CMSC 25300: Mathematical Foundations of ML

Problem Set 6

Hung Le Tran

17 Nov 2023

Problem 6.1 (Problem 1)

Solution

(a) Let

$$f(w) = \|\Phi w - y\|^2 + \lambda \|w\|^2$$

then

$$\begin{aligned} f(w) &= w^T \Phi^T \Phi w - 2w^T \Phi^T y + y^T y + \lambda w^T w \\ \Rightarrow \nabla_w f(w) &= 2\Phi^T \Phi w - 2\Phi^T y + 2\lambda w \end{aligned}$$

is zero when

$$\begin{aligned} 2\Phi^T \Phi w - 2\Phi^T y + 2\lambda w_{krr} &= 0 \\ \Rightarrow w_{krr} &= (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y \\ &= \Phi^T (\Phi^T \Phi + \lambda I)^{-1} y \end{aligned}$$

via SVD of Φ, as similar to regular ridge regression.

(b)(i) We want to write $w_{krr} = \Phi^T \alpha$. From (a),

$$\alpha = (\Phi^T \Phi + \lambda I)^{-1} y \tag{1}$$

$$= (K + \lambda I)^{-1} y \tag{2}$$

(b)(ii) To make prediction on new sample z:

$$\begin{aligned} \hat{\mathbf{y}} &= w_{krr}^T \Phi(z) \\ &= (\Phi^T \alpha)^T \Phi(z) \\ &= \alpha^T (\Phi \Phi(z)) \\ &= \alpha^T k(\mathbf{X}, z) \end{aligned}$$

(c) Mathematically,

$$k(x_i, x_j) = \varphi(x_i)^T \varphi(x_j)$$

and

$$K = \Phi\Phi^T$$

We have Φ to increase the dimensionality of the feature space, so that linear LS on the new feature space can create better-fit decision boundaries that are not achievable in lower dimensions. However, this comes with the cost of more data storage, which k and K solves, by directly computing $\Phi(x_i)^T\Phi(x_j)$, in order to construct $K = \Phi\Phi^T$, which is what we really need to calculate w_{krr}.

(d) I chose d because the simulated data suggests a circular decision boundary, which can be generated by

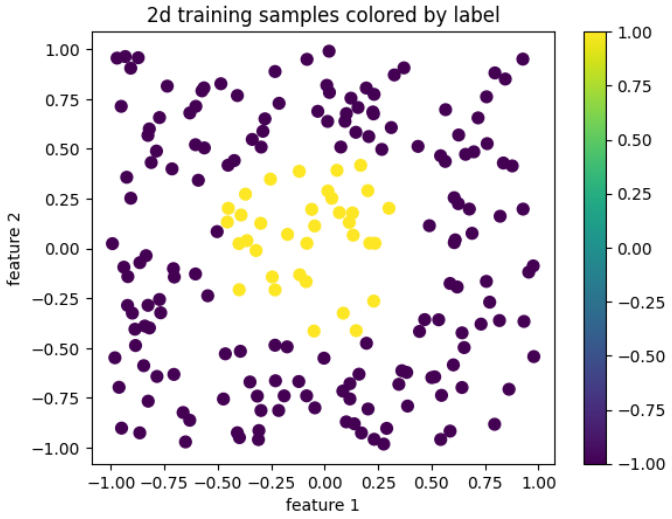
$$(x_1^2 + x_2^2)^2 - r^2 = 0$$

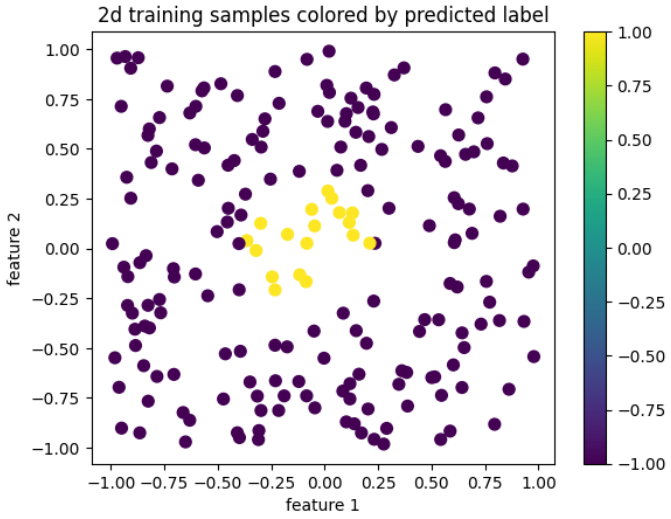
for some r, which is a degree-2 polynomial.

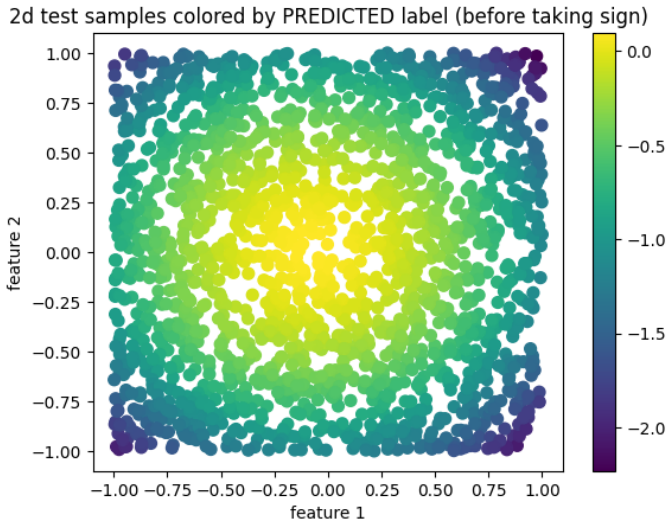
```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la

np.random.seed(10)

n = 200
p = 2
X = 2 * (np.random.rand(n, p) - .5)
radius = 0.5
distances = np.linalg.norm(X, axis = 1)

y = np.where(distances < radius, 1, -1)

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()

c = 1
lam = 1
d = 2
innerProds = X @ X.T

### YOUR CODE STARTS HERE ###
K = np.power(innerProds + c, d)
alpha = la.inv((K + lam * np.identity(n))) @ y
yhat = K @ alpha
### 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()

ntest = 2000
Xtest = 2*(np.random.rand(ntest, p) - .5)

# make predictions on test data
innerProds_test = Xtest@X.T
### YOUR CODE STARTS HERE
K_test = np.power(innerProds_test + c, d)
ytest = K_test @ alpha
### 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()
```

which plots







□

Problem 6.2 (Problem 2)

Solution

(a)(i) If g_1, g_2 are convex then

$$\begin{aligned} g(w) - g(v) &= (g_1(w) - g_1(v)) + (g_2(w) - g_2(v)) \\ &\geq \nabla g_1(v)^T(w - v) + \nabla g_2(v)^T(w - v) \\ &= \nabla g^T(w - v) \end{aligned}$$

which implies g is also convex.

(a)(ii)

$$\begin{aligned} g(w) &= g(v) + \nabla g(v)^T(w - v) + \frac{1}{2}(w - v)^T \nabla_g^2 (w - v) \\ &= v^t K v + 2(Kv)^T(w - v) + (w - v)^T K(w - v) \end{aligned}$$

has $g(v) + \nabla g(v)^T = v^T K v + 2(Kv)^T(w - v)$ as the first-order approximation. The second-order term, $(w - v)^T K(w - v) \geq 0$ since K is positive semidefinite. Therefore

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

(b) If after t steps, all points are correctly classified, then no loss is incurred on the samples. Therefore $f(\alpha^{(t)}) = \lambda \alpha^{(t)^T} K \alpha^{(t)}$ with only the regularization term.

Consequently, $\alpha^{(t+1)} = \alpha^{(t)} - 2\tau\lambda K\alpha^{(t)}$, since there is no contribution from the gradient of loss function term.

(c) If we keep iterating GD, we might no longer correctly classify for all traning samples and there would be loss incurred due to error on the training samples. Therefore it might not remain true that the error term on training samples is zero on later iterations of GD.

At the optimal value α^*,

$$\begin{aligned} 0 &= \nabla_\alpha f|_{\alpha=\alpha^*} \\ &= \sum_{i=1}^{n} \mathbb{1}_{\{y_i k_i^T \alpha^* < 1\}}(-y_i k_i) + 2\lambda K\alpha^* \\ \Rightarrow \sum_{i=1}^{n} \mathbb{1}_{\{y_i k_i^T \alpha^* < 1\}}(y_i k_i) &= 2\lambda K\alpha^* \end{aligned}$$

What this tells us is that at the optimal solution, since $RHS \neq 0$, $LHS \neq 0$ too. It follows that it must classify at least 1 training sample wrongly. □

Problem 6.3 (Problem 3)

Solution

(a)

```
import numpy as np
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
    mat_rec[obs_idcs] = 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
        U, S, Vt = np.linalg.svd(mat_rec)
        S_trunc= np.zeros(S.shape)
        S_trunc[:trunc_rank] = S[:trunc_rank]
        mat_rec_new = U @ np.diag(S_trunc) @ Vt
        # Correct for measurements
        # TODO 3: set observed entries of mat_rec_new to obs_vals
        mat_rec_new[obs_idcs] = 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"]])

D = matrix_completion_svt(dist_mat_ref.shape, obs_dists, obs_idcs, 3 +
   2, 200, True)
```

which prints

```
Iteration  20: relative update size =1.293e-02
Iteration  40: relative update size =3.457e-03
Iteration  60: relative update size =2.576e-03
Iteration  80: relative update size =7.624e-04
Iteration 100: relative update size =2.582e-04
Iteration 120: relative update size =1.058e-04
Iteration 140: relative update size =4.588e-05
Iteration 160: relative update size =2.036e-05
Iteration 180: relative update size =9.171e-06
Iteration 200: relative update size =4.178e-06
```

(b) M has rank d, which means that Σ only has its first d entries as non-zero entries. Therefore one can construct S of shape $n \times d$, with zero entries except for its diagonal

entries as $\sqrt{\sigma_1}, \sqrt{\sigma_2}, \ldots, \sqrt{\sigma_d}$ which would then give

$$\Sigma = SS^T$$

Therefore

$$\begin{aligned} M &= U\Sigma U^T \\ &= USS^TU^T \\ &= (US)(US)^T \end{aligned}$$

Furthermore, $Y = US$ has rank d since its d columns are scaled version of the columns of U, which are linearly independent.

(c) □

```
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 1 D 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]
    # TODO 3: run SVT to fill in the distance matrix
    dist_mat = matrix_completion_svt((sensor_count, sensor_count),
   obs_dists, obs_idcs, dim + 2, svt_iter_count, verbose)
    # 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
    # TODO 4: bring back from the shifted coordinate system used in
   Y_aligned
    sensor_coords = Y_aligned + anchor_coords [0, np.newaxis, :]

    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}" )
```

which prints

```
Iteration  20: relative update size =1.293e-02
```

```
Iteration  40: relative update size =3.457e-03
Iteration  60: relative update size =2.576e-03
Iteration  80: relative update size =7.624e-04
Iteration 100: relative update size =2.582e-04
Iteration 120: relative update size =1.058e-04
Iteration 140: relative update size =4.588e-05
Iteration 160: relative update size =2.036e-05
Iteration 180: relative update size =9.171e-06
Iteration 200: relative update size =4.178e-06
Iteration 220: relative update size =1.921e-06
Iteration 240: relative update size =8.900e-07
Iteration 250: relative update size =6.075e-07

Distance matrix rank error: 3.034e-06
Factorization error: 5.730e-06
Alignment error: 4.373e-06

Distance matrix error: 1.578e-05
Sensor coordinate error: 1.928e-05
```