldsymbol{exp}\left(-\frac{1}{2} * \frac{(\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2}{\sigma^2}\right) \text{ and } \mathbf{P}(\boldsymbol{\beta}) = \boldsymbol{exp}\left(-\frac{1}{2} * \frac{\boldsymbol{\beta}^T\boldsymbol{\beta}}{\frac{\sigma^2}{\lambda}}\right)$$

$P(Y_i|X_i;\boldsymbol{\beta})$ term is the gaussian version of likelihood from the last homework. $P(\boldsymbol{\beta})$ is the form of the gaussian prior for $\boldsymbol{\beta}$.

Then plug those terms into equation 3:

$$\boldsymbol{\beta_{MAP}} = \boldsymbol{argmax_\beta} \prod_{i=1}^{n} \boldsymbol{exp}(-\frac{1}{2} * \frac{(\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2}{\boldsymbol{\sigma^2}})\, \boldsymbol{exp}(-\frac{1}{2} * \frac{\boldsymbol{\beta^T \beta}}{\frac{\boldsymbol{\sigma^2}}{\boldsymbol{\lambda}}}),$$

Now we can perform similar steps from the last homework to again derive the loss function, but with the regularization term. First, use Product Rule of logs to reduce to a summation:

$$\boldsymbol{ln}(\boldsymbol{\beta_{MAP}}) = \boldsymbol{ln}(\boldsymbol{argmax_\beta} \prod_{i=1}^{n} \boldsymbol{exp}(-\frac{1}{2} * \frac{(\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2}{\boldsymbol{\sigma^2}})\, \boldsymbol{exp}(-\frac{1}{2} * \frac{\boldsymbol{\beta^T \beta}}{\frac{\boldsymbol{\sigma^2}}{\boldsymbol{\lambda}}}))$$

$$\boldsymbol{ln}(\boldsymbol{\beta_{MAP}}) = \boldsymbol{argmax_\beta} \sum_{i=1}^{n} \boldsymbol{ln}(\boldsymbol{exp}(-\frac{1}{2} * \frac{(\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2}{\boldsymbol{\sigma^2}})\, \boldsymbol{exp}(-\frac{1}{2} * \frac{\boldsymbol{\beta^T \beta}}{\frac{\boldsymbol{\sigma^2}}{\boldsymbol{\lambda}}}),)$$

Then, nuke exponents:

$$\boldsymbol{ln}(\boldsymbol{\beta_{MAP}}) = \boldsymbol{argmax_\beta} \sum_{i=1}^{n} \left(-\frac{1}{2} * \frac{(\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2}{\boldsymbol{\sigma^2}}\right) + (-\frac{1}{2} * \frac{\boldsymbol{\beta^T \beta}}{\frac{\boldsymbol{\sigma^2}}{\boldsymbol{\lambda}}}),)$$

We can drop the constants and pull out the denominators:

$$\boldsymbol{ln}(\boldsymbol{\beta_{MAP}}) = \boldsymbol{argmax_\beta} \sum_{i=1}^{n} \left((\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2 \frac{1}{\boldsymbol{\sigma^2}}\right) + (\boldsymbol{\beta^T \beta} \frac{\boldsymbol{\lambda}}{\boldsymbol{\sigma^2}}),)$$

Again, pull out a $1/\sigma^2$ term which can be dropped. We multiply by -1 to minimize the function and we get:

$$\boldsymbol{ln}(\boldsymbol{\beta_{MAP}}) = \boldsymbol{argmin_\beta} \sum_{i=1}^{n} \left((\boldsymbol{Y_i} - \boldsymbol{X_i} * \boldsymbol{\beta})^2\right) + (\boldsymbol{\beta^T \beta \lambda}),)$$

$\boldsymbol{\beta^T \beta \lambda}$ is the matrix form of the regularization term which along with the error term can be put into matrix form. This gives us the minimization of the loss function with the regularization term:

$$J_R(\beta) = (X\beta - Y)^T (X\beta - Y) + \lambda||\beta||^2$$

[10pts] What is the probabilistic interpretation of $\lambda \to 0$ under this model? What about $\lambda \to \infty$? Take note: this is asking a related but *different* question than the first part of this problem!

Hint: Consider how the prior $P(\beta)$ is affected by changing λ.

The probabilistic interpretation of $\lambda \to 0$ would be effectively saying that the prior decreases in relation to λ resulting in diminishing impact on defining the parameters β found via minimizing the loss function. At $\lambda = 0$ there is no regularization via the prior as P(β) would be 0 in the loss function.

For $\lambda \to \infty$, you would be increasing the prior by a factor of λ, which as mentioned in the above question, would increasingly implement regularization via the loss function.

In both cases, by increasing or decreasing the prior, you would be effectively restricting ($\lambda \to 0$) or expanding ($\lambda \to \infty$) the distribution of possible parameters to achieve either effect.

[10pts] We have two data points in R^3:

$$\sim x_1 = [2,1]^T, y_1 = 7$$

$$\sim x_2 = [1,2]^T, y_2 = 5$$

We know that for linear regression with a bias/intercept term and mean-squared objective function, there are *infinite* solutions with these two points (i.e., any line in R^3 can be made to cross through these two points).

Give a specific third point < $\sim x_3, y_3$ > such that, when included with the first two above, will cause linear regression to *still have infinite solutions*. Your $\sim x_3$ should not equal $\sim x_1$ nor $\sim x_2$, nor should your y_3 equal either y_1 or y_2.

Calculate slope:

$$\textbf{Slope} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{5-7}{[1,2]^T - [2,1]^T} = \frac{-2}{[-1,1]^T}$$

Come back to this

2 Spectral Clustering [35pts]

The general idea behind spectral clustering is to construct a mapping of data points to an eigenspace of a graph-induced affinity matrix A, with the hope that the points are well-separated in the eigenspace to the point where something simple like k-means will work well on the embedded data.

A very simple affinity matrix can be constructed as follows:

(

$$A_{i,j} = A_{j,i} = \begin{cases} 1 & \text{if } d(\sim x_i, \sim x_j) \leq \Theta \\ 0 & \text{otherwise} \end{cases}$$

where $d(\sim x_i, \sim x_j)$ denotes Euclidean distance between points ~x_i and ~x_j.

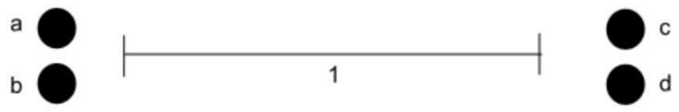

Figure .1: Simple toy dataset.

As an example, consider forming an affinity matrix for the dataset in Figure 1 using the affinity equation above, using Θ = 1. Then we get the affinity matrix in Figure 2.

For this particular example, the clusters $\{a,b\}$ and $\{c,d\}$ show up as nonzero blocks in the affinity matrix. This is, of course, artificial since we could have constructed the matrix A using any ordering of $\{a,b,c,d\}$. For example, another possible affinity matrix A could have been as in Figure 2(b).

$$A = \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 0 & 0 \\ b & 1 & 1 & 0 & 0 \\ c & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 1 & 1 \end{array}\right] \qquad \tilde{A} = \left[\begin{array}{c|cccc} & a & c & b & d \\ \hline a & 1 & 0 & 1 & 0 \\ c & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 0 \\ d & 0 & 1 & 0 & 1 \end{array}\right]$$

(a) (b)

Figure .2: Affinity matrices of Fig. 1 with Θ = 1.

The key insight here is that the eigenvectors of both A and $\tilde{A}$ have the same entries, just permuted. The eigenvectors with nonzero eigenvalues of A are ~$e_1 = [0.7,0.7,0,0]^T$ and
$\vec{e}_2 = [0,0,0.7,0.7]^T$ $_T$. Likewise, the nonzero eigenvectors of $\tilde{A}$ are ~$e_1 = [0.7,0,0.7,0]^T$ and ~$e_2 =$ $[0,0.7,0,0.7]$.

Spectral clustering embeds the original data points in a new space by using the coordinates of these eigenvectors. Specifically, it maps the point ~x_i to the point $[e_1(i), e_2(i), \ldots, e_k(i)]$, where ~$e_1$,...,~$e_k$ are the top k eigenvectors of A. We refer to this mapping as the *spectral embedding*. See Figure 3 for an example.

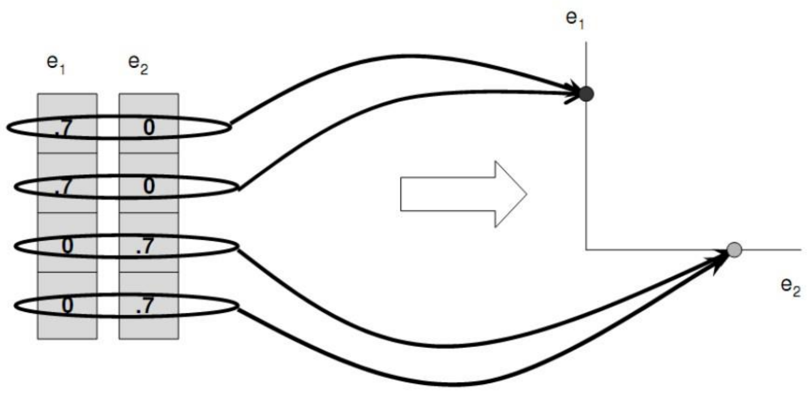

Figure .3: Using the eigenvectors of A to embed the data points. Notice that the points $\{a,b,c,d\}$ are tightly clustered in this space.

In this problem, we'll analyze how spectral clustering works on the simple dataset shown in the next figure.

[5pts] For the dataset in Figure 4, assume that the first cluster has m_1 points in it, and the second one has m_2 points. If we use the affinity equation from before to compute the

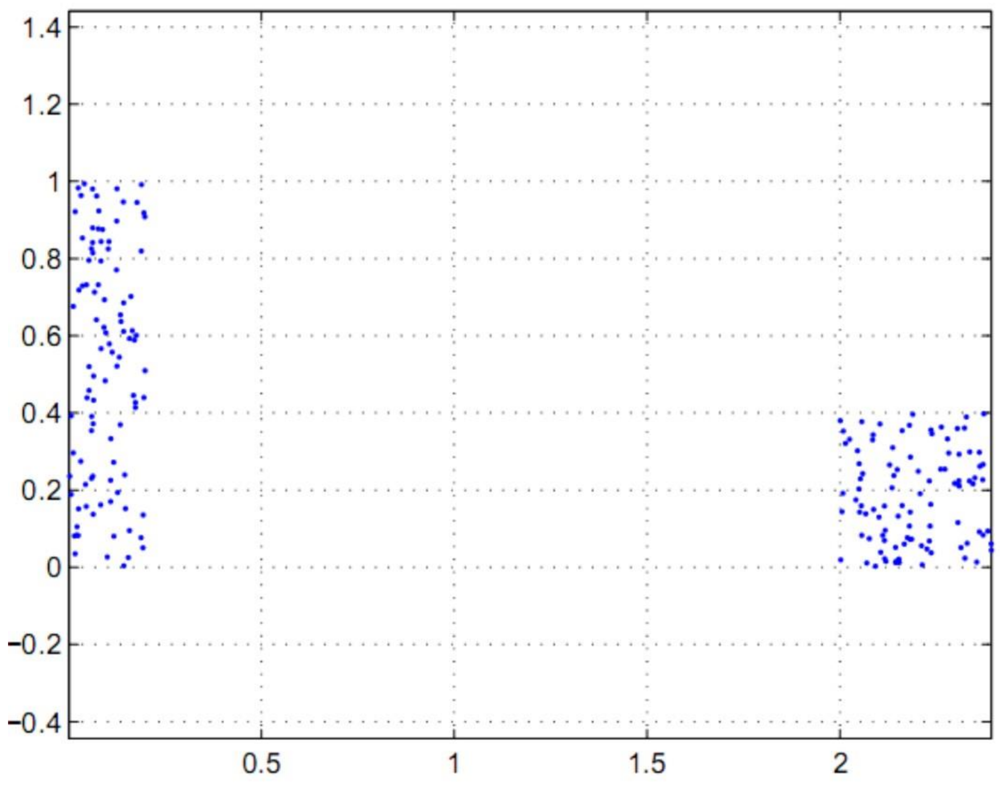

Figure .4: Dataset with rectangles.

affinity matrix A, what Θ value would you choose and why?

Based on the description above, I would assign Θ = 1.5-1.6. Theta must be smaller than the minimum distance between any possible point in m1(roughly 0.25) and m2(roughly 2). Theta also has to be larger than the maximum distance between points within a cluster to ensure that the points within the same cluster are recognized as having strong affinities. With Θ = 1.5-1.6, this appears to be met as for m1 the maximum inter-cluster distance would be slightly over 1, and for m2 appears to be roughly 0.5.

[10pts] The second step is to compute the first k dominant eigenvectors of the affinity matrix, where k is the number of clusters we want to have. For the dataset in the above figure, and the affinity matrix defined by the previous equation, is there a value of Θ for which you can analytically compute the first two eigenvalues and eigenvectors? If not, explain why not. If yes, compute and record these eigenvalues and eigenvectors. What are the other $((m_1 + m_2) - k)$ eigenvalues? Explain briefly.

Given the general lack of pattern within the affinity matrix clusters, it does not appear that even if you had the specific dataset values that you would be able to analytically compute the first two eigenpairs. The other $((m_1 + m_2) - k)$ eigenvalues are the less significant patterns within the dataset that can be mapped to another affinity matrix to eventually perform clustering as well, although given that we mapped the first two eigenvectors, the remaining would capture patterns of less significance.

Spectral clustering algorithms often make use a graph Laplacian matrix, L. A favorite variant is the *normalized* graph Laplacian, $L = D^{-1/2}AD^{-1/2}$, as this formulation has many convenient properties (D is a diagonal matrix whose i^{th} diagonal element, d_{ii}, is the sum of the i^{th} row of A).

[10pts] Show that a vector $\vec{v} = \left[\sqrt{d_{11}}, \sqrt{d_{22}}, ..., \sqrt{d_{nn}}\right]^T$ is an eigenvector of L with corresponding eigenvalue $\lambda = 1$.

Given that the eigenvalue of L is 1, we know that the following eigenvector/value equation should be Lv = v. So, first substitute L (and $\lambda = 1$):

$$\boldsymbol{D^{-\frac{1}{2}}AD^{-\frac{1}{2}} * v = 1 * v}$$

Then, multiply each side by $\boldsymbol{D^{\frac{1}{2}}}$ to eliminate $\boldsymbol{D^{-\frac{1}{2}}}$ term via the identity matrix I made by multiplying the two:

$$\boldsymbol{D^{\frac{1}{2}}D^{-\frac{1}{2}}AD^{-\frac{1}{2}} * v = D^{\frac{1}{2}}v}$$

Since $\boldsymbol{D^{\frac{1}{2}}}$ is the diagonal matrix of elements representing the sum of rows in A, we now get an equation equivalent to:

$$\vec{v} = \left[\sqrt{d_{11}}, \sqrt{d_{22}}, ..., \sqrt{d_{nn}}\right]^T$$

Given that $\boldsymbol{D^{\frac{1}{2}}}$ is the squared diagonal matrix of D, if you implement vector V as say a vector of 1's, you would get the above equation via the process above with an eigenvalue of 1.

One of the convenient properties of normalized graph Laplacians is the eigenvalue λ_1 of the leading eigenvector is, at most, 1; all other eigenvalues $\lambda_2,...,\lambda_n$ have values strictly smaller than 1.

Consider a matrix P, where $P = D^{-1}A$, where A is our affinity matrix and D is the diagonal matrix. Each $p_{ij} = a_{ij}/d_{ii}$. Note the intuition of this operation: we are normalizing each edge by the total degree of the incoming vertex, essentially creating a “transition probability” p_{ij} of transitioning from vertex i to vertex j. In other words, each row of P sums to 1, so it is therefore a valid probability transition matrix. Hence, P^t is a matrix whose $\{i,j\}^{th}$ element shows the probability of being at vertex j after t number of steps, if one started at vertex i.

[10pts] Show that $P^\infty = D^{-1/2}\vec{v}_1\vec{v}_1^T D^{1/2}$. This property shows that if points are viewed as vertices according to a transition probability matrix, then ~v_1 is the only eigenvector needed to compute the probability distribution over P^∞.

So we need to express Pt, which is a sum of the eigenvectors using eigen decomposition. This would look something like:

$$\boldsymbol{P^t} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\lambda_{\boldsymbol{i}}^{\boldsymbol{t}} * \boldsymbol{V_i} * \boldsymbol{V_i^T})$$

Then if we know that the first eigenvector v1 has an eigenvalue of 1, and the remaining eigenvalues for their respective eigenvectors have values strictly smaller than 1 (towards 0), we know that as step t approaches infinity, all other eigenvalues of 0 would essentially negate all values that are not the leading eigenvector in the above equation. So we get left with something like:

$$\boldsymbol{P^\infty} = \sum_{\boldsymbol{n=1}}^{\boldsymbol{n}} (\boldsymbol{1} * \boldsymbol{V_i} * \boldsymbol{V_i^T})$$

Then, if you multiply both sides by the square root and inverse square root matrices you get:

$$\boldsymbol{P^\infty} = \sum_{\boldsymbol{n=1}}^{\boldsymbol{n}} \boldsymbol{D^{-\frac{1}{2}}}(\boldsymbol{1} * \boldsymbol{V_i} * \boldsymbol{V_i^T})\, \boldsymbol{D^{\frac{1}{2}}}$$

Which is the equation from the question.

3 Coding [40pts]

In this question, you’ll be implementing a slightly simplified version of the MultiRankWalk (MRW) semi-supervised learning algorithm discussed in lecture. The paper is here:

https://lti.cs.cmu.edu/sites/default/files/research/reports/2009/cmulti09017. pdf

The basic procedure of MRW is similar to other graph-based random walk algorithms such as PageRank. For a graph G defined by the set of vertices V and edges E, the MRW procedure is as follows:

$$\sim r = (1 - d)\sim u + dW\sim r$$

where W is the weighted transition matrix of graph G from vertex i to j is given by $W_{ij} = A_{ij}/d_{ii}$, where d_{ii} is the degree of the i^{th} vertex. $\sim u$ is the normalized teleportation vector, where $|\sim u| = |V|$ and $||\sim u||_1 = 1$. d is a constant damping factor, controlling how often random jumps are made.

The value A_{ij} comes from our use of an affinity matrix in representing the graph. This is a deviation from the MRW paper, which assumes a simple adjacency matrix. The affinity matrix A will be determined using the radial-basis function kernel, also known as the Gaussian kernel or heat kernel. It has the form $A_{ij} = A_{ji} = e^{-\gamma||\sim x_i - \sim x_j||2}$, and is implemented in scikit-learn's sklearn.metrics.pairwise module as rbf_kernel(). Once you have the affinity matrix A, the diagonal (degree) matrix D can be found by summing the rows of A, i.e. $D_{ii} = \mathrm{P}_j A_{ij}$. Finally, the weighted transition probability matrix W can be found using A and D and the above formulation.

Your task is to solve for the ranking vector $\sim r$ by iteratively substituting $\sim r^{t-1}$ with $\sim r^t$ until convergence or a set number of iterations.

In this implementation, the $\sim u$ vector actually functions as a *seed vector*: this identifies vertices that are labeled and function as seeds for the subsequent label-spreading. "Seeds" are labeled data points used to initiate the label-spreading of the MRW algorithm and predict classes for unlabeled data. The original MRW paper cites several methods, including using PageRank to initially rank labeled vertices in terms of preference as seed vertices to MRW. Your code will need to implement both random seed selection, and degree-based seed selection. In the former, you'll randomly pick k labeled data points from each class and use them as seeds. In the latter, you'll rank the labeled vertices of each class by their degree (i.e. sums of the rows of A) and select the top k in each class.

Critically, you will need to perform MRW for each distinct class c in the data. Specifically, when initializing the labeled seeds in $\sim u$, you need to set each corresponding element $\sim u_i = 1$ such that $\sim y_i = c$. All other entries of $\sim u$ should be 0. Once this step is completed, you will need to normalize $\sim u$ such that $||\sim u||_1 = 1$. Next, you can proceed with MRW. Finally, you will repeat this process again for all unique labels c in your dataset, so that at the end you'll have a set of ranking vectors $\sim r_1, \sim r_2, \ldots, \sim r_c$ for each class. Once you have generated a ranking vector $\sim r$ for each class, you'll then assign labels to all your unlabeled data. For the i^{th} vertex, whichever ranking vector ~r's i^{th} element is largest, assign the corresponding class label represented by that ranking vector to the unlabeled data point. Continue for all unlabeled data.

Your code should be able to process: an input file containing the n m-dimensional data points, the number of labeled data points k to use from each class as seeds, whether to choose seeds randomly or by vertex degree, the damping factor d, and an output file to write the predicted classes for all data.

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. -d: the damping factor (float between 0 and 1)
3. -k: number of data points per class to use as seeds
4. -t: type of seed selection to use, "random" or "degree"
5. -e: the epsilon threshold, or squared difference of ~r^t and ~r^{t+1} to determine convergence
6. -g: value of gamma for the pairwise RBF affinity kernel
7. -o: a file path to an output file, where the predicted labels will be written

The format of the input file will be tab-delimited, where a single data point will be on one line. The first column will be the labels: any unlabeled data will have a label of -1. Functions are already written in the homework3.py-TEMPLATE file that will handle reading in data and parsing command-line arguments.

The format of the output file should be one label prediction per line; therefore, the number of lines in the input file and the output file should match exactly (so for the labeled data, you can either use the labels you read in from the file or the labels that are predicted from your ranking vectors, though in theory they should be the same). Essentially, fill in the -1 values in your initial label vector, then just write the vector to a text file, such that each element of the vector is on its own line. For your convenience, the ground-truth label files y_easy.txt and y_hard.txt for the full datasets are provided; you can use these to check how well your code is predicting the -1 labels.

HINT 1: The value of gamma can substantially affect the accuracy of your method. Larger values shrink the neighborhoods and isolate points from each other; smaller values expand the neighborhoods and make everything look the same distance. If in doubt, plot the affinity matrix using matplotlib.pyplot.imshow, and you should see a blockdiagonal-ish structure. For the easy dataset, try values around 0.5. For the harder dataset, try values in the 10-50 range.

HINT 2: At the same time, adding more seeds per class can help immensely. The default value in the template script is only 1 seeded value per class; while you can still attain high-90s accuracy with proper values of gamma on the hard dataset, it's almost impossible to hit perfect accuracy without increasing the number of seeds.

HINT 3: The two test datasets provided should not require any more than 100 iterations to converge using the default epsilon.

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 3 assessment that is open. When you do, you'll submit two files:

1. homework3.py: the Python script that implements your algorithms, and
2. homework3.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 homework3.tar homework3.py homework3.pdf
```

This will create a new file, homework3.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 October 10, 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! But you can use the pairwise metrics in scikit-learn, as well as the vector norm in SciPy.
- 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.