3a)

Identifying the correct order and arguments to do cross validation:

- shuffle_data(data_train)
- split_data(data_train, num_folds, fold)
- train_model(data_train, lambd)
- predict(data_train, model)
- loss(data_train, model)

  Note that the parameter model is the result of train_model function.

b)

Training error and test errors corresponding to each lambdas are found and plotted below. Also, lambda value of 0.47306122 gives the smallest test error.

In [78]:
```python
tab = {'Lambda': lambdas, 'Train Error': train_error, 'Test Error':test_error, '5 Fold Error':cross_5_error, '10 Fold Error': cross_10_error}

pd.DataFrame(tab)
```

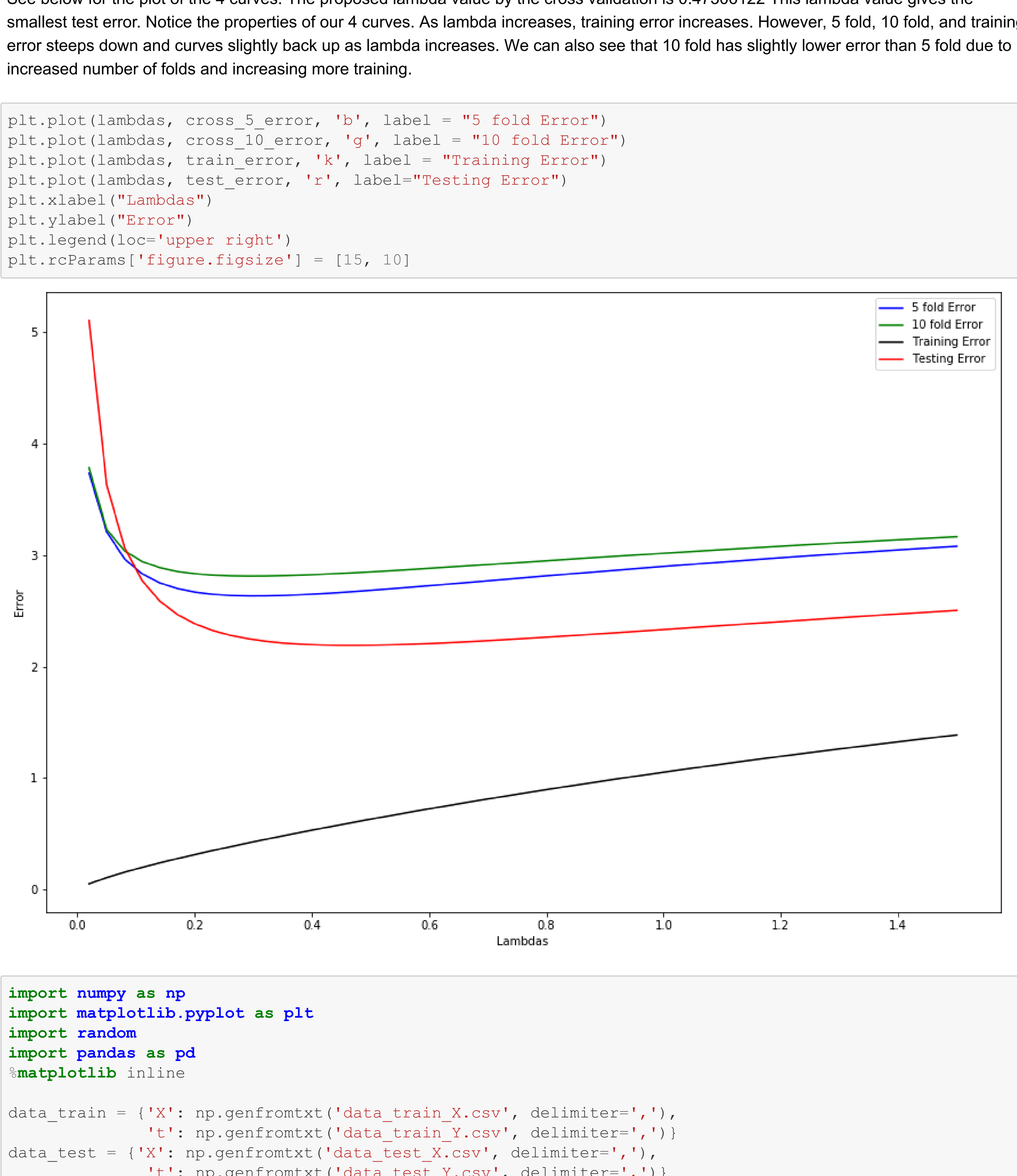
Out[78]:

	Lambda	Train Error	Test Error	5 Fold Error	10 Fold Error
0	0.020000	0.049736	5.106960	3.737667	3.784989
1	0.050204	0.105835	3.630834	3.208935	3.230740
2	0.080408	0.153970	3.070317	2.967352	3.039567
3	0.110612	0.197548	2.770942	2.832474	2.943195
4	0.140816	0.238130	2.587353	2.750857	2.887517
5	0.171020	0.276578	2.466006	2.699879	2.853570
6	0.201224	0.313408	2.382135	2.668094	2.832741
7	0.231429	0.348949	2.322605	2.649033	2.820466
8	0.261633	0.383420	2.279768	2.638754	2.814072
9	0.291837	0.416974	2.248855	2.634717	2.811871
10	0.322041	0.449721	2.226733	2.635216	2.812746
11	0.352245	0.481744	2.211254	2.639064	2.815921
12	0.382449	0.513103	2.200899	2.645414	2.820841
13	0.412653	0.543849	2.194560	2.653649	2.827100
14	0.442857	0.574020	2.191410	2.663308	2.834392
15	0.473061	0.603649	2.190823	2.674045	2.842484
16	0.503265	0.632762	2.192312	2.685593	2.851198
17	0.533469	0.661383	2.195496	2.697747	2.860390
18	0.563673	0.689531	2.200071	2.710347	2.869949
19	0.593878	0.717224	2.205794	2.723266	2.879787
20	0.624082	0.744477	2.212466	2.736403	2.889829
21	0.654286	0.771304	2.219925	2.749680	2.900019
22	0.684490	0.797719	2.228037	2.763032	2.910307
23	0.714694	0.823733	2.236690	2.776407	2.920656
24	0.744898	0.849357	2.245791	2.789766	2.931033
25	0.775102	0.874600	2.255262	2.803074	2.941412
26	0.805306	0.899474	2.265037	2.816304	2.951770
27	0.835510	0.923986	2.275058	2.829437	2.962091
28	0.865714	0.948145	2.285278	2.842453	2.972358
29	0.895918	0.971960	2.295656	2.855340	2.982560
30	0.926122	0.995438	2.306157	2.868088	2.992686
31	0.956327	1.018586	2.316749	2.880686	3.002729
32	0.986531	1.041413	2.327408	2.893130	3.012681
33	1.016735	1.063924	2.338111	2.905415	3.022538
34	1.046939	1.086127	2.348837	2.917536	3.032294
35	1.077143	1.108028	2.359571	2.929491	3.041946
36	1.107347	1.129633	2.370297	2.941280	3.051492
37	1.137551	1.150949	2.381002	2.952901	3.060929
38	1.167755	1.171980	2.391677	2.964355	3.070256
39	1.197959	1.192733	2.402310	2.975641	3.079473
40	1.228163	1.213214	2.412894	2.986761	3.088578
41	1.258367	1.233427	2.423422	2.997716	3.097571
42	1.288571	1.253378	2.433887	3.008508	3.106452
43	1.318776	1.273072	2.444285	3.019138	3.115222
44	1.348980	1.292514	2.454610	3.029609	3.123881
45	1.379184	1.311709	2.464858	3.039922	3.132430
46	1.409388	1.330661	2.475027	3.050080	3.140870
47	1.439592	1.349375	2.485113	3.060086	3.149201
48	1.469796	1.367855	2.495114	3.069941	3.157425
49	1.500000	1.386106	2.505029	3.079649	3.165542

c)

See below for the plot of the 4 curves. The proposed lambda value by the cross validation is 0.47306122 This lambda value gives the smallest test error. Notice the properties of our 4 curves. As lambda increases, training error increases. However, 5 fold, 10 fold, and training error steeps down and curves slightly back up as lambda increases. We can also see that 10 fold has slightly lower error than 5 fold due to increased number of folds and increasing more training.

In [80]:
```python
plt.plot(lambdas, cross_5_error, 'b', label = "5 fold Error")
plt.plot(lambdas, cross_10_error, 'g', label = "10 fold Error")
plt.plot(lambdas, train_error, 'k', label = "Training Error")
plt.plot(lambdas, test_error, 'r', label="Testing Error")
plt.xlabel("Lambdas")
plt.ylabel("Error")
plt.legend(loc='upper right')
plt.rcParams['figure.figsize'] = [15, 10]
```



In [81]:
```python
import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd
%matplotlib inline

data_train = {'X': np.genfromtxt('data_train_X.csv', delimiter=','),
              't': np.genfromtxt('data_train_Y.csv', delimiter=',')}
data_test = {'X': np.genfromtxt('data_test_X.csv', delimiter=','),
             't': np.genfromtxt('data_test_Y.csv', delimiter=',')}
```

In [82]:
```python
def shuffle_data(data):
    """
    Parameters
    ----------
    data : dictionary (bold_t, phi)
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.

    Returns
    -------
    data_shf
    returns its randomly permuted version along the samples.
        preserves the same target-feature pairs.
    """
    # Shuffle the indices of X array.
    index = list(range(len(data['X'])))
    random.shuffle(index)
    # Create a new empty list of X and t
    new_X = []
    new_t = []

    # rearrange the order of X and t array according to shuffled index
    for i in range(len(data['X'])):
        new_X.append(data['X'][index[i]])
        new_t.append(data['t'][index[i]])

    # convert the lists back into array and put them in new dictionary
    data_shf = {'X': np.array(new_X), 't': np.array(new_t)}
    return(data_shf)
```

In [83]:
```python
def split_data(data, num_folds, fold):
    """
    Parameters
    ----------
    data : dictionary
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.
    num_folds : int
        number of partitions
    fold : int
        selected partition

    Returns
    -------
    data_fold: selected partition of data
    data_rest: rest of partition of data
    """
    # fold_length is always an integer since we assume num_folds divides len(data)
    fold_length = len(data['X']) // num_folds

    # indices of the fold block
    fold_index = list(range((fold-1)*fold_length, fold*fold_length))

    # indices of all X array in data
    all_index = list(range(len(data['X'])))

    # Use list comprehension to remove fold index from all index
    rest_index = [item for item in all_index if item not in fold_index]


    data_fold = {'X': data['X'][(fold-1)*fold_length:fold*fold_length],
                 't': data['t'][(fold-1)*fold_length:fold*fold_length]}

    rest_X = []
    rest_t = []
    for i in range(len(rest_index)):
        rest_X.append(data['X'][rest_index[i]])
        rest_t.append(data['t'][rest_index[i]])

    data_rest = {'X': np.array(rest_X), 't':np.array(rest_t)}
    return(data_fold, data_rest)
```

In [84]:
```python
def train_model(data, lambd):
    """
    Parameters
    ----------
    data : dictionary
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.
    lambd : float
        penalty level coefficient.

    Returns
    -------
    model: coefficients of ridge regression.
    """
    # Store number of observations to obs
    obs = len(data['X'])
    # Store number of parameters to var
    var = len(data['X'][0])

    phi = np.reshape(data['X'], (obs, var))
    # Reshape X data array to matrix of obs by var
    # Stack target array to column vector
    t = np.vstack(data['t'])

    phiTphi = np.dot(np.transpose(phi), phi)

    # Compute w_hat
    model = np.dot(np.linalg.inv(phiTphi + lambd*np.identity(var)), np.dot(np.transpose(phi), t))

    # Reshape w_hat back into row vector
    model = np.hstack(model)
    return(model)
```

In [85]:
```python
def predict(data, model):
    """
    Parameters
    ----------
    data : dictionary
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.
    model : TYPE
        DESCRIPTION.

    Returns
    -------
    None.
    """
    # Store number of observations to obs
    obs = len(data['X'])
    # Store number of parameters to var
    var = len(data['X'][0])

    phi = np.reshape(data['X'], (obs, var))

    predictions = np.dot(phi, model)
    return(predictions)
```

In [86]:
```python
def loss(data, model):
    """


    Parameters
    ----------
    data : dictionary
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.
    model : TYPE
        DESCRIPTION.

    Returns
    -------
    None.
    """
    # Observed target vector
    t = data['t']

    # Compute predicted target values
    phi_w = predict(data, model)

    return(sum(np.square(t-phi_w))/len(data['X']))
```

In [87]:
```python
def cross_validation(data, num_folds, lambd_seq):
    """
    Parameters
    ----------
    data : dictionary
        Keys are 'X' and 't'. Contains ndarray of input vectors and
        target vector.
    num_folds : int
        number of CV folds.
    lambd_seq : TYPE
        DESCRIPTION.

    Returns
    -------
    None.
    """
    cv_error = []

    data = shuffle_data(data)
    for lambd in lambd_seq:
        cv_loss_lmd = 0
        for fold in range(1,num_folds+1):
            val_cv, train_cv = split_data(data, num_folds, fold)
            model = train_model(train_cv, lambd)
            cv_loss_lmd += loss(val_cv, model)
        cv_error.append(cv_loss_lmd / num_folds)
    return(cv_error)
```

In [90]:
```python
# Set random seed for plotting
np.random.seed(1)
# Make 50 intervals from 0.02 to 1.5
lambdas = np.linspace(0.02, 1.5, num=50)

# Compute 5 fold and 10 fold error rate
cross_5_error = cross_validation(data_train, 5, lambdas)
cross_10_error = cross_validation(data_train, 10, lambdas)

train_error = []
test_error = []
for lambd in lambdas:
    # Compute 50 model parameters from 50 lambdas
    model = train_model(data_train, lambd)
    # Compute training error using loss function
    train_error.append(loss(data_train, model))
    # Compute testing error using loss function
    test_error.append(loss(data_test, model))

# Finding the lambda proposed by your cross validation procedure.
print(lambdas[test_error.index(min(test_error))])
```

```
0.47306122448979593
```

In [ ]: