CMSC 25300 / 35300
Homework 6

Submission Instructions:

- Please submit your homework in PDF to Gradescope (which can be accessed from the course's website on Canvas);
- Please paste your code in the submitted PDF. In other words, your submission should be a single PDF that contains both your writing solutions and your code. Be sure to include the output of your code when relevant (e.g. text and figures). Keep in mind that we are not easily able to run your code.
- Note that you do not need to copy the problem statements in your solution, *as long as you clearly indicate the problem numbers* (e.g., 1.a, 2.c, etc).

1. Kernel Ridge Regression.

a) (6 pts) In kernel ridge regression (KRR), we convert the data matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ to $\boldsymbol{\Phi} \in \mathbb{R}^{n\times d}$ by running each data x_i through the nonlinear transformation function ϕ. Correspondingly, the goal in KRR is to minimize find a weight vector that minimizes the squared loss and the regularization term:

$$\hat{w}_{KRR} = \underset{w\in\mathbb{R}^d}{\arg\min}\ \|\boldsymbol{\Phi} w - y\|_2^2 + \lambda\|w\|_2^2$$

Derive a closed form formula for $\hat{w}_{KRR}$ in terms of (a subset of) $I, \lambda, \phi, \boldsymbol{\Phi}, y$. We assume $d \geq p \geq n$, and the rows in $\boldsymbol{X}$ and $\boldsymbol{\Phi}$ are linearly independent. (Hint: in regular ridge regression, $\hat{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda I)^{-1}y$).

b) (10 pts) Recall (from Lecture 11 and Recitation 6) that the weights vector $\hat{w}_{KRR}$ is a linear combination of training samples, $\hat{w}_{KRR} = \boldsymbol{\Phi}^T\alpha$, where α is a vector containing the coefficients. Additionally, the kernel function k can be used to construct a matrix $\boldsymbol{K}$, where $\boldsymbol{K}_{ij} = k(x_i, x_j)$.

i) (4 pts) Use your answer from part (a) to derive an expression for α in terms of $\boldsymbol{K}, \lambda, I, y$.

ii) (6 pts) Now we want to make prediction on a new example z, we know that $\hat{y} = w_{KRR}^T\phi(z)$. Rewrite this expression using α and then substitute α with the expression you found in (i). Simplify the expression such that the final version only uses $\boldsymbol{K}, k, \lambda, I, y, X, z$. (Hint: you may find the notation $k(\boldsymbol{X}, z) = \begin{bmatrix} k(x_1, z) \\ k(x_2, z) \\ \cdots \\ k(x_n, z) \end{bmatrix}$ useful.)

c) (6 pts) Explain the relationship between ϕ and k, and between $\boldsymbol{\Phi}$ and $\boldsymbol{K}$ mathematically. And then explain why the relationship is significant. (Hint: Why do we have these two functions / matrices? What benefit do we get by having the kernel function k in kernel methods?)

d) (8 pts) Implement kernel ridge regression: generate training data and train a kernel ridge regression classifier using a polynomial kernel. Recall that polynomial kernel $k(x_i, x_j) = (x_i^T x_j + c)^d$, where c is a parameter and d is the degree of the polynomial. Which d would you use? Why?

You may find the starter code below.

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la
np.random.seed(10)

# Generate training data
n = 200;
p = 2;
X = 2*(np.random.rand(n,p) -.5)
radius = 0.5 # Define the radius of the small circle
distances = np.linalg.norm(X, axis=1)
# Generate labels based on distance from the origin
y = np.where(distances < radius, 1, -1)

# Show training data
plt.figure(1)
plt.scatter(X[:, 0], X[:, 1], 50, c=y)
plt.colorbar()
plt.xlabel('feature␣1')
plt.ylabel('feature␣2')
plt.title('2d␣training␣samples␣colored␣by␣label␣')
plt.show()

# specify parameters
c = 1
lam = 1

# implement kernel ridge regression
innerProds = X@X.T
### YOUR CODE STARTS HERE ###
# K = ...
# alpha = ...
# yhat = ...
### YOUR CODE ENDS HERE ###
```

```
y2 = np.array(np.sign(yhat))
plt.figure(2)
plt.scatter(X[:, 0], X[:, 1], 50, c=y2)
plt.colorbar()
plt.xlabel('feature␣1')
plt.ylabel('feature␣2')
plt.title('2d␣training␣samples␣colored␣by␣PREDICTED␣label')
plt.show()

# generate test data
ntest = 2000;
Xtest = 2*(np.random.rand(ntest ,p) -.5)

# make prediction on test data
innerProds_test = Xtest@X.T
### YOUR CODE STARTS HERE ###
# K_test = ...
# ytest = ...
### YOUR CODE ENDS HERE ###

plt.figure(3)
plt.scatter(Xtest[: ,0], Xtest [: ,1], 50, c=np.array(ytest)
   )
plt.colorbar()
plt.xlabel('feature␣1')
plt.ylabel('feature␣2')
plt.title('2d␣test␣samples␣colored␣by␣PREDICTED␣label␣(
   before␣taking␣sign␣)')
plt.show()
```

2. **Support Vector Machines.** When training a support vector machine, we are given n pairs of training data samples and labels: $\{\boldsymbol{x}_i, y_i\}_{1\leq i\leq n}$, where $\boldsymbol{x}_i \in \mathbb{R}^p$ and labels $y_i \in \{1, -1\}$. If we are using a kernel function $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, then a first step is to create the kernel matrix $\boldsymbol{K}$, where $\boldsymbol{K}_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. In the rest of this problem, we will use $\boldsymbol{k}_i$ for the i^{th} row of $\boldsymbol{K}$. To train the SVM, we are trying to solve this optimization problem:

$$\arg\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \left(1 - y_i \boldsymbol{k}_i^T \boldsymbol{\alpha}\right)_+ + \lambda \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha} \tag{1}$$

We will make two standard assumptions about $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$:

- For any set of samples $\{x_i\}_{1\leq i\leq n}$, $\boldsymbol{K}$ is a positive semidefinite matrix. In this case, we call $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ a *positive semidefinite kernel.*
- $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = k(\boldsymbol{x}_j, \boldsymbol{x}_i)$. In this case, we call $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ a *symmetric kernel.* This means the matrix $\boldsymbol{K}$ is also symmetric.

assume that

a) We will eventually want to discuss training an SVM by optimizing equation 1 using gradient descent. We know that gradient descent works well when optimizing *convex* functions. Remember, a function $g(x)$ is convex if for every $w, v \in \mathbb{R}^p$,

$$g(w) \geq g(v) + \nabla g(v)^T (w - v)$$

One way to think about this definition of a convex function: a function is convex when its first-order Taylor approximation is always less than or equal to the true function value. In this problem, we will prove f in equation 1 is convex.

  i) (10 points) First, suppose that $g(x) = g_1(x) + g_2(x)$, where $g_1(x)$ and $g_2(x)$ are both convex functions. Show that $g(x)$ is also a convex function.

  ii) (10 points) In this problem, we will show that $g(\boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{K} \boldsymbol{w}$ is a convex function of $\boldsymbol{w}$. To prove $g(\boldsymbol{w})$ is convex, we can write out its Taylor expansion, centered at a new point $\boldsymbol{v}$. Because g is a quadratic function, there are only three terms in the Taylor expansion:

$$\begin{aligned} g(\boldsymbol{w}) &= g(\boldsymbol{v}) + \nabla_{\boldsymbol{v}} g^T (\boldsymbol{w} - \boldsymbol{v}) + \frac{1}{2} (\boldsymbol{w} - \boldsymbol{v})^T \nabla^2_{\boldsymbol{v}} g (\boldsymbol{w} - \boldsymbol{v}) \\ &= \boldsymbol{v}^T \boldsymbol{K} \boldsymbol{v} + 2(\boldsymbol{K}\boldsymbol{v})^T (\boldsymbol{w} - \boldsymbol{v}) + (\boldsymbol{w} - \boldsymbol{v})^T \boldsymbol{K} (\boldsymbol{w} - \boldsymbol{v}) \end{aligned}$$

  Looking at this expression, which part is the first-order Taylor approximation? How do we know that g is convex?

b) (10 points) In the previous problem, we convinced ourselves that gradient descent is an appropriate choice for minimizing equation 1. This is because we found that f is a sum of convex functions, so f itself is also convex. Now we will think about minimizing equation 1 using gradient descent with step size τ. Suppose we choose a random initialization for $\boldsymbol{\alpha}^{(1)}$, and after t steps, all of the points are correctly classified with margin > 1. In this case, what is the value of $f(\boldsymbol{\alpha}^{(t)})$? What is the value of $\boldsymbol{\alpha}^{(t+1)}$?

c) (10 points) In the previous problem, some of the terms in the loss function f were zero. If we keep iterating gradient descent and find $\boldsymbol{\alpha}^{(t+2)}, \boldsymbol{\alpha}^{(t+3)}, \boldsymbol{\alpha}^{(t+4)}, \ldots$ will this remain true? Is this true at the optimal value $\boldsymbol{\alpha}^*$? To answer this, remember that $\nabla_{\boldsymbol{\alpha}} f|_{\boldsymbol{\alpha}=\boldsymbol{\alpha}^*} = 0$. Expand this equation using the formula for $\nabla_{\boldsymbol{\alpha}} f$, and explain what this tells us about the optimal solution.

3. Matrix Completion for Sensor Network Localization. In class, we discussed Singular Value Thresholding as a reasonably efficient way to recover missing values in a data matrix, with the assumption that the matrix is relatively low rank. For this question, we will be looking at a setting where a network of sensors is distributed across a d-dimensional space, and the sensors can find how far they are from other sensors.

Consider n sensors located at coordinates $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$. The matrix containing the squared distances between sensors is $D \in \mathbb{R}^{n\times n}$. We can expand this to find

$$\begin{aligned}(D)_{ij} &= \|x_i - x_j\|_2^2 \\ &= \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^T x_j \\ D &= ve^T + ev^T - 2XX^T \end{aligned} \tag{2}$$

where

$$v = \begin{bmatrix} \|x_1\|_2^2 \\ \vdots \\ \|x_n\|_2^2 \end{bmatrix} \in \mathbb{R}^n, \quad e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n, \quad \text{and } X = \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix} \in \mathbb{R}^{n\times d}.$$

Note that $(ve^T)_{ij} = \|x_i\|_2^2$ provides the position norm for the rows, while $(ev^T)_{ij} = \|x_j\|_2^2$ provides the position norm for the columns, and $(XX^T)_{ij} = x_i^T x_j$.

The squared-distance matrix D is rank $d+2$ thanks to the decomposition in equation 2, which is nice because even if most of the sensors' distance measurements are missing or corrupted and need to be thrown out, we can still recover the missing entries of D using low-rank matrix completion techniques. Once we have the distance matrix, we will be able to recover the sensors' locations.

a) (10 points) First, let's implement a matrix-completion algorithm based on Singular Value Thresholding. For simplicity, we'll choose the number of singular values to keep based on the known rank of the distance matrix we're trying to recover (in class, we applied a cutoff based on value, but here we know the target rank and can apply it directly). The data file can be found in `sensor_data.npz`. The idea is to alternately set certain values of a matrix based on observations and to truncate its singular values to a target rank. Below is the starter code, which should only require 4 lines to complete.

```
def matrix_completion_svt(
        mat_shape: tuple,
        obs_vals: np.ndarray,
        obs_idcs: tuple,
        trunc_rank: int,
        iter_count: int = 100,
        verbose: bool = False) -> np.ndarray:
    """Matrix completion with Singular Value Thresholding
    Parameters:
        mat_shape (tuple): shape of the target matrix
        obs_vals (np.ndarray): a flattened array containing the
            observed values
        obs_idcs (tuple of np.ndarrays): describes the indices of
            the observed values in question
             Given as a tuple with an array of indices for each
                 spatial dimension
             Usage: mat[obs_idcs] returns a view of the entries in
                 question (can also be used to set values)
        trunc_rank (int): number of singular values to keep.
             Truncates singular values to set matrix to this rank
        iter_count (int): number of iterations to run singular
            value thresholding
        verbose (bool):   flag for whether to output the step
            sizes at a given iteration
    Output:
        mat_rec (np.ndarray): the recovered matrix
    """
    mat_rec = np.zeros(mat_shape)
    # TODO 1: set observed entries of mat_rec to obs_vals
    for it in range(1, 1+iter_count):
        # Apply rank reduction with SVD
        # TODO 2: truncate singular values of mat_rec, save to
            mat_rec_new
        # Correct for measurements
        # TODO 3: set observed entries of mat_rec_new to obs_vals
        # Basic logging
        if verbose and (it%20==0 or it==iter_count):
            rel_update_size = np.linalg.norm(mat_rec_new - mat_rec
                ) / np.linalg.norm(mat_rec)
            print(f"Iteration␣{it:3d}:␣relative␣update␣size={
                rel_update_size:.3e}")
        mat_rec = mat_rec_new
    return mat_rec

# To load the appropriate values from data (for testing purposes):
sensor_data_file = np.load("sensor_data.npz")
dim          = sensor_data_file["dim"]
sensor_count = sensor_data_file["sensor_count"]
coords_ref   = sensor_data_file["coords_ref"]
dist_mat_ref = sensor_data_file["dist_mat_ref"]
obs_dists    = sensor_data_file["obs_dists"]
obs_idcs     = tuple([*sensor_data_file["obs_idcs"]])
```

b) (5 points) Suppose you have a rank-d symmetric positive-semidefinite matrix $M \in \mathbb{R}^{n\times n}$ and want to factor it such that $M = YY^T$. How could you find a valid $Y \in \mathbb{R}^{n\times d}$ matrix using the SVD $M = U\Sigma U^T$? (note that $U = V$ since M is symmetric)

Hint: you should find a way to split the weighting from the singular values between the two copies of Y. Also keep in mind that we want Y to have just d columns.

c) (15 points) If we also know the true locations of several of the sensors, we can recover the locations of the rest of the sensors using the distance matrix from part (a) as well as the factorization technique from part (b). In this part, assume we know the true locations for the first m sensors, $x_1, \ldots, x_m$ and want to recover the locations for $x_{m+1}, \ldots, x_n$.

In order to use part (b) there is a nice change-of-variables trick to define a second matrix $M \in \mathbb{R}^{n\times n}$ such that

$$\begin{aligned}(M)_{ij} &= \frac{1}{2}((D)_{1j} + (D)_{i1} - (D)_{ij}) \\ &= \frac{1}{2}(\|x_1 - x_j\|_2^2 + \|x_i - x_i\|_2^2 - \|x_i - x_j\|_2^2) \\ &= \cdots = x_i^T x_j - x_1^T x_j - x_i^T x_1 + x_1^T x_1 \\ &= (x_i - x_1)^T (x_j - x_1)\end{aligned}$$

which allows for

$$M = YY^T \quad \text{where} \quad Y = \begin{bmatrix} (x_1 - x_1)^T \\ (x_2 - x_1)^T \\ \vdots \\ (x_n - x_1)^T \end{bmatrix} \in \mathbb{R}^{n \times d}.$$

The matrix Y contains the shifted sensor coordinates (that is, shifted so that the first sensor is at the origin). Y can be computed using part (b). Suppose we know the true locations of the first m sensors, $x_1, x_2, \ldots, x_m$ (call these "anchors"), and would like to use these to find the locations of the rest of the sensors, $x_{m+1}, \ldots, x_n$.

For this question, we provide starter code with several functions to finish:

- `factor_psd` which implements the factorization found in part (b)
- `build_M_mat` which implements the change-of-basis outlined above
- `localize_sensors_from_data` which ties everything together

Once you have completed these functions, run the `localize_sensors_from_data` function on the data from `sensor_data.npz` using the first $m = 4$ points as anchors and report your results.

The provided data uses $n = 300$ sensors in a $d = 3$ space and observing $\approx 15\%$ of the sensor-pair distances (in fact, if there were more sensors we could get away with a smaller fraction of observed sensor-pair distances).

```
def factor_psd(M: np.ndarray, col_count: int) -> np.ndarray:
    """Factors M=YY^T using SVD; output Y has col_count columns"""
    # Since M is symmetric, using np.linalg.eigh is also
        acceptable
    # TODO 1: factor M using SVD to Y
    return Y

def find_alignment(points: np.ndarray, points_target: np.ndarray)
   -> np.ndarray:
    """Finds the global rotation to align points with
        points_target
    Global rotation is given by orthogonal matrix R minimizing
    #   ||points_target - points @ R||_F
    # Closed form for rotation is available in terms of the SVD
    See https://en.wikipedia.org/wiki/
        Orthogonal_Procrustes_problem
    Sidenote: this is kind of analogous to the best rank-k matrix
        approximation where we truncated the singular values, but
        here we set all singular values to 1
    """
    W, s, Zh = np.linalg.svd(points.T @ points_target)
    R = W@Zh
    return R

def build_M_mat(dist_mat: np.ndarray) -> np.ndarray:
    """Builds the M matrix with
    (M)_ij = 0.5 * ((D)_1j + (D)_i1 - (D)_ij)
    """
    # TODO 2: Compute M matrix as requested
    return M_mat
```

```python
def localize_sensors_from_data(
        sensor_count: int,
        dim: int,
        obs_dists: np.ndarray,
        obs_idcs: tuple,
        anchor_coords: np.ndarray,
        svt_iter_count: int = 100,
        verbose: bool = False,
    ) -> (np.ndarray, np.ndarray):
    """Takes in the incomplete distance measurement data and
       recovers the locations of all sensors
    Note that this procedure requires the true location of several
        sensors considered "anchors"
    Parameters:
        sensor_count (int): number of sensors
        dim (int): dimension of the space (e.g., dim=2 or dim=3)
        obs_dists (np.ndarray): a 1D array containing measured
            distances
        obs_idcs (tuple of np.ndarrays): describes the indices of
            the observed values in question
             Given as a tuple with an array of indices for each
                 spatial dimension
        anchor_coords (np.ndarray): coordinates of the "anchor"
            points
             For simplicity let these be the positions of the first
                  few sensors.
        svt_iter_count (int): number of iterations for singular
            value thresholding
        verbose (bool): whether to print logging information
    Outputs:
        coords_rec (np.ndarray): the recovered coordinates of all
            sensors
             Takes shape (sensor_count, dim)
        dist_mat_rec (np.ndarray): the recovered (squared)
            distance matrix between sensors
             Takes shape (sensor_count, sensor_count)
             (Value is returned to facilitate debugging)
    """
    anchor_count = anchor_coords.shape[0]
    dist_mat = # TODO 3: run SVT to fill in the distance matrix

    # Now recover sensor coordinates from the distance matrix
    # (using anchor points to find the right alignment)
    M_mat = build_M_mat(dist_mat)

    # Y matrices represent sensor locations in a shifted
       coordinate system (with x_1 at the origin)
    Y_raw = factor_psd(M_mat, col_count=dim)
    Y_anchor_ref = anchor_coords - anchor_coords[0, np.newaxis, :]

    # The recovered Y matrix is only unique up to a global
    # rotation since M=(YR)(YR)^T=YRR^TY^T=YY^T for orthogonal R
    # so we need to find the appropriate alignment to match
    # the reference positions for the "anchor" sensors
    alignment = find_alignment(Y_raw[:anchor_count], Y_anchor_ref)
    Y_aligned = (Y_raw @ alignment)

    # Shift the coordinates to recover the original X matrix
    sensor_coords = # TODO 4: bring back from the shifted
       coordinate system used in Y_aligned
    if verbose:
        # Extra outputs to help with debugging
        # Find relative magnitude in dist_mat beyond rank (dim+2)
        svals  = np.linalg.svd(dist_mat, compute_uv=False)
        sv_err = np.linalg.norm(svals[dim+2:]) / np.linalg.norm(
            svals)
        print(f"\nDistance␣matrix␣rank␣error:␣{sv_err:.3e}")
        # Find the error incurred by factoring M_mat
        factor_err = np.linalg.norm(Y_raw @ Y_raw.T - M_mat) / np.
            linalg.norm(M_mat)
        print(f"Factorization␣error:␣{factor_err:.3e}")
        # Find the quality of the alignment against anchor points
        Y_anchor_rec = Y_raw[:anchor_count] @ alignment
        align_err = np.linalg.norm(Y_anchor_rec - Y_anchor_ref) /
            np.linalg.norm(Y_anchor_ref)
        print(f"Alignment␣error:␣{np.linalg.norm(align_err):.3e}")
    return sensor_coords, dist_mat

# Example settings to call the function
# Code to load the data is provided in part (a)
anchor_count  = 4
anchor_coords = coords_ref[:anchor_count]
coords_rec, dist_mat_rec = localize_sensors_from_data(
    sensor_count,
    dim,
    obs_dists,
    obs_idcs,
    anchor_coords,
    svt_iter_count = 250,
    verbose = True,
)
dist_mat_err = np.linalg.norm(dist_mat_rec - dist_mat_ref) / np.
   linalg.norm(dist_mat_rec)
coords_err   = np.linalg.norm((coords_rec - coords_ref)[
   anchor_count:]) / np.linalg.norm(coords_ref[anchor_count:])
print()
print(f"Distance␣matrix␣error:␣{dist_mat_err:.3e}")
print(f"Sensor␣coordinate␣error:␣{coords_err:.3e}")
```