shaikat_303527_exercise_4_lab

November 22, 2019

1 Data preprocessing

1.1 Preprocessing Airfare dataset

The data is loaded in tictactoe dataframe

```
In [1]: import pandas as pd
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        pd.options.mode.chained_assignment = None
        from sklearn.model_selection import train_test_split

        colnames=['top-left-square','top-middle-square','top-right-square','middle-left-square
        filename = r"E:\Documents\University of Hildesheim\Machine learning lab\lab4\tic-tac-to
        tictactoe = pd.read_csv(filename,delimiter=',',header=None,names=colnames)
        tictactoe.head()
```

```
Out[1]:   top-left-square top-middle-square top-right-square middle-left-square  \
        0               x                 x                x                  x
        1               x                 x                x                  x
        2               x                 x                x                  x
        3               x                 x                x                  x
        4               x                 x                x                  x

          middle-middle-square middle-right-square bottom-left-square  \
        0                    o                   o                  x
        1                    o                   o                  o
        2                    o                   o                  o
        3                    o                   o                  o
        4                    o                   o                  b

          bottom-middle-square bottom-right-square     Class
        0                    o                   o  positive
        1                    x                   o  positive
        2                    o                   x  positive
        3                    b                   b  positive
        4                    o                   b  positive
```

2 Data Analysis

2.0.1 The dataset has no missing values and all the columns with object values, these could be treated as categorical.

```
In [2]: tictactoe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 958 entries, 0 to 957
Data columns (total 10 columns):
top-left-square         958 non-null object
top-middle-square       958 non-null object
top-right-square        958 non-null object
middle-left-square      958 non-null object
middle-middle-square    958 non-null object
middle-right-square     958 non-null object
bottom-left-square      958 non-null object
bottom-middle-square    958 non-null object
bottom-right-square     958 non-null object
Class                   958 non-null object
dtypes: object(10)
memory usage: 74.9+ KB
```

```
In [3]: tictactoe = tictactoe.astype('category')
```

What can be experimented with is a simple categorical encoding, wherein each unique entry is assigned it's own number. Pandas does with relative ease by assigning desired object columns to a category dtype

```
In [4]: tictactoe.dtypes
```

```
Out[4]: top-left-square         category
        top-middle-square       category
        top-right-square        category
        middle-left-square      category
        middle-middle-square    category
        middle-right-square     category
        bottom-left-square      category
        bottom-middle-square    category
        bottom-right-square     category
        Class                   category
        dtype: object
```

```
In [5]: tictactoe=tictactoe.apply(lambda x: x.cat.codes if x.dtype.name == 'category' else x)
```

```
In [6]: tictactoe.dtypes
```

```
Out[6]: top-left-square         int8
        top-middle-square       int8
```

```
top-right-square        int8
middle-left-square      int8
middle-middle-square    int8
middle-right-square     int8
bottom-left-square      int8
bottom-middle-square    int8
bottom-right-square     int8
Class                   int8
dtype: object
```

2.0.2 It is shown that each categorical column has its unique values

In [7]:
```
tictactoe.head()
```

Out[7]:
```
   top-left-square  top-middle-square  top-right-square  middle-left-square  \
0                2                  2                 2                   2
1                2                  2                 2                   2
2                2                  2                 2                   2
3                2                  2                 2                   2
4                2                  2                 2                   2

   middle-middle-square  middle-right-square  bottom-left-square  \
0                     1                    1                   2
1                     1                    1                   1
2                     1                    1                   1
3                     1                    1                   1
4                     1                    1                   0

   bottom-middle-square  bottom-right-square  Class
0                     1                    1      1
1                     2                    1      1
2                     1                    2      1
3                     0                    0      1
4                     1                    0      1
```

2.0.3 By counting the number of positive and negative we can see that the datasets has more number of postive than negative values hence it is unbalanced

In [8]:
```
tictactoe['Class'].value_counts()
```

Out[8]:
```
1    626
0    332
Name: Class, dtype: int64
```

2.0.4 The solution to make the dataset balanced is using stratified sampling.Stratified sampling is a sampling method where the researcher divides the population into separate groups which is called strata then a probability sample is drawn from each group.

```
In [9]: def stratify_sample(df, col, samples):
            n = min(samples, df[col].value_counts().min())
            df_ = df.groupby(col).apply(lambda x: x.sample(n))
            df_.index = df_.index.droplevel(0)
            return df_
```

```
In [10]: tictactoe_stratified=stratify_sample(tictactoe, 'Class', 1000)
```

2.0.5 After stratified sampling the number of positive and negative is equal which makes the dataset balanced

```
In [11]: tictactoe_stratified['Class'].value_counts()
```

```
Out[11]: 1    332
         0    332
         Name: Class, dtype: int64
```

```
In [12]: tictactoe_stratified.head()
```

```
Out[12]:      top-left-square  top-middle-square  top-right-square  middle-left-square  \
         753                1                  2                 2                   1
         856                1                  0                 2                   1
         673                2                  1                 2                   2
         678                2                  1                 2                   2
         874                1                  0                 0                   1

              middle-middle-square  middle-right-square  bottom-left-square  \
         753                     2                    0                   1
         856                     2                    2                   1
         673                     1                    2                   1
         678                     0                    2                   1
         874                     2                    2                   1

              bottom-middle-square  bottom-right-square  Class
         753                     0                    0      0
         856                     2                    1      0
         673                     1                    0      0
         678                     1                    1      0
         874                     2                    0      0
```

```
In [13]: features=['top-left-square','top-middle-square','top-right-square','middle-left-square
         Xdata = tictactoe_stratified[features]
         Ydata = tictactoe_stratified['Class']

         x_train, x_test, y_train, y_test =train_test_split(Xdata,Ydata,train_size=0.8,test_si
```

```
         print('x_train :',x_train.shape)
         print('x_test :',x_test.shape)
         print('y_train :',y_train.shape)
         print('y_test :',y_test.shape)
```

```
x_train : (531, 9)
x_test : (133, 9)
y_train : (531,)
y_test : (133,)
```

2.0.6 Reshaping the data

```
In [16]: y_train=y_train.reshape(-1,1)
         y_test=y_train.reshape(-1,1)
```

3 Logistic Regression

3.0.1 Logistic Regression is a method in machine learning for classification problems to output discrete values

3.0.2 Step length bolddriver algorithm function

The Bold Driver Heuristic makes the assumption that smaller step sizes are needed when closer to the optimum.It adjusts the step size based on the value of f (x) at time t.If the value of f (x) grows, the step size must decrease.If the value of f (x) decreases, the step size can be larger for faster convergence.

```
In [17]: def bolddriver(x_train,y_train,beta_hat,lr,mhuplus = 1.1, mhuminus = 0.5):

             lr = lr*mhuplus
             contiter = True
             iterations = 0

             y_hat=logistic_function(x_train,beta_hat) # The current predicted value of Y
             l_old=log_likelihood(x_train, y_train, beta_hat)

             while contiter:

                 grad_beta = (np.dot(np.transpose(x_train), y_train - y_hat))
                 beta_hat = beta_hat+learn_rate*(np.dot(np.transpose(x_train), y_train - y_hat


                 y_hat=logistic_function(x_train,beta_hat) # The current predicted value of Y
                 l=log_likelihood(x_train, y_train, beta_hat)


                 if l - l_old <= 0:
```

```
                lr = lr*mhuminus
                iterations += 1
            else:
                return lr

            if iterations == 250:
                break
        return lr
```

```
In [99]: def log_likelihood(x, y, beta):
             z = np.dot(x, beta)
             log = np.sum( y*z - np.log(1 + np.exp(z)) )
             return log

         def gradient_ascent(X, h, y):
             return np.dot(X.T, y - h)

         def logistic_function(X, beta):
             z = x_train.dot(beta)
             return 1 / (1 + np.exp(-z))

         logloss = lambda y,ypred: np.mean((y*np.log(ypred)+(1-y)*np.log(1-ypred)))
         cost = lambda y,ypred: np.mean((y - ypred)**2)
```

4 logistic regression using stochastic gradient decent

```
In [100]: m_train,n_features = x_train.shape

          num_iter      = 1000
          learn_rate    = 0.00001

          beta_hat      = np.random.random(n_features).reshape(-1,1)
          relative_l= []
          relative_loss=[]
          loglosstest=[]

          y_hat=logistic_function(x_train,beta_hat)
          l=0
          l_old=log_likelihood(x_train, y_train, beta_hat)

          chunk_size = 20

          for i in range(num_iter):

              loss_old  = cost(y_train,y_hat)

              for chunk in range(len(x_train)//chunk_size):
```

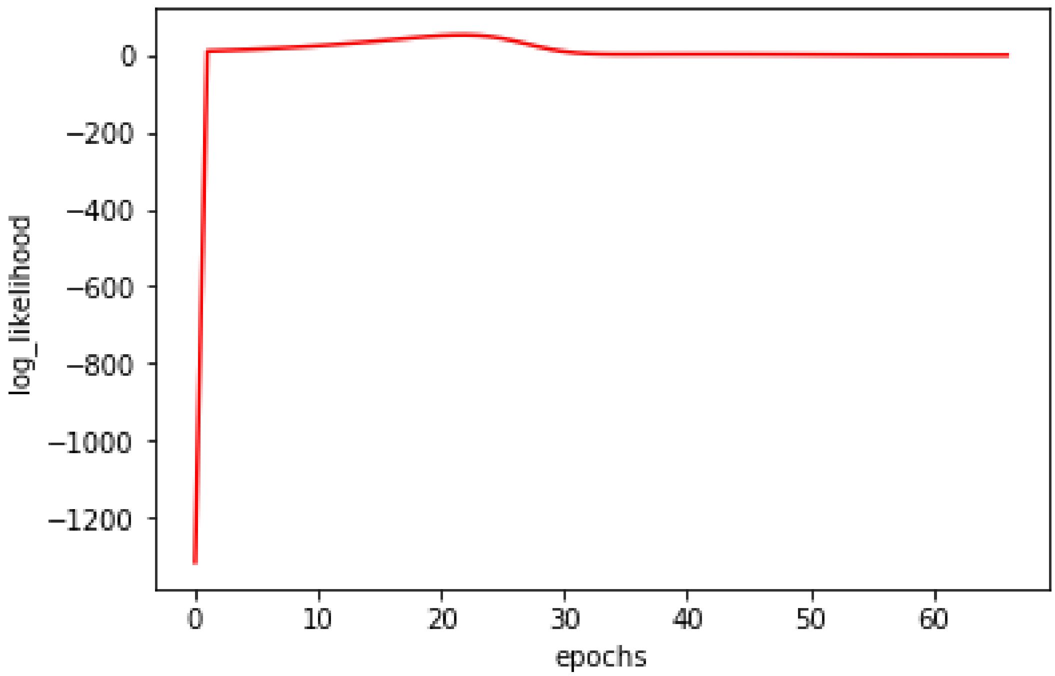
```
            x_chunk = x_train[chunk*chunk_size:min((chunk+1)*chunk_size,len(x_train))]
            y_chunk = y_train[chunk*chunk_size:min((chunk+1)*chunk_size,len(y_train))]

            y_hat=logistic_function(x_chunk,beta_hat)

            beta_hat = beta_hat+learn_rate*(np.dot(np.transpose(x_chunk), y_chunk - y_ha



        learn_rate =  bolddriver(x_train,y_train,beta_hat,learn_rate,mhuplus = 1.1, mhum
        y_hat=logistic_function(x_train,beta_hat)
        loss_new  = cost(y_train,y_hat)


        l_old=l
        l=log_likelihood(x_train, y_train, beta_hat)

        relative_l.append(l-l_old)
        relative_loss.append(np.abs(loss_new.values-loss_old.values))
        loglosstest.append(logloss(y_test,y_hat))

        if i % 5 == 0:
            print(f"epochs: {i} log_likelihood: {(l-l_old)} learning rate: {learn_rate} l

        last_epoch=i+1

        if np.abs(l-l_old) == 0:
            break
```

```
epochs: 0 log_likelihood: -1572.7702094809206 learning rate: 1.1000000000000001e-05 loss:[0.000
epochs: 5 log_likelihood: 15.866499147753984 learning rate: 1.771561000000001e-05 loss:[0.0003
epochs: 10 log_likelihood: 25.29320499976211 learning rate: 2.8531167061100026e-05 loss:[0.000
epochs: 15 log_likelihood: 39.713027286245506 learning rate: 4.594972986357222e-05 loss:[0.001
epochs: 20 log_likelihood: 58.959901423156566 learning rate: 7.400249944258173e-05 loss:[0.006
epochs: 25 log_likelihood: 66.30669580749566 learning rate: 0.00011918176537727237 loss:[0.019
epochs: 30 log_likelihood: 21.28956081438355 learning rate: 0.00019194342495775 1 loss:[0.0134
epochs: 35 log_likelihood: 3.1655026096966594 learning rate: 0.0003091268053287077 loss:[0.002
epochs: 40 log_likelihood: 3.350502702297149 learning rate: 0.0004978518112499372 loss:[0.0028
epochs: 45 log_likelihood: 3.1396682836345917 learning rate: 0.0008017953205361369 loss:[0.002
epochs: 50 log_likelihood: 2.024206539537886 learning rate: 0.001291299381676654 1 loss:[0.001
epochs: 55 log_likelihood: 0.7422588835455599 learning rate: 0.002079650567184069 loss:[0.0006
epochs: 60 log_likelihood: 0.11657514598670105 learning rate: 0.003349298034955616 loss:[7.015
epochs: 65 log_likelihood: 0.0 learning rate: 2.981386663661941e-78 loss:[0.] Logloss : [-0.65
```

```
In [82]: plt.xlabel("epochs")
```

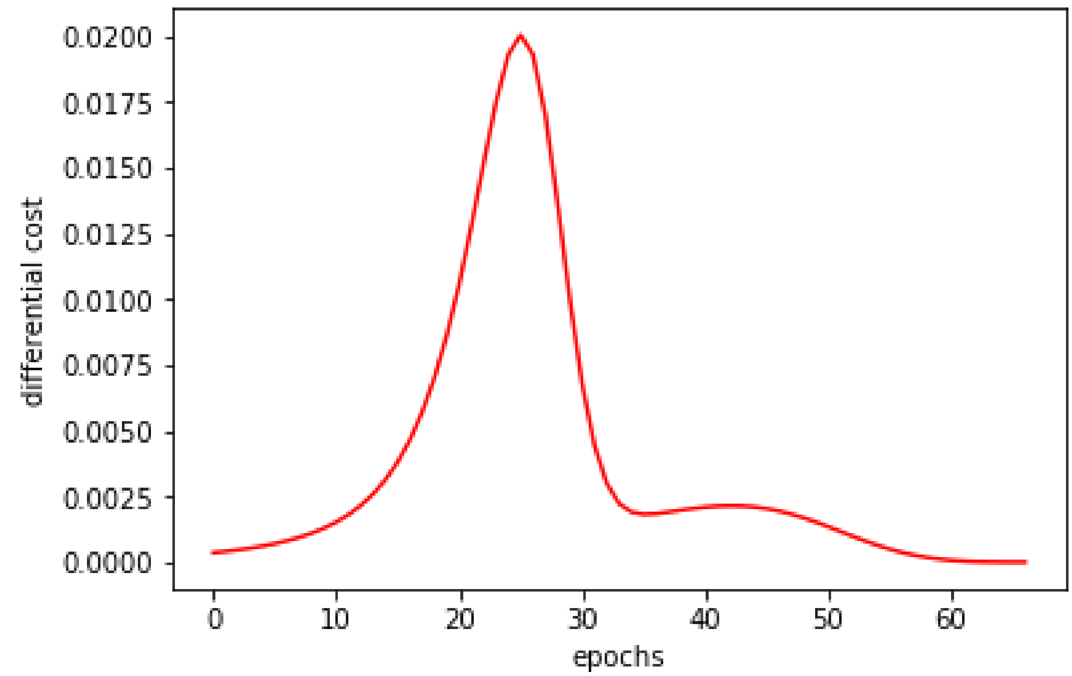
```
plt.ylabel("log_likelihood")
plt.plot( np.arange(last_epoch),relative_l,'r')
```

Out[82]: [<matplotlib.lines.Line2D at 0x1a1ddbc5be0>]



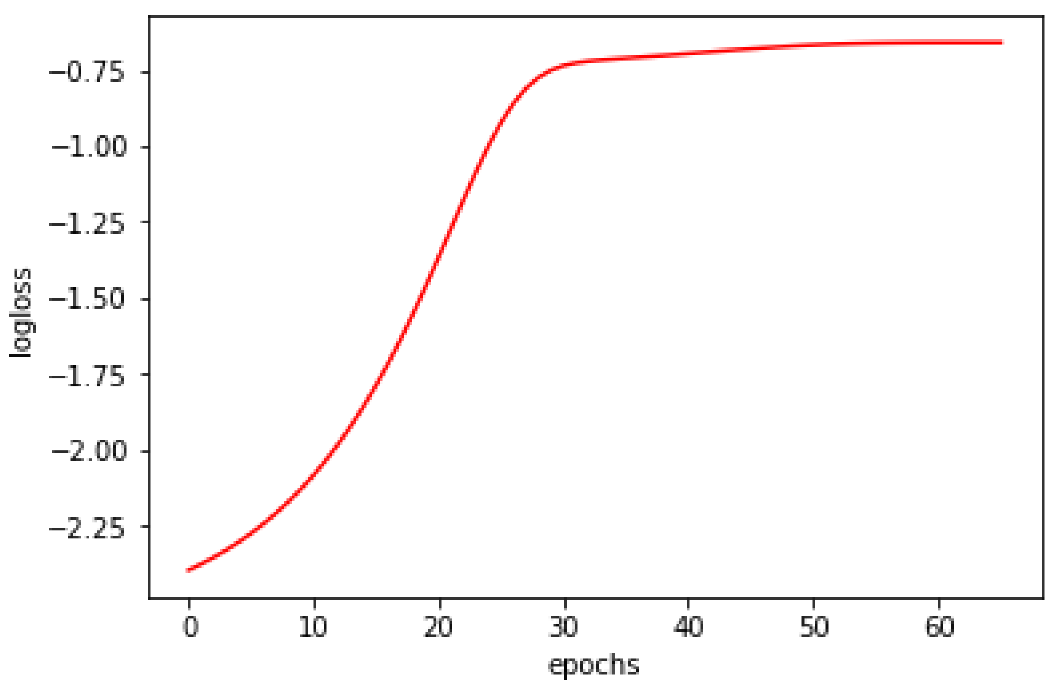
In [83]:
```
plt.xlabel("epochs")
plt.ylabel("differential cost")
plt.plot( np.arange(last_epoch),relative_loss,'r')
```

Out[83]: [<matplotlib.lines.Line2D at 0x1a1ddb203c8>]




```
In [78]: plt.xlabel("epochs")
         plt.ylabel("logloss")
         plt.plot( np.arange(last_epoch),losslosstest,'r')
```

Out[78]: [<matplotlib.lines.Line2D at 0x1a1ddacb160>]



5 Implement Newton Algorithm (learning rate)

Newton's method is a second-order optimization algorithm that can help us find the best weights in our logistic function in fewer iterations compared to batch gradient descent.

The generalization of Newton's method to a multidimensional setting (also called the Newton-Raphson method) is given by:

Where the Hessian is represented by:

For Logistic Regression, the Hessian is given by:

$$Hf(\beta) = -X^T W X$$

and the gradient is:

$$\nabla f(\beta) = X^T(y - p)$$

where

$$W := \text{diag}\left(p(1-p)\right)$$

and p are the predicted probabilites computed at the current value of β.

```
In [93]: def sigmoid(x):
             return 1/(1+np.exp(-x))
```

```
In [94]: def newton(beta0, y, X, lr):

             p = np.array(sigmoid(X.dot(beta0[:,0]))).T
             W = np.diag((p*(1-p)))


             hess = np.dot((np.dot(X.T,W)),X)
             grad = (np.transpose(X)).dot(y-p)


             s =lr*(np.dot(np.linalg.inv(hess), grad))
             beta = beta0 + s

             return beta
```

5.0.1 Using the Newton method we noticed that the convergence is faster because it needed only 5 iteration to converge.

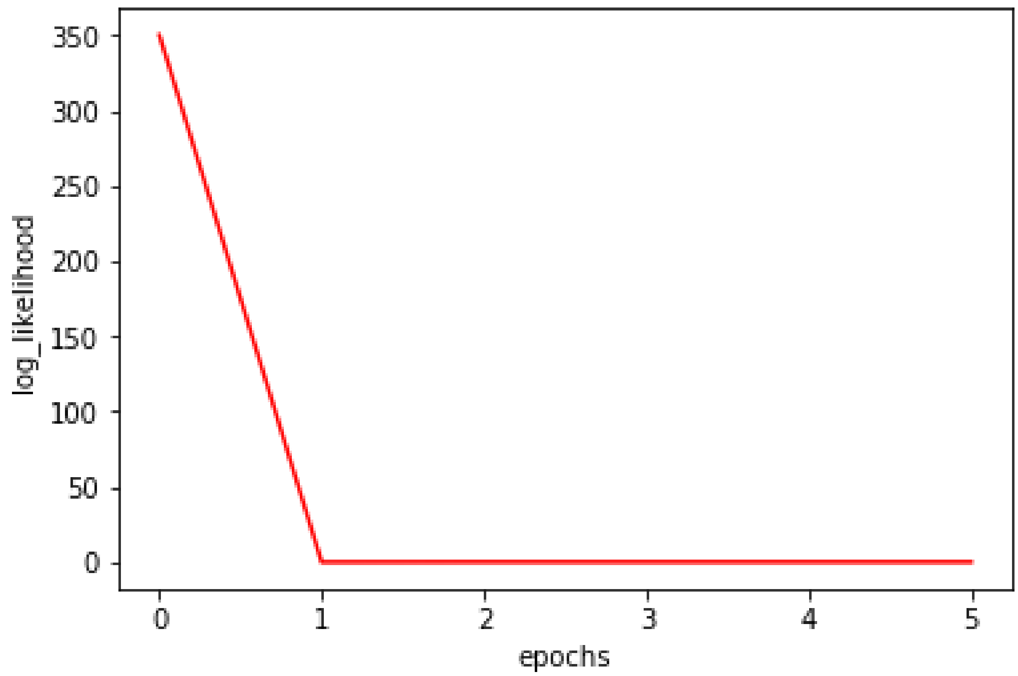
```
In [119]: num_iter = 100

          lr =0.001

          relative_l_newton=[]
```

```
relative_loss_newton=[]
loglosstest_newton=[]

beta_old, beta = np.ones((n_features,1)), np.zeros((n_features,1))
l_newton=0
y_hat_newton=logistic_function(x_train,beta_old)
l_old_newton=-log_likelihood(x_train, y_train, beta_old)

for i in range(num_iter):

    loss_old_newton  = cost(y_train,y_hat_newton)
    beta_old = beta

    beta = newton_step(beta, y_train, x_train, reg_term)
    y_hat_newton=logistic_function(x_train,beta)

    loss_new_newton  = cost(y_train,y_hat_newton)

    l_old_newton=l_newton
    l_newton=-log_likelihood(x_train, y_train, beta)
    relative_l_newton.append(l_newton-l_old_newton)
    relative_loss_newton.append(np.abs(loss_new_newton.values-loss_old_newton.values)


    loglosstest_newton.append(-logloss(y_test,y_hat_newton))

    if i % 1 == 0:
        print(f"epochs: {i} log_likelihood: {(l_newton-l_old_newton)} loss:{np.abs(lo



    if np.abs(l_newton-l_old_newton) == 0:
        break
```

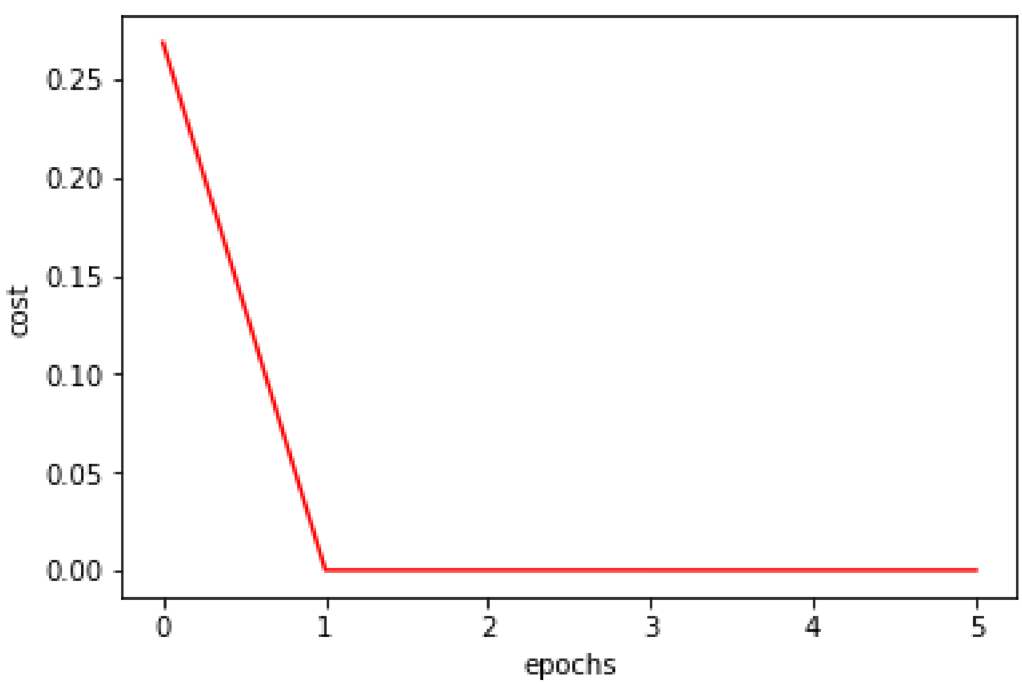
```
epochs: 0 log_likelihood: 350.07633312488116 loss:[0.26892146]
epochs: 1 log_likelihood: -0.08185733152396324 loss:[3.09444334e-06]
epochs: 2 log_likelihood: -4.909220149329485e-05 loss:[1.74056729e-06]
epochs: 3 log_likelihood: -9.544464774080552e-09 loss:[2.29948263e-08]
epochs: 4 log_likelihood: -1.8189894035458565e-12 loss:[2.91486307e-10]
epochs: 5 log_likelihood: 0.0 loss:[3.71214171e-12]
```

In [120]:
```
plt.xlabel("epochs")
plt.ylabel("log_likelihood")
plt.plot( np.arange(len(relative_l_newton)),relative_l_newton,'r')
```

Out[120]:
```
[<matplotlib.lines.Line2D at 0x1a1deea8160>]
```




```
In [121]: plt.xlabel("epochs")
          plt.ylabel("cost")
          plt.plot( np.arange(len(relative_loss_newton)),relative_loss_newton,'r')

Out[121]: [<matplotlib.lines.Line2D at 0x1a1def06198>]
```



```
In [122]: plt.xlabel("epochs")
          plt.ylabel("logloss")
          plt.plot( np.arange(len(loglosstest_newton)),loglosstest_newton,'r')
```

```
Out[122]: [<matplotlib.lines.Line2D at 0x1a1def5a780>]
```