CMSC 25300: Mathematical Foundations of ML

Problem Set 4

Hung Le Tran

03 Nov 2023

Problem 4.1 (Problem 1)

Solution

(a)

```
import numpy as np
def gram_schmidt(X):
    eps = 1e-12
    n, p = X.shape
    U = np.zeros((n, 0))
    for j in range(p):
        v = X[:, j]
        v = v - U @ U.T @ v
        if (np.linalg.norm(v) > eps):
            U = np.hstack((U, (v/np.linalg.norm(v)).reshape(-1, 1)))
    return U
```

(b)

```
def hilbert_matrix (n) :
    X = np.array ([[1.0 / ( i + j - 1) for i in range (1 , n + 1)] for
   j in range (1 , n + 1)])
    return X

U = gram_schmidt(hilbert_matrix(7))
n, r = U.shape
E = np.zeros((r, r))
for i in range(r):
    for j in range(r):
        E[i][j] = U[:, i].T @ U[:, j]

print(np.linalg.norm(E - np.identity(r), 1))
```

which prints

```
0.22871200665499755
```

(c)

```
def modified_gram_schmidt(X):
    # Define a threshold value to identify if a vector is nearly a zero
    vector.
    eps = 1e-12

    n, p = X.shape
    U = np.zeros((n, 0))

    for j in range(p):
        # Get the j-th column of matrix X
        v = X[:, j]
        for i in range(j):
            # Compute and subtract the projection of vector v onto the
   i-th column of U
            v = v - np.dot(U[:, i], v) * U[:, i]
        v = np.reshape(v, (-1, 1))
        # Check whether the vector we get is nearly a zero vector
        if np.linalg.norm(v) > eps:
            # Normalize vector v and append it to U
            U = np.hstack((U, v / np.linalg.norm(v)))

    return U

E = np.zeros((r, r))
U = modified_gram_schmidt(hilbert_matrix(7))
for i in range(r):
    for j in range(r):
        E[i][j] = U[:, i].T @ U[:, j]

print(np.linalg.norm(E - np.identity(r), 1))
```

which prints

```
1.0272454269170171e-08
```

The error is drastically lower for the modified Gram-Schmidt method (around 7 orders of magnitude lower). While both methods in theory make $\mathbf{U}_{n+1}$ orthogonal to all previous $\mathbf{U}_k$ for all $1 \leq k \leq n$, but in the first method, the orthogonality of each specific $\mathbf{U}_{n+1}, \mathbf{U}_{k_0}$ pair is dependent the fact that all $\{\mathbf{U}_k : 1 \leq k \leq n\}$ are orthogonal (since we projected everything at once and were reliant on the fact that $\mathbf{U}^T\mathbf{U} = I$). However, computationally, this means that the numerical errors in the pairwise orthogonality of $\{\mathbf{U}_k : 1 \leq k \leq n\}$ build up to affect the orthogonality of every pair $\mathbf{U}_{n+1}, \mathbf{U}_{k_0}$.

In the second method, since we are projecting $\mathbf{v}$ onto each U_k one by one, the error of the orthogonality of $\mathbf{U}_{n+1}$ and $\mathbf{U}_n$ (the last step when calculating $\mathbf{U}_{n+1}$) is eliminated. This yields a much better result than the first method. □

Problem 4.2 (Problem 2)

Solution

(a) x_1 is not a scaled copy of x_2, so $rank(X) \geq 2$. $x_1, x_2, x_3 \in \mathbb{R}^2$ so $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ trivially form an orthonormal basis.

(b)(i)

$$P = v(v^T v)^{-1} v^T = vv^T/|v|^2 = \frac{1}{v_1^2 + v_2^2} \begin{bmatrix} v_1^2 & v_1 v_2 \\ v_1 v_2 & v_2^2 \end{bmatrix}$$

(ii), (iii) The squared distance for each x_i is

$$\begin{aligned} \|x_i - Px_i\|^2 &= \|(I-P)x_i\|^2 \\ &= x_i^T (I-P)^T (I-P) x_i \\ &= x_i^T (I-P) x_i \\ &= x_i^T x_i - x_i^T P x_i \\ &= x_i^T x_i - \frac{1}{|v|^2} x_i^T v v^T x_i \end{aligned}$$

therefore the sum of squared distance for 3 data points is

$$\begin{aligned} \sum_{i=1}^{3} d_i^2 &= \sum_{i=1}^{3} \left(x_i^T x_i - x_i^T P x_i \right) \\ &= \sum_{i=1}^{3} \left(x_i^T x_i - \frac{1}{v_1^2 + v_2^2} x_i^T v v^T x_i \right) \\ &= \sum_{i=1}^{3} \left(x_i^T x_i - \frac{1}{v_1^2 + v_2^2} (v^T x_i)^2 \right) \end{aligned}$$

(iv) $\|v\| = 1 \Rightarrow v_1^2 + v_2^2 = 1$. Since $\sum x_i^T x_i$ is fixed, to minimize the sum of squared distance we need to maximize

$$\begin{aligned} S &= \sum_{i=1}^{3} (v^T x_i)^2 = \sum_{i=1}^{3} v^T x_i x_i^T v = v^T X X^T v \\ &= \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} 3 & 1 & 0 \\ 0 & 2 & \sqrt{6} \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 1 & 2 \\ 0 & \sqrt{6} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} 10 & 2 \\ 2 & 10 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= 10v_1^2 + 4v_1 v_2 + 10 v_2^2 \\ &= 10 + 4v_1 v_2 \leq 10 + 2 = 12 \end{aligned}$$

with equality only achieved when $v_1 = v_2 = 1/\sqrt{2}$ or $v_1 = v_2 = -1/\sqrt{2}$

Therefore v is not unique. But P is unique:

$$P = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}$$

(c)

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

WLOG, choose $\mathbf{U}_1 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$. Then $\mathbf{U}_2$ has norm 1 and is orthogonal to $\mathbf{U}_1$.

Let $\mathbf{U}_2 = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ then

$$u_1/\sqrt{2} + u_2/\sqrt{2}u_2 = 0, u_1^2 + u_2^2 = 1$$

which implies $u_1 = 1/\sqrt{2}, u_2 = -1/\sqrt{2}$ (WLOG. The other way around works too). Then

$$\sigma_2 = \|X^T u_2\| = \sqrt{8}$$

Therefore

$$\mathbf{U} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}, \mathbf{\Sigma} = \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{8} & 0 \end{bmatrix}$$

□

Problem 4.3 (Problem 3)

Solution

(a)

$$\mathbf{X} = \sum_{i=1}^{r} u_i \sigma_i v_i^T$$

(b)

$$\mathbf{X}_k = \sum_{i=1}^{k} u_i \sigma_i v_i^T$$

□

Problem 4.4 (Problem 4)

Solution

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{U} = \mathbf{V}^T = I, \mathbf{\Sigma} = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix}$$

where we quickly verify that the columns of $\mathbf{U}$ (the 2 standard basis vectors of $\mathbb{R}^2$) indeed

span the columns of $\mathbf{X}$, while the columns of $\mathbf{V}$ (which are also standard basis vectors of $\mathbb{R}^2$) also span the rows of $\mathbf{X}$. □

Problem 4.5 (Problem 5)

Solution

Justification simliar to Problem 4, where the columns of $\mathbf{U}$ span the columns of $\mathbf{X}$, while columns of $\mathbf{V}$ span the rows of $\mathbf{X}$. For this one, note that the singular values have to be positive.

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \mathbf{U} = I, \mathbf{V}^T = -I, \mathbf{\Sigma} = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}$$

□

Problem 4.6 (Problem 6)

Solution

(a), (b)

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
import sys
d = np.load('face_emotion_data.npz')
X = d ['X']
y = d ['y']
n,p = np.shape (X)
# error rate for regularized least squares

# padding with diagonal entries as 0's for easier insertion and
   organization
# row number: index of subset chosen to pick regularization parameter
# column number: index of subset chosen as last holdout
error_RLS = np.zeros ((8 , 8))
# error rate for truncated SVD
error_SVD = np.zeros ((8 , 8))
# SVD parameters to test
k_vals = np.arange (9) + 1
param_err_SVD = np.zeros (len(k_vals))
# RLS parameters to test
lambda_vals = np.array ([0, 0.5, 1, 2, 4, 8, 16])
param_err_RLS = np.zeros (len(lambda_vals))

SUBSET_COUNT = 8
SUBSET_SIZE = int(n/SUBSET_COUNT)
SUBSET_IDCS = range(8)
X = X.reshape(SUBSET_COUNT, SUBSET_SIZE, p)
y = y.reshape(SUBSET_COUNT, SUBSET_SIZE)

for holdout_1 in SUBSET_IDCS:
    for holdout_2 in SUBSET_IDCS:
        if (holdout_1 == holdout_2):
```

```
            break
        Xtrain = np.row_stack([X[j] for j in SUBSET_IDCS if j not in [
    holdout_1, holdout_2]])
        ytrain = np.concatenate([y[j] for j in SUBSET_IDCS if j not in
    [holdout_1, holdout_2]])
        Xval = X[holdout_1]
        yval = y[holdout_1]
        Xtest = X[holdout_2]
        ytest = y[holdout_2]
        U, S, Vh = np.linalg.svd(Xtrain)
        V = Vh.T
        S_pseudo_inv = (S > 0) * (1/S)
        for k in k_vals:
            Uk = U[:, :k]
            Sk = S_pseudo_inv[:k]
            Vk = V[:, :k]
            w = Vk @ np.diag(Sk) @ Uk.T @ ytrain

            # validate on holdout_1 subset
            yval_hat = Xval @ w
            # maps False to -1, True to 1
            yval_pred = 2 * (yval_hat > 0) - 1
            param_err_SVD[k-1] = np.sum(yval_pred != yval)

        for l_index, l in enumerate(lambda_vals):
            S_ridge = S * (1/(S * S + l))
            w = V @ np.diag(S_ridge) @ U[:, :len(S_ridge)].T @ ytrain
            # validate on holdout_1 subset
            yval_hat = Xval @ w
            yval_pred = 2 * (yval_hat > 0) - 1
            param_err_RLS[l_index] = np.sum(yval_pred != yval)

        # SVD: perform prediction on test set with optimal k, adding 1
    from index
        k = np.argmin(param_err_SVD) + 1
        Uk = U[:, :k]
        Sk = S_pseudo_inv[:k]
        Vk = V[:, :k]
        w_svd = Vk @ np.diag(Sk) @ Uk.T @ ytrain
        ytest_hat = Xtest @ w_svd
        ytest_pred =  2 * (ytest_hat > 0) - 1
        error_SVD[holdout_1][holdout_2] = np.sum(ytest_pred != ytest)/
    SUBSET_SIZE

        # RLS: perform prediction on test set with optimal lambda
        l = lambda_vals[np.argmin(param_err_RLS)]
        S_ridge = S * (1/(S * S + l))
        w_rls = V @ np.diag(S_ridge) @ U[:, :len(S_ridge)].T @ ytrain
        ytest_hat = Xtest @ w_rls
        ytest_pred =  2 * (ytest_hat > 0) - 1
        error_RLS[holdout_1][holdout_2] = np.sum(ytest_pred != ytest)/
    SUBSET_SIZE

print(f"SVD average error: {np.sum(error_SVD)/56}, RLS average error: {
    np.sum(error_RLS)/56}")
```

which prints

```
SVD average error: 0.06584821428571429, RLS average error: 0.03125
```

(c)

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
import sys
d = np.load('face_emotion_data.npz')
X = d ['X']
y = d ['y']

X_new = X @ np.random.rand(9, 3)
X = np.column_stack((X, X_new))
n,p = np.shape (X)
# error rate for regularized least squares

# padding with diagonal entries as 0's for easier insertion and
   organization
# row number: index of subset chosen to pick regularization parameter
# column number: index of subset chosen as last holdout
error_RLS = np.zeros ((8 , 8))
# error rate for truncated SVD
error_SVD = np.zeros ((8 , 8))
# SVD parameters to test
k_vals = np.arange (12) + 1
param_err_SVD = np.zeros (len(k_vals))
# RLS parameters to test
lambda_vals = np.array ([0, 0.5, 1, 2, 4, 8, 16])
param_err_RLS = np.zeros (len(lambda_vals))

SUBSET_COUNT = 8
SUBSET_SIZE = int(n/SUBSET_COUNT)
SUBSET_IDCS = range(8)
X = X.reshape(SUBSET_COUNT, SUBSET_SIZE, p)
y = y.reshape(SUBSET_COUNT, SUBSET_SIZE)

for holdout_1 in SUBSET_IDCS:
    for holdout_2 in SUBSET_IDCS:
        if (holdout_1 == holdout_2):
            break
        Xtrain = np.row_stack([X[j] for j in SUBSET_IDCS if j not in [
   holdout_1, holdout_2]])
        ytrain = np.concatenate([y[j] for j in SUBSET_IDCS if j not in
   [holdout_1, holdout_2]])
        Xval = X[holdout_1]
        yval = y[holdout_1]
        Xtest = X[holdout_2]
        ytest = y[holdout_2]
        U, S, Vh = np.linalg.svd(Xtrain)
        V = Vh.T
        S_pseudo_inv = (S > 0) * (1/S)
        for k in k_vals:
            Uk = U[:, :k]
            Sk = S_pseudo_inv[:k]
            Vk = V[:, :k]
            w = Vk @ np.diag(Sk) @ Uk.T @ ytrain
```

```
            # validate on holdout_1 subset
            yval_hat = Xval @ w
            # maps False to -1, True to 1
            yval_pred = 2 * (yval_hat > 0) - 1
            param_err_SVD[k-1] = np.sum(yval_pred != yval)

        for l_index, l in enumerate(lambda_vals):
            S_ridge = S * (1/(S * S + l))

            w = V @ np.diag(S_ridge) @ U[:, :len(S_ridge)].T @ ytrain
            # validate on holdout_1 subset
            yval_hat = Xval @ w
            yval_pred = 2 * (yval_hat > 0) - 1
            param_err_RLS[l_index] = np.sum(yval_pred != yval)

        # SVD: perform prediction on test set with optimal k, adding 1
    from index
        k = np.argmin(param_err_SVD) + 1
        Uk = U[:, :k]
        Sk = S_pseudo_inv[:k]
        Vk = V[:, :k]
        w_svd = Vk @ np.diag(Sk) @ Uk.T @ ytrain
        ytest_hat = Xtest @ w_svd
        ytest_pred =  2 * (ytest_hat > 0) - 1
        error_SVD[holdout_1][holdout_2] = np.sum(ytest_pred != ytest)/
    SUBSET_SIZE

        # RLS: perform prediction on test set with optimal lambda
        l = lambda_vals[np.argmin(param_err_RLS)]
        S_ridge = S * (1/(S * S + l))
        w_rls = V @ np.diag(S_ridge) @ U[:, :len(S_ridge)].T @ ytrain
        ytest_hat = Xtest @ w_rls
        ytest_pred =  2 * (ytest_hat > 0) - 1
        error_RLS[holdout_1][holdout_2] = np.sum(ytest_pred != ytest)/
    SUBSET_SIZE

print(f"SVD average error: {np.sum(error_SVD)/56}, RLS average error: {
    np.sum(error_RLS)/56}")
```

The only significant differences of this part and the previous part being that we have appended augmented features in 3 new columns of $\mathbf{X}$. The k values we can try can therefore be extended from 9 to 12, as reflected in *kvals*. One instance (since we used *rand* to generate the weights to generate new features) of above code prints

```
SVD average error: 0.05803571428571429, RLS average error:
    0.03571428571428571
```

The (linearly combined) augmented features are not helpful for classification. Since we linearly generated these features, they don't represent any new information that is not already embedded in our original 9 columns. □