CS/ECE/ME532 Assignment 6

1. Suppose $\boldsymbol{A}$ is an n-by-n symmetric, *rank-one* matrix with right singular vectors $\boldsymbol{v}_k, k = 1, 2, \ldots, n$. Show that the power method converges to $\boldsymbol{v}_1$ and determine the number of iterations needed to converge within 1% of $\boldsymbol{v}_1$ if your initial vector $\boldsymbol{b}_0 = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$.

2. Use the starter script available for download on the course page. The script loads the 1000 three-dimensional data points in `sdata.csv` into a 1000-by-3 matrix `X`. The second section of the code displays the data using a scatter plot format. We wish to approximate the data using a subspace. Use the rotate tool to view the data cloud from different perspectives.

a) Does the data appear to lie in a low-dimensional subspace? Why or why not? Remember the definition of a subspace.

b) What could you do to the data so that it lies (approximately) in a low-dimensional subspace?

c) The third section of the code removes the mean (average) value of the 1000 data points. Use the rotate tool to inspect the scatterplot of the data with the mean removed. Does the mean-removed data appear to lie in a low-dimensional subspace?

d) The fourth section of the code finds the SVD of the mean-removed data. If `a` is a unit-norm vector representing the best one-dimensional subspace for the mean-removed data, complete the line of code to define `a` in terms of the SVD matrices. The fifth section displays the one-dimensional subspace with the mean-removed data. Note that the length of the vector representing the subspace is scaled by the root-mean-squared value of the data for display purposes. Use the rotate tool to inspect the relationship between the subspace and the data, and comment on how well a one-dimensional subspace captures the data.

e) Let $\boldsymbol{x}_{zi}, i = 1, 2, \ldots, 1000$ be the individual mean-removed data points and $\boldsymbol{a}$ the unit-norm vector representing the best one-dimensional subspace for the data. Thus, $\boldsymbol{x}_{zi} \approx \boldsymbol{a} w_i$. Find w_i in terms of the SVD matrices $\boldsymbol{U}$, $\boldsymbol{S}$, and $\boldsymbol{V}$.

f) Now write the original data $\boldsymbol{x}_i, i = 1, 2, \ldots, 1000$ as $\boldsymbol{x}_i \approx \boldsymbol{a} w_i + \boldsymbol{b}$. What is $\boldsymbol{b}$?

g) Let $\boldsymbol{E}$ be the difference between $\boldsymbol{X}$ and the rank-one approximation. Find a

mathematical expression for $||\boldsymbol{E}||_F^2$ in terms of the singular values of the mean-removed data $\boldsymbol{X}_z$.

h) Now try a rank-two approximation. Use the SVD to find an orthonormal basis for the best plane containing the mean-removed data. Display the mean-removed data and the bases for the plane.

i) Your rank-two approximation for the original data is $\boldsymbol{x}_i \approx \boldsymbol{a}_1 w_{1i} + \boldsymbol{a}_2 w_{2i} + \boldsymbol{b}, i = 1, 2, \ldots, 1000$. Express $w_{2i}, i = 1, 2, \ldots, 1000$ in terms of the SVD of the mean-removed data matrix $\boldsymbol{X}_z$. Display a scatter plot of the original data (in red) and the rank-two approximations in blue. Does the rank-two approximation lie in a plane? Does that plane capture the dominant components of the data?

j) Let $\boldsymbol{E}$ be the difference between $\boldsymbol{X}$ and the rank-two approximation. Find a mathematical expression for $||\boldsymbol{E}||_F^2$ in terms of the singular values of the mean-removed data $\boldsymbol{X}_z$.

k) Find and compare the numerical values for $||\boldsymbol{E}||_F^2$ using both the rank-1 and rank-2 approximation.

3. Consider the face emotion classification problem studied previously. Design and compare the performances of the classifiers proposed in **a** and **b**, below. In each case, divide the dataset into 8 equal sized subsets (e.g., examples $1-16, 17-32$, etc). Use 6 sets of the data to estimate $\boldsymbol{w}$ for each choice of the *regularization parameter*, then select the best value for the regularization parameter by estimating the error on one of the two sets of data held out from training, and finally use the $\boldsymbol{w}$ corresponding to the best value of the regularization parameter to predict the labels of the remaining set of data that was held out. Compute the number of mistakes made on this hold-out set and divide that number by 16 (the size of the set) to estimate the error rate. Note there are $8 \times 7 = 56$ different choices of the two hold-out sets, so repeat this process 56 times and average the error rates across the 56 cases to obtain a final estimate.

a) Truncated SVD. Use the pseudo-inverse $\boldsymbol{V}\boldsymbol{\Sigma}_r^{-1}\boldsymbol{U}^T$, where $\boldsymbol{\Sigma}_r^{-1}$ is computed by inverting the r largest singular values. Hence the regularization parameter r takes values $r = 1, 2, \ldots, 9$.

b) Ridge Regression. Let $\widehat{\boldsymbol{w}}_\lambda = \arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2$, for the following values of the regularization parameter $\lambda = 0, 2^{-1}, 2^0, 2^1, 2^2, 2^3$, and 2^4. Show that $\widehat{\boldsymbol{w}}_\lambda$ can be computed using the SVD and use this fact in your code.

DEVIN BRESSER

CS/ECE/ME532 Assignment 6

1. Suppose $\boldsymbol{A}$ is an n-by-n symmetric, *rank-one* matrix with right singular vectors $\boldsymbol{v}_k$, $k = 1, 2, \ldots, n$. Show that the power method converges to $\boldsymbol{v}_1$ and determine the number of iterations needed to converge within 1% of $\boldsymbol{v}_1$ if your initial vector $\boldsymbol{b}_0 = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$.

1.) $\underset{n\times n}{A} = \underset{n\times n}{U}\ \underset{n\times n}{\Sigma}\ \underset{n\times n}{V^T}$, rank 1, symmetric, $\underline{b}_0 = 1/\sqrt{n} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$

a.) Show that power method converges to $\underline{V}_1$.

Let $\underline{B} = \underline{A}^T\underline{A}$.

for $k = 1, 2, \ldots$ (converge):

$$\underline{b}_k = \frac{\underline{B}\,\underline{b}_{k-1}}{\| \underline{B}\,\underline{b}_{k-1} \|_2}$$

Observe that $\underline{B}\,\underline{b}_{k-1} = \underline{B}\cdot\underline{B}\cdots\underline{B}\,\underline{b}_0 = \underline{B}^k\,\underline{b}_0$

Want right singular vector $\underline{V}_1$, aka eigenvector of $\underline{A}^T\underline{A}$ ($\underline{B}$)

→ Eigendecomposition of $\underline{B}$ is $\underline{V} \begin{bmatrix} \delta_1^2 & & \underline{0} \\ & \ddots & \\ \underline{0} & & \delta_m^2 \end{bmatrix} \underline{V}^T$.

→ In this case, that is just equal to $\underline{V}_1\,\delta_1^2\,\underline{V}_1^T$ (or $\underline{V}_1\,\lambda_1\,\underline{V}_1^T$) because $\underline{A}$ is rank 1, and thus $\underline{A}^T\underline{A}$ is also rank 1. (all δ's except δ_1 are zero).

→ $\underline{B}^k = \underline{V}_1\,\lambda_1\,\underline{V}_1^T\,\underline{V}_1\,\lambda_1\,\underline{V}_1^T \cdots = \underline{V}_1\,\lambda_1^k\,\underline{V}_1^T$

Now express $\underline{b}_0 = \underline{V}\,\underline{g} = \underline{V}\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \underline{v}_1 g_1 + \underline{v}_2 g_2 + \cdots + \underline{v}_n g_n$

So, $\underline{B}^k \underline{b}_0 = \underline{B}^k \left(\underline{v}_1 g_1 + \underline{v}_2 g_2 + \cdots + \underline{v}_n g_n\right)$

Note that, since rank $(\underline{B}) = 1$, $\begin{cases} \underline{B}^k \underline{v}_1 = \lambda_1^k \underline{v}_1 \\ \underline{B}^k \underline{v}_i = 0 \quad \text{for all } i > 1. \end{cases}$

So, $\underline{B}^k \underline{b}_0 = \lambda_1^k \underline{v}_1 g_1$

Now,

$$\underline{b}_k = \frac{\underline{B}\,\underline{b}_{k-1}}{\|\underline{B}\,\underline{b}_{k-1}\|_2} = \frac{\underline{B}^k \underline{b}_0}{\|\underline{B}^k \underline{b}_0\|_2} = \frac{\lambda_1^k \underline{v}_1 g_1}{\|\lambda_1^k \underline{v}_1 g_1\|_2} = \frac{\underline{v}_1}{\|v_1\|_2} = \underline{v}_1$$

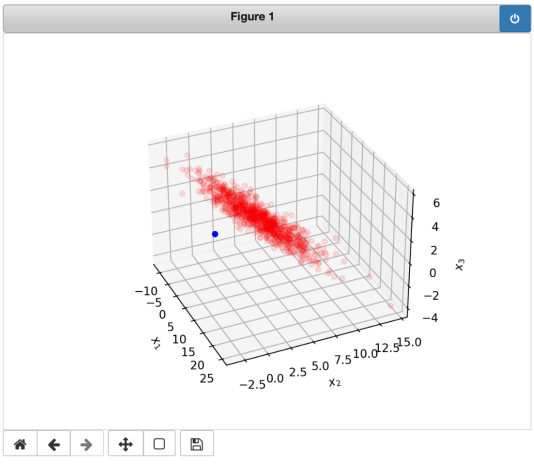
$\uparrow$ = 1 since $\underline{V}$ orthonormal

So, $\underline{b}_k = \underline{v}_1$ after just one iteration, regardless of $\underline{b}_0$.

problem 2.) a.) , b.), c.), d.) : See code snippets below: ↓

In [3]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:,0], X[:,1], X[:,2], c='r', marker='o', alpha=0.1)
ax.scatter(0,0,0,c='b', marker='o')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')

plt.show()
```

Figure 1



Problem 2a comment:
The data appears to be well represented by a low-dimensional plane, but not a subspace. This is because a subspace necessarily must include the origin and this data does not appear to be centered on the origin.

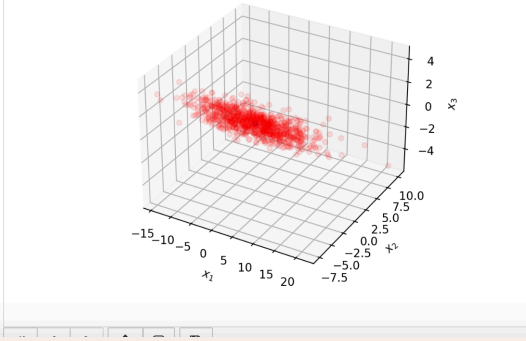
Problem 2b comment:

You could subtract the mean from every point in the dataset to center the data on the origin.

In [4]:
```
# Subtract mean
X_m = X - np.mean(X, 0)
```

In [5]:
```
# display zero mean scatter plot
fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='o', alpha=0.1)

ax.scatter(0,0,0,c='b', marker='o')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')

plt.show()
```

Figure 2



Problem 2c comment:

Yes, now that the mean has been subtracted, the data appears to be well represented by a low-dimensional subspace centered on the origin.

In [ ]:

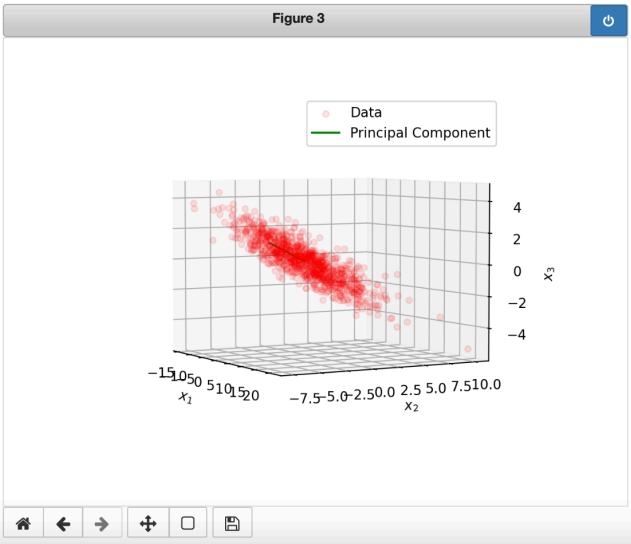
In [6]:
```
# Use SVD to find first principal component

U,s,VT = np.linalg.svd(X_m,full_matrices=False)

# complete the next line of code to assign the first principal component to a

a = VT[0,:]
a
```

Out[6]:
```
array([-0.87325954, -0.43370914,  0.2220679 ])
```

Problem 2d comment (1 of 2):

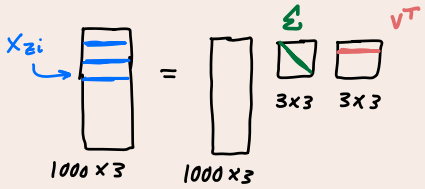
Since the data of matrix X is represented in the rows, we must perform PCA on the rows In this case, the first principal component is given by the first column of V Or equivalently the first row of VT.

In [7]:
```
# display zero mean scatter plot and first principal component

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

#scale length of line by root mean square of data for display
ss = s[0]/np.sqrt(np.shape(X_m)[0])

ax.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='o', label='Data', alpha=0.1)

ax.plot([0,ss*a[0]],[0,ss*a[1]],[0,ss*a[2]], c='g',label='Principal Component')

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')


ax.legend()
plt.show()
```



Problem 2d comment (2 of 2):

As we can see, the data is well represented by this one dimensional subspace.



2e.) $\underline{X}_{2i}$, $i = 1, 2, \ldots, 1000$

$\underline{X}_{2i} \approx \underline{a}\, w_i$. $\underline{a} = \underline{V}_1$

$$\underline{X} = \begin{bmatrix} -\underline{X}_{21}- \\ -\underline{X}_{22}- \\ \vdots \\ -\underline{X}_{2,1000}- \end{bmatrix} = \begin{bmatrix} -d_1^T- \\ -d_2^T- \\ \vdots \\ -d_{1000}^T- \end{bmatrix} \begin{bmatrix} -\underline{V}_1^T- \\ -\underline{V}_2^T- \\ -\underline{V}_3^T- \end{bmatrix} \quad \underline{V}^T \ 3\times3$$

$\underline{D} = \underline{U}\,\underline{\Sigma}$, 1000×3

$$\rightarrow \underline{X}_{2i} = \sum_{j=1}^{3} \underline{V}_j^T \left[\underline{d}_i\right]_j$$

$$\rightarrow \underline{X}_{2i} = \sum_{j=1}^{3} V_j^T \delta_j \left[\underline{U}\right]_{i,j}$$

→ When $p = 1$, $\underline{X}_{2i} \approx \sum_{j=1}^{1} V_j^T \delta_j \left[\underline{U}\right]_{i,j}$

→ $\underline{X}_{2i} \approx V_1^T \delta_1 \left[\underline{U}\right]_{i,1}$

→ $\underline{a} = \underline{V}_1^T$ 1×3

$w_i = \delta_1 \left[\underline{U}\right]_{i,1}$ 1×1

2f.) $\underline{X}_i \approx \underline{a}\, w_i + \underline{b}$.

1×3 1×3 1×3

$\underline{b}$ is the mean vector of $\underline{X}$, i.e. $\underline{b} = \frac{1}{1000}\left[\sum_{i=1}^{1000} x_{i,1} \quad \sum_{i=1}^{1000} x_{i,2} \quad \sum_{i=1}^{1000} x_{i,3}\right]$

2g.) $\underline{E} = \underline{X} - \underline{X}_{r1}$

$$\underline{E} = \sum_{i=2}^{3} \delta_i \underline{u}_i \underline{v}_i^T = \underline{X} - \delta_1 \underline{u}_1 \underline{v}_1^T = \underline{X} - \underline{X}_{r1}$$

$$\|\underline{E}\|_F^2 = \sum_{i=1}^{1000} \sum_{j=1}^{3} \left(\underline{E}_{i,j}\right)^2$$

Apply Eckart Young theorem: $\|X - X_r\|_F^2 = \sum_{i=k+1}^{r} \delta_i^2$

$\rightarrow \|E\|_F^2 = \sum_{i=k+1}^{r} \delta_i^2 = \sum_{i=2}^{3} \delta_i^2 = \delta_2^2 + \delta_3^2$

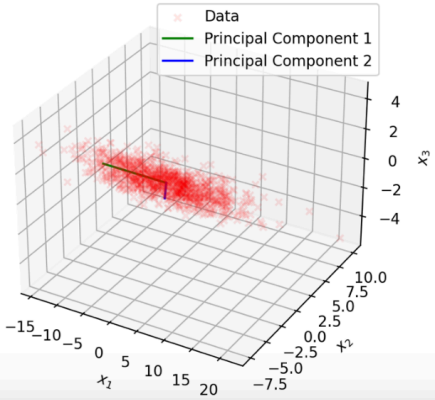
2h.) See Code snippet: ↓

Problem 2h

In [17]:
```
# Use SVD to find first and second principal components

U,s,VT = np.linalg.svd(X_m,full_matrices=False)
a2 = VT[0:2,:]
a2
```

Out[17]:
```
array([[-0.87325954, -0.43370914,  0.2220679 ],
       [-0.24467994, -0.00379784, -0.96959646]])
```

In [9]:
```
# display zero mean scatter plot and first & second principal components

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

#scale length of line by root mean square of data for display
s_sum = np.sum(s)
ss1 = s[0]/s_sum*10
ss2 = s[1]/s_sum*10

ax.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='x', label='Data', alpha=0.1)

ax.plot([0,ss1*a2[0,0]],[0,ss1*a2[0,1]],[0,ss1*a2[0,2]], c='g',label='Principal Component 1')
ax.plot([0,ss2*a2[1,0]],[0,ss2*a2[1,1]],[0,ss2*a2[1,2]], c='b',label='Principal Component 2')


ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')


ax.legend()
plt.show()
```



2i) See Code snippet: ↓

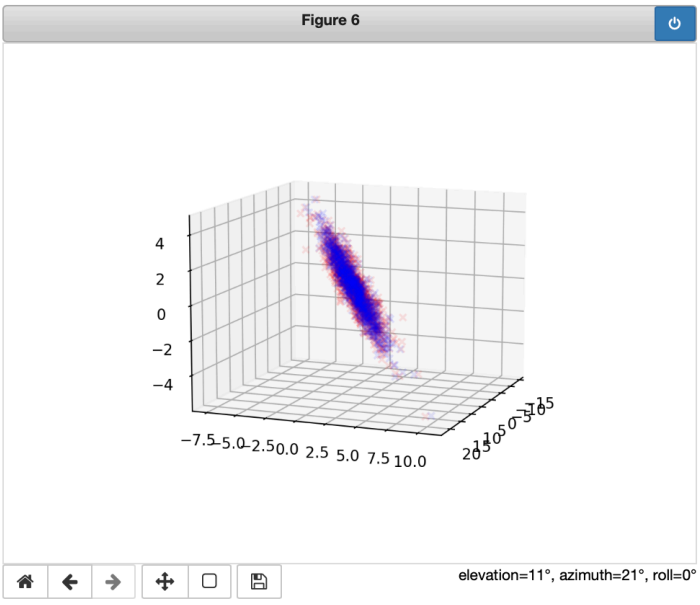
Problem 2i

In [10]:
```
# As follows from problem 2e solution,
# The rank-2 approximation is given by:
# x_z1 ~= v1^T * sigma_1 * [U]i,1 + v2^T * sigma_2 * [U]i,2

U,s,VT = np.linalg.svd(X_m,full_matrices=False)

#x_z1_rank2 = (s[0] * U[0,0] * VT[0,:]) + (s[1] * U[i,0] * VT[0,:])

# define rank1 and rank2 approximations

def x_zi_rank1(i):
    return (s[0] * U[i,0] * VT[0,:])

def x_zi_rank2(i):
    return ( (s[0] * U[i,0] * VT[0,:]) + (s[1] * U[i,1] * VT[1,:]) )

# This should reconstruct the original X_m matrix (for testing)
def x_zi_rank3(i):
    return ( (s[0] * U[i,0] * VT[0,:]) + (s[1] * U[i,1] * VT[1,:]) + (s[2] * U[i,2] * VT[2,:]))

print(x_zi_rank3(10), "\n", X_m[10])
```

```
[-5.96555563 -2.78729511  0.72139203]
 [-5.96555563 -2.78729511  0.72139203]
```

In [11]:
```
# compute the rank-2 approximated data points
x_zi_rank2s = []
for a in range(np.shape(X)[0]):
    x_zi_rank2s.append(x_zi_rank2(a))
x_zi_rank2s = np.vstack(x_zi_rank2s)
```

In [13]:
```
fig = plt.figure()
ax2i = fig.add_subplot(111, projection='3d')

ax2i.scatter(X_m[:,0], X_m[:,1], X_m[:,2], c='r', marker='x', label='Data', alpha=0.1)
ax2i.scatter(x_zi_rank2s[:,0], x_zi_rank2s[:,1], x_zi_rank2s[:,2], c='b', marker='x', label='Data', alpha=0.1)


ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$x_3$')

ax.legend()
plt.show()
```



Problem 2i comment:

The rank-2 approximation by definition lies in a 2-d plane. The plane appears to capture the dominant components of the data very well as variance in a 3rd orthogonal direction is comparatively low.

Apply Eckart Young theorem: $\| X - X_{r_2} \|_F^2 = \sum_{i=k+1}^{r} \delta_i^2$

2j.) $E = X - X_{r2}$

$$E = \sum_{i=3}^{3} \delta_i u_i v_i^T = \delta_3 u_3 v_3^T$$

$$\| E \|_F^2 = \sum_{i=3}^{3} \delta_i^2 = \delta_3^2.$$

2K.) See code snippet ↓

Problem 2k comment: We observe that the numerical value of (Frobenius norm of error)^2 decreases starkly as the rank of the approximation increases from 1 to 2.

In [19]:
```
E_frobsquared_rank1 = s[1]**2 + s[2]**2
E_frobsquared_rank2 = s[2]**2

print(f"||E||frob ^2, rank1: {E_frobsquared_rank1}")
print(f"||E||frob ^2, rank2: {E_frobsquared_rank2}")
```

```
||E||frob ^2, rank1: 626.6899203862769
||E||frob ^2, rank2: 152.94557577886462
```

P3 (SVD)

Classifier a.) Truncated SVD

Known result from lecture:

w_min = sum(i=1, i=p) 1/sigma_i * v_i * (u_i^T * d)

Truncate as follows:

w_min_r = sum(i=1, i=r) 1/sigma_i * v_i * (u_i^T * d)

85]:
```
# Split data into 8 equal sized slices

X_slices = np.split(X,8)
y_slices = np.split(y,8)


# split slices into 6-slice stacks
from itertools import combinations

X_stacks = []
y_stacks = []
X_holdouts = []
y_holdouts = []

# use combinations to get all possible 6-slice combinations
for combo in combinations(range(8), 6):
    X_stack = np.vstack([X_slices[i] for i in combo])
    y_stack = np.vstack([y_slices[i] for i in combo])
    X_stacks.append(X_stack)
    y_stacks.append(y_stack)

    # designate the X_holdouts and y_holdouts as the slices not assigned to each stack
    holdout_indices = [i for i in range(8) if i not in combo]
    X_holdout = np.vstack([X_slices[i] for i in holdout_indices])
    y_holdout = np.vstack([y_slices[i] for i in holdout_indices])
    X_holdouts.append(X_holdout)
    y_holdouts.append(y_holdout)

# convert to nparrays for better processing
X_stacks = np.array(X_stacks)
y_stacks = np.array(y_stacks)
X_holdouts = np.array(X_holdouts)
y_holdouts = np.array(y_holdouts)
```

414]:
```
# this function computes a list of w_min's for the all values of r from 1 to rank(input)

def w_min_rs(X_, y_):
    U_,s_,VT_ = np.linalg.svd(X_, full_matrices = False)
    UT_ = U_.T
    V_ = VT_.T
    sigma_ = np.diag(s_)
    rows_ = 9
    cols_ = 9
    sigma_ = np.zeros_like(np.zeros((rows_, cols_)));
    np.fill_diagonal(sigma_, s_)
    w_min_rs_ = []

    for i in range(np.linalg.matrix_rank(X_)):
        # truncate sigma
        sigma_inv_ = np.linalg.inv(sigma_)
        sigma_i_ = sigma_inv_ * (np.arange(sigma_inv_.shape[0]) < i+1)[:, None]
        w_min_rs_.append(V_ @ sigma_i_ @ UT_ @ y_)

    return w_min_rs_

# this function estimates the error rate some training and test data
# it also returns the best value of r
def estimate_error(X_train, y_train, X_test, y_test, regularization_type):

    # compute the w_mins with the function above
    w_min_rs_ = w_min_rs(X_train, y_train)

    error_rates = []

    # for each value of r
    if(regularization_type == "truncated_SVD"):

        for w_min_r in w_min_rs_:

            # compute y_pred by using the w_min_r value on X_test
            y_pred = X_test @ w_min_r

            # compute the proportion of errors for that value of r
            # this is a binary classifier so use sign of predictions, and take mean
            error = np.mean(np.sign(y_pred) != np.sign(y_test))

            error_rates.append(error)
```

```
    # for each value of r
    elif(regularization_type == "ridge_regression"):

        lambdas_ = [2**i for i in range(-1, 5)]
        w_min_lambdas_ = ridge_regression(X_train, y_train, lambdas_)

        for w_min_lambda in w_min_lambdas_:

            # compute y_pred by using the w_min_lambda value on X_test
            y_pred = X_test @ w_min_lambda

            # compute the proportion of errors for that value of lambda
            # this is a binary classifier so use sign of predictions, and take mean
            error = np.mean(np.sign(y_pred) != np.sign(y_test))

            error_rates.append(error)

    # find the (first) value of r that minimizes error
    best_r = np.argmin(error_rates)

    # find the error rate corresponding to the optimal value of r
    best_error_rate = error_rates[best_r]

    return best_r, best_error_rate
```

[ ]:
```
# for each X_stack (6-long), 28 total :
    # -select the best value of r
    # -use the w corresponding to the best value of r to predict the labels of holdout set 1
    # -compute the % error of these predicted labels
    # function estimate_error above does all of this
    # run estimate_error with each X_stack, y_stack, X_holdout 1, y_holdout 1

    # -use the w corresponding to the best value of r to predict the labels of holdout set 2
    # -compute the % error of these predicted labels
    # function estimate_error above does all of this
    # run estimate_error with each X_stack, y_stack, X_holdout 2, y_holdout 2
```

434]:
```
error_rates_holdout1_SVD = []
error_rates_holdout2_SVD = []

# for each X_stack (6-long, 28 total):
for i in range(len(X_stacks)):
    # append the computed error rate at the optimal value of r
    error_rates_holdout1_SVD.append(estimate_error(X_stacks[i], y_stacks[i], X_holdouts[i][:16],
                                                   y_holdouts[i][:16], "truncated_SVD")[1])
    error_rates_holdout2_SVD.append(estimate_error(X_stacks[i], y_stacks[i], X_holdouts[i][16:],
                                                   y_holdouts[i][16:], "truncated_SVD")[1])

error_rates_overall_SVD = np.concatenate([error_rates_holdout1_SVD, error_rates_holdout2_SVD])
print(f"Truncated SVD: \nError rate average over both holdout sets: {np.mean(error_rates_overall_SVD)}")
```

```
Truncated SVD:
Error rate average over both holdout sets: 0.03459821428571429
```

P3 (Ridge Regression)

Classifier b.) - Ridge Regression

several of the functions above can be re-used so I will call them as needed

Also, apply result from Lecture video 4.1:

w_min = V (Sigma^2 + lambda*I)^-1 @ Sigma @ U^T @ y

This result is derived from taking the expression:

w_min = (A^T A + lambda*I)^-1 @ A^T @ y

and plugging in the SVD for A, A = U @ Sigma @ V^T

In [428]:
```
def ridge_regression(X_, y_, lambdas_):

    U_, s_, VT_ = np.linalg.svd(X_, full_matrices=False)
    V_ = VT_.T
    sigma_ = np.diag(s_)
    dimension_ = len(s_)
    w_min_lambdas_ = []

    for lambda_ in lambdas_:
        w_min_lambda = V_ @ np.linalg.inv(sigma_**2 + lambda_ * np.identity(dimension_)) @ sigma_ @ U_.T @ y_
        w_min_lambdas_.append(w_min_lambda)

    return w_min_lambdas_
```

In [432]:
```
error_rates_holdout1_ridge = []
error_rates_holdout2_ridge = []

# for each X_stack (6-long, 28 total):
for i in range(len(X_stacks)):
    # append the computed error rate at the optimal value of r
    error_rates_holdout1_ridge.append(estimate_error(X_stacks[i], y_stacks[i], X_holdouts[i][:16],
                                                     y_holdouts[i][:16], "ridge_regression")[1])
    error_rates_holdout2_ridge.append(estimate_error(X_stacks[i], y_stacks[i], X_holdouts[i][16:],
                                                     y_holdouts[i][16:], "ridge_regression")[1])

error_rates_overall_ridge = np.concatenate([error_rates_holdout1_ridge, error_rates_holdout2_ridge])
print(f"Ridge Regression: \nError rate average over both holdout sets: {np.mean(error_rates_overall_ridge)}")
```

```
Ridge Regression:
Error rate average over both holdout sets: 0.03571428571428571
```