In [1]:
```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
from dnn_app_utils_v3 import *
from public_tests import *
import pandas as pd
import seaborn as sns
import os

import tensorflow as tf

import sklearn
import mpl_toolkits



%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)


!pwd;
```

```
/Users/jhynes1/Documents/GitHub/Kaggle-Competition/titanic
```

5.0) Deep Learning Neural Networks

5.1) Prepare data:

```
* Steps 0: prepare data inputs & network model parameters,
ensure correct dimenssions
```

5.2) Write helper functions:

```
* Steps 1: parameters = def initialize parameters(layers_dim)
* Steps 2: AL, caches = L_model_forward(X, parameters):
* Steps 3: cost = compute_cost(AL, Y):
* Steps 4: grads = L_model_backward(AL, Y, caches):
* Steps 6: parameters =  update_parameters(parameters, grads,
learning_rate):
```

5.3) Build model & beta test:

* Step 7: parameters, cost = L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 1, print_cost=False)
* Step 8: test : print("Cost after first iteration: " + str(costs[0]))

5.4) Train model:

* Step 9: parameters, costs = L_layer_model(train_x, train_y, layers_dims, num_iterations = 1000, print_cost = True)

5.5) Test model:

* Step 10: parameters, costs = L_layer_model(test_x, test_y, layers_dims, num_iterations = 1000, print_cost = True

Generate prediction file:

* Step 11A: pred_target_y = L_Forward_model(test_x,paramters)
  * Step 11B: threshold
  * Step 11C: creat df and csv file. [passengerid predictions]

In [2]:

```
import os


train_df = pd.read_csv("/Users/jhynes1/Documents/GitHub/Kaggle-Competition/tita
test_df = pd.read_csv("/Users/jhynes1/Documents/GitHub/Kaggle-Competition/titan

train_df = train_df.copy()
test_df = test_df.copy()



train_df['Embarked'].fillna(0, inplace=True)  # unkown
train_df['Embarked'].replace('Q', 1,inplace=True)
train_df['Embarked'].replace('S', 2,inplace=True)
train_df['Embarked'].replace('C', 3,inplace=True)

test_df['Embarked'].fillna(0, inplace=True)
test_df['Embarked'].replace('Q', 1,inplace=True)
test_df['Embarked'].replace('S', 2,inplace=True)
test_df['Embarked'].replace('C', 3,inplace=True)

train_df['Sex'].replace('male', 0,inplace=True)
train_df['Sex'].replace('female', 1,inplace=True)

test_df['Sex'].replace('male', 0,inplace=True)
test_df['Sex'].replace('female', 1,inplace=True)

#for data in combined_data:
train_df.Fare.fillna(train_df.Fare.mean(), inplace = True)
test_df.Fare.fillna(train_df.Fare.mean(), inplace = True)
```

```
train_df.Age.fillna(method = 'ffill', inplace = True)
test_df.Age.fillna(method='ffill', inplace = True)

print(test_df.isnull().sum()) # inspect data types: any missing or null data
print(train_df.isnull().sum()) # inspect data types: any missing or null data

## TEST PREDICITVE POWER


from sklearn.feature_selection import SelectKBest, f_classif

columns_features_final = ['Sex', 'Pclass' ,'Embarked' ,  'Parch', 'SibSp', 'Age

selector = SelectKBest(f_classif, k='all')

selector.fit(train_df[columns_features_final],train_df['Survived'])
scores = -np.log10(selector.pvalues_)
indices = np.argsort(scores)[::-1]

print('Features importance:')
for i in range(len(scores)):
    print('%.2f %s' % (scores[indices[i]], columns_features_final[indices[i]]))
```

```
PassengerId      0
Pclass           0
Name             0
Sex              0
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          327
Embarked         0
dtype: int64
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         0
dtype: int64
Features importance:
68.85 Sex
24.60 Pclass
14.21 Fare
3.13 Embarked
1.83 Parch
1.30 Age
0.53 SibSp
```

In [3]:
```
## Feature Engineering - Select final features and scale

columns_features_final = ['Sex', 'Pclass' ,'Embarked' , 'SibSp','Age'  ]

train_df_features = train_df[['Survived'] + columns_features_final ]
test_df_features = test_df[columns_features_final]

# For submission scoring (i.e., don't normalize 'PassenderID' feature during su
test_df_features_Match = test_df[['PassengerId'] + columns_features_final ]
```

In [4]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler ()
scaler.fit(train_df_features)
scaled = scaler.fit_transform(train_df_features)
train_df_features_norm = pd.DataFrame(scaled, columns=train_df_features. column

scaler.fit(test_df_features)
scaled = scaler.fit_transform(test_df_features)
test_df_features_norm = pd.DataFrame(scaled, columns=test_df_features. columns
```

In [50]:
```
from sklearn.model_selection import train_test_split


print(np.shape(test_df))
 ## SWAP OUT REG DATA FOR PCA ###
#train_df_norm = pca_plot_df[['PC1', 'PC2','PC3','PC4', 'Survived']]

#columns_to_be_added_as_features = ['PC1', 'PC2',  'PC3', 'PC4']
#### SWAP OUT REG DATA FOR PCA  ###
#train_df_features_norm = pca_plot_df


#train_df_features_final = train_df_features_final.sample(frac=1).reset_index(


validation_set_ratio = 0.20 # 30%
validation_set_size = int(len(train_df_features_norm)*validation_set_ratio)
training_set_size = len(train_df_features_norm) - validation_set_size

# test vs train = 20% split
nn_train, nn_val = train_test_split(train_df_features_norm, test_size=validatic

nn_train_x = nn_train[columns_features_final]
nn_train_y = nn_train['Survived']

nn_val_x = nn_val[columns_features_final]
nn_val_y = nn_val['Survived']

print("Total set size: {}".format(len(train_df_features_norm)))
print("Training set size: {}".format(training_set_size))
print("Validation set size: {}".format(validation_set_size))
```

```
(418, 11)
Total set size: 891
Training set size: 713
Validation set size: 178
```

In [51]:
```
train_x = nn_train_x.to_numpy().T
train_y = nn_train_y.to_numpy()[np.newaxis,:]
```

```
val_x = nn_val_x.to_numpy().T
val_y = nn_val_y.to_numpy()[np.newaxis,:]

print(np.shape(train_x), np.shape(train_y))
```

```
(5, 712) (1, 712)
```

5 - L-layer Neural Network

Question: Use the helper functions you have implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]* $\times$ *(L-1) -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters_deep(layer_dims):
    ...
    return parameters
def L_model_forward(X, parameters):
    ...
    return AL, caches
def compute_cost(AL, Y):
    ...
    return cost
def L_model_backward(AL, Y, caches):
    ...
    return grads
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

5.1) write helper functions

In [52]:
```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer i

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ...,
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(1)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] =  np.random.randn(layer_dims[l], layer_dims[l
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))
```

```
    return parameters
```

Step 2: calculate forward propagation through hidden layers

This involves passing the input through a linear integration and non-linear activiation layer. Cache the weights (W) and biases for later, and pass the activiations on to the next layer.

Z, cache = linear_forward(A, W, b) A, cache = linear_activation_forward(A_prev, W, b, activation):

In [53]:
```
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous lay
    W -- weights matrix: numpy array of shape (size of current layer, size of p
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation paran
    cache -- a python dictionary containing "A", "W" and "b" ; stored for compu
    """

    Z = W.dot(A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

In [54]:
```
def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
```

```
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the back
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache
```

In [55]:

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previou
    W -- weights matrix: numpy array of shape (size of current layer, size of p
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text str

    Returns:
    A -- the output of the activation function, also called the post-activation
    cache -- a python dictionary containing "linear_cache" and "activation_cach
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    else:
        print("\033[91mError! Please make sure you have passed the value correc

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

Step 3: generate the complete forward model, inluding the output neuron.

L1: [LINEAR -> RELU] -> L2: [LINEAR -> RELU] -> L3: [LINEAR -> SIGMOID] ->

In [56]:

```
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOII

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
```

```
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, in
                the cache of linear_sigmoid_forward() (there is one, indexed L-
    """

    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)],
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], paramete
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches
```

Step 4: Calculate the mismatch (cost) between the output predictions (AL) and the target values (nn_Y)

In [57]:

```
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1,
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat),

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    # Compute loss from aL and y.
    cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we ex
    assert(cost.shape == ())

    return cost
```

Step 5: calculate the backwards propagation through the hidden layers.

In [58]:

```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (la

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current la
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation
```

```
    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape
    db -- Gradient of the cost with respect to b (current layer l), same shape
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1./m * np.dot(dZ,A_prev.T)
    db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

In [59]:

```
def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)
```

```
    return dZ
```

In [60]:
```
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for comp
    activation -- the activation to be used in this layer, stored as a text str

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape
    db -- Gradient of the cost with respect to b (current layer l), same shape
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    else:
        print("\033[91mError! Please make sure you have passed the value correc

    return dA_prev, dW, db
```

Step 6: add it to a backwards propagation model

In [61]:
```
def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forwar
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (there a
                the cache of linear_activation_forward() with "sigmoid" (there

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
```

```
    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA"
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

Step 7: use the new gradients to modify the weights and biases of the nework.

In [62]:

```
def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_bac

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network

    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate

    return parameters
```

In [63]:

```
def predict(X, y, parameters):
    """
    This function is used to predict the results of a  L-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2 # number of layers in the neural network
    p = np.zeros((1,m))

    # Forward propagation
```

```
    probas, caches = L_model_forward(X, parameters)
    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0


    accuracy = str(np.sum((p == y)/m))
    # print results
    # print ("predictions: " + str(p))
    # print ("true labels: " + str(y))
    print("Accuracy: "  + str(np.sum((p == y)/m)))

    return p, accuracy
```

step 8: Build Full Model & run beta test

Step 8: parameters, cost = L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 1, print_cost=False)

In [64]:

```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1,
    layers_dims -- list containing the input size and each layer size, of lengt
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

    Returns:
    parameters -- parameters learnt by the model. They can then be used to pred
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    parameters =initialize_parameters_deep(layers_dims)


    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        AL, caches = L_model_forward(X, parameters)

        cost = compute_cost(AL, Y)

        grads = L_model_backward(AL, Y, caches)

        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the cost every 100 iterations
```

```
        if print_cost and i % 500 == 0 or i == num_iterations - 1:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if i % 100 == 0 or i == num_iterations:
            costs.append(cost)

    return parameters, costs
```

5.3) Train Model & Generate Predictions

X input should be (b y

In [94]:
```
### Constants for 3-layer Neural Network
input_dim = np.shape(train_x)

m = input_dim[1]
n_x = input_dim[0]
n_y = 1

layers_dim = (n_x, 5,3,1)
learning_rate = 0.005

print('layers_dim = ', layers_dim)
```

```
layers_dim =  (5, 5, 3, 1)
```

In [95]:
```
parameters =  initialize_parameters_deep(layers_dim)
print(parameters)
```

```
{'W1': array([[ 0.72642933, -0.27358579, -0.23620559, -0.47984616,  0.3870220
6],
       [-1.0292794 ,  0.78030354, -0.34042208,  0.14267862, -0.11152182],
       [ 0.65387455, -0.92132293, -0.14418936, -0.17175433,  0.50703711],
       [-0.49188633, -0.07711224, -0.39259022,  0.01887856,  0.26064289],
       [-0.49221186,  0.51193601,  0.40320363,  0.2247223 ,  0.40287503]]), 'b
1': array([[0.],
       [0.],
       [0.],
       [0.],
       [0.]]), 'W2': array([[-0.30577239, -0.05495818, -0.41848881, -0.1198031
9,  0.23718218],
       [-0.30932009, -0.17743357, -0.30731297, -0.37798745, -0.3001904 ],
       [-0.00566378, -0.49967638,  0.10483389,  0.7422861 ,  0.33185224]]), 'b
2': array([[0.],
       [0.],
       [0.]]), 'W3': array([[-0.11075631, -0.51247282, -0.43137204]]), 'b3': a
rray([[0.]])}
```

In [96]:
```
# Test helper funcion: parameters = def initialize parameters(layers_dim)

parameters, costs = L_layer_model(train_x, train_y, layers_dim, num_iterations

pred_train, accuracy = predict(train_x, train_y, parameters)
```

```
Cost after iteration 0: 0.6831277289877913
Accuracy: 0.6362359550561798
```

Train Deep Neural Network Model

In [97]:
```
parameters, costs = L_layer_model(train_x, train_y, layers_dim, num_iterations
pred_test = predict(val_x, val_y, parameters)
print(np.shape(val_x),np.shape(val_y) )
```

```
Cost after iteration 0: 0.6831277289877913
Accuracy: 0.5363128491620112
(5, 179) (1, 179)
```

In [99]:
```
for n_it in range(2000, 6000, 100):

    parameters, costs = L_layer_model(train_x, train_y, layers_dim, num_iterat:
    pred_test, accuracy = predict(train_x, train_y, parameters)

    pred_test, accuracy = predict(val_x, val_y, parameters)
```

```
Cost after iteration 1999: 0.5212182391940341
Accuracy: 0.794943820224719
Accuracy: 0.7150837988826816
Cost after iteration 2099: 0.5171351642534391
Accuracy: 0.797752808988764
Accuracy: 0.7206703910614526
Cost after iteration 2199: 0.5133336290798147
Accuracy: 0.8047752808988764
Accuracy: 0.7206703910614526
Cost after iteration 2299: 0.5097884931403207
Accuracy: 0.8103932584269662
Accuracy: 0.7262569832402235
Cost after iteration 2399: 0.5064735385224884
Accuracy: 0.8132022471910112
Accuracy: 0.7262569832402235
Cost after iteration 2499: 0.5033623582906309
Accuracy: 0.8117977528089888
Accuracy: 0.7262569832402235
Cost after iteration 2599: 0.5004375132813866
Accuracy: 0.8146067415730337
Accuracy: 0.7262569832402235
Cost after iteration 2699: 0.49769093129472114
Accuracy: 0.8188202247191011
Accuracy: 0.7206703910614525
Cost after iteration 2799: 0.4951084256975978
Accuracy: 0.8202247191011236
Accuracy: 0.7318435754189945
Cost after iteration 2899: 0.4926745689350587
Accuracy: 0.8202247191011236
Accuracy: 0.7318435754189945
Cost after iteration 2999: 0.4903776031890755
Accuracy: 0.8230337078651685
Accuracy: 0.7318435754189945
Cost after iteration 3099: 0.4882026023251778
Accuracy: 0.8230337078651686
Accuracy: 0.7374301675977654
Cost after iteration 3199: 0.4861473929399566
Accuracy: 0.824438202247191
Accuracy: 0.7430167597765363
Cost after iteration 3299: 0.4842022547362159
Accuracy: 0.827247191011236
Accuracy: 0.7374301675977654
Cost after iteration 3399: 0.4823592153903401
Accuracy: 0.824438202247191
Accuracy: 0.7430167597765363
Cost after iteration 3499: 0.48061722788921846
Accuracy: 0.824438202247191
Accuracy: 0.7486033519553073
Cost after iteration 3599: 0.47896679851959095
Accuracy: 0.8230337078651685
Accuracy: 0.7597765363128492
Cost after iteration 3699: 0.4773963945620511
Accuracy: 0.8202247191011236
Accuracy: 0.7597765363128492
Cost after iteration 3799: 0.4759007213190581
Accuracy: 0.8202247191011236
Accuracy: 0.7597765363128492
Cost after iteration 3899: 0.47447698572566555
Accuracy: 0.8160112359550562
Accuracy: 0.7597765363128492
```

```
Cost after iteration 3999: 0.47311934853748794
Accuracy: 0.8160112359550562
Accuracy: 0.7597765363128492
Cost after iteration 4099: 0.47182143000126203
Accuracy: 0.8174157303370786
Accuracy: 0.7597765363128492
Cost after iteration 4199: 0.4705840444126143
Accuracy: 0.8160112359550562
Accuracy: 0.7653631284916201
Cost after iteration 4299: 0.46940593479920417
Accuracy: 0.8146067415730337
Accuracy: 0.7653631284916201
Cost after iteration 4399: 0.4682776079503983
Accuracy: 0.8146067415730337
Accuracy: 0.7653631284916201
Cost after iteration 4499: 0.46720101320566304
Accuracy: 0.8132022471910112
Accuracy: 0.7653631284916201
Cost after iteration 4599: 0.4661673157013879
Accuracy: 0.8117977528089888
Accuracy: 0.7653631284916201
Cost after iteration 4699: 0.4651756709267787
Accuracy: 0.8117977528089888
Accuracy: 0.7653631284916201
Cost after iteration 4799: 0.46422268662539307
Accuracy: 0.8117977528089888
Accuracy: 0.7653631284916201
Cost after iteration 4899: 0.4633135051754869
Accuracy: 0.8089887640449438
Accuracy: 0.7653631284916201
Cost after iteration 4999: 0.462444072175786
Accuracy: 0.8061797752808988
Accuracy: 0.7653631284916201
Cost after iteration 5099: 0.4616133014736198
Accuracy: 0.8061797752808988
Accuracy: 0.7653631284916201
Cost after iteration 5199: 0.4608048221397119
Accuracy: 0.8033707865168539
Accuracy: 0.7653631284916201
Cost after iteration 5299: 0.4600269689945815
Accuracy: 0.8019662921348314
Accuracy: 0.770949720670391
Cost after iteration 5399: 0.45928036628762636
Accuracy: 0.8019662921348314
Accuracy: 0.770949720670391
Cost after iteration 5499: 0.458561532366756
Accuracy: 0.800561797752809
Accuracy: 0.770949720670391
Cost after iteration 5599: 0.45786516384800663
Accuracy: 0.800561797752809
Accuracy: 0.7653631284916201
Cost after iteration 5699: 0.45718706440691853
Accuracy: 0.800561797752809
Accuracy: 0.7653631284916201
Cost after iteration 5799: 0.45652585649663296
Accuracy: 0.800561797752809
Accuracy: 0.7653631284916201
Cost after iteration 5899: 0.45588171964673746
Accuracy: 0.8019662921348314
Accuracy: 0.7653631284916201
```

6.0) Winning Model: Selection, Testing, Submission

Generate predictions for competition test dataset with unknown ground-truth labels.

In [100…

```
#y_target_predict = [xxxx].predict(test_df_features_norm[columns_final_features

#print('Predicted result: ', y_target_predict)
#print(len(y_target_predict))
```

In [101…

```
#submission = pd.DataFrame({'PassengerId':test_df_features_Match.PassengerId.va
#submission.Survived = submission.Survived.astype(int)

#print(submission.head()) # make sure we are submitting integers and not floats
#print(submission.shape)

#filename = 'Titanic Predictions_pub.csv'
#submission.to_csv(filename, index=False)
#print('Saved file: ' + filename)
```

In [ ]:

In [ ]: