In [152]:
```
import numpy as np
import numpy.matlib as mat
import numpy.linalg as la
import matplotlib.pyplot as plt
import scipy.io as sio
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.rcParams['figure.figsize'] = [8, 4]
plt.rcParams['figure.dpi'] = 100 # 200 e.g. is really fine, but slower
```

Question 1

Gram-Schmidt Orthogonalization Algorithm

In [2]:
```
def gram_schmidt(X):
    p = X.shape[1]
    n = X.shape[0]
    U = np.zeros((n, p))

    # set the first column of orthonormal basis
    u1 = X[:, 0] / la.norm(X[:, 0])
    U[:, 0] = u1.ravel()

    # compute other columns
    for j in range(1, p):
        res = X[:, j] - U[:, 0:j] @ U[:, 0:j].T @ X[:, j]

        # append 0 if residual is 0
        if all(np.round(res, 5) == 0):
            uj = np.zeros(n).ravel()
        else:
            uj = res / la.norm(res)
        U[:, j] = uj.ravel()

    # remove 0 columns
    U = U[:, ~np.all(U == 0, axis=0)]
    return U
```

In [3]:
```
X = np.matrix([[2,0,0],
               [0,2,0],
               [0,0,2]])
print(X, '\n\n', gram_schmidt(X))
```

```
[[2 0 0]
 [0 2 0]
 [0 0 2]]

 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [4]:
```
# first and last columns are linearly dependent
X1 = np.matrix([[1,-3,5,2],
                [0,1,-4,0],
                [9,0,-4,18],
                [0,0,2,0]])
print(X1, '\n\n', gram_schmidt(X1))
```

```
[[ 1 -3  5  2]
 [ 0  1 -4  0]
 [ 9  0 -4 18]
 [ 0  0  2  0]]

 [[ 0.11043153 -0.94229941 -0.22737251]
 [ 0.          0.31797758 -0.68211752]
 [ 0.99388373  0.10469993  0.02526361]
 [ 0.          0.          0.69453523]]
```

In [5]:
```
# p is greater than n
X2 = np.random.random((3,5))
print(X2, '\n\n', gram_schmidt(X2))
```

```
[[0.80060004 0.22442678 0.9217612  0.4131072  0.62807938]
 [0.87751708 0.61799057 0.57360841 0.11735741 0.55228937]
 [0.0728151  0.96372602 0.94001776 0.67995768 0.54353799]]

 [[ 0.67272595 -0.23311828  0.70220771]
 [ 0.73735759  0.13274777 -0.6623306 ]
 [ 0.06118487  0.96334516  0.2611944 ]]
```

Question 2

a)

We can use the `gram_schmidt` function I created in the earlier question.

In [224]:
```
X = np.matrix([[np.sqrt(2), 3, 0],
               [0, 2, np.sqrt(7)]])
print(X)
```

```
[[1.41421356 3.         0.        ]
 [0.         2.         2.64575131]]
```

In [225]:
```
U = gram_schmidt(X)
print(U)
```

```
[[1. 0.]
 [0. 1.]]
```

The rank of the matrix X is 2 since min(n, p) = 2 and no other columns or rows are linearly dependent. Therefore we can just take the 1st and 3rd columns of X and deduce that an orthonormal basis for this data coould be the matrix U printed above.

b)

i) $\underline{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ → subspace $V = \begin{bmatrix} \alpha v_1 \\ \alpha v_2 \end{bmatrix}$ → subspace is a line in 2-D space

$P = \underline{v}(\underline{v}^T\underline{v})^{-1}\underline{v}^T$

ii) $\|\underline{x}_i - Px_i\|_2^2$ for $i \in 1,2,3$

iii) if $\|v\|_2 = 1$ then $P = \underline{v}\,\underline{v}^T$ $\quad \underline{v} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$

$$\underset{v}{\arg\max} \frac{\underline{v}^T X^T X \underline{v}}{\underbrace{\underline{v}^T \underline{v}}_{1}} \rightarrow 11a_1^2 + 12a_1a_2 + 11a_2^2 = 12a_1a_2 + 11$$

$$12a_1\sqrt{1-a_1} + 11 \rightarrow \nabla_v f = -\frac{18a_1 - 12}{\sqrt{1-a_1}} = 0$$

$$a_1 = \frac{12}{18} = \frac{2}{3} \Rightarrow a_1$$

$$a_2 = \sqrt{\frac{5}{9}} = \frac{\sqrt{5}}{3}$$

c)

In [229]:
```
U, S, Vt = la.svd(X, full_matrices=True)
```

In [230]:
```
print(U)
```

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

In [231]:
```
print(np.diag(S))
```

```
[[4.12310563 0.        ]
 [0.         2.23606798]]
```

In [13]:
```
print((U @ np.diag(S) @ Vt).round(5))
```

```
[[ 1.41421  3.       0.     ]
 [-0.       2.       2.64575]]
```

Question 3

Question 3

a) Let $r = \min(n, p)$, then $X = \sum_{i=1}^{r} \underline{u}_i \sigma_i \underline{v}_i^T$

where u_i and v_i are i'th columns of U and V.

b) $X_k = \sum_{i=1}^{k} \underline{u}_i \sigma_i \underline{v}_i^T$

Question 4

In [14]:
```
X = np.matrix([[5, 0],
               [0, 1]])
```

In [15]:
```
U, S, Vt = la.svd(X, full_matrices=False)
```

In [16]:
```
print(U)
```

```
[[1. 0.]
 [0. 1.]]
```

In [17]:
```
print(np.diag(S))
```

```
[[5. 0.]
 [0. 1.]]
```

In [18]:
```
print(Vt)
```

```
[[1. 0.]
 [0. 1.]]
```

The columns and vectors of the matrix are already orthogonal to each other and most variation is displayed in the first column of the matrix. So orthonormal basis for the column and rows of the matrix are identity matrices with the same size as X.

Question 5

In [19]:
```
X = np.matrix([[-3, 0],
               [0, -1]])
```

In [20]:
```
U, S, Vt = la.svd(X, full_matrices=False)
```

In [21]:
```
print(U)
```

```
[[1. 0.]
 [0. 1.]]
```

In [22]:
```
print(np.diag(S))
```

```
[[3. 0.]
 [0. 1.]]
```

In [23]:
```
print(Vt)
```

```
[[-1. -0.]
 [-0. -1.]]
```

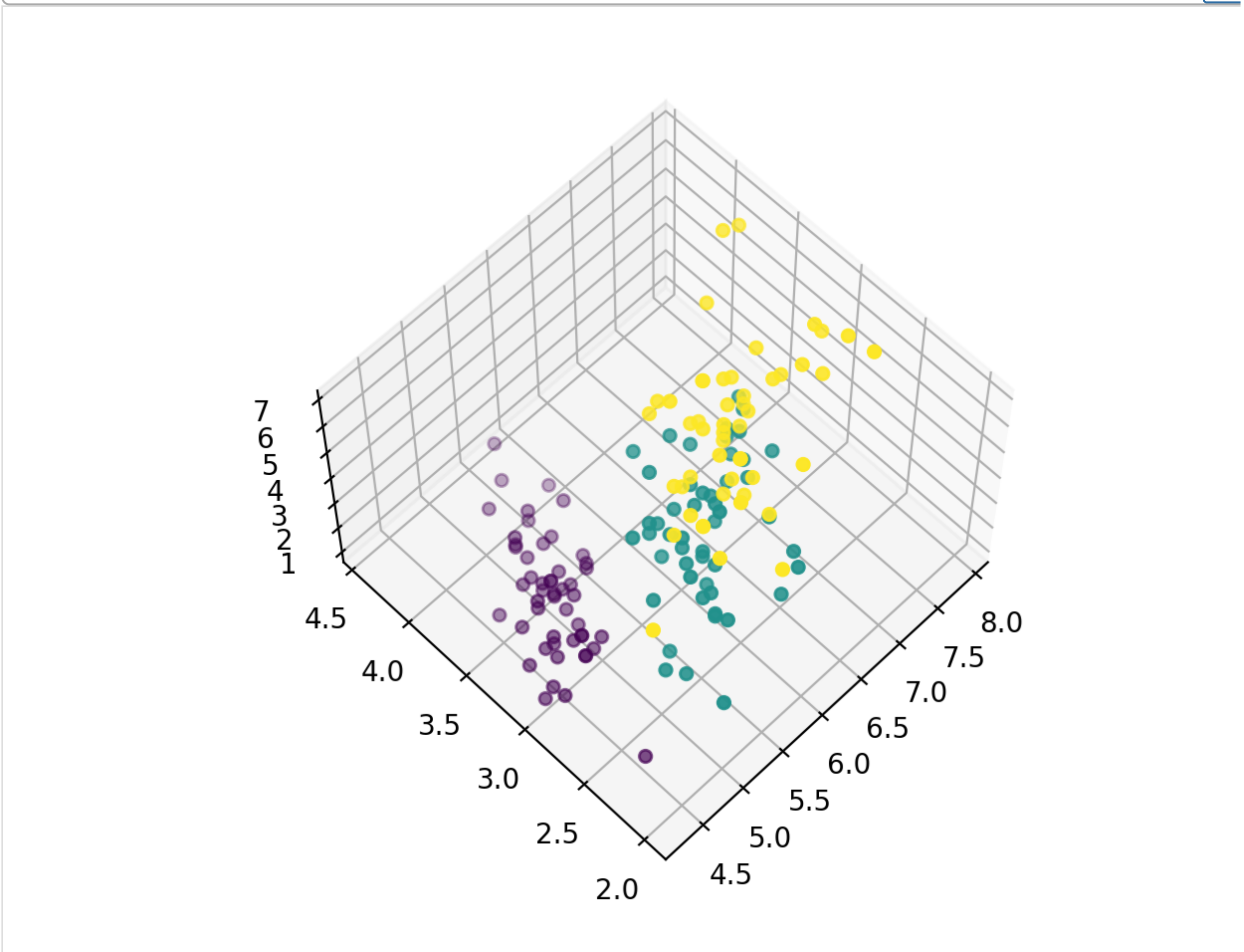
Question 6

In [24]:
```
matlab_data_file = sio.loadmat ('../hw3/fisheriris.mat')
meas = matlab_data_file['meas']
species = matlab_data_file['species']
y = np.matrix([-1.] * 50 + [0.] * 50 + [1.] * 50).T
```

a)

In [25]:
```
%matplotlib notebook
fig = plt.figure(21)
ax = fig.add_subplot(111, projection="3d")
p = ax.scatter(meas[:, 0], meas[:, 1], meas[:, 2], c=y)
```

Figure 21



We can find a 2-d plane that approximately contains the data points at this angle.

In [26]:
```
def reduce_dimensions(X, k):
    U, S, Vt = la.svd(X, full_matrices=False)
    S = np.diag(S)
    Uk = U[:, :k]
    Sk = S[:k, :k]
    Vtk = Vt[:k, :]
    Xk = Uk @ Sk @ Vtk
    Z = Sk @ Uk.T
    return Z.T
```

In [27]:
```
Z = reduce_dimensions(meas[:, :3], 2)
```

In [28]:
```
Z.shape
```

Out[28]:
```
(150, 2)
```

b)

In [29]:
```
def map_bins(x):
    if x < -.5:
        return -1
    elif x <= .5:
        return 0
    return 1

def compute_test_error(x, y, iterations, train_size):
    errors = np.zeros(iterations)
    for i in range(iterations):
        train_i = np.concatenate([
            np.random.choice(range(50), train_size, replace=False),
            np.random.choice(range(50, 100), train_size, replace=False),
            np.random.choice(range(100, 150), train_size, replace=False)
        ])
        train_x = x[train_i]
        train_y = y[train_i]

        test_i = np.array(list(set(range(150))- set(train_i)))

        test_x = x[test_i]
        test_y = y[test_i]

        w = la.inv(train_x.T @ train_x) @ train_x.T @ train_y
        results = test_x @ w
        errors[i] = (np.apply_along_axis(map_bins, 1, results) !=
                     np.array(test_y)[:, 0]).sum() / (150 - train_size * 3)
    return errors.mean()
```

Average test error with reduced dimensions:

In [30]:
```
compute_test_error(Z, y, 100, 40)
```

Out[30]:
```
0.03633333333333333
```

Question 7

In [31]:
```
# load the training data X and the training labels y
matlab_data_file = sio.loadmat ('../hw2/face_emotion_data.mat')
X = matlab_data_file['X']
y = matlab_data_file['y']
# n = number of data points
# p = number of features
n, p = np.shape(X)
```

In [203]:
```
Xs = np.array(np.split(X, 8))
ys = np.array(np.split(y, 8))
```

a) Truncated SVD

In [204]:
```
holdout_errors = []

full_idx = set(range(8))
for i in range(8):
    for j in range(8):
        if i == j:
            continue

        idx = list(full_idx - {i, j})
        X_train = np.concatenate(Xs[idx])
        y_train = np.concatenate(ys[idx])
        X_test = Xs[j]
        y_test = ys[j]
        X_holdout = Xs[i]
        y_holdout = ys[i]

        U, S, Vt = la.svd(X_train, full_matrices=False)

        best_error = 1
        for k in range(1, 10):
            Sk = np.diag(np.pad(S[:k]  ** -1,
                                 pad_width=(0, 9 - k),
                                constant_values=0))
            w = Vt.T @ Sk.T @ U.T @ y_train
            preds = np.sign(X_test @ w)
            error_rate = (sum(preds != y_test) / len(y_test))[0]
            if error_rate <= best_error:
                best_error = error_rate
                best_k = k
                best_w = w

        holdout_preds = np.sign(X_holdout @ best_w)
        holdout_error = (sum(holdout_preds != y_holdout) / len(y_holdout))[
        holdout_errors.append(holdout_error)

final_error = sum(holdout_errors) / len(holdout_errors)
print(f'Average error rate is: {final_error}')
```

```
Average error rate is: 0.05133928571428571
```

b) Ridge Regression

In [200]:
```python
holdout_errors = []
lambda_vals = np.array ([0 , 0.5 , 1 , 2 , 4 , 8 , 16])

full_idx = set(range(8))
for i in range(8):
    for j in range(8):
        if i == j:
            continue

        idx = list(full_idx - {i, j})
        X_train = np.concatenate(Xs[idx])
        y_train = np.concatenate(ys[idx])
        X_test = Xs[j]
        y_test = ys[j]
        X_holdout = Xs[i]
        y_holdout = ys[i]

        U, S, Vt = la.svd(X_train, full_matrices=False)
        S = np.diag(S)

        best_error = 1
        for l in lambda_vals:
            w = (Vt.T @ la.inv(S.T @ S + l * np.identity(X_train.shape[1]))
                @ S.T @ U.T @ y_train)

            preds = np.sign(X_test @ w)
            error_rate = (sum(preds != y_test) / len(y_test))[0]
            if error_rate < best_error:
                best_error = error_rate
                best_l = l
                best_w = w

        holdout_preds = np.sign(X_holdout @ best_w)
        holdout_error = (sum(holdout_preds != y_holdout) / len(y_holdout))[
        holdout_errors.append(holdout_error)

final_error = sum(holdout_errors) / len(holdout_errors)
print(f'Average error rate is: {final_error}')
```

```
Average error rate is: 0.04799107142857143
```

c) With generated features

They would not be helpful except purely by chance. Basically by taking a random combination of the original 9 features, we are introducing some noise, which would not result in improved or reduced performance necessarily as there is no additional information gained with these new features that is not present in the original dataset.

In [221]:
```python
X_new = np.concatenate([X, X @ np.random.random((9,3))], axis=1)
Xs = np.array(np.split(X_new, 8))
```

Truncated SVD

In [222]:

```
holdout_errors = []

full_idx = set(range(8))
for i in range(8):
    for j in range(8):
        if i == j:
            continue

        idx = list(full_idx - {i, j})
        X_train = np.concatenate(Xs[idx])
        y_train = np.concatenate(ys[idx])
        X_test = Xs[j]
        y_test = ys[j]
        X_holdout = Xs[i]
        y_holdout = ys[i]

        U, S, Vt = la.svd(X_train, full_matrices=False)

        best_error = 1
        for k in range(1, 10):
            Sk = np.diag(np.pad(S[:k]  ** -1,
                                 pad_width=(0, 12 - k),
                                constant_values=0))
            w = Vt.T @ Sk.T @ U.T @ y_train
            preds = np.sign(X_test @ w)
            error_rate = (sum(preds != y_test) / len(y_test))[0]
            if error_rate <= best_error:
                best_error = error_rate
                best_k = k
                best_w = w

        holdout_preds = np.sign(X_holdout @ best_w)
        holdout_error = (sum(holdout_preds != y_holdout) / len(y_holdout))[
        holdout_errors.append(holdout_error)

final_error = sum(holdout_errors) / len(holdout_errors)
print(f'Average error rate is: {final_error}')
```

```
Average error rate is: 0.049107142857142856
```

In [223]:
```
holdout_errors = []
lambda_vals = np.array ([0 , 0.5 , 1 , 2 , 4 , 8 , 16])

full_idx = set(range(8))
for i in range(8):
    for j in range(8):
        if i == j:
            continue

        idx = list(full_idx - {i, j})
        X_train = np.concatenate(Xs[idx])
        y_train = np.concatenate(ys[idx])
        X_test = Xs[j]
        y_test = ys[j]
        X_holdout = Xs[i]
        y_holdout = ys[i]

        U, S, Vt = la.svd(X_train, full_matrices=False)
        S = np.diag(S)

        best_error = 1
        for l in lambda_vals:
            w = (Vt.T @ la.inv(S.T @ S + l * np.identity(X_train.shape[1]))
                @ S.T @ U.T @ y_train)

            preds = np.sign(X_test @ w)
            error_rate = (sum(preds != y_test) / len(y_test))[0]
            if error_rate < best_error:
                best_error = error_rate
                best_l = l
                best_w = w

        holdout_preds = np.sign(X_holdout @ best_w)
        holdout_error = (sum(holdout_preds != y_holdout) / len(y_holdout))[
        holdout_errors.append(holdout_error)

final_error = sum(holdout_errors) / len(holdout_errors)
print(f'Average error rate is: {final_error}')
```

```
Average error rate is: 0.06361607142857142
```

In [ ]: