[1]:
```python
import numpy as np
import copy
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
```

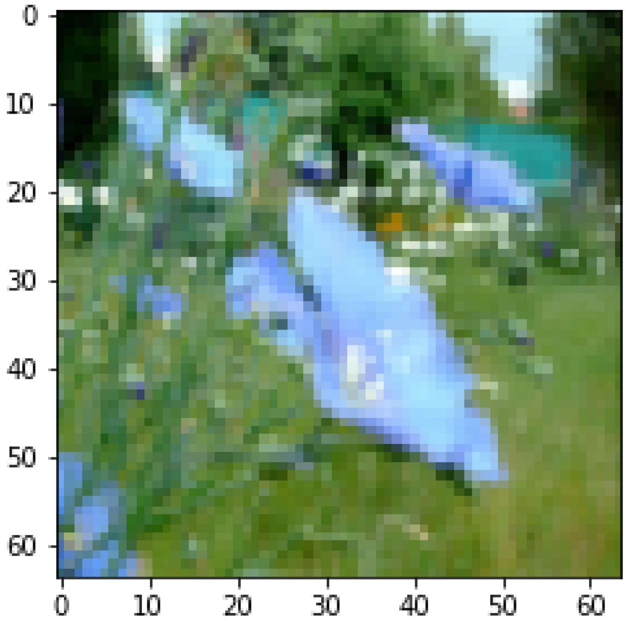
[2]:
```python
def load_dataset():
    train_ds = h5py.File('train_catvnoncat.h5', 'r')
    train_set_x = np.array(train_ds['train_set_x'][:])
    train_set_y = np.array(train_ds['train_set_y'][:])

    test_ds = h5py.File('test_catvnoncat.h5', 'r')
    test_set_x = np.array(test_ds['test_set_x'][:])
    test_set_y = np.array(test_ds['test_set_y'][:])

    classes = np.array(test_ds['list_classes'][:])

    train_set_y = train_set_y.reshape((1, train_set_y.shape[0]))
    test_set_y = test_set_y.reshape((1, test_set_y.shape[0]))

    return train_set_x, train_set_y, test_set_x, test_set_y, classes
```

[3]:
```python
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

[4]:
```python
index = 20
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.squeeze(train_set_y[:, index])].decode("utf-8") +  "' picture.")
""
```

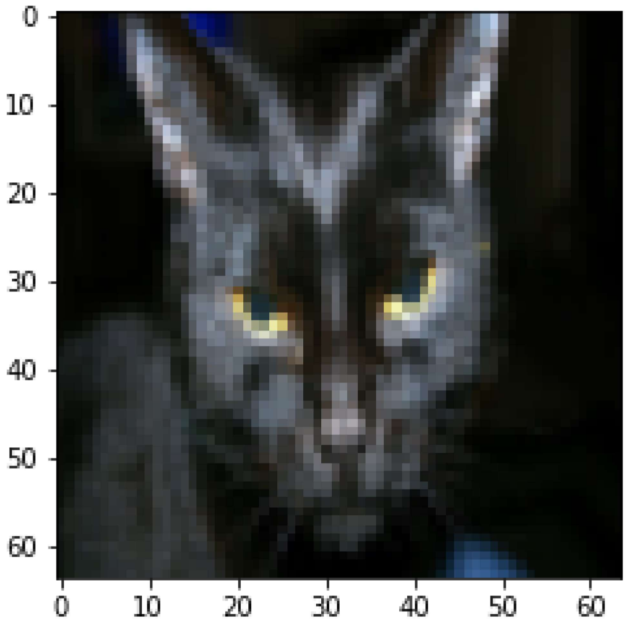
```
y = [0], it's a 'non-cat' picture.
```

[4]: ''



```
[5]: index = 25
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.
 ↪squeeze(train_set_y[:, index])].decode("utf-8") +  "' picture.")
```

```
y = [1], it's a 'cat' picture.
```



[6]:
```
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

[7]:
```
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
```

[8]:
```
# Let's standardize our dataset.

train_set_x = train_set_x_flatten / 255.
test_set_x = test_set_x_flatten / 255.
```

[9]:
```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.
```

```python
    Return:
    s -- sigmoid(z)
    """
    s = 1 / (1 + np.exp(-z))
    return s
```

[10]:
```python
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0, 2]) = [0.5        0.88079708]
```

[11]:
```python
x = np.array([0.5, 0, 2.0])
output = sigmoid(x)
print(output)
```

```
[0.62245933 0.5        0.88079708]
```

[12]:
```python
def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) for w and␣
 ↪initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias) of type float
    """
    w = np.zeros(shape=(dim, 1), dtype=np.float32)
    b = 0.0

    return w, b
```

[13]:
```python
dim = 2
w, b = initialize_with_zeros(dim)

assert type(b) == float
print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0.]
 [0.]]
b = 0.0
```

[14]:
```python
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained␣
 ↪above
```

```
    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1,␣
↪number of examples)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b

    Tips:
    - Write your code step by step for the propagation. np.log(), np.dot()
    """

    m = X.shape[1]

    # forward propagation (from x to cost)
    # compute activation
    A = sigmoid(w.T @ X + b)
    # compute cost by using np.dot to perform multiplication
    cost = np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A)) / -m

    # backward propagation (to find grad)
    dw = X @ (A - Y).T / m
    db = np.sum(A - Y) / m

    cost = np.squeeze(np.array(cost))

    grads = {'dw': dw, 'db': db}
    return grads, cost
```

[15]:
```
w =  np.array([[1.], [2]])
b = 1.5
X = np.array([[1., -2., -1.], [3., 0.5, -3.2]])
Y = np.array([[1, 1, 0]])
grads, cost = propagate(w, b, X, Y)

assert type(grads["dw"]) == np.ndarray
assert grads["dw"].shape == (2, 1)
assert type(grads["db"]) == np.float64


print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

```
print ("cost = " + str(cost))
```

```
dw = [[ 0.25071532]
 [-0.06604096]]
db = -0.12500404500439652
cost = 0.15900537707692405
```

[16]:
```
def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009,␣
↪print_cost=False):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1,␣
↪number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with␣
↪respect to the cost function
    costs -- list of all the costs computed during the optimization, this will␣
↪be used to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use␣
↪propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """

    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        # cost and gradient calculation
        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
```

```
        db = grads["db"]

        # update rule
        w -= learning_rate * dw
        b -= learning_rate * db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

            # Print the cost every 100 training iterations
            if print_cost:
                print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, cost
```

[17]:
```
params, grads, costs = optimize(w, b, X, Y, num_iterations=100, learning_rate=0.
↪009, print_cost=False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print("Costs = " + str(costs))
```

```
w = [[0.80956046]
 [2.0508202 ]]
b = 1.5948713189708588
dw = [[ 0.17860505]
 [-0.04840656]]
db = -0.08888460336847771
Costs = 0.10579008649578009
```

[18]:
```
def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression␣
↪parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
```

```
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for␣
↪the examples in X
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1, m))
    w = w.reshape(X.shape[0], 1)

    # compute vector 'A' predicting the probabilities of a cat being present in␣
↪the picture
    A = sigmoid(w.T @ X + b)

    for i in range(A.shape[1]):
        # convert probabilities A[0, i] to actual predictions p[0, i]
        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
            Y_prediction[0, i] = 0

    return Y_prediction
```

[19]:
```
w = np.array([[0.1124579], [0.23106775]])
b = -0.3
X = np.array([[1., -1.1, -3.2],[1.2, 2., 0.1]])
print ("predictions = " + str(predict(w, b, X)))
```

```
predictions = [[1. 1. 0.]]
```

[20]:
```
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000,␣
↪learning_rate=0.5, print_cost=False):
    """
    Builds the logistic regression model by calling the function you've␣
↪implemented previously

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px *␣
↪num_px * 3, m_train)
    Y_train -- training labels represented by a numpy array (vector) of shape␣
↪(1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px * num_px *␣
↪3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of shape (1,␣
↪m_test)
```

```
    num_iterations -- hyperparameter representing the number of iterations to␣
↪optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the␣
↪update rule of optimize()
    print_cost -- Set to True to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """
    w, b = initialize_with_zeros(dim=X_train.shape[0])

    # Gradient descent
    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,␣
↪learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "params"
    w = params['w']
    b = params['b']

    # Predict test/train set examples
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    # Print train/test Errors
    if print_cost:
        print("train accuracy: {} %".format(100 - np.mean(np.
↪abs(Y_prediction_train - Y_train)) * 100))
        print("test accuracy: {} %".format(100 - np.mean(np.
↪abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

[21]:
```
logistic_regression_model = model(train_set_x, train_set_y, test_set_x,␣
↪test_set_y, num_iterations=10000, learning_rate=0.002, print_cost=True)
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.555752
Cost after iteration 200: 0.506847
```

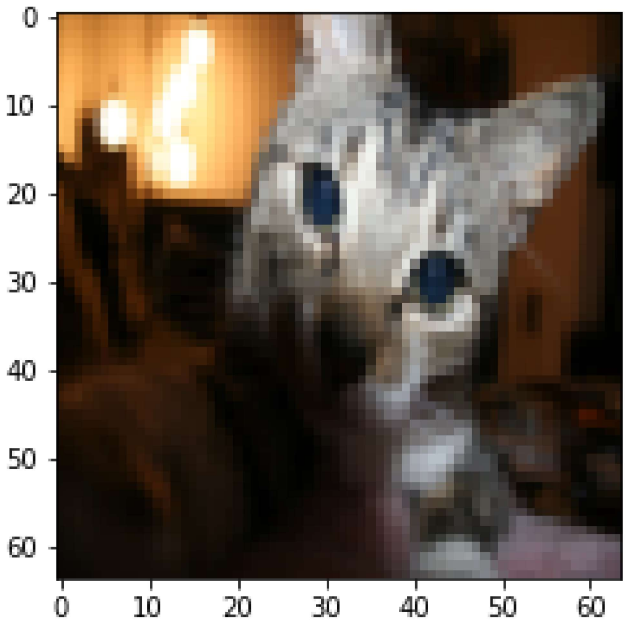
```
Cost after iteration 300: 0.471079
Cost after iteration 400: 0.442324
Cost after iteration 500: 0.418139
Cost after iteration 600: 0.397247
Cost after iteration 700: 0.378867
Cost after iteration 800: 0.362480
Cost after iteration 900: 0.347718
Cost after iteration 1000: 0.334308
Cost after iteration 1100: 0.322043
Cost after iteration 1200: 0.310759
Cost after iteration 1300: 0.300326
Cost after iteration 1400: 0.290639
Cost after iteration 1500: 0.281610
Cost after iteration 1600: 0.273167
Cost after iteration 1700: 0.265247
Cost after iteration 1800: 0.257798
Cost after iteration 1900: 0.250775
Cost after iteration 2000: 0.244139
Cost after iteration 2100: 0.237856
Cost after iteration 2200: 0.231897
Cost after iteration 2300: 0.226234
Cost after iteration 2400: 0.220846
Cost after iteration 2500: 0.215710
Cost after iteration 2600: 0.210808
Cost after iteration 2700: 0.206124
Cost after iteration 2800: 0.201643
Cost after iteration 2900: 0.197351
Cost after iteration 3000: 0.193236
Cost after iteration 3100: 0.189286
Cost after iteration 3200: 0.185492
Cost after iteration 3300: 0.181844
Cost after iteration 3400: 0.178334
Cost after iteration 3500: 0.174953
Cost after iteration 3600: 0.171695
Cost after iteration 3700: 0.168552
Cost after iteration 3800: 0.165519
Cost after iteration 3900: 0.162590
Cost after iteration 4000: 0.159760
Cost after iteration 4100: 0.157023
Cost after iteration 4200: 0.154375
Cost after iteration 4300: 0.151811
Cost after iteration 4400: 0.149329
Cost after iteration 4500: 0.146923
Cost after iteration 4600: 0.144590
Cost after iteration 4700: 0.142328
Cost after iteration 4800: 0.140133
Cost after iteration 4900: 0.138001
Cost after iteration 5000: 0.135931
```

```
Cost after iteration 5100: 0.133920
Cost after iteration 5200: 0.131965
Cost after iteration 5300: 0.130063
Cost after iteration 5400: 0.128214
Cost after iteration 5500: 0.126414
Cost after iteration 5600: 0.124662
Cost after iteration 5700: 0.122956
Cost after iteration 5800: 0.121294
Cost after iteration 5900: 0.119675
Cost after iteration 6000: 0.118096
Cost after iteration 6100: 0.116557
Cost after iteration 6200: 0.115055
Cost after iteration 6300: 0.113590
Cost after iteration 6400: 0.112161
Cost after iteration 6500: 0.110765
Cost after iteration 6600: 0.109403
Cost after iteration 6700: 0.108072
Cost after iteration 6800: 0.106772
Cost after iteration 6900: 0.105501
Cost after iteration 7000: 0.104259
Cost after iteration 7100: 0.103044
Cost after iteration 7200: 0.101857
Cost after iteration 7300: 0.100695
Cost after iteration 7400: 0.099559
Cost after iteration 7500: 0.098446
Cost after iteration 7600: 0.097358
Cost after iteration 7700: 0.096292
Cost after iteration 7800: 0.095248
Cost after iteration 7900: 0.094226
Cost after iteration 8000: 0.093224
Cost after iteration 8100: 0.092243
Cost after iteration 8200: 0.091281
Cost after iteration 8300: 0.090338
Cost after iteration 8400: 0.089414
Cost after iteration 8500: 0.088508
Cost after iteration 8600: 0.087619
Cost after iteration 8700: 0.086747
Cost after iteration 8800: 0.085892
Cost after iteration 8900: 0.085053
Cost after iteration 9000: 0.084229
Cost after iteration 9100: 0.083420
Cost after iteration 9200: 0.082626
Cost after iteration 9300: 0.081847
Cost after iteration 9400: 0.081081
Cost after iteration 9500: 0.080329
Cost after iteration 9600: 0.079591
Cost after iteration 9700: 0.078865
Cost after iteration 9800: 0.078152
```

```
Cost after iteration 9900: 0.077451
train accuracy: 99.52153110047847 %
test accuracy: 70.0 %
```

[23]:
```python
index = 40
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a \"" +
 classes[int(logistic_regression_model['Y_prediction_test'][0,index])].
 decode("utf-8") +  "\" picture.")
```

```
y = 1, you predicted that it is a "cat" picture.
```



[ ]: