In [27]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
```

In [28]:
```
df = pd.read_csv('Churn_Modelling.csv', delimiter=',')
df.shape
```

Out[28]:
```
(10000, 14)
```

In [29]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   RowNumber        10000 non-null  int64  
 1   CustomerId       10000 non-null  int64  
 2   Surname          10000 non-null  object 
 3   CreditScore      10000 non-null  int64  
 4   Geography        10000 non-null  object 
 5   Gender           10000 non-null  object 
 6   Age              10000 non-null  int64  
 7   Tenure           10000 non-null  int64  
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64  
 10  HasCrCard        10000 non-null  int64  
 11  IsActiveMember   10000 non-null  int64  
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64  
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

In [30]:
```
df = df.drop(["RowNumber", "CustomerId", "Surname"], axis = 1)
```

In [31]:
```
df.head(7)
```

Out[31]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
0	619	France	Female	42	2	0.00	1	1	
1	608	Spain	Female	41	1	83807.86	1	0	
2	502	France	Female	42	8	159660.80	3	1	
3	699	France	Female	39	1	0.00	2	0	
4	850	Spain	Female	43	2	125510.82	1	1	
5	645	Spain	Male	44	8	113755.78	2	1	
6	822	France	Male	50	7	0.00	2	1	

In [33]:

```
X=df.drop(["Exited"],axis=1)
X.head()
Y=df["Exited"]
```

In [34]:

```
categories = ['Geography', 'Gender']
for i in categories:
    for j in X[i].unique():
        X[i+'-'+j] = np.where(X[i] == j,1,0)
X = X.drop(categories, axis=1)
X.head()
```

Out[34]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSala
0	619	42	2	0.00	1	1	1	101348.8
1	608	41	1	83807.86	1	0	1	112542.5
2	502	42	8	159660.80	3	1	0	113931.5
3	699	39	1	0.00	2	0	0	93826.6
4	850	43	2	125510.82	1	1	1	79084.1

In [35]:

```
X['TenureByAge'] = X.Tenure/(X.Age)
X['CreditScoreGivenAge'] = X.CreditScore/(X.Age)
X['BalanceSalaryRatio'] = X.Balance/X.EstimatedSalary
X.loc[X.HasCrCard == 0, 'HasCrCard'] = -1
X.loc[X.IsActiveMember == 0, 'IsActiveMember'] = -1
X=X.drop(["CreditScore","Balance","Age","EstimatedSalary"],axis=1)
X.head()
```

Out[35]:

	Tenure	NumOfProducts	HasCrCard	IsActiveMember	Geography-France	Geography-Spain	Geography-Germany	Gen Fen
0	2	1	1	1	1	0	0	
1	1	1	-1	1	0	1	0	
2	8	3	1	-1	1	0	0	
3	1	2	-1	-1	1	0	0	
4	2	1	1	1	0	1	0	

In [59]:

```
x_cols=['Tenure', 'NumOfProducts', 'HasCrCard', 'IsActiveMember',
       'Geography-France', 'Geography-Spain', 'Geography-Germany',
       'Gender-Female', 'Gender-Male', 'TenureByAge', 'CreditScoreGivenAge',
       'BalanceSalaryRatio']

def standardize(X_tr):
    for i in range(X_tr.shape[1]):
        X_tr[:,i] = (X_tr[:,i] - np.mean(X_tr[:,i]))/np.std(X_tr[:,i])

Y=np.array(Y)
X.head()
Y=Y.reshape(-1,1)
X=X.values
standardize(X)
```

In [60]:

```
x_train,x_test,y_train,y_test = train_test_split(X, Y, train_size = .8)
```

In [72]:

```
    def initialize_parameters_deep(layer_dims):
        np.random.seed(3)
        parameters = {}
        L = len(layer_dims)

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



        return parameters

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

        return A, cache

    def relu(Z):
        A = np.maximum(0, Z)
        cache = Z

        return A, cache

    def linear_forward(A_prev, W, b):
        Z = A_prev.dot(W)
        Z = Z + b
```

```
    cache = (A_prev, W, b)

    return Z, cache


def linear_activation_forward(A_prev, W, b, activation):
    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

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

    cache = (linear_cache, activation_cache)

    return A, cache


def L_model_forward(X, parameters):
    A = X
    caches = []
    L = len(parameters) // 2

    for l in range(1, L):
        A_prev = A

        A, cache = linear_activation_forward(
            A_prev, parameters["W" + str(l)], parameters["b" + str(l)], "relu")
        caches.append(cache)

    AL, cache = linear_activation_forward(
        A, parameters["W" + str(L)], parameters["b" + str(L)], "sigmoid")
    caches.append(cache)

    return AL, caches

def compute_cost(AL, Y):
    m = Y.shape[1]
    cost = -(np.sum(Y * np.log(AL) + (1.0 - Y) * np.log(1.0 - AL))) / m
    cost = np.squeeze(cost)

    return cost

def sigmoid_backward(dA, cache):
    Z = cache
    s = 1/(1 + np.exp(-Z))
    dZ = dA * s * (1-s)

    return dZ

def relu_backward(dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    return dZ
def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[0]
```

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

    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    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)

    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches)
    Y = Y.reshape(AL.shape)

    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    current_cache = caches[L - 1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_cache
    grads["dA" + str(L-1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp

    for l in range(L-2,-1,-1):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(
            grads["dA" + str(l + 1)], current_cache, "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

def update_parameters(params, grads, learning_rate):
    parameters = params.copy()
    L = len(parameters) // 2
    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 *
        #print(grads["db" + str(l+1)].shape)

    return parameters

def L_layer_model( X, Y, layers_dims, learning_rate=0.1, num_iterations = 20000, p
    np.random.seed(1)
    costs = []

    parameters = initialize_parameters_deep(layers_dims)

    for i in range(0, num_iterations):
        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)

        if i % 3000 == 0:
            learning_rate = learning_rate / 2

        if print_cost and i % 100 == 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

#parameters, costs = L_layer_model(train_set_x, y_train, layers_dims = [3072, 5, 5

def accuracy_score(Y,Y_pred):
    tp,tn,fp,fn = 0,0,0,0
    for i in range(len(Y)):
        if Y[i] == 1 and Y_pred[i] == 1:
            tp += 1
        elif Y[i] == 1 and Y_pred[i] == 0:
            fn += 1
        elif Y[i] == 0 and Y_pred[i] == 1:
            fp += 1
        elif Y[i] == 0 and Y_pred[i] == 0:
            tn += 1
    accuracy=(tp+tn)/(tp+tn+fp+fn)
    return accuracy

def predict(X, y, parameters):
    m = X.shape[0]
    n = len(parameters) // 2
    p = np.zeros((1, m))

    probas, caches = L_model_forward(X, parameters)

    for i in range(0, probas.shape[1]):
        if probas[0, i] > 0.5:
            p[0, i] = 1
        else:
            p[0, i] = 0
    #print("Accuracy: " + str(np.sum((p == y)/m)))

    return p

#pred_train = predict(test_set_x, y_test, parameters)
#pred_test = predict(train_set_x, y_train, parameters)
```

In [73]:
```
parameters, costs = L_layer_model(x_train, y_train, layers_dims = [12, 6, 1], learning
```

```
Cost after iteration 0: 7975.021337408711
Cost after iteration 100: 3931.9729002932836
Cost after iteration 200: 3771.334623656054
Cost after iteration 300: 3694.0846020505137
Cost after iteration 400: 3646.0670536042253
Cost after iteration 500: 3615.08298745058
Cost after iteration 600: 3590.1491395107923
Cost after iteration 700: 3571.1871977729706
Cost after iteration 800: 3557.680677030494
Cost after iteration 900: 3547.096060213259
Cost after iteration 1000: 3538.0547853118082
Cost after iteration 1100: 3530.76780183529
Cost after iteration 1200: 3524.563988861911
Cost after iteration 1300: 3519.096668950635
Cost after iteration 1400: 3515.012287379108
Cost after iteration 1500: 3511.467914775605
Cost after iteration 1600: 3508.392116964762
Cost after iteration 1700: 3505.4798658462064
Cost after iteration 1800: 3502.791938971657
Cost after iteration 1900: 3500.2930957763742
Cost after iteration 2000: 3497.6300566352234
Cost after iteration 2100: 3494.984804840934
Cost after iteration 2200: 3492.2573566147075
Cost after iteration 2300: 3488.9787161510394
Cost after iteration 2400: 3485.2759129813944
Cost after iteration 2500: 3481.3132907465774
Cost after iteration 2600: 3477.320717968694
Cost after iteration 2700: 3473.4475485825924
Cost after iteration 2800: 3469.4127664466378
Cost after iteration 2900: 3465.4049976741558
Cost after iteration 3000: 3461.175752397043
Cost after iteration 3100: 3459.5228395971303
Cost after iteration 3200: 3457.9269937646723
Cost after iteration 3300: 3456.4545876910734
Cost after iteration 3400: 3455.331554312507
Cost after iteration 3500: 3454.407489631536
Cost after iteration 3600: 3453.633770897769
Cost after iteration 3700: 3452.951373985986
Cost after iteration 3800: 3452.3424496311054
Cost after iteration 3900: 3451.8048558402525
Cost after iteration 4000: 3451.3327292129575
Cost after iteration 4100: 3450.903361655349
Cost after iteration 4200: 3450.531694402024
Cost after iteration 4300: 3450.2106463729115
Cost after iteration 4400: 3449.9194794849664
Cost after iteration 4500: 3449.641880263711
Cost after iteration 4600: 3449.389504959363
Cost after iteration 4700: 3449.160039007937
Cost after iteration 4800: 3448.9467747373947
Cost after iteration 4900: 3448.7523703023107
Cost after iteration 5000: 3448.574847530693
Cost after iteration 5100: 3448.4073713907887
Cost after iteration 5200: 3448.245100822987
Cost after iteration 5300: 3448.105031909685
Cost after iteration 5400: 3447.978391492741
Cost after iteration 5500: 3447.8394310347962
Cost after iteration 5600: 3447.700486093284
Cost after iteration 5700: 3447.572047094395
Cost after iteration 5800: 3447.464326906328
Cost after iteration 5900: 3447.3619678313794
```

```
Cost after iteration 6000: 3447.262103415582
Cost after iteration 6100: 3447.2099768923663
Cost after iteration 6200: 3447.1607966929123
Cost after iteration 6300: 3447.115539953586
Cost after iteration 6400: 3447.071176794785
Cost after iteration 6500: 3447.028039832692
Cost after iteration 6600: 3446.984959009054
Cost after iteration 6700: 3446.941417854435
Cost after iteration 6800: 3446.8986933235883
Cost after iteration 6900: 3446.856983180958
Cost after iteration 7000: 3446.8139975519675
Cost after iteration 7100: 3446.7669840415424
Cost after iteration 7200: 3446.721297510779
Cost after iteration 7300: 3446.6769018654013
Cost after iteration 7400: 3446.6305970793555
Cost after iteration 7500: 3446.5820482338577
Cost after iteration 7600: 3446.533391627636
Cost after iteration 7700: 3446.4802538614076
Cost after iteration 7800: 3446.4254678787147
Cost after iteration 7900: 3446.377064594337
Cost after iteration 8000: 3446.3286037529683
Cost after iteration 8100: 3446.289374101504
Cost after iteration 8200: 3446.2536399645965
Cost after iteration 8300: 3446.2206529681343
Cost after iteration 8400: 3446.188775100454
Cost after iteration 8500: 3446.158020877323
Cost after iteration 8600: 3446.127423436435
Cost after iteration 8700: 3446.0963348493906
Cost after iteration 8800: 3446.0672694919203
Cost after iteration 8900: 3446.0381044105725
Cost after iteration 9000: 3446.007214202815
Cost after iteration 9100: 3445.9913191171586
Cost after iteration 9200: 3445.9758211823255
Cost after iteration 9300: 3445.9602821820126
Cost after iteration 9400: 3445.944048897414
Cost after iteration 9500: 3445.9275245208337
Cost after iteration 9600: 3445.9108956728023
Cost after iteration 9700: 3445.894663337752
Cost after iteration 9800: 3445.8787347754674
Cost after iteration 9900: 3445.862693603118
Cost after iteration 9999: 3445.8473377895907
```

In [ ]:

```
pred_train = predict(x_test, y_test, parameters)
print(pred_train[0][15])
score = accuracy_score(y_test, pred_train[0])
```

In [ ]: