2. The SVD

$x_1 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$, $x_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $x_3 = \begin{bmatrix} 0 \\ \sqrt{6} \end{bmatrix}$

$X = [x_1 \ x_2 \ x_3] \in \mathbb{R}^{2\times 3}$

(a.) $-\frac{\sqrt{6}}{6} x_1 + \frac{\sqrt{6}}{2}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = x_3$

$\underline{u}_1 = \frac{x_1}{\|x_1\|_2} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}/3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$\underline{x}_2' = \underline{x}_2 - \underline{u}_1(u_1^T \underline{x}_2) = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix}\left([1 \ 0]\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$

$= \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$= \begin{bmatrix} 0 \\ 2 \end{bmatrix}$

$\underline{u}_2 = \frac{x_2'}{\|x_2'\|_2} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}/2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

$\Rightarrow U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

(b.)

(i) $\in \mathbb{R}^2$

$V = \text{span}(v)$

$P_v = v(v^Tv)^{-1}v^T$

$P_v^T = v(v^{-1}(v^T)^{-1})^T v^T$
$= v(v^{-1}(v^{-1})^T)^T v^T$
$= v(v^{-1}(v^T)^{-1})v^T$
$= v(v^T v)^{-1} v^T = P_v$

(ii) $d_i^2 = \|x_i - P_v x_i\|_2^2 = (x_i - P_v x_i)^T(x_i - P_v x_i)$

$= x_i^T x_i - x_i^T P_v^T x_i - x_i^T P_v x_i + x_i^T P_v^T P_v x_i$

$= \|x_i\|_2^2 - 2x_i^T P_v x_i + x_i^T P_v^2 x_i$

$= \|x_i\|_2^2 - 2x_i^T P_v x_i + x_i^T P_v x_i$

$= \|x_i\|_2^2 - x_i^T P_v x_i$

(iii)

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \quad x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix}$$

$$d_i^2 = \|x_i\|_2^2 - x_i^T P_v x_i$$

$$x_i^T P_v x_i = x_i^T v (v^T v)^{-1} v^T x_i$$

$$= x_i^T \frac{v v^T}{(v^T v)} x_i$$

$$= \frac{(v^T x_i)^2}{v^T v}$$

$$= \frac{(v_1 x_{i1} + v_2 x_{i2})^2}{v_1^2 + v_2^2}$$

$$\Rightarrow d_i^2 = x_i^T x_i - \frac{(v_1 x_{i1} + v_2 x_{i2})^2}{v_1^2 + v_2^2}$$

(c)

$$X = \begin{bmatrix} 3 & 1 & 0 \\ 0 & 2 & \sqrt{6} \end{bmatrix}$$

$$\arg\min_v \sum_{i=1}^{3} d_i^2 = \arg\max_{v: v^T v = 1} \sum_{i=1}^{3} \frac{(v_1 x_{i1} + v_2 x_{i2})^2}{v_1^2 + v_2^2}$$

$$= \arg\max_{v: v^T v = 1} \sum_{i=1}^{3} (v_1 x_{i1} + v_2 x_{i2})^2$$

$$\sum_{i=1}^{3} (v_1 x_{i1} + v_2 x_{i2})^2 = (3v_1)^2 + (v_1 + 2v_2)^2 + (\sqrt{6} v_2)^2$$

$$= 9v_1^2 + v_1^2 + 4v_1 v_2 + 4v_2^2 + 6v_2^2$$

$$= 10v_1^2 + 4v_1 v_2 + 10 v_2^2$$

$$= 10(v_1^2 + v_2^2) + 4v_1 v_2$$

$$= 10 + 4v_1 v_2$$

$$\mathcal{L} = 10 + 4v_1 v_2 - \lambda(v_1^2 + v_2^2 - 1)$$

$$\frac{\partial \mathcal{L}}{\partial v_1} = 4v_2 - 2\lambda v_1 = 0$$

$$\frac{\partial \mathcal{L}}{\partial v_2} = 4v_1 - 2\lambda v_2 = 0$$

$$\Rightarrow v_1 = v_2 \Rightarrow 2v_1^2 = 1 \Rightarrow v_1 = v_2 = \frac{1}{\sqrt{2}} \Rightarrow v = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

$$u_1 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \quad u_2 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \Rightarrow U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

Include part of (iv)

(iv) $v' = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$ also minimize d_i^2. So v is not unique. But P is unique as $v v^T = v' v'^T$.

$$\sigma_1 = \|X^T \underline{u}_1\|_2$$

$$= \left\| \begin{bmatrix} 3 & 0 \\ 1 & 2 \\ 0 & \sqrt{6} \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} 3/\sqrt{2} \\ 3/\sqrt{2} \\ \sqrt{6}/\sqrt{2} \end{bmatrix} \right\|_2 = \sqrt{\frac{9}{2} + \frac{9}{2} + \frac{6}{2}} = \sqrt{12} = 2\sqrt{3}$$

$$\sigma_2 = \|X^T \underline{u}_2\|_2$$

$$= \left\| \begin{bmatrix} 3 & 0 \\ 1 & 2 \\ 0 & \sqrt{6} \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} 3/\sqrt{2} \\ -1/\sqrt{2} \\ -\sqrt{6}/\sqrt{2} \end{bmatrix} \right\|_2 = \sqrt{\frac{9}{2} + \frac{1}{2} + \frac{6}{2}} = \sqrt{8} = 2\sqrt{2}$$

$\Rightarrow X \in \mathbb{R}^{2\times 3}$

$$\Rightarrow \Sigma = \begin{bmatrix} 2\sqrt{3} & 0 & 0 \\ 0 & 2\sqrt{2} & 0 \end{bmatrix}$$

3. $X \in \mathbb{R}^{n\times d}$ $\quad V = [v_1\ v_2\ \cdots\ v_r]$ $\quad \underbrace{X}_{n\times d} = \underbrace{U}_{n\times r}\,\underbrace{\Sigma}_{r\times r}\,\underbrace{V^T}_{r\times d}$

$U = [u_1\ u_2\ \cdots\ u_r]$

(a) $$X = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

(b) $$X_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

---

4. $$X = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix}$$

X is a diagonal matrix

$$\Rightarrow U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \Sigma = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix},\ V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$X = U\Sigma V = I\Sigma I^T = \Sigma$$

---

5. $$X = \begin{bmatrix} -3 & 0 \\ 0 & -1 \end{bmatrix}$$

X is a diagonal matrix

$$\Rightarrow U = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix},\ \Sigma = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix},\ V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

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

cmsc253_hw4

November 5, 2023

[ ]:
```python
import numpy as np
import numpy.linalg as la # matrix rank, inverse import scipy.io
import matplotlib.pyplot as plt # plots
import scipy.io as sio
import sys
from matplotlib import colors
```

Q1 Gram-Schmidt. (a)

[ ]:
```python
def gram_schmidt(X):
    # X is an n-by-p matrix.
    # Returns U an orthonormal matrix.
    # eps is 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]
        # Write your own code here: Perform the
        # orthogonalization by subtracting the projections on
        # all columns of U. And then check whether the vector
        # you get is nearly a zero vector.
        v = v - U@U.T@v
        v = np.reshape(v, (-1, 1))
        if np.linalg.norm(v) > eps:
            # Normalize vector v and append it to U
            U = np.hstack((U, v / la.norm(v)))
    return U
X = np.array([[1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7],
              [1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8],
              [1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9],
              [1/4, 1/5, 1/6, 1/7, 1/8, 1/9, 1/10],
              [1/5, 1/6, 1/7, 1/8, 1/9, 1/10, 1/11],
              [1/6, 1/7, 1/8, 1/9, 1/10, 1/11, 1/12],
```

```
              [1/7, 1/8, 1/9, 1/10, 1/11, 1/12, 1/13]])
gram_schmidt(X)
```

```
[ ]: array([[ 8.13304645e-01, -5.43794290e-01,  1.99095308e-01,
        -5.51132525e-02,  1.19578493e-02, -1.96870730e-03,
        -2.10219152e-04],
       [ 4.06652323e-01,  3.03317262e-01, -6.88560548e-01,
         4.75965265e-01, -1.97392654e-01,  5.41151990e-02,
         3.01288446e-03],
       [ 2.71101548e-01,  3.93949644e-01, -2.07134626e-01,
        -4.90111167e-01,  6.10839674e-01, -3.26893986e-01,
         1.54577469e-02],
       [ 2.03326161e-01,  3.81744394e-01,  1.12389156e-01,
        -4.39598473e-01, -2.54179063e-01,  6.41045574e-01,
        -2.08187091e-01],
       [ 1.62660929e-01,  3.51412668e-01,  2.91518455e-01,
        -1.12276608e-01, -4.99206689e-01, -2.02224426e-01,
         6.15118567e-01],
       [ 1.35550774e-01,  3.20235052e-01,  3.89191824e-01,
         2.30886766e-01, -1.50594721e-01, -5.41817789e-01,
        -7.06123659e-01],
       [ 1.16186378e-01,  2.92095792e-01,  4.40744903e-01,
         5.20623748e-01,  5.01273338e-01,  3.80535581e-01,
         2.81830797e-01]])
```

(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
     n = 7
     X = hilbert_matrix(n)
     U = gram_schmidt(X)
     L1 = np.identity((U.T@U).shape[0]) - U.T@U
     print(L1)
     err = la.norm(L1, ord=1)
     print(f'L1 matrix norm is {err}')
```

```
[[ 0.00000000e+00  6.73072709e-16 -2.36616282e-15  2.80331314e-14
  -1.92849903e-12  1.25247715e-10 -9.71088800e-09]
 [ 6.73072709e-16  0.00000000e+00 -2.76723089e-14  3.71008779e-13
  -2.63988831e-12 -4.05814549e-11  5.49175827e-09]
 [-2.36616282e-15 -2.76723089e-14  1.11022302e-16  8.52865001e-12
  -2.31620473e-10  5.20528393e-09 -1.33284293e-07]
 [ 2.80331314e-14  3.71008779e-13  8.52865001e-12 -2.22044605e-16
  -9.04092928e-09  4.29613409e-07 -1.80981921e-05]
 [-1.92849903e-12 -2.63988831e-12 -2.31620473e-10 -9.04092928e-09
```

```
  0.00000000e+00  2.66371580e-05 -2.30322707e-03]
 [ 1.25247715e-10 -4.05814549e-11  5.20528393e-09  4.29613409e-07
   2.66371580e-05  0.00000000e+00 -2.27098006e-01]
 [-9.71088800e-09  5.49175827e-09 -1.33284293e-07 -1.80981921e-05
  -2.30322707e-03 -2.27098006e-01  1.11022302e-16]]
L1 matrix norm is 0.22941947980869065
```

(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 / la.norm(v)))
         return U
     n = 7
     X = hilbert_matrix(n)
     U_modified = modified_gram_schmidt(X)
     L1_modified = np.identity((U_modified.T@U_modified).shape[0]) - \
         U_modified.T@U_modified

     L1_modified
     print(L1_modified)
     err_modified = la.norm(L1_modified, ord=1)
     print(f'Modified L1 matrix norm is {err_modified}')
```

```
[[ 0.00000000e+00  1.54737334e-15 -2.16146545e-14  3.62931907e-13
  -7.97459321e-12  3.16995839e-10 -1.90386089e-08]
 [ 1.54737334e-15  0.00000000e+00  1.02695630e-15 -2.63955524e-14
   9.00696184e-13 -2.74882339e-11  1.03917761e-09]
 [-2.16146545e-14  1.02695630e-15 -2.22044605e-16  3.05311332e-15
  -2.71893619e-13  1.51737511e-11 -6.93317848e-10]
 [ 3.62931907e-13 -2.63955524e-14  3.05311332e-15  0.00000000e+00
  -1.66533454e-15 -2.34368080e-13  4.83647289e-11]
 [-7.97459321e-12  9.00696184e-13 -2.71893619e-13 -1.66533454e-15
```

```
   0.00000000e+00  2.01227923e-14 -2.16494878e-12]
 [ 3.16995839e-10 -2.74882339e-11  1.51737511e-11 -2.34368080e-13
   2.01227923e-14  2.22044605e-16 -1.46549439e-14]
 [-1.90386089e-08  1.03917761e-09 -6.93317848e-10  4.83647289e-11
  -2.16494878e-12 -1.46549439e-14 -2.22044605e-16]]
Modified L1 matrix norm is 2.0821648873819987e-08
```

In the traditional approach to Gram-Schmidt, each vector in the sequence is made orthogonal to all previously processed vectors by subtracting their projections. The modified Gram-Schmidt re-orthogonalizes each vector against all previously orthogonalized vectors, one at a time. This stepwise approach reduces the accumulation of errors and maintains better orthogonality. Therefore, the modified Gram-Schmidtis process is generally superior to the original process.

Q6 (a)

[ ]:
```
d = np.load('face_emotion_data.npz')
X = d['X']
y = d['y']
n, p = np.shape(X)
# error rate for truncated SVD
error_SVD = np.zeros((8, 7))
# SVD parameters to test
k_vals = np.arange(9) + 1
param_err_SVD = np.zeros(len(k_vals))

subset_count = 8

subsets = np.array_split(np.arange(n), 8)

for i in range(subset_count):
    X_hold_out = X[subsets[i]]
    y_hold_out = y[subsets[i]]

    for idx, j in enumerate([j for j in range(subset_count) if j != i]):
        X_train = np.concatenate([X[subsets[p]] for p in range(8) if p != i
                                  and p != j])
        y_train = np.concatenate([y[subsets[p]] for p in range(8) if p != i
                                  and p != j])

        U, Sigma, VT = la.svd(X_train, full_matrices=False)

        X_test = X[subsets[j]]
        y_test = y[subsets[j]]

        w_hat_lst = list()

        for k_idx, k in enumerate(k_vals):
            Sigma_plus = np.diag(1 / Sigma[:k])
```

```
            VT_k = VT[:k, :]
            U_k = U[:, :k]

            w_hat = VT_k.T@Sigma_plus@U_k.T@y_train
            w_hat_lst.append(w_hat)
            y_hat = X_test@w_hat

            param_err_SVD[k_idx] += np.sum(np.sign(y_hat) != y_test) / 16

        # print(param_err_SVD)
        k_best = np.argmin(param_err_SVD)
        w_hat_best = w_hat_lst[k_best]

        y_hat_hold_out = X_hold_out@w_hat_best
        error_SVD[i][idx] = np.sum(np.sign(y_hat_hold_out) != y_hold_out) / 16
print(f"Error estimate: {np.mean(error_SVD)}")
```

```
Error estimate: 0.052455357142857144
```

(b)

[ ]:

```
d = np.load('face_emotion_data.npz')
X = d['X']
y = d['y']
n, p = np.shape(X)
# error rate for regularized least squares
error_RLS = np.zeros((8, 7))
# 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

subsets = np.array_split(np.arange(n), 8)

for i in range(subset_count):
    X_hold_out = X[subsets[i]]
    y_hold_out = y[subsets[i]]

    for idx, j in enumerate([j for j in range(subset_count) if j != i]):
        X_train = np.concatenate([X[subsets[p]] for p in range(8) if p != i
                                  and p != j])
        y_train = np.concatenate([y[subsets[p]] for p in range(8) if p != i
                                  and p != j])

        X_test = X[subsets[j]]
        y_test = y[subsets[j]]
```

```
        w_hat_lst = list()

        for lmd_idx, lmd in enumerate(lambda_vals):
            w_hat = la.inv(X_train.T@X_train + lmd*np.identity(p))@X_train.
↪T@y_train
            w_hat_lst.append(w_hat)
            y_hat = X_test@w_hat

            param_err_RLS[lmd_idx] += np.sum(np.sign(y_hat) != y_test) / 16

        # print(param_err_SVD)
        k_best = np.argmin(param_err_RLS)
        w_hat_best = w_hat_lst[k_best]

        y_hat_hold_out = X_hold_out@w_hat_best
        error_RLS[i][idx] = np.sum(np.sign(y_hat_hold_out) != y_hold_out) / 16
print(f"Error estimate: {np.mean(error_RLS)}")
```

```
Error estimate: 0.0546875
```

(c)

[ ]:
```
X = d['X']
y = d['y']
new_features = X@np.random.rand(9,3)
X_new = np.concatenate((X, new_features), axis = 1)

### (a)' ###
n, p = np.shape(X_new)

# error rate for regularized least squares
error_SVD = np.zeros((8, 7))
# SVD parameters to test
k_vals = np.arange(9) + 1
param_err_SVD = np.zeros(len(k_vals))

subset_count = 8

subsets = np.array_split(np.arange(n), 8)

for i in range(subset_count):
    X_hold_out = X_new[subsets[i]]
    y_hold_out = y[subsets[i]]

    for idx, j in enumerate([j for j in range(subset_count) if j != i]):
        X_train = np.concatenate([X_new[subsets[p]] for p in range(8) if p != i
                                  and p != j])
        y_train = np.concatenate([y[subsets[p]] for p in range(8) if p != i
```

```
                                          and p != j])

        U, Sigma, VT = la.svd(X_train, full_matrices=False)

        X_test = X_new[subsets[j]]
        y_test = y[subsets[j]]

        w_hat_lst = list()

        for k_idx, k in enumerate(k_vals):
            Sigma_plus = np.diag(1 / Sigma[:k])
            VT_k = VT[:k, :]
            U_k = U[:, :k]

            w_hat = VT_k.T@Sigma_plus@U_k.T@y_train
            w_hat_lst.append(w_hat)
            y_hat = X_test@w_hat

            param_err_SVD[k_idx] += np.sum(np.sign(y_hat) != y_test) / 16

        # print(param_err_SVD)
        k_best = np.argmin(param_err_SVD)
        w_hat_best = w_hat_lst[k_best]

        y_hat_hold_out = X_hold_out@w_hat_best
        error_SVD[i][idx] = np.sum(np.sign(y_hat_hold_out) != y_hold_out) / 16
print(f"Error estimate: {np.mean(error_SVD)}")
```

Error estimate: 0.049107142857142856

[ ]:
```
### (b)' ###
# error rate for regularized least squares
error_RLS = np.zeros((8, 7))
# 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

subsets = np.array_split(np.arange(n), 8)

for i in range(subset_count):
    X_hold_out = X_new[subsets[i]]
    y_hold_out = y[subsets[i]]

    for idx, j in enumerate([j for j in range(subset_count) if j != i]):
        X_train = np.concatenate([X_new[subsets[p]] for p in range(8) if p != i
```

```
                                 and p != j])
        y_train = np.concatenate([y[subsets[p]] for p in range(8) if p != i
                                 and p != j])

        X_test = X_new[subsets[j]]
        y_test = y[subsets[j]]

        w_hat_lst = list()

        for lmd_idx, lmd in enumerate(lambda_vals):
            w_hat = la.inv(X_train.T@X_train + lmd*np.identity(p))@X_train.
 ↪T@y_train
            w_hat_lst.append(w_hat)
            y_hat = X_test@w_hat

            param_err_RLS[lmd_idx] += np.sum(np.sign(y_hat) != y_test) / 16

        # print(param_err_SVD)
        k_best = np.argmin(param_err_RLS)
        w_hat_best = w_hat_lst[k_best]

        y_hat_hold_out = X_hold_out@w_hat_best
        error_RLS[i][idx] = np.sum(np.sign(y_hat_hold_out) != y_hold_out) / 16
print(f"Error estimate: {np.mean(error_RLS)}")
```

```
Error estimate: 0.05357142857142857
```

The average error rate of Truncated SVD solution is 0.052455357142857144. The average error rate of Regularized LS is 0.0546875. With new features, Avg error rate of SVD_new is 0.049107142857142856 and Avg error rate of RLS_new is 0.05357142857142857. It seems that these new features are helpful for classification, but not significantly so. One possible explanation is that since the new features are generated as random linear combinations of the original features, they might not necessarily add any meaningful or informative variance to the model that would improve classification.