DSC 140B - Homework 05

Due: Wednesday, February 11

Instructions:

- Write your solutions to the following problems **by hand**, either on on another piece of paper that you scan or using a tablet. Typed solutions will not be accepted for credit!
- Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer.
- Homework problems are graded pass/fail on completeness and effort, not correctness.
- Homeworks are due via Gradescope at 11:59 PM.

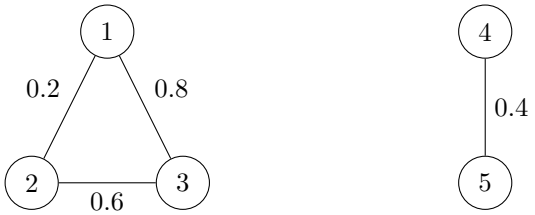
Problem 1. (0.5 credits)

In lecture, we saw that one minimizer of the cost function

$$\text{Cost}(\vec{f}) = \frac{1}{2}\sum_i \sum_j w_{ij}(f_i - f_j)^2$$

is the vector $\vec{f} = \frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$ which embeds all of the nodes of the graph to exactly the same number. The cost of this trivial embedding is zero, and we typically ignore it in favor of the eigenvector of the graph Laplacian with the *next* smallest positive eigenvalue for the embedding.

In the case where the graph has multiple connected components, however, there may be additional embeddings which have a cost of zero. For example, consider the similarity graph G shown below:



The weight of each edge is shown; the weight of non-existing edges is zero.

Find a normalized embedding vector $\vec{g}$ for the above graph such that $\text{Cost}(\vec{g}) = 0$ and $\vec{g} \perp \frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$; that is, $\vec{g}$ is orthogonal to the vector of all ones. Check that this is indeed a zero-cost embedding and show your work. Explain in words: why is this not a good choice for an embedding of the graph?

Solution:

For a connected graph, we can achieve a cost of zero by embedding each node to the exact same place. For a disconnected graph, we can achieve a cost of zero by embedding all nodes *in the same connected component* to the same place.

Let $\vec{g} = (g_1, g_2, g_3, g_4, g_5)^T$. In this case, we have a connected component of three nodes and another of two nodes. We can embed each of the three nodes in the first component to the same number, a, and the two nodes of the second component to the same number, b. That is: $\vec{g} = (a, a, a, b, b)^T$.

There are two constraints: $\vec{g}$ is orthogonal to the vector of all ones, and $\|\vec{g}\| = 1$. The latter tells us that $3a^2 + 2b^2 = 1$, and the former tells us that $\vec{g} \cdot (1, 1, 1, 1, 1)^T = 3a + 2b = 0$. Solving for b, we find

$b = -3a/2$. Substituting into the first equation:

$$1 = 3a^2 + 2b^2 \implies 1 = 3a^2 + 18a^2/4 \implies 30a^2/4 = 1 \implies a = \sqrt{2/15}$$

And therefore

$$b = -3a/2 = -\frac{3}{2}\sqrt{2/15} = -\sqrt{18/60} = -\sqrt{3/10}$$

So

$$\vec{g} = \begin{pmatrix} \sqrt{2/15} \\ \sqrt{2/15} \\ \sqrt{2/15} \\ -\sqrt{3/10} \\ -\sqrt{3/10} \end{pmatrix}$$

Checking that this is orthogonal to the vector of all ones:

$$3a + 2b = 3\sqrt{2/15} - 2\sqrt{3/10} = \sqrt{18/15} - \sqrt{12/10} = \sqrt{6/5} - \sqrt{6/5} = 0.$$

Checking that this is normalized:

$$3a^2 + 2b^2 = 3 \times (2/15) + 2 \times (3/10) = 6/15 + 6/10 = 2/5 + 3/5 = 1$$

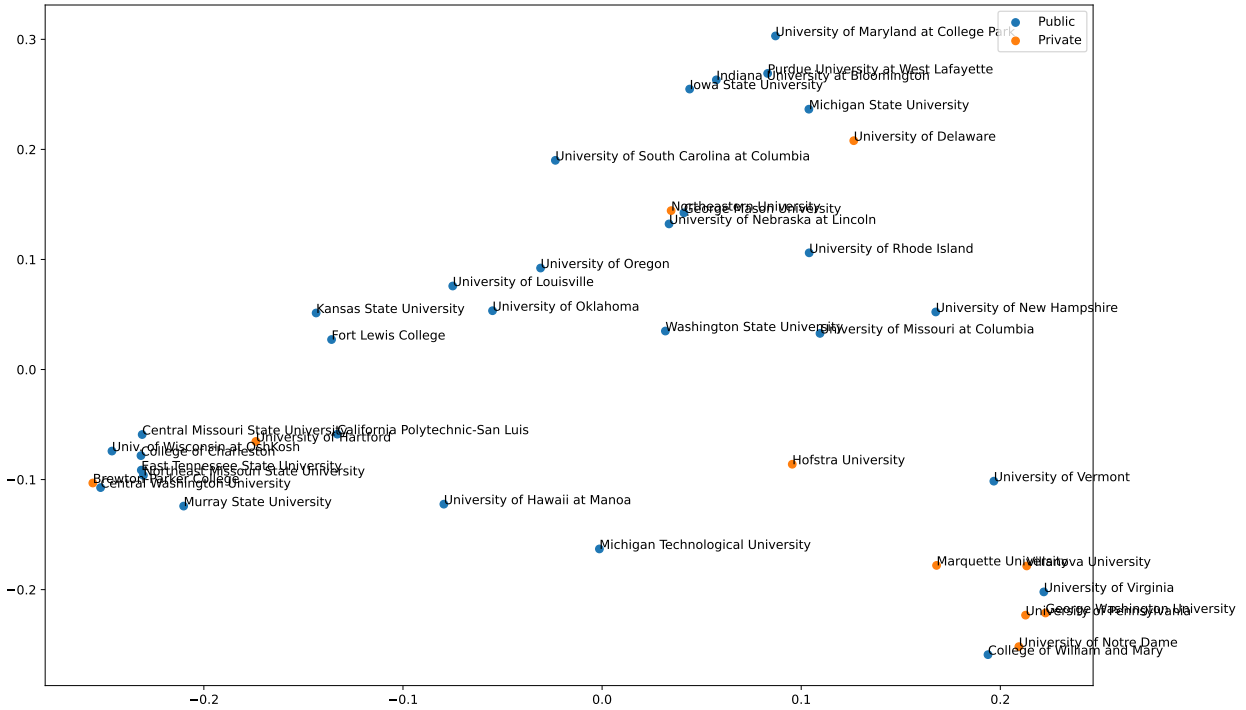
Problem 2. (1 credit)

The data at the link below contains information on 40 colleges in the US:

`https://f000.backblazeb2.com/file/jeldridge-data/006-colleges/colleges-sample.csv`

The data contains 17 numerical features measuring the size of the student body of each college, the graduation rate, etc., and one Boolean feature ("Private") saying whether the school is private or public. If we consider only the numerical features, each college is a point in $\mathbb{R}^{17}$.

Using Laplacian Eigenmaps, embed each college as a point in two dimensions and plot the result to obtain a similar figure to that shown below. You should construct a *symmetric* similarity matrix by building a k-neighbors graph (you will need to choose k). You should also consider whether the data needs to be standardized. You should drop the "Private" column for the purpose of computing the embedding – use only the numerical data.



Like the plot above, each point in your plot should be annotated with the name of the college, and the color of the point should show whether the college is public or private. However, your plot may look significantly different from the plot above due to the sign of the eigenvectors your code finds (it is arbitrary), as well as your choice of k in constructing the k-neighbors graph, etc. We will look to see if it shows the same general *patterns* as the plot above. For example, notice how there is a group of large midwestern state schools (including "Michigan State University", etc.), another group of large private universities (including "Notre Dame"), etc. Your plot should show the same.

You may use packages like `sklearn` to compute the k-neighbors graph, but not to do the actual embedding itself (e.g., do not use anything from `sklear.manifold`). Hint: the output of some `sklearn` functions is a sparse matrix; it is OK in this case to convert it to a dense `numpy` array. Also note that `sklearn`'s function for building a k-neighbors graph returns a *directed* graph, not an undirected graph, and therefore produces an asymmetric weight matrix. You will need to think about how to "symmetrize" it.

Note: you do not need to handwrite your solution to this problem, but you should include a screenshot of your code as part of your submission.

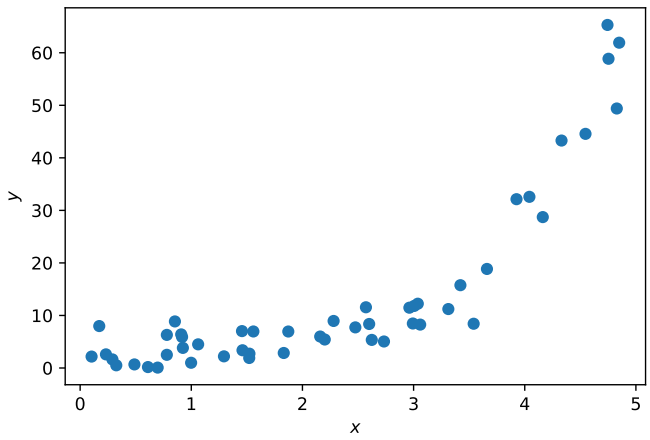
Solution:

```
import sklearn.neighbors
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

data = pd.read_csv("./colleges-sample.csv", index_col=0)

# standardize the data
Z = (data - data.mean(axis=0)) / data.std(axis=0)

W = np.array(sklearn.neighbors.kneighbors_graph(Z, n_neighbors=10).todense())
```

```
# the below line symmetrizes W. In the result, an entry (i,j) will be 1
# if W_{ij} was one or W_{ij}.T = W_{ji} was one.
W = np.maximum(W, W.T)

D = np.diag(W.sum(axis=0))
L = D - W

evals, evecs = np.linalg.eigh(L_norm)
embedding = evecs[:, [1, 2]]

# draw the plot
plt.figure(figsize=(15, 10))
plt.scatter(*embedding[private == "No"].T, label="Public")
plt.scatter(*embedding[private == "Yes"].T, label="Private")

for i, (x, y) in enumerate(embedding):
    plt.annotate(data.index[i], (x, y))

plt.legend()
```

Problem 3. (1.5 credits)

Consider fitting a prediction function to the data shown below:



The pattern in this data is not linear, but we can still fit it using a linear prediction function and an appropriate choice of basis functions.

The data for this problem can be found at:

`https://f000.backblazeb2.com/file/jeldridge-data/007-noisy_exponential/data.csv`

Note: for this problem, you do not need to handwrite any code; you can include screenshots of your code as part of your submission.

a) For this and the next few subproblems, consider fitting a function of the form $H_1(x; w) = we^x$. Note that this function has just one parameter to be learned: w.

Write a Python function that takes in one argument, `w`, and computes the average square loss (that is, the mean squared error) of H_1 on the data set with respect to `w`. Use your function to compute the average square loss of $w = 0.2$.

Hint: the correct output of your function when called on `0.2` is a number between 100 and 200.

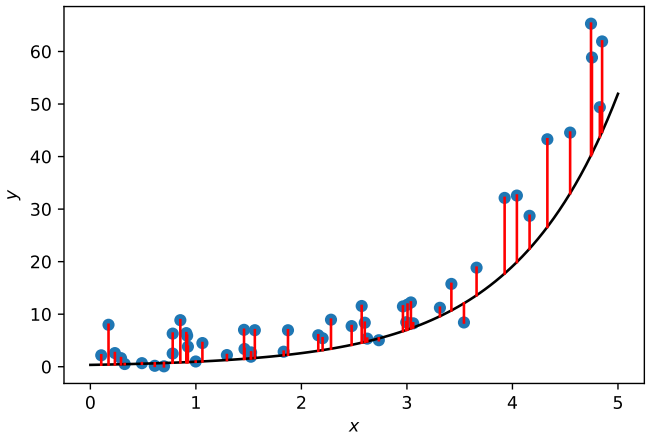
Solution:

```
def expected_square_loss(w):
    p = w * np.exp(x)
    return np.mean((p - y)**2)

expected_square_loss(0.2)
```

The expected square loss at $w = 0.2$ is approximately 174.2.

b) The parameter w controls the value of we^x when x is close to zero. In that sense, it is a scale parameter. Different values of w will change the shape of H_1, and will incur different amounts of error.

For example, the plot below shows H_1 for a particular choice of w that happens to be too small; H_1 is not a good fit to the data. The red lines show the residuals. Your Python function from the last part computes the average squared length of these red lines.



One way to find the optimal value of w (that is, the value resulting in the smallest average squared residual) is to use a numerical optimizer, such as `scipy.optimize.minimize`.

Use `scipy.optimize.minimize` to find the value of w which minimizes the average square loss (mean squared error).

Solution: We can find the optimimum by writing:

```
w_opt = scipy.optimize.minimize(expected_square_loss, .5).x
```

We find $w^* = 0.4949$

c) Another way to find the optimal value of w is to map the data to "feature space" and to fit a straight line in that space. In this case, we have a single basis function $\phi(x) = e^x$. Therefore, $H_1(x) = we^x = w\phi(x)$.

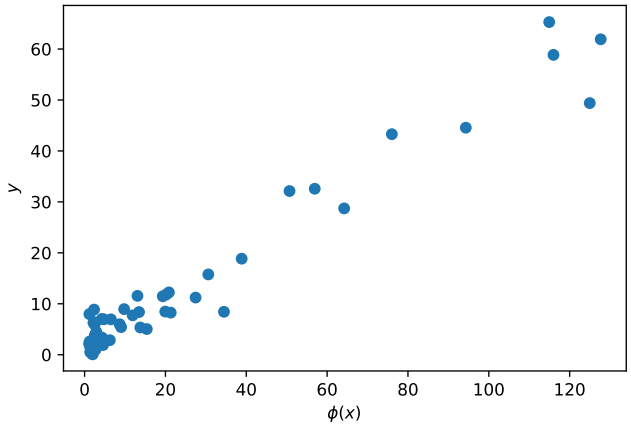
Plot the data in feature space; that is, create a "new" data set where each original point (x, y) becomes $(\phi(x), y)$.

Hint: you should see a linear trend.

Solution: We can plot the data in feature space with

```
plt.scatter(np.exp(x), y)
```

We will see:



d) Perform linear least squares in feature space. That is, fit a straight line to the "new" data

$$(\phi(x^{(1)}), y_1), \ldots, (\phi(x^{(n)}), y_n)$$

and report its slope, w.

Note that the discussion shows you how to perform least squares regression using `np.linalg.lstsq`. In this case, *do not* augment the data, since we are fitting a model we^x *without* a bias term.

Solution: We can perform least squares regression in feature space with the following code:

```
X = np.exp(x)[:,None]
np.linalg.lstsq(X, y)[0]
```

This gives the same value of w^* as before: roughly 0.4949.

Note that `np.linalg.lstsq` expects its first argument to be a two-dimensional array, but our data is just a one-dimensional array. The `[:,None]` part in the first line has the effect of adding a dimension to the one dimensional array; that is, it turns the array `[1, 2, 3]` into the array `[[1], [2], [3]]`.

You should have found the same value of w in the last part as you did when using `scipy.optimize.minimize`. When we perform least squares regression in feature space, we are finding the value of w which minimizes the average squared residual between the "new" data set and a straight line with slope w. When we used `scipy.optimize.minimize`, we were finding the value of w to minimize the average squared residual between the original data and the curved line, we^x. The fact is – these are the same optimization problem! Whether you fit a straight line in feature space, or a curved line in original space, you will find the same optimal w.

This happened because the function $H_1(x) = we^x$ is a linear function of the parameter, w. Now let's try the same experiment, but with a *different* prediction function, $H_2(x) = e^{wx}$. This is also a function of one parameter, but the parameter is now in the exponential. This is *not* a linear function of w.

e) As before, write a Python function that takes in one argument, `w`, and computes the average square loss (that is, mean squared error) of H_2 on the data set with respect to `w`. Use your function to compute the average square loss of $w = 0.5$.

Hint: the correct output of your function when called on `0.5` is a number between 200 and 300.

Solution:

```
def expected_square_loss(w):
    p = np.exp(w * x)
    return np.mean((y - p)**2)
```

When called on $w = 0.5$, the function returns 286.3.

f) Using `scipy.optimize.minimize`, find the value of w which minimizes the mean squared error of H_2.

Solution: We get a solution of $w^* = 0.844$ with the following code:

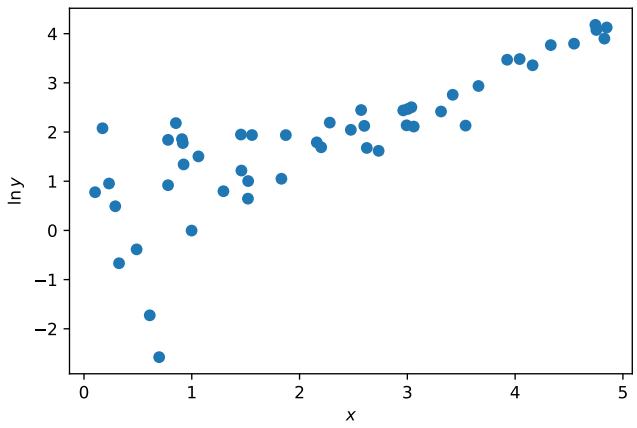
```
w_opt_direct = scipy.optimize.minimize(expected_square_loss, .5).x
```

g) The prediction function of the form $H_2(x; w) = e^{wx}$ does not involve a feature map in the sense that we've seen in lecture. Because of that, we cannot map the data to feature space and fit a linear predictor there. However, there is trick: we can "linearize" the data in another way. Note that if $y \approx e^{wx}$, then $\ln y \approx \ln e^{wx} = wx$.

Plot the "transformed" data set in which the point (x, y) becomes $(x, \ln y)$.

Hint: you should see a linear trend.

Solution: When we plot the data with `plt.scatter(x, np.log(y))`, we see:



h) Fit a straight line to the "transformed" data set by minimizing mean squared error, and report the slope of this line, w.

Solution: Using the code below, we find a slope of $w = 0.81$.

```
w_opt_transformed = np.linalg.lstsq(x[:, None], y_prime)[0]
```

i) You should observe that the w you found in the last step is slightly different than the w that was found "directly" using `scipy.optimize.minimize`. Using your Python function that computes the mean squared error of H_2, given a choice of `w`, calculate the mean squared error of both values of w.

Solution: For the "direct" solution, we find an expected square loss of 15.122. For the solution obtained through least squares on the "linearized" data, we get 20.613.

This is to be expected, since the "direct" solution was directly minimizing the mean squared error – assuming that the numerical optimizer did it's job and found the solution, there's no choice of w that can get smaller mean squared error than 15.122.

You should have found that the w found by "linearizing" the data has a larger mean squared error – it doesn't "fit" the data quite as well as the true minimizer. That is, linearizing the data and performing least squares *does not* minimize the mean squared error of H_2. This is because H_2 is not a linear function of the parameter w.

This isn't to say that it is a *bad* way of fitting the model. The plot below shows H_2 with both choices of w: the "optimal" choice found by the numerical solver, and the choice found by least squares on the "transformed" data. They are not too different!