HERIOT WATT UNIVERSITY

BIOLOGICALLY INSPIRED COMPUTATION

COURSEWORK 1A

Logistic Regression model using Gradient Descendent

Author
Mohit VAISHNAV

Supervisor
Dr. Marta VALLEJO

Submission Last Date 26th, Sept

1 Introduction

Logistic Regression is the building block for any Deep learning model. It is like a linear classifier which uses logits called scores for predicting target class. LR uses Softmax function to relate categorical dependent variable and independent variables. Dependent variables are to be predicted whereas independent variables are the features used for prediction. They are used to calculate the likelihood occurrence.

2 Procedure

2.1 Loading Dataset

Test and training dataset has been provided which are labelled as *cat* and *non cat*. Dimension of the image is $64 \times 64 \times 3$ representing them as RGB image.

```
# Loading the dataset

# training label:
train_dataset = h5py.File('trainCats.h5', "r")
trainSetX = np.array(train_dataset["train_set_x"][:]) # your train set features
trainSetY = np.array(train_dataset["train_set_y"][:]) # your train set labels

# Reading the test data and test labels:
test_dataset = h5py.File('testCats.h5', "r")
testSetX = np.array(test_dataset["test_set_x"][:]) # your test set features
testSetY = np.array(test_dataset["test_set_y"][:]) # your test set labels
```

```
def normalize255(X):
    X_new = X/255

    return X_new

def normalizeL1(X):
    X_normalized = preprocessing.normalize(X, norm='l1')

    return X_normalized

def normalizeL2(X):
    X_normalized = preprocessing.normalize(X, norm='l2')

    return X_normalized
```

2.2 Processing

Once the data is in place it has to be converted in the form desired then normalization is done by dividing the whole matrix by 255. Other option could be to use *l*1 or *l*2 normalization from the *Scikit* library.

```
# To check the shape of the input data and convert in desired
    shape:
trainSetY = trainSetY.reshape((1, trainSetY.shape[0]))
testSetY = testSetY.reshape((1, testSetY.shape[0]))

num_var = trainSetX.shape[1]*trainSetX.shape[2]*trainSetX.shape[3]
trainSetX_new = np.zeros((trainSetX.shape[0],num_var)).T
for i in range(trainSetX.shape[0]):
    trainSetX_new[:, i] = trainSetX[i].reshape(-1)

testSetX_new = np.zeros((testSetX.shape[0],num_var)).T

for i in range(testSetX.shape[0]):
    testSetX_new[:, i] = testSetX[i].reshape(-1)
```

```
# To check the shape of the input data and convert in desired
    shape:
trainSetY = trainSetY.reshape((1, trainSetY.shape[0]))
testSetY = testSetY.reshape((1, testSetY.shape[0]))

num_var = trainSetX.shape[1]*trainSetX.shape[2]*trainSetX.shape[3]
trainSetX_new = np.zeros((trainSetX.shape[0],num_var)).T
for i in range(trainSetX.shape[0]):
    trainSetX_new[:, i] = trainSetX[i].reshape(-1)

testSetX_new = np.zeros((testSetX.shape[0],num_var)).T

for i in range(testSetX.shape[0]):
    testSetX_new[:, i] = testSetX[i].reshape(-1)
```

2.3 Implementation

There are two ways of implementing the Logistic Regression, one is to hard code everything and the other is to use the inbuilt function already designed with the best optimization techniques. Logistic regression function can be used from the Scikit Library which can ease the implementation to a greater extent. It can be done as follows:

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1000.0, random_state=0, max_iter = 1000)
lr.fit(trainSetX_new.T, trainSetY.T.ravel())

# For prediction of test set
Y_prediction = lr.predict(testSetX_new.T)
Y_prediction.shape
```

While on the other hand, each of the module can be modelled step by step helping create the basic of a neural network from scratch. Each of the building block can be defined as a definition to be used further. Below mentioned bibliography shows a list of web links used for the reference purpose.

```
# Define sigmoid function:
def sigmoid(z):
    sig = 1/(1+np.exp(-(z)))
    return sig

# Calculating cross entropy
def propagate(w, b, SetX, SetY):
    m = SetX.shape[1]
    A = sigmoid(np.dot(w.T,SetX)+b)
    L = -1/m * np.sum(SetY*np.log(A)+(1-SetY)*np.log(1-A))
    dw = (1/m) * (np.dot(SetX,(A-SetY).T))
    db = (1/m) * (np.sum(A-SetY))

    # Removes the extra dimension:
    L = np.squeeze(L)

    # Creating a dictionary in python
    # https://www.pythonforbeginners.com/dictionary/how-to-use-
    dictionaries-in-python/
    grads = {"dw": dw, "db": db}

    return grads, L

# To find optimal w and b
def converge(SetX, SetY, w, b, num_iter, learning_rate):
    costs = []
    for i in range(num_iter):
        grads, L = propagate(w, b, SetX, SetY)
        dw = grads["dw"]
        db = grads["db"]
        w = w-learning_rate*dw
        b = b-learning_rate*db
        costs.append(L)

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

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

    return params, grads, costs

def predict(w, b, SetX):
    m = SetX.shape[1]
    predict_Y = np.zeros((1, m))
    w = w.reshape(SetX.shape[0], 1)
    A = sigmoid( np.dot(w.T, SetX) + b)
    for i in range(A.shape[1]):
        if A[0,i] <= 0.5:
            predict_Y[0,i] = 0
        else:
            predict_Y[0,i] = 1

    return predict_Y

# Creating the model:
def model(trainSetX, trainSetY, testSetX, testSetY, num_iterations
    , learning_rate):

    #Initialize paramters with 0
    w = np.zeros((trainSetX.shape[0],1))
    b = 0

    # Gradient Descent
    params, grads, costs = converge(trainSetX, trainSetY, w, b,
    num_iterations, learning_rate)

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

    #Predict test/train set examples
    predict_testSetY = predict(w,b,testSetX)
    predict_trainSetY = predict(w,b,trainSetX)

    final = {"costs": costs, "predict_testSetY": predict_testSetY,
     "predict_trainSetY": predict_trainSetY,
         "b": b, "w": w, "num_iterations": num_iterations, "
    learning_rate": learning_rate}

    #Print train/test errors
    print("Accuracy for training set is: {} %".format(100-np.mean(
    np.abs(predict_trainSetY-trainSetY))*100))
    print("Accuracy for testing set is: {} %".format(100-np.mean(
    np.abs(predict_testSetY-testSetY))*100))

    return final
```

```


# https://mashimo.wordpress.com/2017/10/19/cat-or-not-cat/
# http://dataaspirant.com/2017/04/15/implement-logistic-regression
    -model-python-binary-classification/
```

Different modules are created such as: sigmoid, initialize_params, propogate, converge, predict and finally the model. In *sigmoid*, this is used as the activation function, which could be replaced with softmax function in case of multiclass classification. Initialization of the parameters are done by assigining them values as 0. Further they are propagated by calculating *dw* & *db* as explained in the class. To find the least cost solution, convergence is carried out using the function *converge*. It requires the parameters as *num_iter, learning_rate* along with others. Parameters are updated after each iteration for all the samples in the data-set. Finally all the functions are combined to form the full fledged linear regression function for any kind of input data.

2.4 Run the code

To run the code, input parameters are to be converted in the desired shape and then passed in the respective model along with the desired learning rate and number of iteration.

```
# To run the code:
d = model(trainSetX_new, trainSetY, testSetX_new, testSetY,
    num_iterations=1000, learning_rate=0.05)
```

To display the cost of each iteration:

```
%matplotlib notebook
plt.figure()
plt.plot(np.arange(len(d['costs'])), d['costs'])
```

3 Observation and Result

While following the inbuilt function of Scikit library *confusion_matrix* can be created which is found as *diag(137,72)*. By default the number of iteration is 100 which results in accuracy of 72%. Even after changing this value to any higher value there is not any change in the test accuracy. Confusion Matrix for the test data set is: [13, 4; 10, 23].

```
print("test accuracy: {} %".format(100 - np.mean(np.abs(
    Y_prediction - testSetY)) * 100))
```

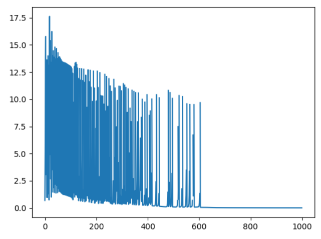
```
# To run the code in Part I:
d = model(trainSetX_new, trainSetY, testSetX_new, testSetY,
    num_iterations=1000, learning_rate=0.25)

# To run the code in part II:
# Pass the training set and index
lr.fit(trainSetX_new.T, trainSetY.T.ravel())

# Test data
Y_prediction = lr.predict(testSetX_new.T)
```

Logistic Regression model comes out to be:

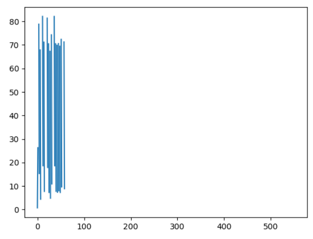
$$\begin{aligned} LogisticRegression(C = 1000.0, class\_weight = None, dual = False, \\ fit\_intercept = True, intercept\_scaling = 1, max\_iter = 100, \\ multi\_class =' ovr', n\_jobs = 1, penalty =' l2', random\_state = 0, \\ solver =' liblinear', tol = 0.0001, verbose = 0, warm\_start = False) \end{aligned} \tag{1}$$

Training set consists of 209 images and test set has 50 images. For training purpose, vector of length 64x64x3 (12288) is used which is equal to number of parameters needed to be optimized. Classes in the data set are: non cat and cat.

In the part II of the implementation when learning rate is .05, even after increasing the number of iteration, results does not seems to be promising wrt to the part I, direct function. If the learning rate is increased to .25, same result is obtained as the previous one when test data set is used. From Fig. 1, it can be seen that with an optimal learning rate, result can be achieved with very few iteration (less than 100 in this case).



(a) Learning rate .05



(b) Learning Rate .25

Figure 1: Cost vs Number of iteration for Normalization as /255

Another alteration can be done by changing different initialization of weights. Few combination tried are with values as -1, 0, +1. By default all the results cited are with values 0 whereas in Table. 1 training and testing accuracy can be seen with respect to different iteration from 100 to 700.

While looking at the values of "costs" obtained, there are presence of many values as *nan*.

Number of Iteration	Train Accuracy	Test Accuracy
100	65.5	34
500	96	74
600	98	70
700	100	72

Table 1: Mean Accuracy with weights initialization of +1 and -1 for Norm = /255

It can be said that creating the whole module is very much essential to learn the implementation of a basic neural network but in a longer run one should also know to use the commands too.

3.1 Trial 2: L2 Normalization

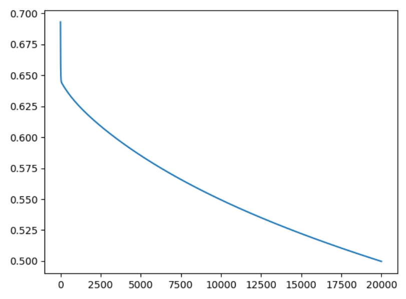
In second trial normalization is changed to two more values, *l1 & l2*. With L1 normalization and learning rate of 0.25 even after increasing the number of iteration to about 20000, there is no change in the test set accuracy. It does not moves beyond 56% while for the training set 100% accuracy is achieved after 4000 iterations (Table. 2). Various costs v/s number of iteration values can be observed in Fig. 2.

3.1.1 Changing Learning rate

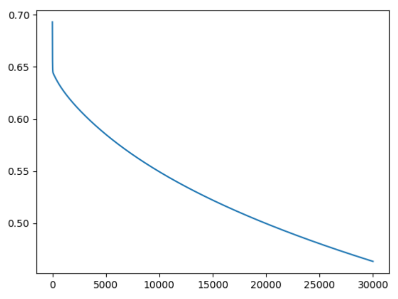
Changing the learning rate from .01 to .5 and consecutively altering the number of iteration till 15000, maximum test case accuracy comes out to be 58%.

3.2 L1 Normalization

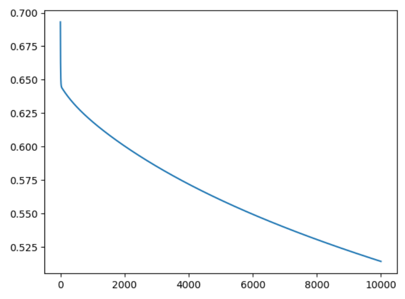
Following the same as above with the normalization function *l*1 gives the following result in Table. 3. Their various combination of costs value can be visualized in Fig. 3.



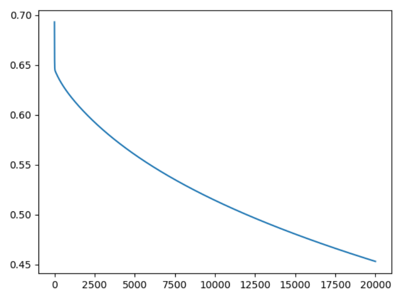
(a) (.15, 20000)



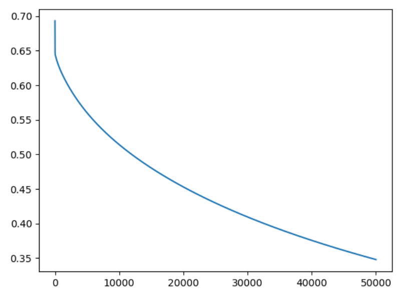
(b) (.15, 30000)



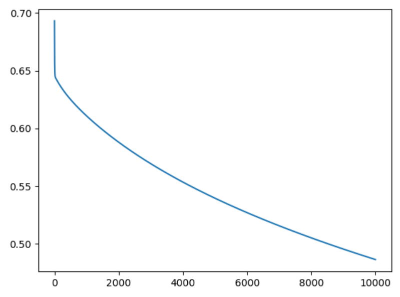
(c) (.25, 10000)



(d) (.25, 20000)

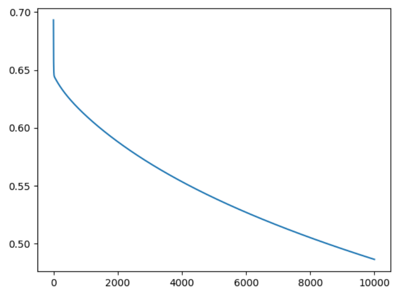


(e) (.25, 50000)

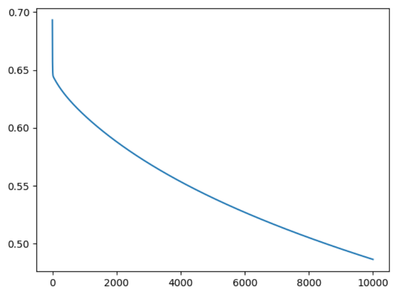


(f) (.35, 100000)

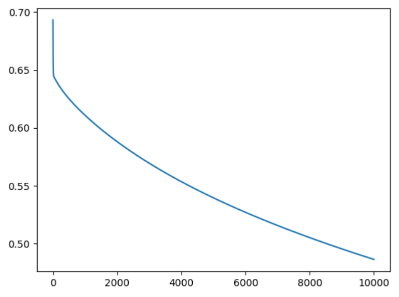
Figure 2: Comparison of Cost v/s learning rate and number of iteration for L2



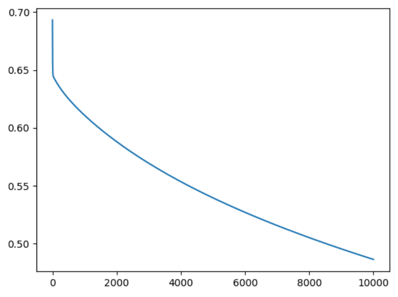
(a) (.05, 10000)



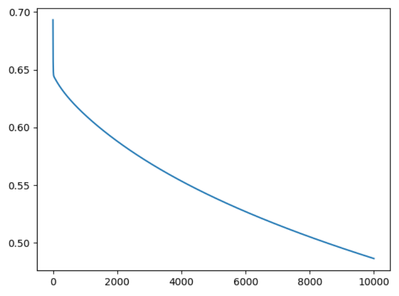
(b) (.03, 10000)



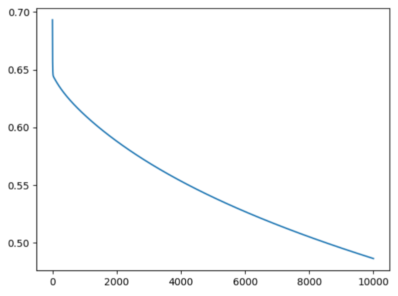
(c) (.15, 10000)



(d) (.25, 10000)

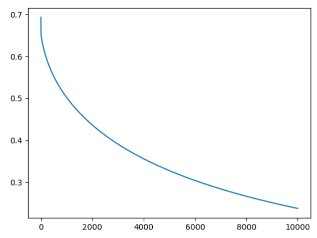


(e) (.025, 20000)

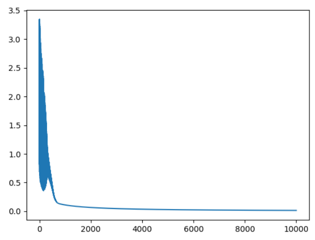


(f) (.025, 10000)

Figure 3: Comparison of Cost v/s learning rate and number of iteration for L2



(a) Learning rate .02



(b) Learning Rate .5

Figure 4: Cost vs Number of iteration

Learning Rate	Iteration	Train Accuracy	Test Accuracy
.05	1000	85.64	64
.05	10000	99.04	56
.05	20000	100	56
.15	10000	100	56
.15	20000	100	56
.025	10000	96	56
.03	10000	97	58
.035	10000	98	58
.035	10000	99.52	56

Table 2: Comparative study with L2 Norm

4 Self Analysis

Table. 4 shows the markings.

5 Conclusion

This is the first step to put forward towards learning neural networks. For implementation purpose, *python* is used. Two types of implementations are done in this coursework, one, where the code is developed from the scratch and all the modules are defined to be used finally as a logistic regression function. While in the second kind of implementation inbuilt functions are used to test on the data set. On changing different parameters it can be seen that learning rate plays a very important role in convergence. For this

Learning Rate	Iteration	Train Accuracy	Test Accuracy
.05	1000	65.5	34
.05	10000	65.5	72
.05	1000	65.5	34
.15	10000	69.38	64
.15	20000	76.5	62
.15	30000	78	62
.15	100000	91	60
.25	10000	73.7	64
.25	20000	79.9	62
.25	50000	90	62
.35	10000	77	62
.35	1000	65.55	72

Table 3: Comparative study with L1 norm

Question	S_Mark	Final
Did you code the entire algorithm?	x	
Did you vectorise your code?	x	
Is your code runnable?	x	
Did you collect data to measure convergence?	x	
Did you create a report explaining what you did?	x	

Table 4: Coursework 1a

particular type of dataset $l1$ norm seems to be the promising one. Though with general normalization (dividing by 255) result is somewhat misleading where the convergence is achieved in less than 100 iterations and result on test data set is very high. With a proper learning rate, number of iteration could be decreased drastically. Tuning of hyper parameter is one of the research topics now a days.

6 Code

6.1 Part I

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
import h5py
import os

# Loading the dataset
os.getcwd()

# training label:
train_dataset = h5py.File('trainCats.h5', "r")
trainSetX = np.array(train_dataset["train_set_x"][:]) # your train set features
trainSetY = np.array(train_dataset["train_set_y"][:]) # your train set labels

# Reading the test data and test labels:
test_dataset = h5py.File('testCats.h5', "r")
testSetX = np.array(test_dataset["test_set_x"][:]) # your test set features
testSetY = np.array(test_dataset["test_set_y"][:]) # your test set labels


# To check the shape of the input data and convert in desired shape:
trainSetY = trainSetY.reshape((1, trainSetY.shape[0]))
testSetY = testSetY.reshape((1, testSetY.shape[0]))

num_var = trainSetX.shape[1]*trainSetX.shape[2]*trainSetX.shape[3]
trainSetX_new = np.zeros((trainSetX.shape[0],num_var)).T
for i in range(trainSetX.shape[0]):
    trainSetX_new[:, i] = trainSetX[i].reshape(-1)

testSetX_new = np.zeros((testSetX.shape[0],num_var)).T

for i in range(testSetX.shape[0]):
    testSetX_new[:, i] = testSetX[i].reshape(-1)


## Calling different normalization function

#testSetX_new = normalize255(testSetX_new)
#trainSetX_new = normalize255(trainSetX_new)
```

```

#testSetX_new = normalizeL1(testSetX_new)
#trainSetX_new = normalizeL1(trainSetX_new)

testSetX_new = normalizeL2(testSetX_new)
trainSetX_new = normalizeL2(trainSetX_new)

# Defining normalization function
def normalize255(X):
    X_new = X/255

    return X_new

def normalizeL1(X):
    X_normalized = preprocessing.normalize(X, norm='l1')

    return X_normalized

def normalizeL2(X):
    X_normalized = preprocessing.normalize(X, norm='l2')

    return X_normalized

# Define sigmoid function:
def sigmoid(z):
    sig = 1/(1+np.exp(-(z)))
    return sig

# Calculating cross entropy
def propagate(w, b, SetX, SetY):
    m = SetX.shape[1]
    A = sigmoid(np.dot(w.T,SetX)+b)
    L = -1/m * np.sum(SetY*np.log(A)+(1-SetY)*np.log(1-A))
    dw = (1/m) * (np.dot(SetX,(A-SetY).T))
    db = (1/m) * (np.sum(A-SetY))

    # Removes the extra dimension:
    L = np.squeeze(L)

    # Creating a dictionary in python
    # https://www.pythonforbeginners.com/dictionary/how-to-use-
    dictionaries-in-python/
    grads = {"dw": dw, "db": db}

    return grads, L

# To find optimal w and b
def converge(SetX, SetY, w, b, num_iter, learning_rate):
    costs = []
    for i in range(num_iter):
        grads, L = propagate(w, b, SetX, SetY)
        dw = grads["dw"]
        db = grads["db"]
        w = w-learning_rate*dw
        b = b-learning_rate*db
        costs.append(L)

    params = {"w":w, "b":b}
    grads = {"dw":dw, "db":db}

    return params, grads, costs

def predict(w, b, SetX):
    m = SetX.shape[1]
    predict_Y = np.zeros((1, m))
    w = w.reshape(SetX.shape[0], 1)
    A = sigmoid( np.dot(w.T, SetX) + b)
    for i in range(A.shape[1]):
        if A[0,i] <= 0.5:
            predict_Y[0,i] = 0
        else:
            predict_Y[0,i] = 1

    return predict_Y

# Creating the model:
def model(trainSetX, trainSetY, testSetX, testSetY, num_iterations
    , learning_rate):

    #Initialize paramters with 0
    w = np.zeros((trainSetX.shape[0],1))
    b = 0

    # Gradient Descent
    params, grads, costs = converge(trainSetX, trainSetY, w, b,
    num_iterations, learning_rate)

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

    #Predict test/train set examples
    predict_testSetY = predict(w,b,testSetX)
    predict_trainSetY = predict(w,b,trainSetX)

    final = {"costs": costs, "predict_testSetY": predict_testSetY,
     "predict_trainSetY": predict_trainSetY,
        "b": b, "w": w, "num_iterations": num_iterations, "
    learning_rate": learning_rate}

    #Print train/test errors
    print("Accuracy for training set is: {} %".format(100-np.mean(
    np.abs(predict_trainSetY-trainSetY))*100))
    print("Accuracy for testing set is: {} %".format(100-np.mean(
    np.abs(predict_testSetY-testSetY))*100))

    return final


# https://mashimo.wordpress.com/2017/10/19/cat-or-not-cat/
# http://dataaspirant.com/2017/04/15/implement-logistic-regression
    -model-python-binary-classification/

# To run the code:
d = model(trainSetX_new, trainSetY, testSetX_new, testSetY,
    num_iterations=20000, learning_rate=0.035)

# Plot graph
%matplotlib notebook
plt.figure()
plt.plot(np.arange(len(model_param['costs'])), model_param['costs'
    ])
plt.savefig("fooT403520000.png")
#plt.plot(d['costs'])
```

6.2 Part II

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1000.0, random_state=0, max_iter = 100)
lr.fit(trainSetX_new.T, trainSetY.T.ravel())

# Confusion Matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(trainSetY.T.ravel(), lr.predict(trainSetX_new.T))

# Prediction
Y_prediction = lr.predict(testSetX_new.T)
Y_prediction.shape

print("test accuracy: {} %".format(100 - np.mean(np.abs(
    Y_prediction - testSetY)) * 100))

confusion_matrix(testSetY.T.ravel(), lr.predict(testSetX_new.T))
```

References

[1] https://mashimo.wordpress.com/2017/10/19/cat-or-not-cat/

[2] http://dataaspirant.com/2017/04/15/implement-logistic-regression-model-python-binary-classification/

[3] https://github.com/JB1984/Logistic-Regression-Cat-Classifier/blob/master/LogisticRegression.py

[4] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html