Problem 1: Support Vector Machines

Instructions:

1. Please use this q1.ipynb file to complete hw5-q1 about SVMs
2. You may create new cells for discussions or visualizations

In [1]:

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix, solvers
```

a): Linearly Separable Dataset

In [2]:

```
data = np.loadtxt('clean_lin.txt', delimiter='\t')
x = data[:, 0:2]
y = data[:, 2].reshape(-1, 1)

# adding a column of 1 to accomodate bias
x1 = np.ones((data.shape[0], 1))
x = np.concatenate([x1, x], axis = 1)

# defining matrices for cvxopt
Q = matrix(np.identity(x.shape[1]))
p = matrix(np.zeros((x.shape[1], 1)))

h  = -1 / np.abs(y)
h = matrix(h)
G = matrix(-x*y)

#training
out = solvers.qp(Q, p, G, h)
z = np.asarray(out['x'])
z
```

```
     pcost       dcost       gap    pres   dres
 0:  1.7890e+00  3.9171e+01  1e+02  2e+00  3e+01
 1:  1.4170e+01  1.4208e+01  4e+01  6e-01  8e+00
 2:  2.2524e+01  2.5627e+01  4e+01  5e-01  7e+00
 3:  4.8660e+01  5.0039e+01  1e+01  1e-01  2e+00
```

```
 4:  5.7717e+01  5.7639e+01  8e-01  6e-03  8e-02
 5:  5.8215e+01  5.8178e+01  9e-02  4e-04  6e-03
 6:  5.8237e+01  5.8237e+01  1e-03  4e-06  5e-05
 7:  5.8237e+01  5.8237e+01  1e-05  4e-08  5e-07
 8:  5.8237e+01  5.8237e+01  1e-07  4e-10  5e-09
Optimal solution found.
```

Out[2]:

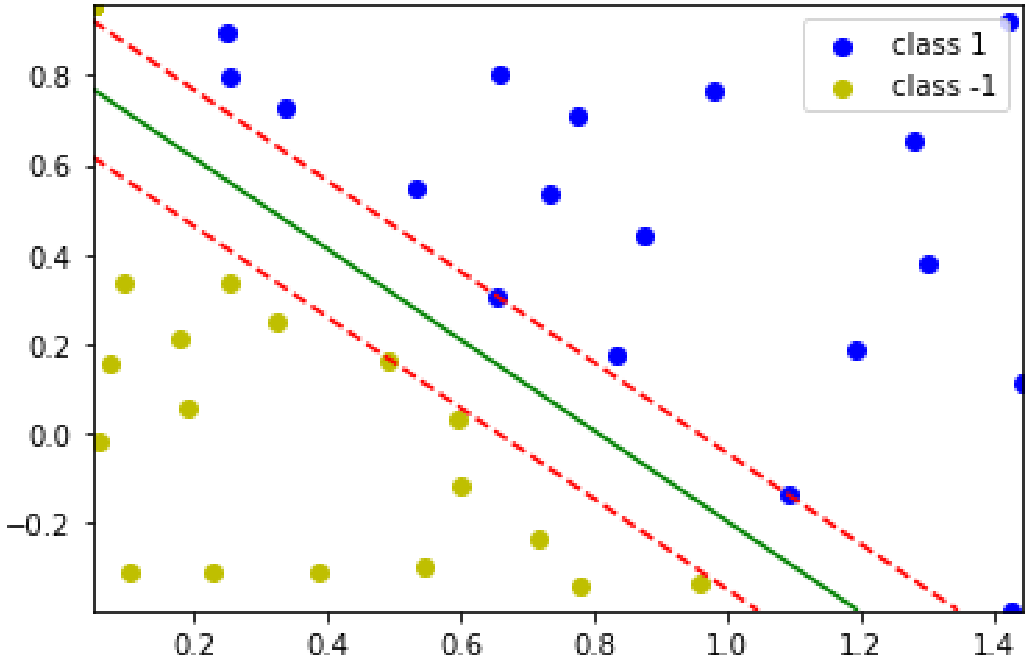
```
array([[-5.3742034 ],
       [ 6.6678144 ],
       [ 6.56755772]])
```

In [3]:

```python
# getting a, b, c from the weight matrix.
b = z[2]
a = z[1]
c = z[0]

X_1 = np.linspace(np.min(x[:,1]), np.max(x[:,1]), 20)
X_2 = np.linspace(np.min(x[:,2]), np.max(x[:,2]), 20)
x_, y_ = np.meshgrid(X_1, X_2)

# finding z as a*x1 + b*x2 + c
z_ = a*x_ + b*y_ + c

fig, ax = plt.subplots(1)
ax.contour(x_, y_, z_, levels=[0], colors='g')
ax.contour(x_, y_, z_, levels=[-1], colors='r',linestyles='dashed')
ax.contour(x_, y_, z_, levels=[1], colors='r',linestyles='dashed')
ax.scatter(x[np.where(y == 1),1], x[np.where(y == 1),2], color='b', label="class 1")
ax.scatter(x[np.where(y == -1),1], x[np.where(y == -1),2], color='y', label="class -1")
ax.legend()
```

Out[3]:

```
<matplotlib.legend.Legend at 0x7fd7a8cb2460>
```



b) and c) : Linearly Non-separable Dataset

In [4]:

```
# Load the data set that is not linearly separable
data = np.loadtxt('dirty_nonlin.txt', delimiter='\t')
x = data[:, 0:2]
y = data[:, 2].reshape(-1, 1)

# appending 1 to make the bias
x_ones = np.ones((x.shape[0],1))
x = np.concatenate([x_ones, x], axis=1)
```

In [5]:

```
#constructing G
e = np.identity(x.shape[0])
x_y = y*x
```

In [6]:

```
G = np.concatenate([x_y, e], axis=1)
h = np.ones(x.shape[0])
Q = np.identity(x.shape[0]+3)
p = np.ones(x.shape[0]+3)
p[:3] = 0
```

In [7]:

```
G = matrix(-G)
Q = matrix(Q)
p_ = matrix(0.05*p)
h = matrix(-h)
```

In [8]:

```
def solve(G, Q, p, h):
    solvers.options['show_progress'] = False
    out = solvers.qp(Q, p, G, h)
    z = np.asarray(out['x'])
    return z[:3]
```

In [9]:

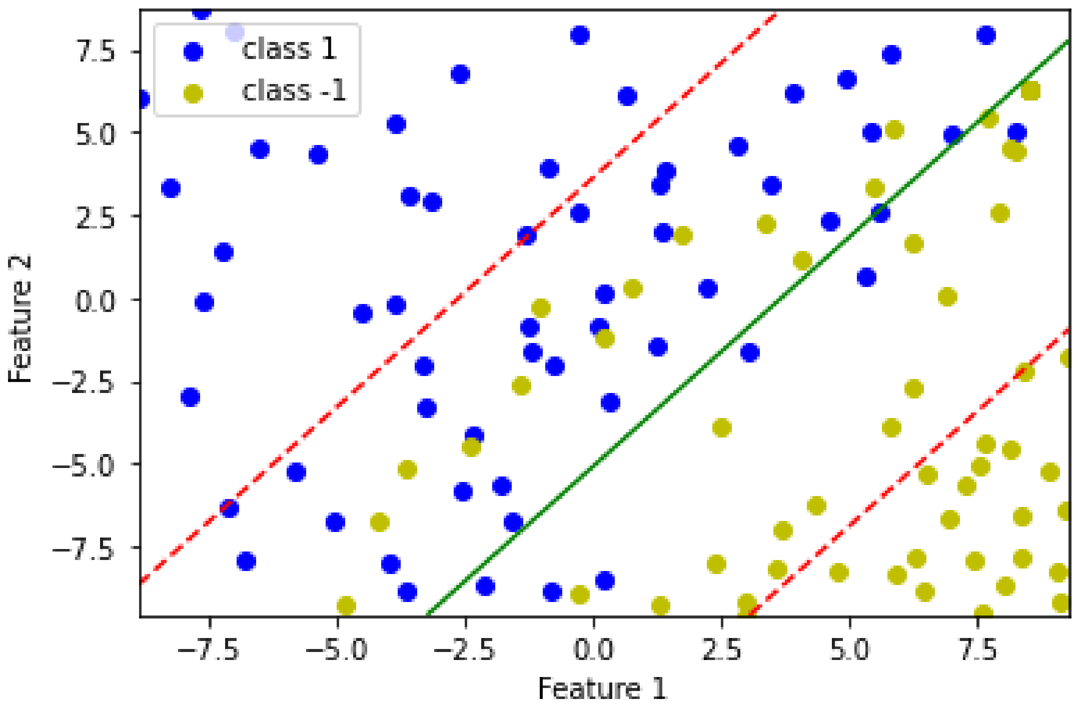
```
z = solve(G, Q, p_, h)
```

In [10]:

```
#plotting the decision boundary
def plot(x, y, z):
    X_1 = np.linspace(np.min(x[:,1]), np.max(x[:,1]), 20)
```

```python
    b = z[2]
    a = z[1]
    c = z[0]

    X_2 = np.linspace(np.min(x[:,2]), np.max(x[:,2]), 20)
    x_, y_ = np.meshgrid(X_1, X_2)

    z_ = a*x_ + b*y_ + c

    fig, ax = plt.subplots(1)
    ax.contour(x_, y_, z_, levels=[0], colors='g')
    ax.contour(x_, y_, z_, levels=[-1], colors='r',linestyles='dashed')
    ax.contour(x_, y_, z_, levels=[1], colors='r',linestyles='dashed')

    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")
    ax.scatter(x[np.where(y == 1),1], x[np.where(y == 1),2], color='b', label="class 1")
    ax.scatter(x[np.where(y == -1),1], x[np.where(y == -1),2], color='y', label="class -1")
    ax.legend()

print("Decision boundary when c = 0.05")
plot(x, y, z)
```

```
Decision boundary when c = 0.05
```

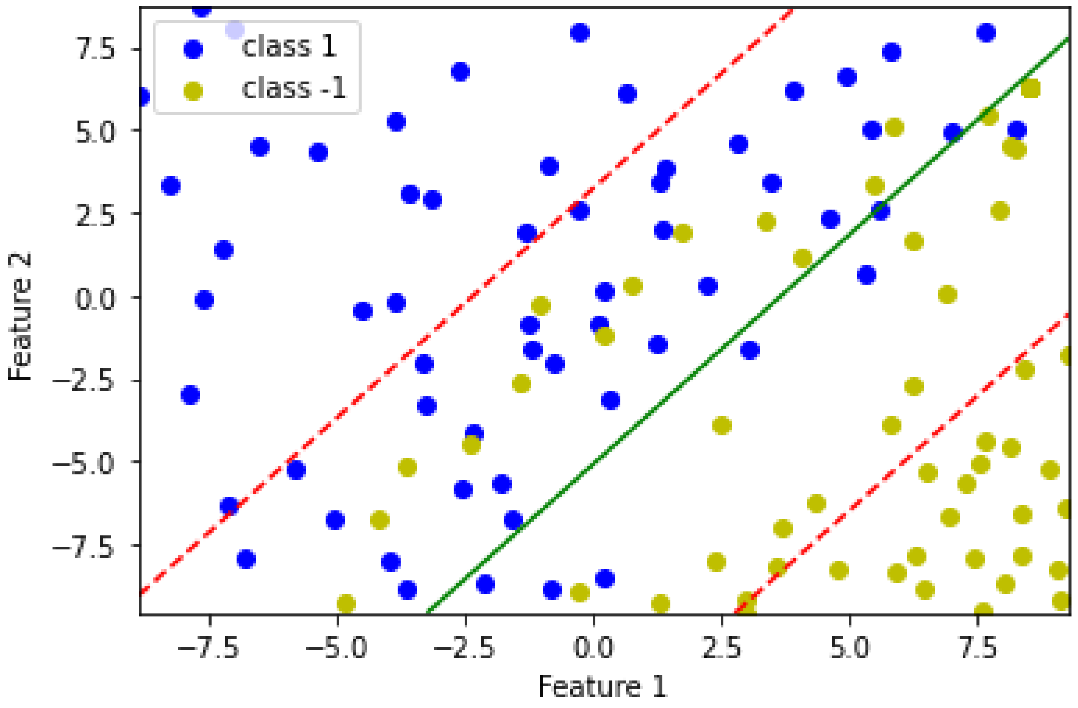


In [11]:

```python
# for different c values
C = [0.1, 1, 100, 1000000]
print("Decision boundary when c = ", str(C[0]))
p_ = matrix(C[0]*p)
```

```
z = solve(G, Q, p_, h)
plot(x, y, z)
```

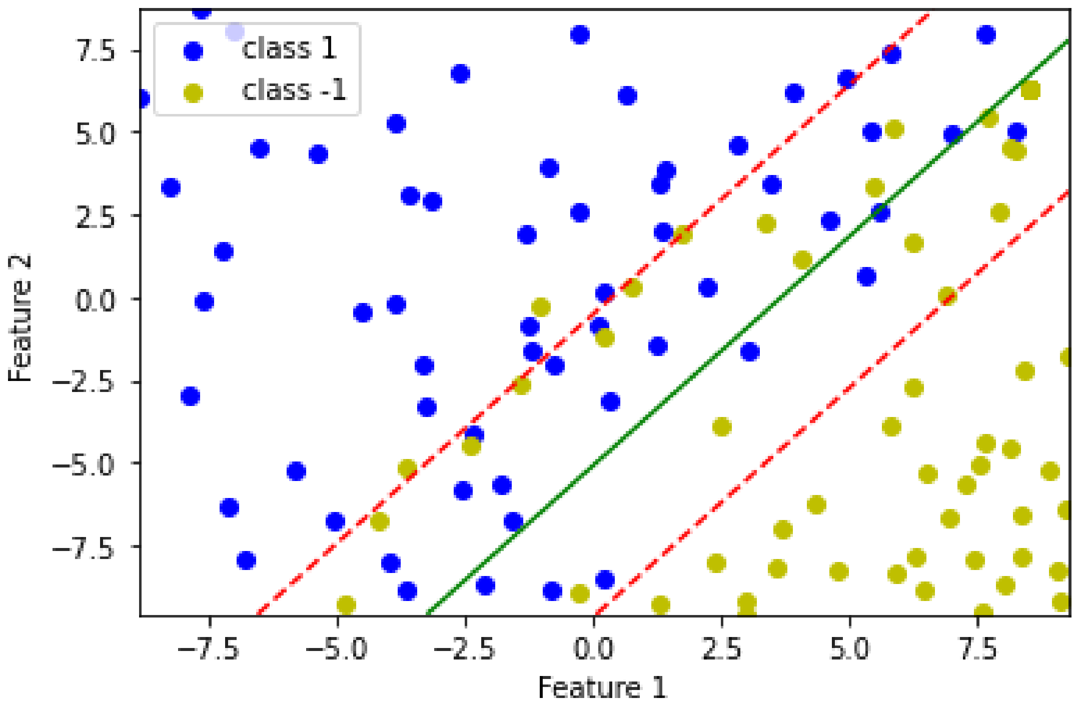
Decision boundary when c = 0.1



In [12]:

```
print("Decision boundary when c = ", str(C[1]))
p_ = matrix(C[1]*p)
z = solve(G, Q, p_, h)
plot(x, y, z)
```

Decision boundary when c = 1

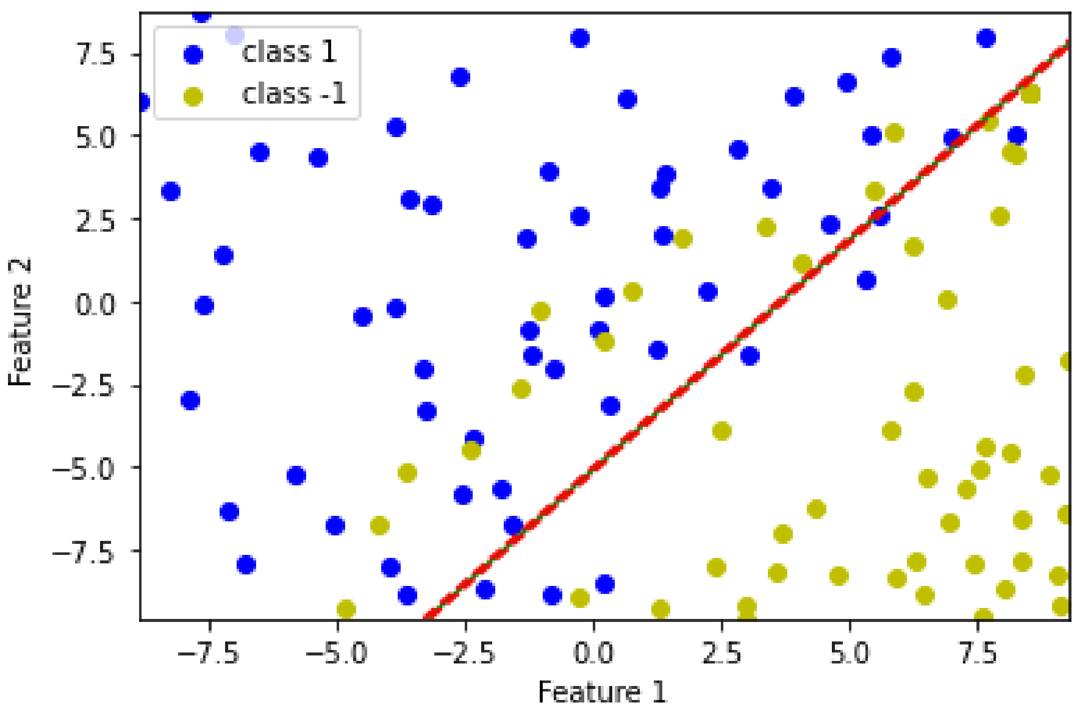


In [13]:

```
print("Decision boundary when c = ", str(C[2]))
```

```
p_ = matrix(C[2]*p)
z = solve(G, Q, p_, h)
plot(x, y, z)
```

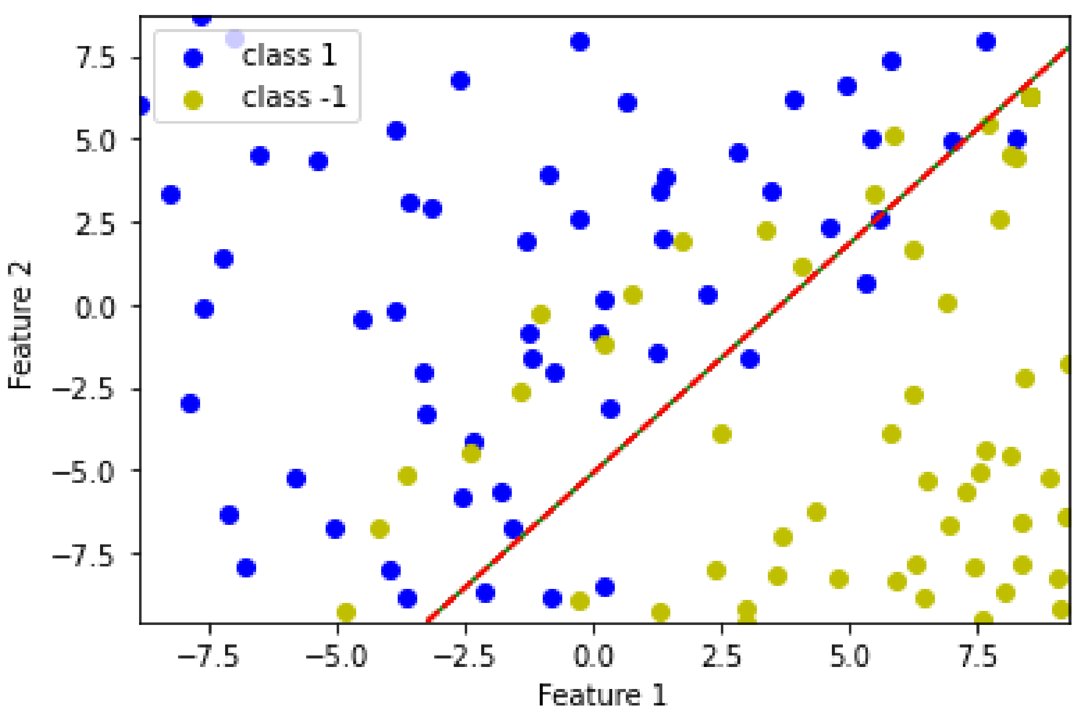
```
Decision boundary when c =  100
```



In [14]:

```
print("Decision boundary when c = ", str(C[3]))
p_ = matrix(C[3]*p)
z = solve(G, Q, p_, h)
plot(x, y, z)
```

```
Decision boundary when c =  1000000
```



Explain your observations here:

As the value of C increases the margin width reduces. This is because a higher value of C heavily penalizes the misclassified points. So the term which tries to reduce the hinge loss, will dominate the term which maximizes the margin. Hence the margin becomes narrow and almost coincide for higher c values.

In [44]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, KFold
```

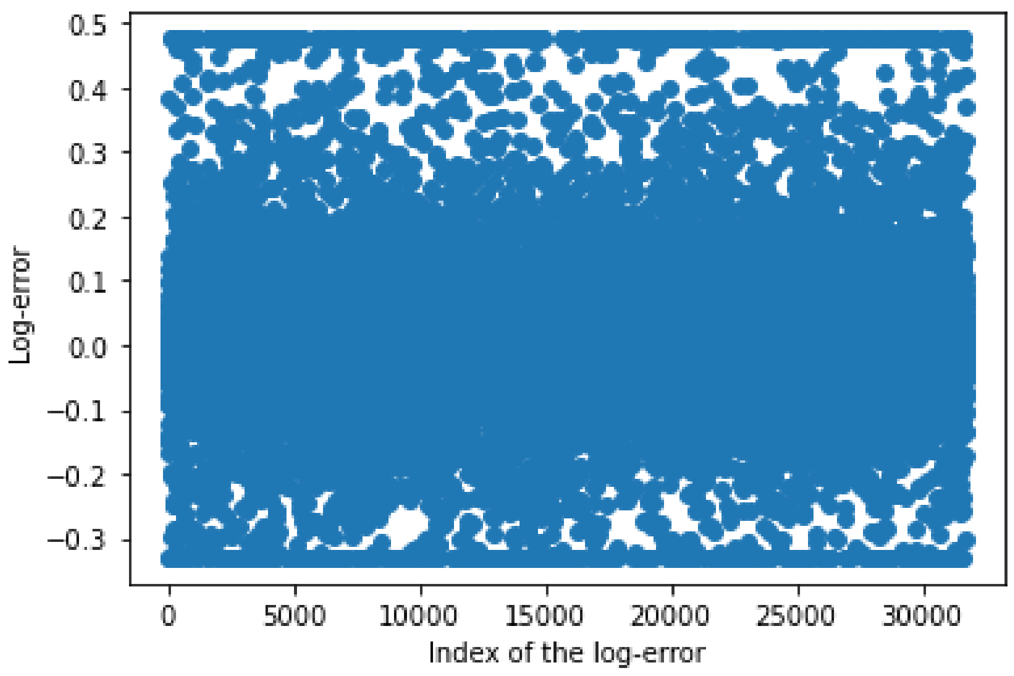
a) load/merge data and visualize logerror

In [2]:
```
# load data into DataFrames
train = pd.read_csv("/home/akshay/Downloads/MAIL/HW5/HW5/q2_data/train.csv")
properties = pd.read_csv("/home/akshay/Downloads/MAIL/HW5/HW5/q2_data/properties.csv")
data_pd = train.merge(properties, on="id", how="inner")
data = data_pd.to_numpy()
```

In [3]:
```
# eliminate outliers
# taking column 2, which is the log error
one_percent = np.percentile(data[:,1], 1)
ninety_nine_percent = np.percentile(data[:,1], 99)
low_idx = np.where(data[:,1] < one_percent)
high_idx = np.where(data[:,1] > ninety_nine_percent)
data[low_idx, 1] = one_percent
data[high_idx, 1] = ninety_nine_percent
```

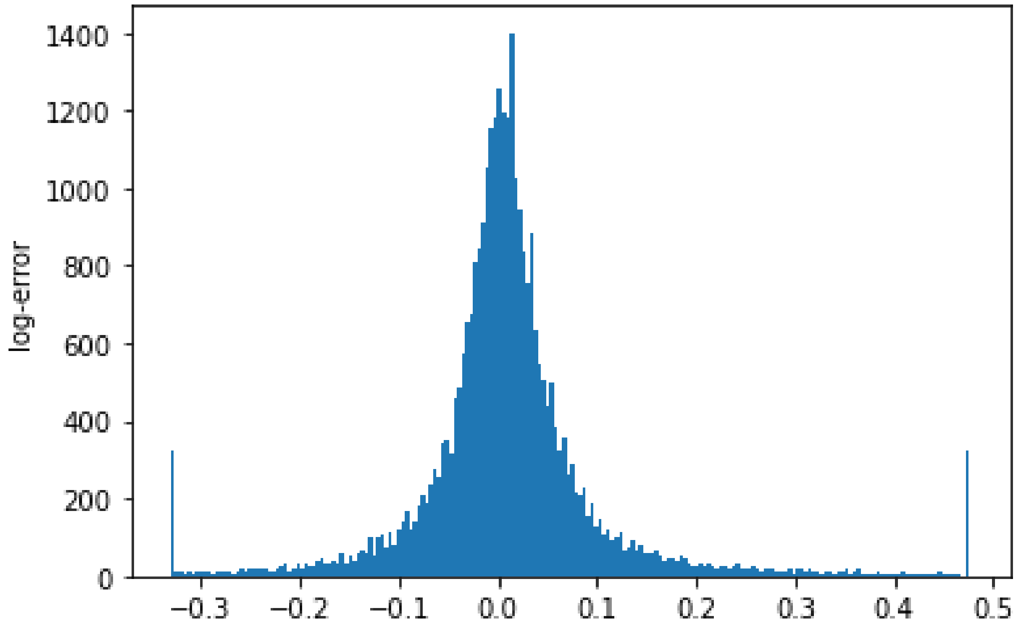
In [4]:
```
# scatter of logerr
data_no = np.arange(1, data.shape[0]+1)
fig1, ax1 = plt.subplots(1)
ax1.scatter(data_no, data[:, 1])
ax1.set_xlabel("Index of the log-error")
ax1.set_ylabel("Log-error")
```

Out[4]:
```
Text(0, 0.5, 'Log-error')
```



In [5]:
```python
# histogram of logerr
fig2, ax2 = plt.subplots(1)
ax2.hist(data[:,1], bins="auto")
ax2.set_ylabel("log-error")
```

Out[5]: Text(0, 0.5, 'log-error')



b) data cleaning

In [6]:
```python
#assisgning the updated column to the data_frame
data_pd.iloc[:,1] = data[:,1]
```

In [7]:
```python
# build new data frame
```

```
clean_df = pd.DataFrame(columns=['column_name', 'missing_count'])
clean_df['column_name'] = [*data_pd]
missing_count = []

for column in [*data_pd]:
    missing_count.append(data_pd[column].isnull().sum())

clean_df['missing_count'] = missing_count
clean_df.iloc[:5,:]

missing_ratio = []
total_no_of_data = data_pd.shape[0]

for column, miss_count in zip([*data_pd], missing_count):
    missing_ratio.append((miss_count / total_no_of_data))

clean_df['missing_ratio'] = missing_ratio
```

In [8]:

```
# fill missing data
i = 0
for column, miss_no in zip(data_pd, clean_df['missing_count']):
    i += 1
    if pd.api.types.is_numeric_dtype(data_pd[column]) and miss_no != 0:
        col_mean = data_pd[column].mean()
        idx = pd.isna(data_pd[column])
        data_pd[column].iloc[idx] = col_mean

#print(data_pd.iloc[:1][:])
```

```
/home/akshay/anaconda3/envs/cve/lib/python3.9/site-packages/pandas/core/indexing.py:1732: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

c) univariate analysis

In [65]:

```
# make bar chart

correlation = []
logerror_column = pd.DataFrame(data_pd['logerror'])
columns = []

# for each column finding the correlation
for column in data_pd:
    if pd.api.types.is_numeric_dtype(data_pd[column]) and column != 'logerror':
        temp_df = pd.DataFrame(data_pd[column])
```

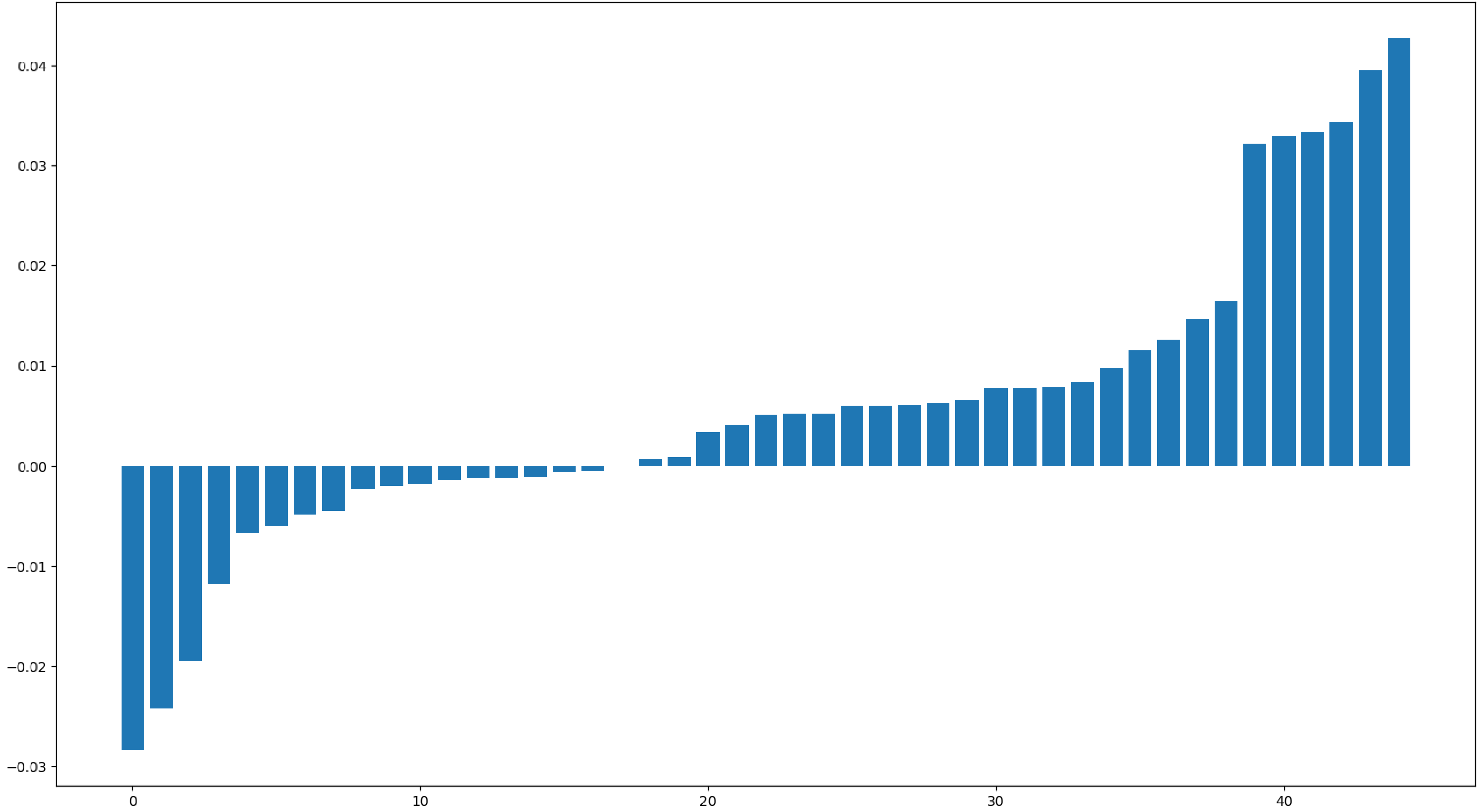
```
        corr_curr = (data_pd[column].astype('float64').corr(data_pd['logerror'].astype('float64')))
        correlation.append(corr_curr)
        columns.append(column)
        print(corr_curr, column)

correlation = np.asarray(correlation)
correlation = np.sort(correlation)

x_coords = np.arange(0, correlation.shape[0])

fig, ax = plt.subplots(1)
ax.bar(x_coords, correlation, align='center')
fig.set_size_inches(18.5, 10.5)

fig.set_dpi(100)
#plot below
correlation.shape, len(columns)
```

```
0.006561980129046172 id
0.00632751143687425 airconditioningtypeid
-0.001233864838286728 architecturalstyletypeid
0.005238522619389635 basementsqft
0.033445022352849414 bathroomcnt
0.03216818790391722 bedroomcnt
nan buildingclasstypeid
-0.001839668292540492 buildingqualitytypeid
0.03434458555574457 calculatedbathnbr
nan decktypeid
0.000806680347630661 finishedfloor1squarefeet
0.04284113799626631 calculatedfinishedsquarefeet
0.03950434052382624 finishedsquarefeet12
0.012608065999967201 finishedsquarefeet13
0.01468666231333262 finishedsquarefeet15
0.0006214820079056785 finishedsquarefeet50
-0.000655915058639708 finishedsquarefeet6
0.007862640449805601 fips
0.0050994131654547034 fireplacecnt
0.032985811417016356 fullbathcnt
-3.923165394283734e-05 garagecarcnt
0.005227481132968134 garagetotalsqft
-0.01951067953867454 heatingorsystemtypeid
0.003277310285116709 latitude
0.0077821661685649815 longitude
0.00609256083520044 lotsizesquarefeet
nan poolcnt
-0.0014421181560562095 poolsizesum
nan pooltypeid10
nan pooltypeid2
nan pooltypeid7
-0.028458981971990892 propertylandusetypeid
```

```
0.007821236068795489 rawcensustractandblock
-0.0005421521922660375 regionidcity
-0.004874229836429321 regionidcounty
-0.004454391213189863 regionidneighborhood
-0.0012533585054891423 regionidzip
0.00976205417010197 roomcnt
nan storytypeid
-0.0011334267143213678 threequarterbathnbr
-0.002361254031611235 typeconstructiontypeid
0.005964198993407032 unitcnt
-0.006088177733782849 yardbuildingsqft17
0.004130658446371897 yardbuildingsqft26
0.016441744841236786 yearbuilt
0.011565216615242782 numberofstories
0.00595004133672608 structuretaxvaluedollarcnt
-0.0020597874316461403 taxvaluedollarcnt
nan assessmentyear
-0.006758495880139079 landtaxvaluedollarcnt
-0.02428161798443687 taxamount
-0.011825579502366615 taxdelinquencyyear
0.008389054234153235 censustractandblock
```

Out[65]:

```
((53,), 53)
```



explain reason

Some features have correlation as 'nan' in the above calculation. There are namely, pooltypeid10,pooltypeid2,pooltypeid7 assessmentyear etc. This shows that there is no relation between log error and these variables. Precisely, the change to log-error cannot be related in anyway to these features. This is because these feature are like id's or assessmentyear or types, which cannot directly contribute to the regression problem. These are more of idendifier features. So there change is independent of the change of log-error

d) non-linear regression model

In [27]:

```
# drop categorical features
# ("hashottuborspa", "propertycountylandusecode", "propertyzoningdesc", "fireplaceflag", "taxdelinquencyflag")
# drop "id" and "transactiondate"
columns_to_drop = ["hashottuborspa", 'propertycountylandusecode', 'propertyzoningdesc', 'fireplaceflag',
                   'taxdelinquencyflag', 'id', 'transactiondate', 'logerror']

data_n_ = data_pd.drop(columns_to_drop, axis = 1)
data_n_.shape
```

Out[27]:

```
(31725, 52)
```

In [28]:

```
y = data_pd['logerror']
```

In [29]:

```
#converting to numpy
data_n = data_n_.to_numpy()
```

In [50]:

```
# split and train
X_train, X_test, y_train, y_test = train_test_split(data_n, y, test_size=0.30, random_state=2)
random_forest_clf = RandomForestRegressor(n_estimators=10)
random_forest_clf.fit(X_train, y_train)
print("Train Score:", random_forest_clf.score(X_train, y_train), "Test Score:", random_forest_clf.score(X_test, y_test))
```

```
Train Score: 0.8002237693795773 Test Score: -0.13100544955322402
```

In [51]:

```
def calculate_mse(X_test, y_test, random_forest_clf):
    y_pred = random_forest_clf.predict(X_test)
    mse_ = np.sum(np.power(y_test - y_pred, 2)) / y_test.shape[0]
    return mse_
```

report importances and msedata_pd

In [52]:

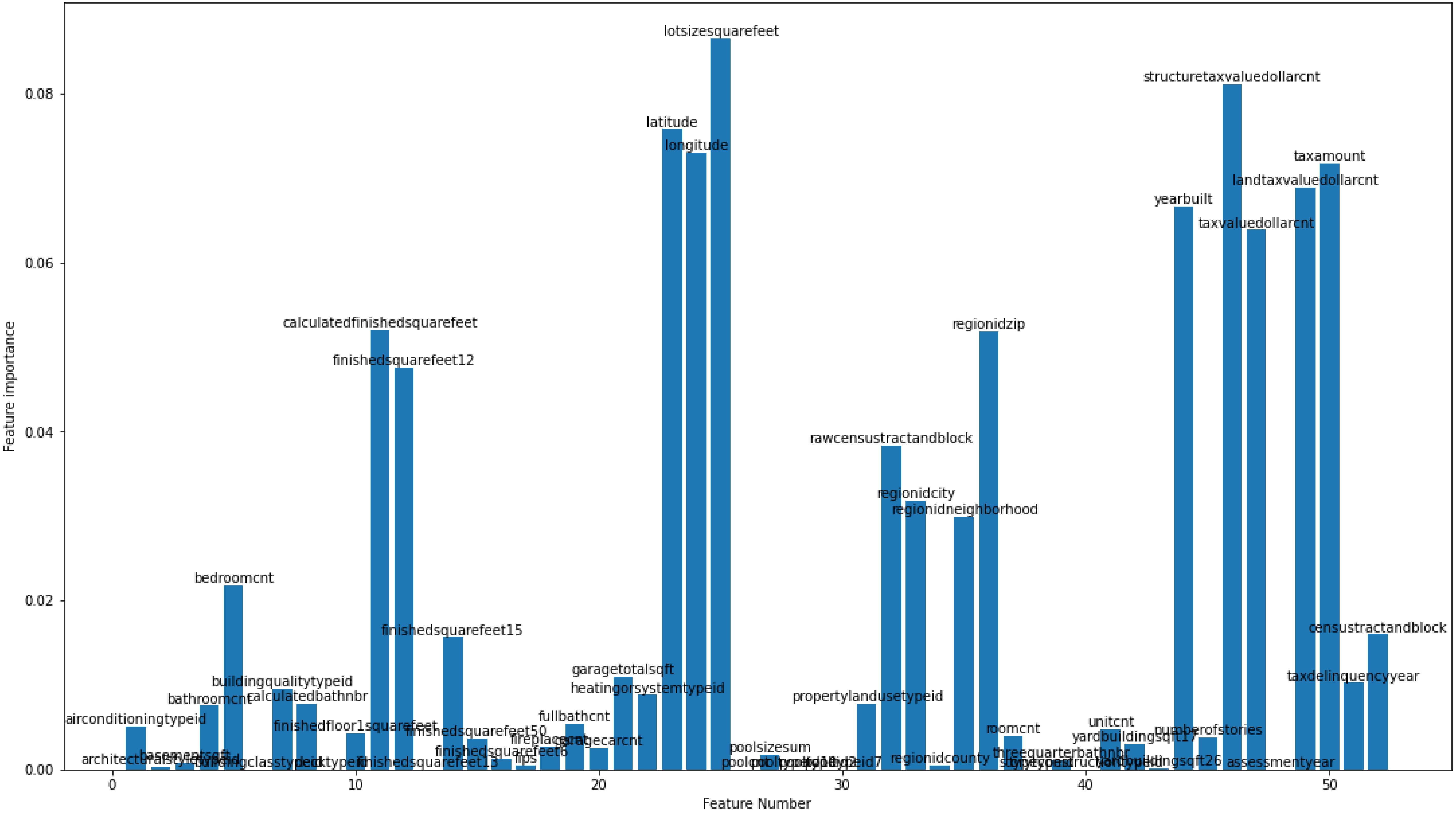
```
mse_loss_ = calculate_mse(X_test, y_test, random_forest_clf)
feature_importance = random_forest_clf.feature_importances_
# feature_importance_args = np.argsort(feature_importance)
print("MSE_loss:", mse_loss_)
```

```
MSE_loss: 0.011309687526971694
```

In [53]:

```
# plotting bar charts
x_coords = np.arange(1,53,1)
```

```
s = list(data_n_.columns)

fig, ax = plt.subplots(1)
line = ax.bar(x_coords, feature_importance)
ax.set_xlabel('Feature Number')
ax.set_ylabel("Feature importance")
fig.set_size_inches(18.5, 10.5)

for i in range(len(s)):
    ax.annotate(str(s[i]), xy=(x_coords[i],feature_importance[i]), ha='center', va='bottom')

plt.show()
```



e) KFold

In [66]:

```
# KFold, k = 5
# taking the first 500 samples
X_k_fold = data_n[:500, :]
y_k_fold = y[:500]
mse_loss_list = []
kf = KFold(5)
random_forest_clf_kf = RandomForestRegressor(n_estimators=10)
```

```
for train_index, test_index in kf.split(X_k_fold):
    X_train_kf, X_test_kf = X_k_fold[train_index], X_k_fold[test_index]
    y_train_kf, y_test_kf = y_k_fold[train_index], y_k_fold[test_index]
    random_forest_clf_kf.fit(X_train_kf, y_train_kf)
    mse_loss_list.append(calculate_mse(X_test_kf, y_test_kf, random_forest_clf_kf))

overall_mse_loss_kf = sum(mse_loss_list) / len(mse_loss_list)
print("overall Loss:", overall_mse_loss_kf)
```

```
overall Loss: 0.014702641136324312
```

In [56]:

```
def random_forest(data_n, y, r):
    X_train, X_test, y_train, y_test = train_test_split(data_n, y, test_size=0.30, random_state=r)
    random_forest_clf = RandomForestRegressor(n_estimators=10, random_state=r)
    random_forest_clf.fit(X_train, y_train)
    #calculating the mse of test
    return calculate_mse(X_test, y_test, random_forest_clf)
```

In [59]:

```
# Run d2 for 100 times
mse_loss_100 = []
for i in range(100):
    mse_loss_100.append(random_forest(data_n, y, i))
    print("Iteration/Random State:", i, ", MSE_LOSS:", mse_loss_100[i])
```

```
Iteration/Random State: 0 , MSE_LOSS: 0.011352893200932699
Iteration/Random State: 1 , MSE_LOSS: 0.011245934026848845
Iteration/Random State: 2 , MSE_LOSS: 0.011260543458676628
Iteration/Random State: 3 , MSE_LOSS: 0.011231872059675673
Iteration/Random State: 4 , MSE_LOSS: 0.010864840322324374
Iteration/Random State: 5 , MSE_LOSS: 0.0113770708898309
Iteration/Random State: 6 , MSE_LOSS: 0.011463349780889379
Iteration/Random State: 7 , MSE_LOSS: 0.011251050441579293
Iteration/Random State: 8 , MSE_LOSS: 0.011262338272529409
Iteration/Random State: 9 , MSE_LOSS: 0.011066511139263038
Iteration/Random State: 10 , MSE_LOSS: 0.011045854514874498
Iteration/Random State: 11 , MSE_LOSS: 0.011165183919149226
Iteration/Random State: 12 , MSE_LOSS: 0.010625989603921212
Iteration/Random State: 13 , MSE_LOSS: 0.011404065131116437
Iteration/Random State: 14 , MSE_LOSS: 0.011241071940108642
Iteration/Random State: 15 , MSE_LOSS: 0.011140717880262295
Iteration/Random State: 16 , MSE_LOSS: 0.01098434546429527
Iteration/Random State: 17 , MSE_LOSS: 0.01079368200437472
Iteration/Random State: 18 , MSE_LOSS: 0.011384277152876633
Iteration/Random State: 19 , MSE_LOSS: 0.011199133027143144
Iteration/Random State: 20 , MSE_LOSS: 0.011104603979645522
Iteration/Random State: 21 , MSE_LOSS: 0.011779303738798848
Iteration/Random State: 22 , MSE_LOSS: 0.01077546237112527
Iteration/Random State: 23 , MSE_LOSS: 0.011297342471516812
```

```
Iteration/Random State: 24 , MSE_LOSS: 0.011300483157457748
Iteration/Random State: 25 , MSE_LOSS: 0.011184128362371823
Iteration/Random State: 26 , MSE_LOSS: 0.011416091173829074
Iteration/Random State: 27 , MSE_LOSS: 0.011370520843415305
Iteration/Random State: 28 , MSE_LOSS: 0.010755854150378764
Iteration/Random State: 29 , MSE_LOSS: 0.01108946181527962
Iteration/Random State: 30 , MSE_LOSS: 0.011338032559223951
Iteration/Random State: 31 , MSE_LOSS: 0.011607797866342818
Iteration/Random State: 32 , MSE_LOSS: 0.01124614367509932
Iteration/Random State: 33 , MSE_LOSS: 0.011274479535306416
Iteration/Random State: 34 , MSE_LOSS: 0.011825508785561876
Iteration/Random State: 35 , MSE_LOSS: 0.011389227176429378
Iteration/Random State: 36 , MSE_LOSS: 0.011625807596295898
Iteration/Random State: 37 , MSE_LOSS: 0.010915476603866917
Iteration/Random State: 38 , MSE_LOSS: 0.011102473283645239
Iteration/Random State: 39 , MSE_LOSS: 0.011296636206547068
Iteration/Random State: 40 , MSE_LOSS: 0.01129334423302723
Iteration/Random State: 41 , MSE_LOSS: 0.011401956171570252
Iteration/Random State: 42 , MSE_LOSS: 0.011581765423706057
Iteration/Random State: 43 , MSE_LOSS: 0.011135594135117124
Iteration/Random State: 44 , MSE_LOSS: 0.011501300313094641
Iteration/Random State: 45 , MSE_LOSS: 0.010847294437157737
Iteration/Random State: 46 , MSE_LOSS: 0.011038690166691323
Iteration/Random State: 47 , MSE_LOSS: 0.011193645546564794
Iteration/Random State: 48 , MSE_LOSS: 0.01129612823613629
Iteration/Random State: 49 , MSE_LOSS: 0.010852247173643552
Iteration/Random State: 50 , MSE_LOSS: 0.011379366728367233
Iteration/Random State: 51 , MSE_LOSS: 0.011247023861534842
Iteration/Random State: 52 , MSE_LOSS: 0.011644484877822362
Iteration/Random State: 53 , MSE_LOSS: 0.010895535829210953
Iteration/Random State: 54 , MSE_LOSS: 0.011122728872024444
Iteration/Random State: 55 , MSE_LOSS: 0.011226218040208373
Iteration/Random State: 56 , MSE_LOSS: 0.011454442231688271
Iteration/Random State: 57 , MSE_LOSS: 0.011187873473565469
Iteration/Random State: 58 , MSE_LOSS: 0.010826824782040615
Iteration/Random State: 59 , MSE_LOSS: 0.011476930865088225
Iteration/Random State: 60 , MSE_LOSS: 0.011128805749988861
Iteration/Random State: 61 , MSE_LOSS: 0.011177640776488468
Iteration/Random State: 62 , MSE_LOSS: 0.010598804712862038
Iteration/Random State: 63 , MSE_LOSS: 0.011034482959083795
Iteration/Random State: 64 , MSE_LOSS: 0.011138451650878048
Iteration/Random State: 65 , MSE_LOSS: 0.011331901676034078
Iteration/Random State: 66 , MSE_LOSS: 0.011161679604631046
Iteration/Random State: 67 , MSE_LOSS: 0.011256830948433874
Iteration/Random State: 68 , MSE_LOSS: 0.011221844036665495
Iteration/Random State: 69 , MSE_LOSS: 0.01124322538317819
Iteration/Random State: 70 , MSE_LOSS: 0.01117977965421899
Iteration/Random State: 71 , MSE_LOSS: 0.011454783014887813
Iteration/Random State: 72 , MSE_LOSS: 0.010937316812612171
Iteration/Random State: 73 , MSE_LOSS: 0.010805420903786616
```

```
Iteration/Random State: 74 , MSE_LOSS: 0.010932635922904813
Iteration/Random State: 75 , MSE_LOSS: 0.010485640570874826
Iteration/Random State: 76 , MSE_LOSS: 0.01102734884161853
Iteration/Random State: 77 , MSE_LOSS: 0.011259543056220515
Iteration/Random State: 78 , MSE_LOSS: 0.011563775569705559
Iteration/Random State: 79 , MSE_LOSS: 0.011079065541065965
Iteration/Random State: 80 , MSE_LOSS: 0.011277360794544137
Iteration/Random State: 81 , MSE_LOSS: 0.011226835110114616
Iteration/Random State: 82 , MSE_LOSS: 0.01125121700765508
Iteration/Random State: 83 , MSE_LOSS: 0.011293252085354758
Iteration/Random State: 84 , MSE_LOSS: 0.01082440515647261
Iteration/Random State: 85 , MSE_LOSS: 0.011153547602890107
Iteration/Random State: 86 , MSE_LOSS: 0.011744304471865101
Iteration/Random State: 87 , MSE_LOSS: 0.011126812151611676
Iteration/Random State: 88 , MSE_LOSS: 0.011246732196672285
Iteration/Random State: 89 , MSE_LOSS: 0.01129282657273669
Iteration/Random State: 90 , MSE_LOSS: 0.010933238676596268
Iteration/Random State: 91 , MSE_LOSS: 0.011286654894041074
Iteration/Random State: 92 , MSE_LOSS: 0.011243070433628039
Iteration/Random State: 93 , MSE_LOSS: 0.011419530565924173
Iteration/Random State: 94 , MSE_LOSS: 0.010892649739136794
Iteration/Random State: 95 , MSE_LOSS: 0.011530914519973194
Iteration/Random State: 96 , MSE_LOSS: 0.011559041651040741
Iteration/Random State: 97 , MSE_LOSS: 0.011347434957696679
Iteration/Random State: 98 , MSE_LOSS: 0.011204605257822055
Iteration/Random State: 99 , MSE_LOSS: 0.011495684899137203
```

Advantage of Cross Validation

As the cross validation divides data into k-folds, all the data will come under training and validation, so the model learns more from the data and avoids over fitting to train set. It will give a more accurate measure of the loss, and both train and test data are considered, while without kfold, only the train data is considred by the model while training.

In [ ]:

```
```

In [ ]:

```
```

Question 3 Flower Classification using CNN

- Please **do not** change the default variable names in this problem, as we will use them in different parts.
- The default variables are initially set to "None".
- You only need to modify code in the "TODO" part. We added some "assertions" to check your code. **Do not** modify them.

```
import numpy as np # linear algebra
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torchvision
from torchvision import datasets, transforms, models
from torch.utils.data import *
import random
from tqdm import tqdm
import warnings
```

You can upload your image folder on Google drive and access image folder from it. **Skip it if you run on local machine.** To mount google drive to your current colab page, use the following command

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# check pytorch cuda and use cuda if possible
device = torch.cuda.is_available()
print('*' * 50)
if torch.cuda.is_available():
  print('CUDA is found! Tranining on %s.......'%torch.cuda.get_device_name(0))
else:
  warnings.warn('CUDA not found! Training may be slow......')
```

```
**************************************************
CUDA is found! Tranining on Tesla K80.......
```

P1. Data augmentation and plotting

TODO

- Design your image augmentation method for transform_image
- Load train and test data, and split them into train_loader and test_loader
- Visualize your augmented image

```
# TODO: define your image augmentation method
# Make sure to crop the image in (3,224,224) using transforms.RandomResizedCrop(224)
transform_image = transforms.Compose([transforms.RandomVerticalFlip(p=0.2),
```

```
                                      transforms.RandomHorizontalFlip(p=0.2),
                                      transforms.RandomResizedCrop((224)),
                                      transforms.ToTensor()])


# TODO: Load data using ImageFolder. Specify your image folder path
path = "/content/drive/MyDrive/flowers/flowers/"


dataset = datasets.ImageFolder(path,transform=transform_image)

n = len(dataset)
n_test = int(0.1 * n)

# Split data into features(pixels) and labels(numbers from 0 to 4)
train_dataset, test_dataset = random_split(dataset, (n-n_test,n_test))
train_loader, test_loader = DataLoader(train_dataset, batch_size=16, shuffle=True), DataLoader(test_dataset, batch_size=16, shuffle=True)
```

```
# Sample output
label_map = [['daisy'],['dandelion'],['rose'],['sunflower'],['tulip']]
random_image = random.randint(0,len(train_dataset))
image = train_dataset.__getitem__(random_image)

assert np.array_equal(image[0].detach().numpy().shape, [3,224,224])
plt.imshow(image[0].permute(1,2,0))
plt.title(f"Training example {label_map[image[1]]}")
plt.axis('off')
```

```
(-0.5, 223.5, 223.5, -0.5)
```

Training example ['daisy']

P2. Build you own CNN model

TODO

- Design your own model class in **CNNModel(nn.Module)** and write forward pass in **forward(self, x)**
- Create loss function in **error**, optimizer in **optimizer**
- Define hyparparameters: **learning_rate**, **num_epochs**
- Plot your **loss vs num_epochs** and **accuracy vs num_epochs**
- Plot your first convolution layer kernels using **plot_filters_multi_channel()**

Hints

- Start with low number of epochs for debugging. (eg. num_epochs=1)
- You may want to use small learning rate for training. (eg. 1e-5)
- Be careful with the input dimension of fully connected layer.
- The dimension calculation of the output tensor from the input tensor is
  $D_{out} = \frac{D_{in} - K + 2P}{S} + 1$
  D_{out} : Dimension of output tensor
  D_{in} : Dimension of input tensor
  K : width/height of the kernel
  S : stride
  P : padding

Convolutional and Pooling Layers

A convolutional layer using pyTorch:

```
torch.nn.Conv2d(num_in_channels, num_out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='z
```

For example:

```
torch.nn.Conv2d(3, 32, 3)
```

It applies a 2D convolution over an input signal composed of several input planes. If we have input size with (N, C_{in}, H, W) and output size with $(N, C_{out}, H_{out}, W_{out})$, the 2D convolution can described as

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

num_in_channels: is the number of channels of the input tensor. If the previous layer is the input layer, num_in_channels is the number of channels of the image (3 channels for RGB images), otherwise num_in_channels is equal to the number of feature maps of the previous layer.

num_out_channels: is the number of filters (feature extractor) that this layer will apply over the image or feature maps generated by the previous layer.

kernel_size: is the size of the convolving kernel So for instance, if we have an RGB image and we are going to apply 32 filters of 3x3:

stide: is the stride of the convolution. Default: 1

padding: is the padding added to all four sides of the input. Default: 0

dilation: is the spacing between kernel elements. Default: 1

group: is the number of blocked connections from input channels to output channels. Default: 1

bias: If True, adds a learnable bias to the output. Default: True

A Simple Convolutional Neural Network

In our convnet we'll use the next structure shown in the comment:

input -> convolution -> pooling -> fully connected -> output

Convolution #1

16 kernels of 5x5; *Width/Height:* (224 - 5 + 2x0) / 1 + 1 = 220; *Output dimensions:* (16, 220, 220)

Max Pooling #1

filter size = 2, stride = 2; *Width/Height:* (220 - 2) / 2 + 1 = 110; *Output dimensions:* (16, 110, 110)

So at the end of the last convolutional layer we get a tensor of dimension (16, 110, 110). And since now we are going to feed it to fully connected classifier, we need to convert it into a 1-D vector, and for that we use the reshape method:

```
x = x.view(x.size(0), -1)
```

The way of calculating size of the output size from previous convolution layer can be formulized as below:

$$H_{output} = \frac{H_{in} + 2 \times padding - kernel\_Size}{stride} + 1$$

For more details, you can refer to this link:
https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

```
class CNNModel(nn.Module):
  def __init__(self):
    super(CNNModel, self).__init__()
    # TODO: Create CNNModel using 2D convolution. You should vary the number of convolution layers and fully connected layers
    # Example:
    # self.cnn1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=0)
    # self.relu1 = nn.ReLU()
    # self.maxpool1 = nn.MaxPool2d(kernel_size=2)
    self.cnn1 = nn.Conv2d(3, 32, kernel_size=(3,3), padding=(1,1))
    self.relu1 = nn.ReLU()
    self.maxpool1 = nn.MaxPool2d((2,2), stride=2, padding=(0,0))

    self.cnn2 = nn.Conv2d(32, 64, (3,3), padding=(1,1))
    self.relu2 = nn.ReLU()
    self.maxpool2 = nn.MaxPool2d((2,2), stride=2, padding=(0,0))

    self.cnn3 = nn.Conv2d(64, 128, (3,3), padding=(1,1))
    self.relu3 = nn.ReLU()
    self.maxpool3 = nn.MaxPool2d((2,2), stride=2, padding=(0,0))

    self.cnn4 = nn.Conv2d(128, 256, (3,3), padding=(1,1))
    self.relu4 = nn.ReLU()
    self.maxpool4 = nn.MaxPool2d((2,2), stride=2, padding=(0,0))

    self.cnn5 = nn.Conv2d(256, 512, (3,3), padding=(1,1))
    self.relu5 = nn.ReLU()
    self.maxpool5 = nn.MaxPool2d((2,2), stride=2, padding=(0,0))
    self.dropout1 = nn.Dropout(0.2)

    # TODO: Create Fully connected layers. You should calculate the dimension of the input tensor from the previous layer
    # Example:
    # self.fc1 = nn.Linear(16 *110 * 110, 5)
    # Fully connected 1
```

```
    self.fc1 = nn.Linear(7*7*512, 64)
    self.relu6 = nn.ReLU()
    self.fc2 = nn.Linear(64, 5)

  def forward(self,x):
    # TODO: Perform forward pass in blow section
    # Example:
    # out = self.cnn1(x)
    # out = self.relu1(out)
    # out = self.maxpool1(out)
    # out = out.view(out.size(0), -1)
    # out = self.fc1(out)

    out = self.maxpool1(self.relu1(self.cnn1(x)))
    out = self.maxpool2(self.relu2(self.cnn2(out)))
    out = self.maxpool3(self.relu3(self.cnn3(out)))
    out = self.maxpool4(self.relu4(self.cnn4(out)))
    out = self.maxpool5(self.relu5(self.cnn5(out)))
    out = torch.flatten(out, 1)
    out = self.dropout1(out)
    out = self.fc2(self.relu6(self.fc1(out)))
    return out
```

Starting Up Our Model

We'll send the model to our GPU if you have one so we need to create a CUDA device and instantiate our model. Then we will define the loss function and hyperparameters that we need to train the model:

TODO

- Define Cross Entropy Loss
- Create Adam Optimizer
- Define hyperparameters

```
# Create CNN
device = "cuda" if torch.cuda.is_available() else "cpu"
model = CNNModel()
model.to(device)

# TODO: define Cross Entropy Loss
error = nn.CrossEntropyLoss()

# TODO: create Adam Optimizer and define your hyperparameters
learning_rate = 1e-5
optimizer = torch.optim.Adam(model.parameters(), learning_rate)
num_epochs = 100
```

Training the Model

TODO

- Make predictions from your model

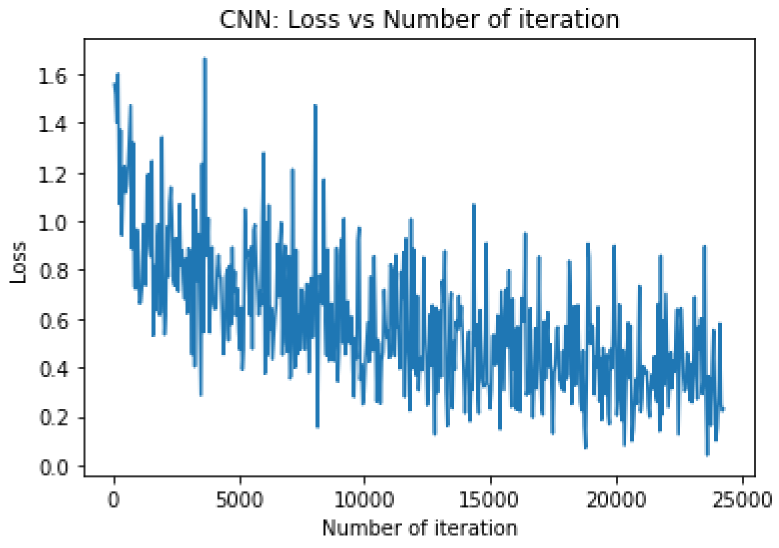
- Calculate Cross Entropy Loss from predictions and labels

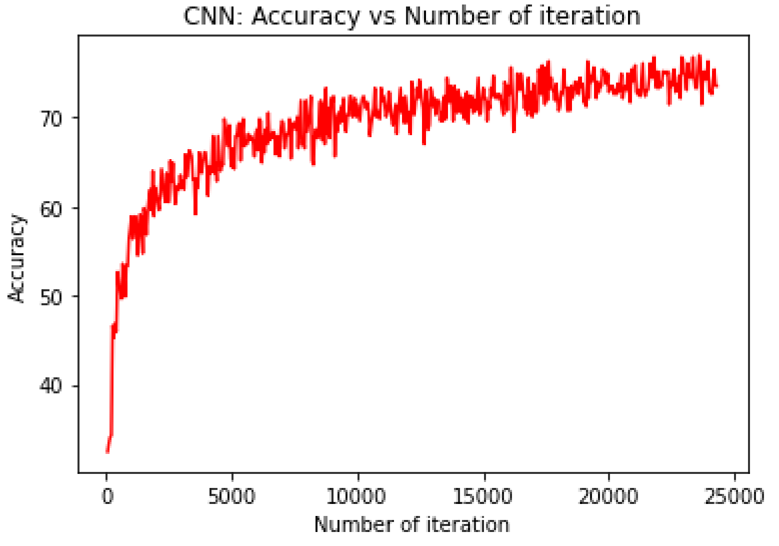
```
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in tqdm(range(num_epochs)):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Clear gradients
        optimizer.zero_grad()
        # TODO: Forward propagation
        outputs = model(images)

        # TODO: Calculate softmax and cross entropy loss
        loss = error(outputs, labels)

        # Backprop agate your Loss
        loss.backward()

        # Update CNN model
        optimizer.step()

        count += 1

        if count % 50 == 0:
            model.eval()
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)

                # Forward propagation
                outputs = model(images)

                # Get predictions from the maximum value
                predicted = torch.argmax(outputs,1)

                # Total number of labels
                total += len(labels)

                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

```
  2%|          | 2/100 [12:12<8:24:03, 308.61s/it] Iteration: 500  Loss: 1.117492437362671  Accuracy: 51.04408264160156 %
  4%|          | 4/100 [13:30<3:24:23, 127.75s/it]Iteration: 1000  Loss: 0.8753722906112671  Accuracy: 58.93271255493164 %
  6%|          | 6/100 [14:49<2:00:00, 76.60s/it]Iteration: 1500  Loss: 0.8561764359474182  Accuracy: 59.86078643798828 %
  8%|          | 8/100 [16:05<1:25:26, 55.72s/it]Iteration: 2000  Loss: 0.7537195086479187  Accuracy: 61.252899169921875 %
 10%|          | 10/100 [17:27<1:11:46, 47.85s/it]Iteration: 2500  Loss: 0.9296541810035706  Accuracy: 60.55684280395508 %
 12%|          | 12/100 [18:47<1:04:14, 43.80s/it]Iteration: 3000  Loss: 0.8885321021080017  Accuracy: 62.180973052978516 %
 14%|          | 14/100 [20:08<1:00:18, 42.07s/it]Iteration: 3500  Loss: 0.28579601645469666  Accuracy: 63.109046936035156 %
 16%|          | 16/100 [21:24<56:22, 40.27s/it]Iteration: 4000  Loss: 0.6603484749794006  Accuracy: 64.7331771850586 %
 18%|          | 18/100 [22:44<54:43, 40.05s/it]Iteration: 4500  Loss: 0.7712033987045288  Accuracy: 65.89327239990234 %
 20%|          | 20/100 [24:03<52:55, 39.70s/it]Iteration: 5000  Loss: 0.5460932850837708  Accuracy: 64.50115966796875 %
 22%|          | 22/100 [25:19<50:15, 38.66s/it]Iteration: 5500  Loss: 0.8951508402824402  Accuracy: 66.82134246826172 %
 24%|          | 24/100 [26:38<49:32, 39.11s/it]Iteration: 6000  Loss: 1.2795366048812866  Accuracy: 68.677490234375 %
 26%|          | 26/100 [27:59<49:03, 39.78s/it]Iteration: 6500  Loss: 0.6571154594421387  Accuracy: 67.51740264892578 %
 28%|          | 28/100 [29:17<47:31, 39.60s/it]Iteration: 7000  Loss: 0.8564727902412415  Accuracy: 69.6055679321289 %
 30%|          | 30/100 [30:34<45:38, 39.12s/it]Iteration: 7500  Loss: 0.7212187051773071  Accuracy: 69.37355041503906 %
 32%|          | 32/100 [31:54<44:51, 39.57s/it]Iteration: 8000  Loss: 0.6982813477516174  Accuracy: 68.44547271728516 %
 34%|          | 34/100 [33:15<43:50, 39.86s/it]Iteration: 8500  Loss: 0.44950783252716064  Accuracy: 71.69373321533203 %
 37%|          | 37/100 [35:11<41:11, 39.23s/it]Iteration: 9000  Loss: 0.46283119916915894  Accuracy: 71.69373321533203 %
 39%|          | 39/100 [36:31<40:09, 39.50s/it]Iteration: 9500  Loss: 0.6084745526313782  Accuracy: 68.90950775146484 %
 41%|          | 41/100 [37:50<38:48, 39.46s/it]Iteration: 10000  Loss: 0.2958071529865265  Accuracy: 71.46171569824219 %
 43%|          | 43/100 [39:05<36:33, 38.49s/it]Iteration: 10500  Loss: 0.259946346282959  Accuracy: 67.98143768310547 %
 45%|          | 45/100 [40:24<35:36, 38.84s/it]Iteration: 11000  Loss: 0.4397362768650055  Accuracy: 71.92575073242188 %
 47%|          | 47/100 [41:42<34:25, 38.97s/it]Iteration: 11500  Loss: 0.6854530572891235  Accuracy: 71.46171569824219 %
 49%|          | 49/100 [43:00<33:06, 38.96s/it]Iteration: 12000  Loss: 0.4584358334541321  Accuracy: 69.37355041503906 %
 51%|          | 51/100 [44:15<31:16, 38.30s/it]Iteration: 12500  Loss: 0.24524320662021637  Accuracy: 73.31786346435547 %
 53%|          | 53/100 [45:34<30:28, 38.90s/it]Iteration: 13000  Loss: 0.6326792240142822  Accuracy: 71.22969818115234 %
 55%|          | 55/100 [46:53<29:13, 38.96s/it]Iteration: 13500  Loss: 0.232758566737175  Accuracy: 69.83758544921875 %
 57%|          | 57/100 [48:11<27:56, 38.99s/it]Iteration: 14000  Loss: 0.21395474672317505  Accuracy: 73.08584594726562 %
 59%|          | 59/100 [49:26<26:12, 38.35s/it]Iteration: 14500  Loss: 0.30431875586509705  Accuracy: 71.22969818115234 %
 61%|          | 61/100 [50:46<25:31, 39.26s/it]Iteration: 15000  Loss: 0.2308058887720108  Accuracy: 73.31786346435547 %
 63%|          | 63/100 [52:06<24:28, 39.69s/it]Iteration: 15500  Loss: 0.4942813813686371  Accuracy: 72.15776824951172 %
 65%|          | 65/100 [53:22<22:37, 38.79s/it]Iteration: 16000  Loss: 0.556371808052063  Accuracy: 73.54988098144531 %
 67%|          | 67/100 [54:41<21:27, 39.03s/it]Iteration: 16500  Loss: 0.3122072219848633  Accuracy: 74.9419937133789 %
 69%|          | 69/100 [56:00<20:16, 39.25s/it]Iteration: 17000  Loss: 0.3951760232448578  Accuracy: 73.78189849853516 %
 72%|          | 72/100 [57:55<17:55, 38.43s/it]Iteration: 17500  Loss: 0.13021281361579895  Accuracy: 75.63804626464844 %
 74%|          | 74/100 [59:13<16:50, 38.86s/it]Iteration: 18000  Loss: 0.5700092315673828  Accuracy: 70.76565551757812 %
 76%|          | 76/100 [1:00:32<15:38, 39.12s/it]Iteration: 18500  Loss: 0.6565026640892029  Accuracy: 73.54988098144531 %
 78%|          | 78/100 [1:01:51<14:25, 39.34s/it]Iteration: 19000  Loss: 0.4967154860496521  Accuracy: 71.69373321533203 %
 80%|          | 80/100 [1:03:08<12:58, 38.91s/it]Iteration: 19500  Loss: 0.48062264919281006  Accuracy: 72.15776824951172 %
 82%|          | 82/100 [1:04:28<11:51, 39.55s/it]Iteration: 20000  Loss: 0.5037093758583069  Accuracy: 72.85382843017578 %
 84%|          | 84/100 [1:05:48<10:36, 39.80s/it]Iteration: 20500  Loss: 0.58393794298172  Accuracy: 74.9419937133789 %
 86%|          | 86/100 [1:07:06<09:07, 39.11s/it]Iteration: 21000  Loss: 0.21914945542812347  Accuracy: 75.4060287475586 %
 88%|          | 88/100 [1:08:26<07:54, 39.57s/it]Iteration: 21500  Loss: 0.31155067682266235  Accuracy: 73.08584594726562 %
 90%|          | 90/100 [1:09:47<06:40, 40.06s/it]Iteration: 22000  Loss: 0.7024110555648804  Accuracy: 73.31786346435547 %
 92%|          | 92/100 [1:11:08<05:23, 40.47s/it]Iteration: 22500  Loss: 0.12648507952690125  Accuracy: 73.54988098144531 %
 94%|          | 94/100 [1:12:27<04:00, 40.01s/it]Iteration: 23000  Loss: 0.4158187806606293  Accuracy: 74.9419937133789 %
 96%|          | 96/100 [1:13:49<02:41, 40.32s/it]Iteration: 23500  Loss: 0.3429071307182312  Accuracy: 73.31786346435547 %
 98%|          | 98/100 [1:15:08<01:19, 39.99s/it]Iteration: 24000  Loss: 0.10150668770074844  Accuracy: 72.85382843017578 %
100%|          | 100/100 [1:16:24<00:00, 45.85s/it]
```

```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red")
```

```
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.show()
```

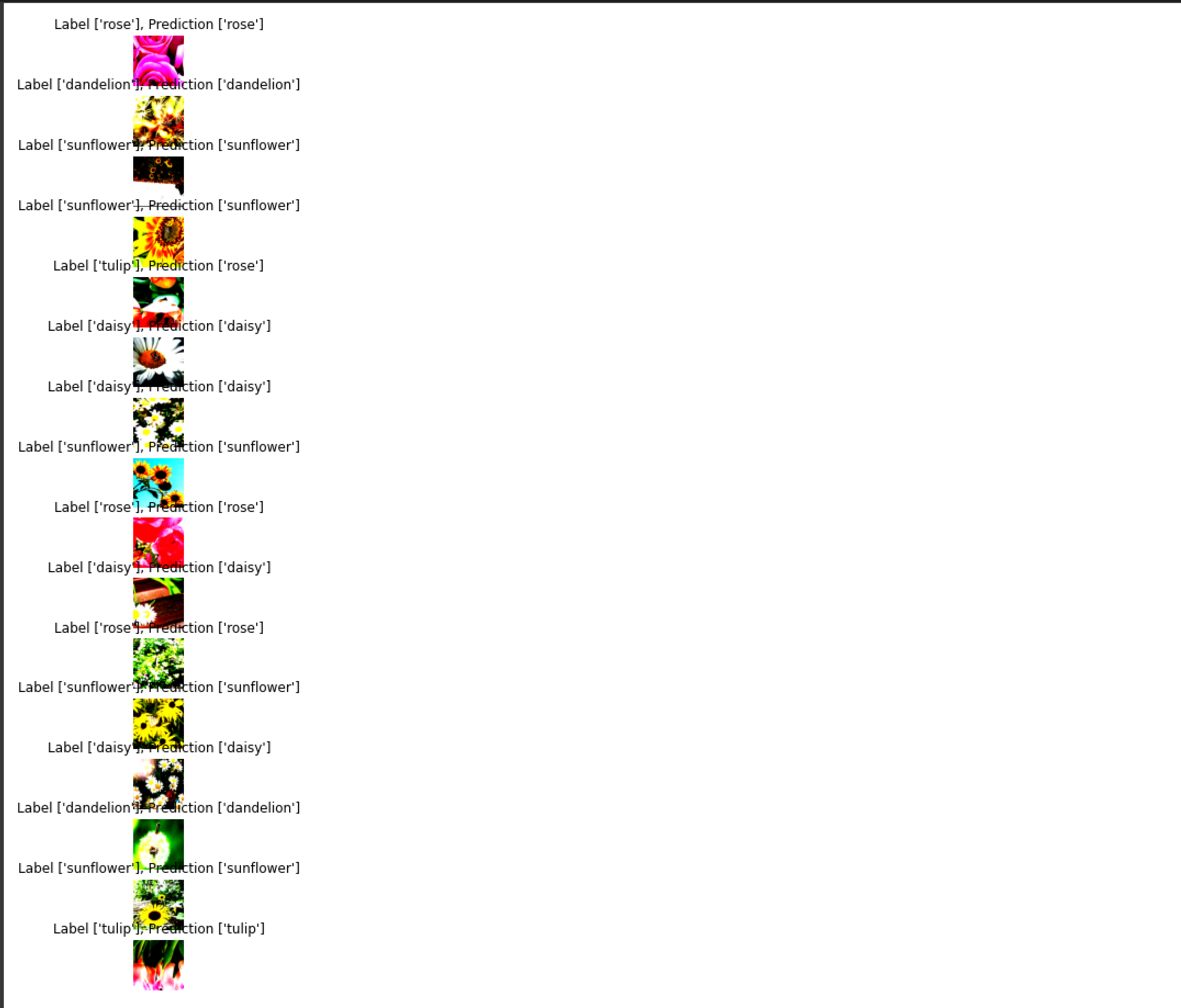




Evaluating the Model

```
# Evaluate your model
random_image = random.randint(0,len(train_dataset))
image = train_dataset.__getitem__(random_image)
model.eval()
images, labels = next(iter(train_loader))
images, labels = images.to(device), labels.to(device)
predictions = torch.argmax(model(images),1)
num_cols=1
num_rows = len(labels)
fig = plt.figure(figsize=(num_cols,num_rows))
for idx in range(num_rows):
  ax1 = fig.add_subplot(num_rows,num_cols,idx+1)
  img = images.cpu().detach()[idx].numpy()
  img = (img - np.mean(img)) / np.std(img)
  img = np.minimum(1, np.maximum(0, (img + 0.5)))
  ax1.imshow(img.transpose((1,2,0)))
```

```
    ax1.set_title(f"Label {label_map[labels[idx]]}, Prediction {label_map[predictions[idx]]}")
    ax1.axis('off')
plt.savefig('Prediction.png', dpi=100)
plt.show()
```



Visualizing your first layer filter

```python
# plot your first layer kernels

def plot_filters_multi_channel(t):
    #make sure the input channel is 3
    assert(t.shape[1]==3)

    #get the number of kernals
    num_kernels = t.shape[0]

    #define number of columns for subplots
    num_cols = 12

    #rows = num of kernels
    num_rows = num_kernels

    #set the figure size
    fig = plt.figure(figsize=(num_cols,num_rows))

    #looping through all the kernels
    for i in range(t.shape[0]):
        ax1 = fig.add_subplot(num_rows,num_cols,i+1)

        #for each kernel, we convert the tensor to numpy
        npimg = np.array(t[i].cpu().detach().numpy(), np.float32)

        #standardize the numpy image
        npimg = (npimg - np.mean(npimg)) / np.std(npimg)
        npimg = np.minimum(1, np.maximum(0, (npimg + 0.5)))
        npimg = npimg.transpose((1, 2, 0))
        ax1.imshow(npimg)
        ax1.axis('off')
        ax1.set_title(str(i))
        ax1.set_xticklabels([])
        ax1.set_yticklabels([])

    plt.savefig('Filter.png', dpi=100)
    plt.tight_layout()
    plt.show()

plot_filters_multi_channel(list(model.parameters())[0])
```