1)

a) $x_1 = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix}$ $x_2 = \begin{pmatrix} 7 \\ 1 \\ 5 \end{pmatrix}$

$\|x_1\| = \sqrt{3^2+4^2+0^2} = 5$

$u_1 = \frac{x_1}{\|x_1\|} = \frac{x_1}{5} = \frac{1}{5}\begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} \\ \frac{4}{5} \\ 0 \end{pmatrix}$

$\bar{x}_2 = x_2 - u_1 u_1^T x_2 = \begin{pmatrix} 7 \\ 1 \\ 5 \end{pmatrix} - \begin{pmatrix} \frac{3}{5} \\ \frac{4}{5} \\ 0 \end{pmatrix} \begin{pmatrix} \frac{3}{5} & \frac{4}{5} & 0 \end{pmatrix} \begin{pmatrix} 7 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \\ 5 \end{pmatrix}$

$\|\bar{x}_2\| = \sqrt{4^2+(-3)^2+5^2} = 5\sqrt{2}$

$u_2 = \frac{\bar{x}_2}{\|\bar{x}_2\|} = \frac{\bar{x}_2}{5\sqrt{2}} = \frac{1}{5\sqrt{2}}\begin{pmatrix} 4 \\ -3 \\ 5 \end{pmatrix} = \begin{pmatrix} \frac{4}{5\sqrt{2}} \\ \frac{-3}{5\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$

$$\boxed{u_1 = \begin{pmatrix} \frac{3}{5} \\ \frac{4}{5} \\ 0 \end{pmatrix} \quad u_2 = \begin{pmatrix} \frac{4}{5\sqrt{2}} \\ \frac{-3}{5\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}}$$

b) Since U is the orthogonalization of X, $\text{span}(u_1, \ldots, u_k) = \text{span}(x_1, \ldots, x_k)$. In other words, the columns of U span the same subspace spanned by the columns of X, so a projection of y onto this subspace is the same regardless of whether we use the projection matrix for X or its orthogonalization U.

c) $\hat{y} = U(U^TU)^{-1}U^Ty$

Since U is an orthogonal matrix, $U^TU = I$, so $(U^TU)^{-1} = I$.

$$U = \begin{pmatrix} \frac{3}{5} & \frac{4}{5\sqrt{2}} \\ \frac{4}{5} & \frac{-3}{5\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$\hat{y} = UU^Ty = \begin{pmatrix} \frac{3}{5} & \frac{4}{5\sqrt{2}} \\ \frac{4}{5} & \frac{-3}{5\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{3}{5} & \frac{4}{5} & 0 \\ \frac{4}{5\sqrt{2}} & \frac{-3}{5\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 6 \\ -2 \end{pmatrix} = \begin{pmatrix} \frac{3}{5} & \frac{4}{5\sqrt{2}} \\ \frac{4}{5} & \frac{-3}{5\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{27}{5} \\ \frac{-12\sqrt{2}}{5} \end{pmatrix} = \boxed{\begin{pmatrix} \frac{33}{25} \\ \frac{144}{25} \\ \frac{-12}{5} \end{pmatrix}}$$

It is easier to compute the LS estimate using U instead of X because the $(U^TU)^{-1}$ term simplifies to I, leaving UU^Ty instead $U(U^TU)^{-1}U^Ty$ as we no longer have to find the inverse of a matrix, which may be computationally intensive.

2)

a) $S = \{x \in \mathbb{R}^3 : 2x_1 - 3x_2 + x_3 = 0\}$

This is a plane in $\mathbb{R}^3$, so any 2 linearly independent vectors from the origin to the plane would provide a basis for the subspace since the dimension of a plane is 2.

(1,0,-2) and (0,1,3) lie in S, so set: $\boxed{x_1 = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} \quad x_2 = \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix}}$

x_1 and x_2 are linearly independent (not multiples of each other), so they form a basis for S.

$P = X(X^TX)^{-1}X^T$

$$P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -2 & 3 \end{pmatrix}\left(\begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 3 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -2 & 3 \end{pmatrix}\right)^{-1}\begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 3 \end{pmatrix} = \boxed{\begin{pmatrix} \frac{5}{7} & \frac{3}{7} & \frac{-1}{7} \\ \frac{3}{7} & \frac{5}{14} & \frac{3}{14} \\ \frac{-1}{7} & \frac{3}{14} & \frac{13}{14} \end{pmatrix}}$$

b) $\|x_1\| = \sqrt{1^2 + 0^2 + (-2)^2} = \sqrt{5}$

$$u_1 = \frac{x_1}{\|x_1\|} = \frac{x_1}{\sqrt{5}} = \frac{1}{\sqrt{5}}\begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{5}} \\ 0 \\ \frac{-2}{\sqrt{5}} \end{pmatrix}$$

$$\bar{x}_2 = x_2 - u_1u_1^Tx_2 = \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix} - \begin{pmatrix} \frac{1}{\sqrt{5}} \\ 0 \\ \frac{-2}{\sqrt{5}} \end{pmatrix}\begin{pmatrix} \frac{1}{\sqrt{5}} & 0 & \frac{-2}{\sqrt{5}} \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix} = \begin{pmatrix} \frac{6}{5} \\ 1 \\ \frac{3}{5} \end{pmatrix}$$

$\|\bar{x}_2\| = \sqrt{(\frac{6}{5})^2 + 1^2 + (\frac{3}{5})^2} = \sqrt{\frac{14}{5}}$

$$u_2 = \frac{\bar{x}_2}{\|\bar{x}_2\|} = \frac{\bar{x}_2}{\sqrt{\frac{14}{5}}} = \sqrt{\frac{5}{14}}\begin{pmatrix} \frac{6}{5} \\ 1 \\ \frac{3}{5} \end{pmatrix} = \begin{pmatrix} \frac{6}{5}\sqrt{\frac{5}{14}} \\ \sqrt{\frac{5}{14}} \\ \frac{3}{5}\sqrt{\frac{5}{14}} \end{pmatrix}$$

$$\boxed{u_1 = \begin{pmatrix} \frac{1}{\sqrt{5}} \\ 0 \\ \frac{-2}{\sqrt{5}} \end{pmatrix} \quad u_2 = \begin{pmatrix} \frac{6}{5}\sqrt{\frac{5}{14}} \\ \sqrt{\frac{5}{14}} \\ \frac{3}{5}\sqrt{\frac{5}{14}} \end{pmatrix}}$$

$$P = UU^T = \begin{pmatrix} \frac{1}{\sqrt{5}} & \frac{6\sqrt{5}}{5\sqrt{14}} \\ 0 & \frac{\sqrt{5}}{\sqrt{14}} \\ \frac{-2}{\sqrt{5}} & \frac{3\sqrt{5}}{5\sqrt{14}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{5}} & 0 & \frac{-2}{\sqrt{5}} \\ \frac{6\sqrt{5}}{5\sqrt{14}} & \frac{\sqrt{5}}{\sqrt{14}} & \frac{3\sqrt{5}}{5\sqrt{14}} \end{pmatrix} = \boxed{\begin{pmatrix} \frac{5}{7} & \frac{3}{7} & -\frac{1}{7} \\ \frac{3}{7} & \frac{5}{14} & \frac{3}{14} \\ -\frac{1}{7} & \frac{3}{14} & \frac{13}{14} \end{pmatrix}}$$

c) $$P_S x = \begin{pmatrix} \frac{5}{7} & \frac{3}{7} & -\frac{1}{7} \\ \frac{3}{7} & \frac{5}{14} & \frac{3}{14} \\ -\frac{1}{7} & \frac{3}{14} & \frac{13}{14} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$d = \|x - P_S x\| = \sqrt{(1-1)^2 + (1-1)^2 + (1-1)^2} = \boxed{0}$$

The distance between x and the projection of x onto S is 0 because the projection of x is x itself.

Evidently, x already lies in the subspace: $2(1) - 3(1) + 1 = 0$

3) all parts in Jupyter Notebook

4)

a) We know rank(A) = # of linearly columns of A = # of linearly independent rows of A.

$$P_S = \begin{pmatrix} 4 & 6 & 0 \\ 2 & 3 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

Let the rows of P_S be $P_S{}^T{}_1$, $P_S{}^T{}_2$, and $P_S{}^T{}_3$.
Clearly, $P_S{}^T{}_1 = 2P_S{}^T{}_2$, so row 2 of P_S is redundant. $P_S{}^T{}_1$ and $P_S{}^T{}_3$ are not scalar multiples of each other, so they are linearly independent.
Thus, $\text{rank}(P_S) = 2$.

This means P_S projects a vector in $\mathbb{R}^3$ onto a subspace of dimension 2 in $\mathbb{R}^3$, or a plane in $\mathbb{R}^3$
If we find 2 linearly independent vectors that lie in S, they form a basis for S.

Let $y_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ and $y_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$.

$$v_1 = P_S y_1 = \begin{pmatrix} 4 & 6 & 0 \\ 2 & 3 & 0 \\ 0 & 3 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 0 \end{pmatrix}$$

$$v_2 = P_S y_2 = \begin{pmatrix} 4 & 6 & 0 \\ 2 & 3 & 0 \\ 0 & 3 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 6 \\ 3 \\ 3 \end{pmatrix}$$

Both v_1 and v_2 lie in S and are not sc lar multiples of each other, so they are linearly independent.

Thus, a basis for S is: $\boxed{v_1 = \begin{pmatrix} 4 \\ 2 \\ 0 \end{pmatrix} \quad v_2 = \begin{pmatrix} 6 \\ 3 \\ 3 \end{pmatrix}}$

b) First, find some vector n that is orthogonal to v_1 and v_2, meaning it is orthogonal to S.

let $n = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \Rightarrow$ $v_1^T n = 0$: $4a + 2b = 0 \Rightarrow b = -2a$

$v_2^T n = 0$: $6a + 3b + 3c = 0$

$6a + 3(-2a) + 3c = 0$

$6a - 6a + 3c = 0 \Rightarrow c = 0$

This gives: $b = -2a$

set $a = 1 \Rightarrow b = -2$ and we know $c = 0$: $\boxed{x_1 - 2x_2 = 0}$ gives an equation for S

S must contain $(0,0,0)$, $(4,2,0)$, and $(6,3,3)$.

$(0,0,0)$: $0 - 2(0) = 0$

$(4,2,0)$: $4 - 2(2) = 0$

$(6,3,3)$: $6 - 2(3) = 0$

c) code at end

In [1]:

```python
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from matplotlib import colors

import numpy as np
import numpy.linalg as la
```

3a

Since there are three types of flowers, we can label the setosa flower as -1, the versicolor flower as 0, and the virginica flower as 1. Then, once we produce a real-valued prediction, we can turn that into a discrete prediction by finding whether our real-valued prediction is closest to -1, 0, or 1.

3b

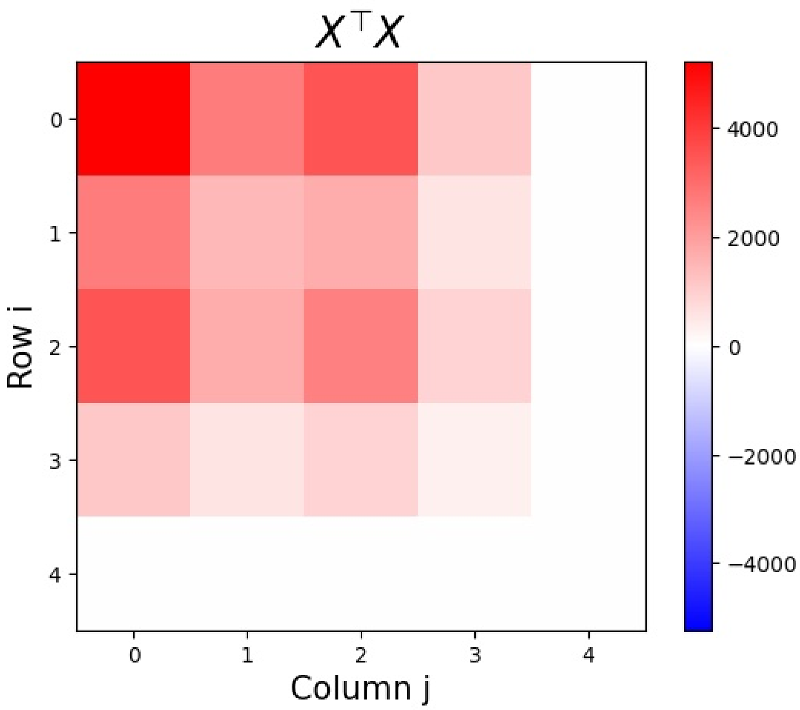
In [2]:

```python
def plot_matrix_pairwise_column_inner_prods(X: np.ndarray) -> None:
    """
    This function plots X^T X.

    Inputs: X (np.ndarray): has shape (n, p).
    """
    plt_arr = X.T @ X
    print(plt_arr)

    # find the max of the absolute vals of X^T X to set the colorscale
    max_val = np.max(np.abs(plt_arr))

    # This color norm ensures that 0 is always plotted as white
    color_norm = colors.TwoSlopeNorm(vmin=-1 * max_val, vcenter=0, vmax=max_val)

    plt.imshow(plt_arr, cmap="bwr", norm=color_norm)
    plt.ylabel("Row i", size=15)
    plt.xlabel("Column j", size=15)
    plt.colorbar()
    plt.title("$X^\\top X$", size=20)
    plt.show()
```

In [3]:

```python
# STARTER CODE
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from matplotlib import colors

# load data, make sure `fisheriris.mat` is in your working directory
data = scipy.io.loadmat("fisheriris.mat")
X = data["meas"]
y_text = data["species"]

############################
# YOUR CODE BELOW
# Process and assign numerical values to
# `y` according to your (a), make sure `y` is a 1d numpy array.
# If dimensions are mismatching, you may find `y = y.flatten()` useful.
y = np.copy(y_text)
y[y == "setosa"] = -1
y[y == "versicolor"] = 0
y[y == "virginica"] = 1
y = np.array(y, dtype=float)

# Compute the least squares weights
w = la.inv(X.T @ X) @ X.T @ y

# Compute the residuals
r = y - X @ w

# Make a plot
A = np.column_stack((X, r))
plot_matrix_pairwise_column_inner_prods(A)
```

```
[[ 5.22385000e+03  2.67343000e+03  3.48376000e+03  1.12814000e+03
   9.38660038e-13]
 [ 2.67343000e+03  1.43040000e+03  1.67430000e+03  5.31890000e+02
  -1.50990331e-13]
 [ 3.48376000e+03  1.67430000e+03  2.58271000e+03  8.69110000e+02
  -1.08657305e-13]
 [ 1.12814000e+03  5.31890000e+02  8.69110000e+02  3.02330000e+02
  -1.42968982e-16]
 [ 9.38660038e-13 -1.50990331e-13 -1.08657305e-13 -1.42968982e-16
   7.71772902e+00]]
```



In this method, we used the given code from the recitation to plot rows against columns for $X^T \cdot X$. Here, the only entries that are of concern are where row $i = 0, 1, 2, 3$ and column $j = 4$, representing X_1, X_2, X_3, X_4 and r, respectively. To ensure that r is orthogonal to the columns of X, we compute the inner product between X_i and r and see that they are all 0, which we printed out above the plot.

Nonetheless, another way to calculate this without using a plot is to simply compute $X^T \cdot r$ and see that we approximately get the 0 vector as follows:

In [4]:
```
X.T @ r
```

Out[4]:
```
array([[ 9.40136857e-13],
       [-1.49213975e-13],
       [-1.15463195e-13],
       [-2.99760217e-15]])
```

Problem 3c

In [5]:
```
# STARTER CODE
import numpy as np
import scipy.io

# load data, make sure `fisheriris.mat` is in your working directory
data = scipy.io.loadmat("fisheriris.mat")
# training data
X = data["meas"]
y_text = data["species"]

############################
# YOUR CODE BELOW
# Process and assign numerical values to
# `y` according to your (a), make sure `y` is a 1d numpy array.
# If dimensions are mismatching, you may find `y = y.flatten()` useful.
y = np.copy(y_text)
y[y == "setosa"] = -1
y[y == "versicolor"] = 0
y[y == "virginica"] = 1
y = np.array(y, dtype=float)

# number of random trials
N = 10_000
# array to store errors
errs = np.zeros(N)
# size of training set
num_train = 40

for i in np.arange(N):
    # initialize 0-length arrays for the train and holdout indices. These
    # arrays will be filled in the inner loop.
    idx_train = np.zeros(0, dtype=np.intp)
    idx_holdout = np.zeros(0, dtype=np.intp)

    # There are 3 label types and 50 samples of each type
    for label_type in range(3):
        # Choose a random ordering of the 50 samples
        r = np.random.permutation(50)
        # Add the first num_train indices of the random ordering to
```

```
        # the idx_train array
        idx_train = np.concatenate((idx_train, 50 * label_type + r[:num_train]))
        # Add the rest of the indices to the idx_holdout array
        idx_holdout = np.concatenate((idx_holdout, 50 * label_type + r[num_train:]))

    # divide data and labels into the train and holdout sets
    Xt = X[idx_train]
    yt = y[idx_train]
    Xh = X[idx_holdout]
    yh = y[idx_holdout]

    ########################
    # YOUR CODE BELOW
    # Solve for the LS weights
    w = la.inv(Xt.T @ Xt) @ Xt.T @ yt

    # Make predictions using the LS weights
    predictions = Xh @ w

    # Turn the real-valued predictions into class labels
    for k in range(len(predictions)):
        dist_neg, dist_zero, dist_pos = (
            abs(predictions[k] + 1),
            abs(predictions[k]),
            abs(predictions[k] - 1),
        )
        min_dist = min(dist_neg, dist_zero, dist_pos)
        if min_dist == dist_neg:
            predictions[k] = -1
        if min_dist == dist_zero:
            predictions[k] = 0
        if min_dist == dist_pos:
            predictions[k] = 1

    # Compute the errors
    count = 0
    for k in range(len(predictions)):
        if predictions[k] != yh[k]:
            count += 1
    errs[i] = count / 30
```

In [6]:

```
sum(errs) / len(errs)
```

Out[6]:

```
0.031970000000001116
```

As seen here, the average test error is roughly 3.2%, which indicates our model was quite successful.

Problem 3d

In [7]:

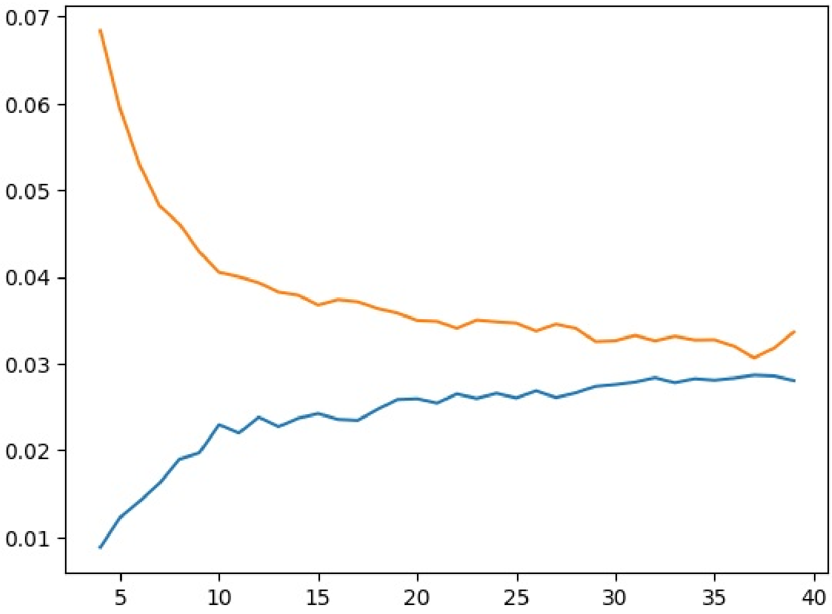
```
# STARTER CODE
import numpy as np
import scipy.io

# load data, make sure `fisheriris.mat` is in your working directory
data = scipy.io.loadmat("fisheriris.mat")
# training data
X = data["meas"]
y_text = data["species"]
############################
# YOUR CODE BELOW
# Process and assign numerical values to
# `y` according to your (a), make sure `y` is a 1d numpy array.
# If dimensions are mismatching, you may find `y = y.flatten()` useful.
y = np.copy(y_text)
y[y == "setosa"] = -1
y[y == "versicolor"] = 0
y[y == "virginica"] = 1
y = np.array(y, dtype=float)


# number of random trials
N = 1_000
# Min / Max size of the training set
min_num_train = 4
max_num_train = 40

# Arrays to store error rates
train_errs = np.zeros((max_num_train - min_num_train, N))
test_errs = np.zeros((max_num_train - min_num_train, N))

n_train_vals = np.arange(min_num_train, max_num_train)


for j, n_train in enumerate(n_train_vals):
    for i in np.arange(N):
```

```python
        # initialize 0-length arrays for the train and holdout indices.
        # These arrays will be filled in the inner loop.
        idx_train = np.zeros(0, dtype=np.intp)
        idx_holdout = np.zeros(0, dtype=np.intp)

        # There are 3 label types and 50 samples of each type
        for label_type in range(3):
            # Choose a random ordering of the 50 samples
            r = np.random.permutation(50)
            # Add the first num_train indices of the random ordering to
            # the idx_train array
            idx_train = np.concatenate((idx_train, 50 * label_type + r[:n_train]))
            # Add the rest of the indices to the idx_holdout array
            idx_holdout = np.concatenate((idx_holdout, 50 * label_type + r[n_train:]))

        # divide data and labels into the train and holdout sets
        Xt = X[idx_train]
        yt = y[idx_train]
        Xh = X[idx_holdout]
        yh = y[idx_holdout]

        ########################
        # YOUR CODE BELOW
        # Solve for the LS weights
        w = la.inv(Xt.T @ Xt) @ Xt.T @ yt

        # Make predictions using the LS weights
        predictions_train = Xt @ w
        predictions_test = Xh @ w

        # Turn the real-valued predictions into class labels
        for k in range(len(predictions_train)):
            dist_neg, dist_zero, dist_pos = (
                abs(predictions_train[k] + 1),
                abs(predictions_train[k]),
                abs(predictions_train[k] - 1),
            )
            min_dist = min(dist_neg, dist_zero, dist_pos)
            if min_dist == dist_neg:
                predictions_train[k] = -1
            if min_dist == dist_zero:
                predictions_train[k] = 0
            if min_dist == dist_pos:
                predictions_train[k] = 1

        for k in range(len(predictions_test)):
            dist_neg, dist_zero, dist_pos = (
                abs(predictions_test[k] + 1),
                abs(predictions_test[k]),
                abs(predictions_test[k] - 1),
            )
            min_dist = min(dist_neg, dist_zero, dist_pos)
            if min_dist == dist_neg:
                predictions_test[k] = -1
            if min_dist == dist_zero:
                predictions_test[k] = 0
            if min_dist == dist_pos:
                predictions_test[k] = 1

        # Compute the errors
        count_train = 0
        for k in range(len(predictions_train)):
            if predictions_train[k] != yt[k]:
                count_train += 1

        count_test = 0
        for k in range(len(predictions_test)):
            if predictions_test[k] != yh[k]:
                count_test += 1

        train_errs[j][i] = count_train / Xt.shape[0]
        test_errs[j][i] = count_test / Xh.shape[0]
```

In [8]:

```python
##################
# YOUR CODE BELOW
# Make a plot of the train and test errors as a function of
# training set size
avg_train_errs = [
    sum(train_errs[j]) / len(train_errs[j])
    for j in range(max_num_train - min_num_train)
]
avg_test_errs = [
    sum(test_errs[j]) / len(test_errs[j]) for j in range(max_num_train - min_num_train)
]

plt.plot(n_train_vals, avg_train_errs)
plt.plot(n_train_vals, avg_test_errs)
```

Out[8]: [<matplotlib.lines.Line2D at 0x7fc470205fd0>]



In the plot above, the train error is in blue and test error is in orange. We see that as the size of the training set increases, the train error increases which makes sense since it is a lot easier to classify the data accurately when there are fewer things to classify. On the other hand, as the size of the training set increases, the test error decreases which makes sense because we are able to train our data more effectively using the larger training set.

The end goal is to predict a new data point accurately, and we observe that the train and test set error are nearly identical when the number of samples is around 40. Therefore, it would be a good idea to use 40 as the training size, since then we get roughly similar performance on train and test data, and there is sufficient data for the training to occur properly. We also see that at this point, we stop improving our training accuracy and decreasing our test accuracy, so it suggests asymptotic behavior where we will get little improvement if we keep increasing our training size.

3e

In [9]:

```
# STARTER CODE
import numpy as np
import scipy.io

# load data, make sure `fisheriris.mat` is in your working directory
data = scipy.io.loadmat("fisheriris.mat")
# training data
X = data["meas"]
y_text = data["species"]

############################
# YOUR CODE BELOW
# Process and assign numerical values to
# `y` according to your (a), make sure `y` is a 1d numpy array.
# If dimensions are mismatching, you may find `y = y.flatten()` useful.
y = np.copy(y_text)
y[y == "setosa"] = -1
y[y == "versicolor"] = 0
y[y == "virginica"] = 1
y = np.array(y, dtype=float)
X = X[:, [0, 2]]

# number of random trials
N = 10_000
# array to store errors
errs = np.zeros(N)
# size of training set
num_train = 40

for i in np.arange(N):
    # initialize 0-length arrays for the train and holdout indices. These
    # arrays will be filled in the inner loop.
    idx_train = np.zeros(0, dtype=np.intp)
    idx_holdout = np.zeros(0, dtype=np.intp)

    # There are 3 label types and 50 samples of each type
    for label_type in range(3):
        # Choose a random ordering of the 50 samples
        r = np.random.permutation(50)
        # Add the first num_train indices of the random ordering to
        # the idx_train array
```

```python
        idx_train = np.concatenate((idx_train, 50 * label_type + r[:num_train]))
        # Add the rest of the indices to the idx_holdout array
        idx_holdout = np.concatenate((idx_holdout, 50 * label_type + r[num_train:]))

    # divide data and labels into the train and holdout sets
    Xt = X[idx_train]
    yt = y[idx_train]
    Xh = X[idx_holdout]
    yh = y[idx_holdout]

    ########################
    # YOUR CODE BELOW
    # Solve for the LS weights
    w = la.inv(Xt.T @ Xt) @ Xt.T @ yt

    # Make predictions using the LS weights
    predictions = Xh @ w

    # Turn the real-valued predictions into class labels
    for k in range(len(predictions)):
        dist_neg, dist_zero, dist_pos = (
            abs(predictions[k] + 1),
            abs(predictions[k]),
            abs(predictions[k] - 1),
        )
        min_dist = min(dist_neg, dist_zero, dist_pos)
        if min_dist == dist_neg:
            predictions[k] = -1
        if min_dist == dist_zero:
            predictions[k] = 0
        if min_dist == dist_pos:
            predictions[k] = 1

    # Compute the errors
    count = 0
    for k in range(len(predictions)):
        if predictions[k] != yh[k]:
            count += 1
    errs[i] = count / 30
```

In [10]:

```python
sum(errs) / len(errs)
```

Out[10]:

```
0.05715666666667161
```

As seen here, the average test error is roughly 5.7%, which indicates our model was quite successful. This error is higher than when we used all features, which intuitively makes sense because we are losing information by dropping some features, making it harder to classify the data points accurately. Note that the only change we made to this code from before was changing X to only include the first and third columns: $X = X[:, [0, 2]]$.

4c

In [11]:

```python
### STARTER CODE
import numpy as np
import numpy.linalg as la

p = np.array([[4, 6, 0.0], [2, 3, 0.0], [0.0, 3, 1.0]])
### YOUR CODE BELOW

y1 = np.array([[1], [0], [0]])
y2 = np.array([[0], [1], [0]])
y3 = np.array([[0], [0], [1]])

v1 = p @ y1
v2 = p @ y2
v3 = p @ y3

u1 = v1 / la.norm(v1)

v2bar = v2 - (u1.T @ v2) * u1
u2 = v2bar / la.norm(v2bar)

v3bar = v3 - (u1.T @ v3) * u1 - (u2.T @ v3) * u2
v3bar
```

Out[11]:

```
array([[0.],
       [0.],
       [0.]])
```

In this case, it is evident that the third vector is 0 which aligns with our understanding that the subspace is a plane in R^3, so it can be spanned by 2 linearly independent vectors that lie in the plane.

In [12]:

```python
print(u1)
```

```
print(u2)
```

```
[[0.89442719]
 [0.4472136 ]
 [0.        ]]
[[0.]
 [0.]
 [1.]]
```

Processing math: 100%