Logistic Regression with a Neural Network mindset

Welcome! You will build a logistic regression classifier to recognize cats. This assignment will step you through how to do this with a Neural Network mindset, and so will also hone your intuitions about deep learning.

Instructions:

- Do not use loops (for/while) in your code, unless the instructions explicitly ask you to do so.

You will learn to:

- Build the general architecture of a learning algorithm, including:
  - Initializing parameters
  - Calculating the cost function and its gradient
  - Using an optimization algorithm (gradient descent)
- Gather all three functions above into a main model function, in the right order.

1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- h5py is a common package to interact with a dataset that is stored on an H5 file.
- matplotlib is a famous library to plot graphs in Python.
- PIL and scipy are used here to test your model with your own picture at the end.

```
import numpy as np
import h5py
import os

def load_dataset():
    train_dataset = h5py.File('../data/train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # 
your train set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # 
your train set labels

    test_dataset = h5py.File('../data/test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your 
test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your 
test set labels

    classes = np.array(test_dataset["list_classes"][:]) # the list of 
classes
```

```
    train_set_y_orig = train_set_y_orig.reshape((1, 
train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, 
test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, 
test_set_y_orig, classes
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as img
import h5py
import scipy
from PIL import Image
from scipy import ndimage
#from lr_utils import load_dataset

%matplotlib inline
```

2 - Overview of the Problem set

Problem Statement: You are given a dataset ("data.h5") containing:

- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test images labeled as cat or non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num_px) and (width = num_px).

You will build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat.

Let's get more familiar with the dataset. Load the data by running the following code.

```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = 
load_dataset()
```

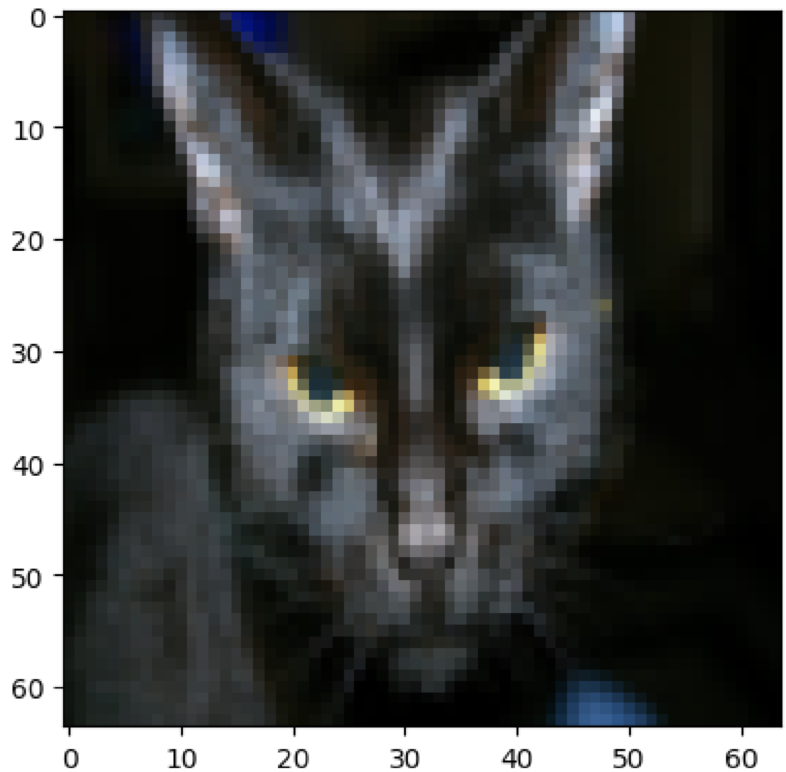
We added "_orig" at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with train_set_x and test_set_x (the labels train_set_y and test_set_y don't need any preprocessing).

Each line of your train_set_x_orig and test_set_x_orig is an array representing an image. You can visualize an example by running the following code. Feel free also to change the `index` value and re-run to see other images.

```
# Example of a picture
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.")


print(train_set_x_orig.shape)
```

```
y = [1], it's a 'cat' picture.
(209, 64, 64, 3)
```



Many software bugs in deep learning come from having matrix/vector dimensions that don't fit. If you can keep your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.

1. Exercise: Find and print the values for:

- m_train (number of training examples)
- m_test (number of test examples)
- num_px (= height = width of a training image)

```
### START CODE HERE ###
print(test_set_x_orig.shape)
print(train_set_x_orig.shape)
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = test_set_x_orig.shape[1]
print(m_train, m_test, num_px, end = "\n")
### END CODE HERE ###
```

```
(50, 64, 64, 3)
(209, 64, 64, 3)
209 50 64
```

Expected Output for m_train, m_test and num_px: **m_train** 209

m_test 50

num_px 64

For convenience, you should now reshape images of shape (num_px, num_px, 3) in a numpy-array of shape (num_px ¿ num_px ¿ 3, 1). After this, our training (and test) dataset is a numpy-array where each column represents a flattened image. There should be m_train (respectively m_test) columns.

2. Exercise: Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px ¿ num_px ¿ 3, 1).

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b ¿ c ¿ d, a) is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T is the transpose of
X
```

```
# Reshape the training and test examples

### START CODE HERE ### (≈ 2 lines of code)
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
### END CODE HERE ###

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))
print ("sanity check after reshaping: " +
str(train_set_x_flatten[0:5,0]))
```

```
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)
sanity check after reshaping: [17 31 56 22 33]
```

Expected Output:

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's standardize our dataset.

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

What you need to remember:

Common steps for pre-processing a new dataset are:

- Figure out the dimensions and shapes of the problem (m_train, m_test, num_px, ...)
- Reshape the datasets such that each example is now a vector of size (num_px * num_px * 3, 1)
- "Standardize" the data

3 - General Architecture of the learning algorithm

It's time to design a simple algorithm to distinguish cat images from non-cat images.

You will build a Logistic Regression, using a Neural Network mindset. The following Figure explains why **Logistic Regression is actually a very simple Neural Network!**

Mathematical expression of the algorithm:

For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$

$$L(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$$

Key steps: In this exercise, you will carry out the following steps:

- Initialize the parameters of the model
- Learn the parameters for the model by minimizing the cost

- Use the learned parameters to make predictions (on the test set)

- Analyse the results and conclude

4 - Building the parts of our algorithm

The main steps for building a Neural Network are:

1. Define the model structure (such as number of input features)
2. Initialize the model's parameters
3. Loop:
   - Calculate current loss (forward propagation)
   - Calculate current gradient (backward propagation)
   - Update parameters (gradient descent)

You often build 1-3 separately and integrate them into one function we call `model()`.

4.1 - Helper functions

3. Exercise: Implement `sigmoid()`. As you've seen in the figure above, you need to compute $sigmoid\left(w^T x+b\right)=\frac{1}{1+e^{-\left(w^T x+b\right)}}$ to make predictions. Use np.exp().

```
# GRADED FUNCTION: sigmoid

def sigmoid(z):
    """
    Compute the sigmoid of z

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

    Return:
    s -- sigmoid(z)
    """

    ### START CODE HERE ### (≈ 1 line of code)
    s = 1/(1 + np.exp(-z))
    ### END CODE HERE ###

    return s
```

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

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

Expected Output:

4.2 - Initializing parameters

4. Exercise: Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros. If you don't know what numpy function to use, look up np.zeros() in the Numpy library's documentation.

```
# GRADED FUNCTION: initialize_with_zeros

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)
    """

    ### START CODE HERE ### (≈ 1 line of code)
    w = np.zeros((dim, 1))
    b = 0.0
    ### END CODE HERE ###

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b

dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

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

Expected Output:

For image inputs, w will be of shape (num_px × num_px × 3, 1).

4.3 - Forward and Backward propagation

Now that your parameters are initialized, you can do the "forward" and "backward" propagation steps for learning the parameters.

5. Exercise: Implement a function `propagate()` that computes the cost function and its gradient.

```
# GRADED FUNCTION: propagate

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)
    ### START CODE HERE ### (≈ 2 lines of code)
    A = sigmoid(np.dot(w.T, X) + b)  # compute activation
    cost = -np.sum((Y*np.log(A) + (1 - Y)*np.log(1 - A)))/m    # 
compute cost
    ### END CODE HERE ###

    # BACKWARD PROPAGATION (TO FIND GRAD)
    ### START CODE HERE ### (≈ 2 lines of code)
    dw = np.dot(X, (A - Y).T)/m
    db = np.sum(A - Y)/m
    ### END CODE HERE ###

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

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

```
    return grads, cost
```

```
w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-
3.2]]), np.array([[1,0,1]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

```
dw = [[0.99845601]
 [2.39507239]]
db = 0.00145557813678
cost = 5.801545319394553
```

Expected Output:

4.4 - Optimization

- You have initialized your parameters.
- You are also able to compute a cost function and its gradient.
- Now, you want to update the parameters using gradient descent.

6. Exercise: Write down the optimization function. The goal is to learn w and b by minimizing the cost function J. For a parameter θ, the update rule is $ \theta = \theta - \alpha \text{ } d\theta$, where α is the learning rate.

```
# GRADED FUNCTION: optimize

def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost =
False):


    costs = []

    for i in range(num_iterations):

        # Cost and gradient calculation
        ### START CODE HERE ###
        grads, cost = propagate(w, b, X, Y)
        ### END CODE HERE ###

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

        # update rule for w and b
        ### START CODE HERE ###
        w = w - learning_rate*dw
        b = b - learning_rate*db
        ### END CODE HERE ###
```

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

        # Print the cost every 100 training iterations
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

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

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

    return params, grads, costs
```

```
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"]))
```

```
w = [[0.19033591]
 [0.12259159]]
b = 1.9253598300845747
dw = [[0.67752042]
 [1.41625495]]
db = 0.21919450454067654
```

Expected Output:

7. Exercise: The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. Implement the `predict()` function. There are two steps to computing predictions:

1. Calculate $\hat{Y} = A = \sigma\left(w^T X + b\right)$

2. Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector `Y_prediction`. If you wish, you can use an `if/else` statement in a `for` loop (though there is also a way to vectorize this).

```
# GRADED FUNCTION: predict

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
    ### START CODE HERE ###
    A = sigmoid(np.dot(w.T, X) + b)
    ### END CODE HERE ###

    for i in range(A.shape[1]):
      # Convert probabilities to actual predictions
      ### START CODE HERE ###
      Y_prediction[0, i] = 1 if A[0, i] > 0.5 else 0
      ### END CODE HERE ###

    assert(Y_prediction.shape == (1, m))

    return Y_prediction

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.]]
```

Expected Output:

What to remember:

You've implemented several functions that:

- Initialize (w,b)
- Optimize the loss iteratively to learn parameters (w,b):
  - computing the cost and its gradient
  - updating the parameters using gradient descent
- Use the learned (w,b) to predict the labels for a given set of examples

5 - Merge all functions into a model

You will now see how the overall model is structured by putting together all the building blocks (functions implemented in the previous parts) together, in the right order.

8. Exercise: Implement the model function. Use the following notation:

- Y_prediction_test for your predictions on the test set
- Y_prediction_train for your predictions on the train set
- w, costs, grads for the outputs of optimize()

```
# GRADED FUNCTION: model

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.
    """
    
    ### START CODE HERE ###
    # initialize parameters with zeros (≈ 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent (≈ 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, 
num_iterations, learning_rate, True)
    
    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]
```

```
    # Predict test/train set examples (≈ 2 lines of code)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    ### END CODE HERE ###

    # Print train/test Errors
    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
```

Run the following cell to train your model.

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, 
num_iterations = 2000, learning_rate = 0.005, print_cost = True)
```

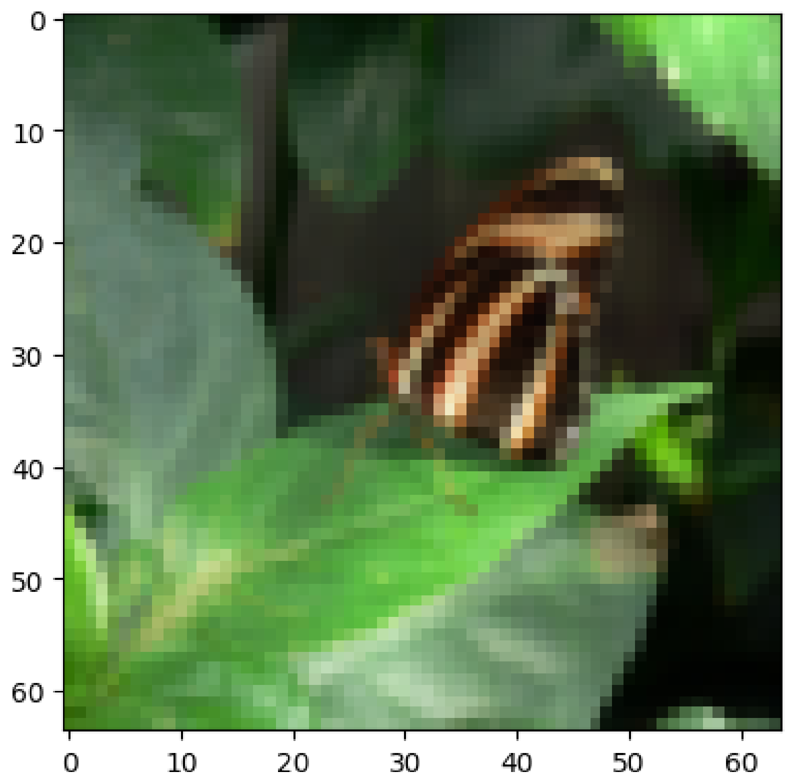
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694%
test accuracy: 70.0%
```

Expected Output:

Comment: Training accuracy is close to 100%. This is a good sanity check: your model is working and has high enough capacity to fit the training data. Test accuracy is 70%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier.

```
# Example of a picture that was wrongly classified.
index = 5
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))

outputClass = classes[int(d["Y_prediction_test"][0, 
index])].decode("utf-8")

print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is 
a \"" + outputClass +  "\" picture.")
```

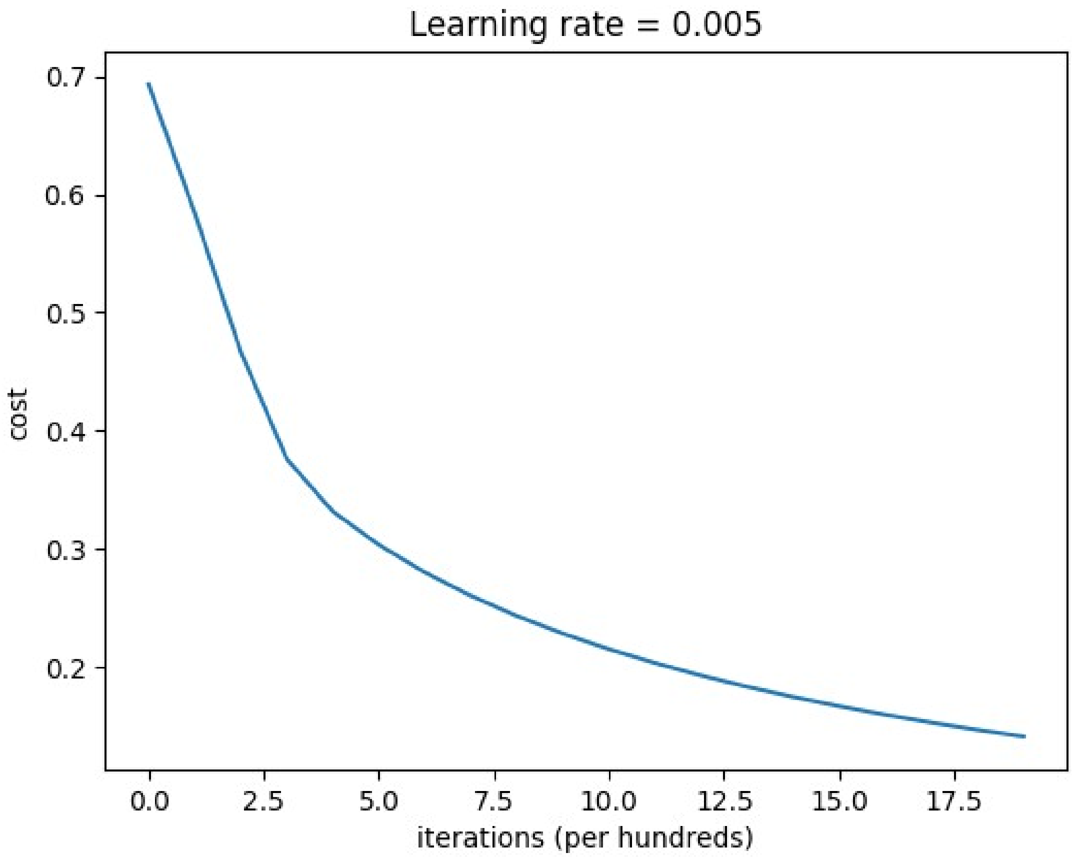
```
y = 0, you predicted that it is a "cat" picture.
```



Let's also plot the cost function and the gradients.

```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
```

```
plt.title("Learning rate = " + str(d["learning_rate"]))
plt.show()
```



Interpretation: You can see the cost decreasing. It shows that the parameters are being learned. However, you see that you could train the model even more on the training set. Try to increase the number of iterations in the cell above and rerun the cells. You might see that the training set accuracy goes up, but the test set accuracy goes down. This is called overfitting.

7 - Test with custom images

Create/find a database of at least 200 images, test your classifier and report your result. Your database will have the following characteristics:

- 100 at least images of the positive class; i.e., images with fingerprints.
- 100 at least images of the negative class; i.e., images without fingerprints.

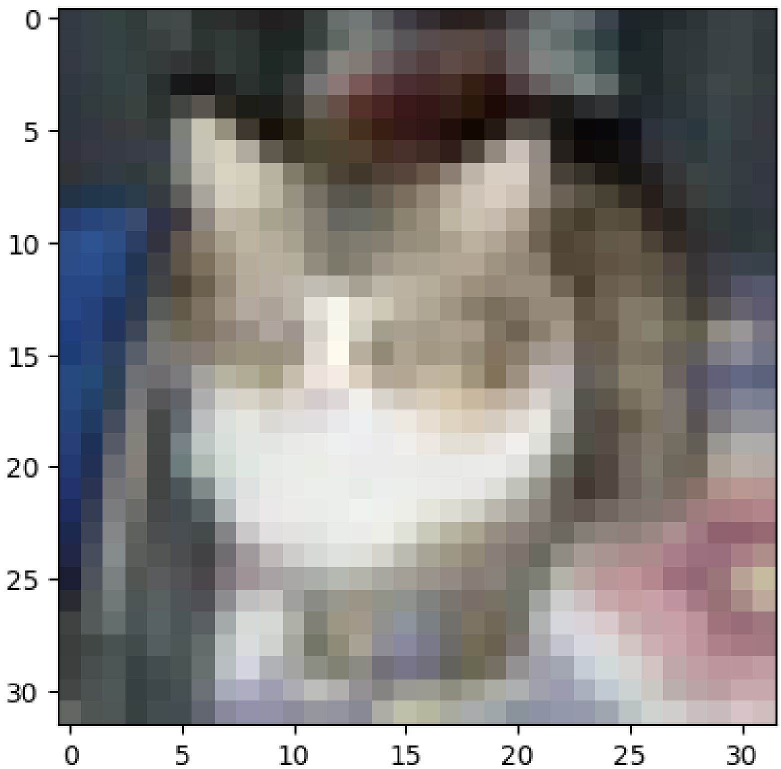
```
def get_dataset(train_size = 500, test_size = 100):
    train_x_o, train_y = np.zeros((train_size, 32, 32, 3)),
np.zeros((train_size))
    test_x_o, test_y = np.zeros((test_size, 32, 32, 3)),
np.zeros((test_size))
    for i in range(train_size):
```

```python
        path, isCat = None, None
        if np.random.uniform() < 0.5:
            path = "../../data/cifar10/train/airplane/" + f"{(i + 1):04d}" + ".png"
            isCat = False
        else:
            path = "../../data/cifar10/train/cat/" + f"{(i + 1):04d}" + ".png"
            isCat = True
        train_x_o[i] = img.imread(path)
        train_y[i] = isCat
    for i in range(test_size):
        path, isCat = None, None
        if np.random.uniform() < 0.5:
            path = "../../data/cifar10/test/airplane/" + f"{(i + 1):04d}" + ".png"
            isCat = False
        else:
            path = "../../data/cifar10/test/cat/" + f"{(i + 1):04d}" + ".png"
            isCat = True
        test_x_o[i] = img.imread(path)
        test_y[i] = isCat
    return train_x_o, train_y, test_x_o, test_y

train_x_o, train_y, test_x_o, test_y = get_dataset()
num_px = 32
print(train_x_o.shape, train_y.shape, test_x_o.shape, test_y.shape, sep = "\n")
```

```
(500, 32, 32, 3)
(500,)
(100, 32, 32, 3)
(100,)
```

```python
# Example of a picture
index = np.random.randint(1, 500)
plt.imshow(train_x_o[index])
print(train_x_o.shape)
print("It's a", "cat" if train_y[index] else "non-cat")
```

```
(500, 32, 32, 3)
It's a cat
```



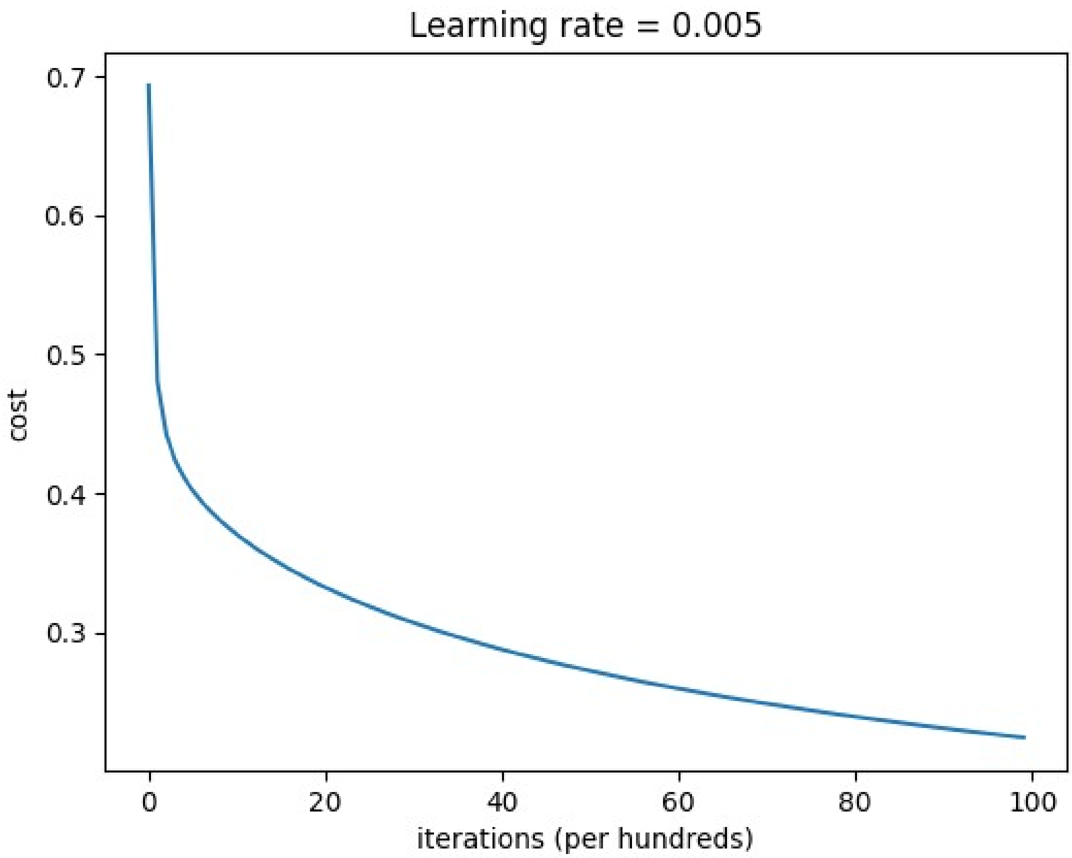

```
train_x = train_x_o.reshape(train_x_o.shape[0], -1).T
test_x = test_x_o.reshape(test_x_o.shape[0], -1).T

# Standardize
#train_x = train_x/255.
#test_x = test_x/255.

d = model(train_x, train_y, test_x, test_y, num_iterations = 10000,
learning_rate = 0.005, print_cost = False)
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.480321
Cost after iteration 200: 0.442196
Cost after iteration 300: 0.423458
Cost after iteration 400: 0.411071
Cost after iteration 500: 0.401592
Cost after iteration 600: 0.393728
Cost after iteration 700: 0.386890
Cost after iteration 800: 0.380772
Cost after iteration 900: 0.375194
Cost after iteration 1000: 0.370045
Cost after iteration 1100: 0.365250
Cost after iteration 1200: 0.360754
Cost after iteration 1300: 0.356517
Cost after iteration 1400: 0.352506
Cost after iteration 1500: 0.348696
```

```
Cost after iteration 1600: 0.345066
Cost after iteration 1700: 0.341600
Cost after iteration 1800: 0.338281
Cost after iteration 1900: 0.335097
Cost after iteration 2000: 0.332036
Cost after iteration 2100: 0.329091
Cost after iteration 2200: 0.326251
Cost after iteration 2300: 0.323509
Cost after iteration 2400: 0.320859
Cost after iteration 2500: 0.318294
Cost after iteration 2600: 0.315809
Cost after iteration 2700: 0.313400
Cost after iteration 2800: 0.311061
Cost after iteration 2900: 0.308789
Cost after iteration 3000: 0.306581
Cost after iteration 3100: 0.304432
Cost after iteration 3200: 0.302340
Cost after iteration 3300: 0.300302
Cost after iteration 3400: 0.298315
Cost after iteration 3500: 0.296377
Cost after iteration 3600: 0.294486
Cost after iteration 3700: 0.292639
Cost after iteration 3800: 0.290835
Cost after iteration 3900: 0.289072
Cost after iteration 4000: 0.287348
Cost after iteration 4100: 0.285661
Cost after iteration 4200: 0.284011
Cost after iteration 4300: 0.282395
Cost after iteration 4400: 0.280813
Cost after iteration 4500: 0.279262
Cost after iteration 4600: 0.277743
Cost after iteration 4700: 0.276253
Cost after iteration 4800: 0.274793
Cost after iteration 4900: 0.273360
Cost after iteration 5000: 0.271954
Cost after iteration 5100: 0.270574
Cost after iteration 5200: 0.269218
Cost after iteration 5300: 0.267888
Cost after iteration 5400: 0.266580
Cost after iteration 5500: 0.265296
Cost after iteration 5600: 0.264034
Cost after iteration 5700: 0.262793
Cost after iteration 5800: 0.261572
Cost after iteration 5900: 0.260372
Cost after iteration 6000: 0.259192
Cost after iteration 6100: 0.258030
Cost after iteration 6200: 0.256887
Cost after iteration 6300: 0.255762
Cost after iteration 6400: 0.254654
Cost after iteration 6500: 0.253563
```

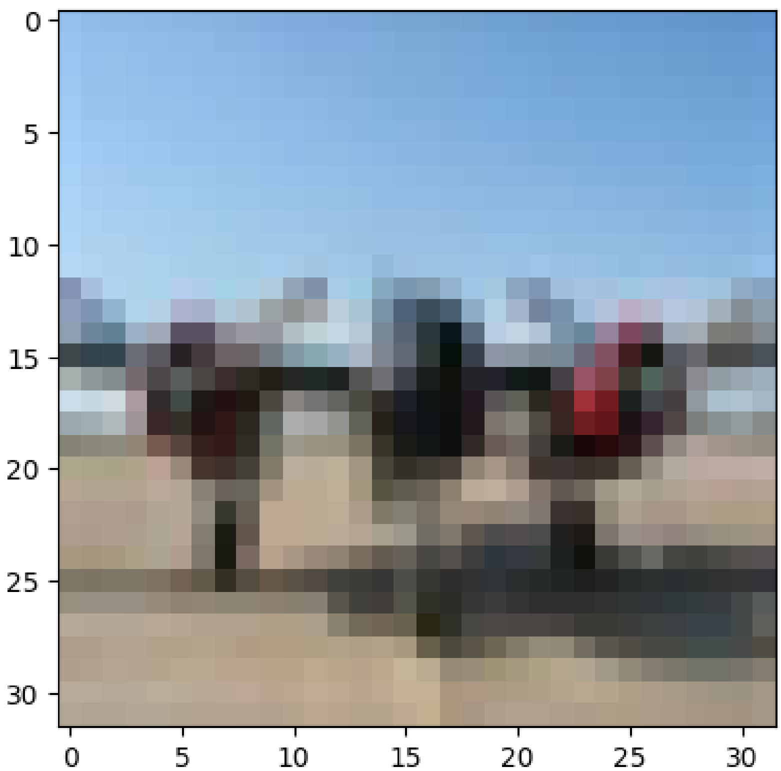
```
Cost after iteration 6600: 0.252488
Cost after iteration 6700: 0.251430
Cost after iteration 6800: 0.250387
Cost after iteration 6900: 0.249359
Cost after iteration 7000: 0.248347
Cost after iteration 7100: 0.247348
Cost after iteration 7200: 0.246364
Cost after iteration 7300: 0.245393
Cost after iteration 7400: 0.244436
Cost after iteration 7500: 0.243491
Cost after iteration 7600: 0.242560
Cost after iteration 7700: 0.241640
Cost after iteration 7800: 0.240733
Cost after iteration 7900: 0.239838
Cost after iteration 8000: 0.238954
Cost after iteration 8100: 0.238081
Cost after iteration 8200: 0.237220
Cost after iteration 8300: 0.236369
Cost after iteration 8400: 0.235528
Cost after iteration 8500: 0.234698
Cost after iteration 8600: 0.233878
Cost after iteration 8700: 0.233067
Cost after iteration 8800: 0.232267
Cost after iteration 8900: 0.231475
Cost after iteration 9000: 0.230693
Cost after iteration 9100: 0.229920
Cost after iteration 9200: 0.229155
Cost after iteration 9300: 0.228399
Cost after iteration 9400: 0.227652
Cost after iteration 9500: 0.226912
Cost after iteration 9600: 0.226181
Cost after iteration 9700: 0.225458
Cost after iteration 9800: 0.224742
Cost after iteration 9900: 0.224034
train accuracy: 95.6%
test accuracy: 80.0%
```

```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate = " + str(d["learning_rate"]))
plt.show()
```



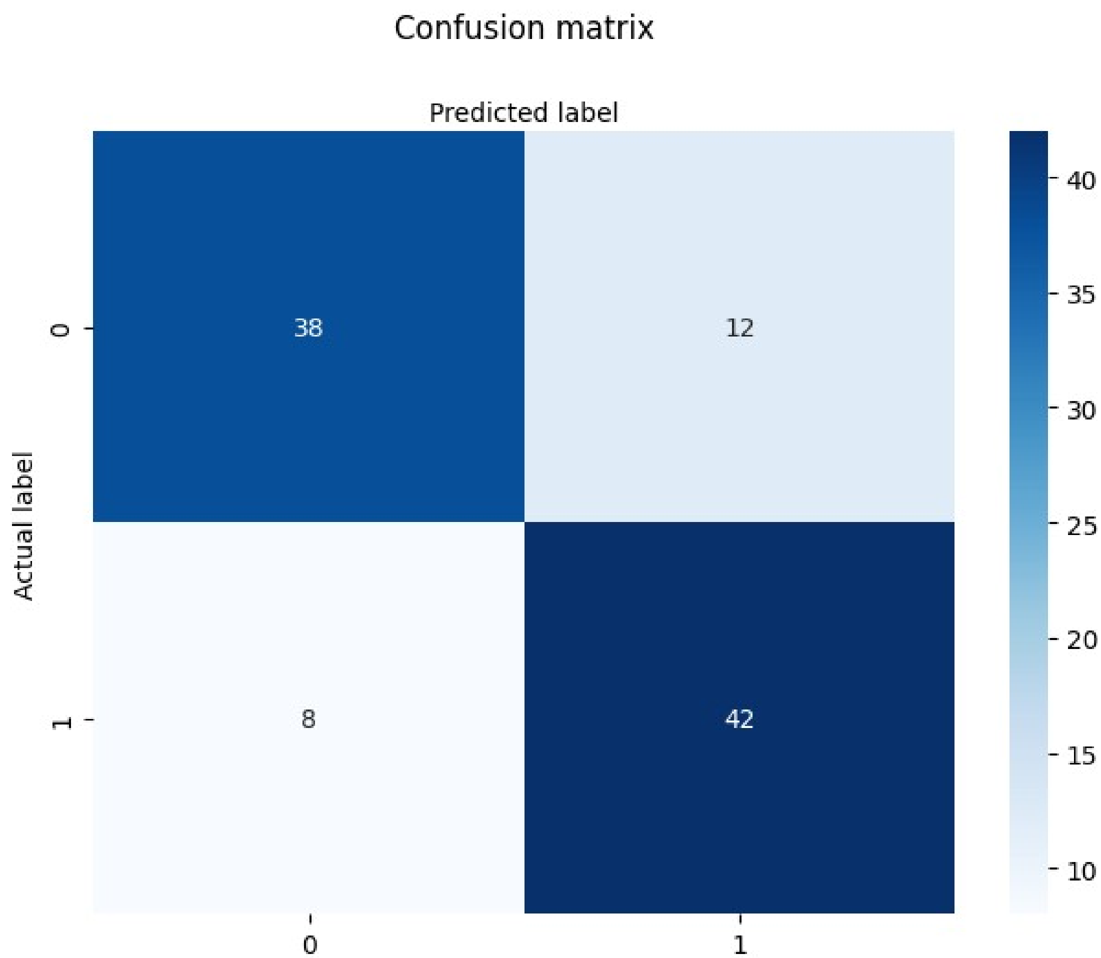

```
index = np.random.randint(1, 100)
plt.imshow(test_x_o[index])
print(test_x_o.shape)
print("It's a", "cat" if test_y[index] else "non-cat")
print("The model predicted:", "cat" if d["Y_prediction_test"][0,
index] else "non-cat")
```

```
(100, 32, 32, 3)
It's a non-cat
The model predicted: non-cat
```




```
from sklearn import metrics
import seaborn as sns
import pandas as pd

cnf_matrix = metrics.confusion_matrix(test_y,
np.squeeze(d["Y_prediction_test"]))
class_names=[0,1] # name  of classes
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
# create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True,
cmap="Blues" ,fmt='g') #another color map for confusion matrix: YlGnBu
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

```
Text(0.5, 427.9555555555555, 'Predicted label')
```



Report

Please report and discuss:

- Description of the used dataset, e.g., number of training and testing samples, kind of data, problem, etc.
- Train and test accuracy.
- Play with the learning rate and the number of iterations.
- Try different initialization methods and compare the results.
- Test other preprocessings (center the data, or divide each row by its standard deviation)

Dataset

Name: CIFAR 10

Sizes:

- The training set contains about 250 images of airplanes and 250 images of cats.

- The testing set contains about 50 images of airplanes and 50 images of cats.
- Each image is 32x32x3.

Accuracy:

I would say accuracy is kind of decent based on the fact that logistic regression is one of the most primitives of its kind.

- Train accuracy: 95.0%
- Test accuracy: 80.0%

Experimentiation

Iterations

- 100,000 iterations improved the train accuracy to 99% and the test accuracy slightly increased. I would say the extra time is not worth it but I was impressed by the fact that overfitting didn't happen.

Step size

- Step size of 0.05 fitted better training data but didn't do any better on test data.
- Step size of 0.0005 is slower, so it doesn't reach the same results with that number of iterations.

Changing initial data

The dataset is randomly chosen, it gives different results, sometimes better than the average, sometimes worse.

Modifying data

Standardizing data gave terrifying results. 50% of accuracy in both cases.