**LAB - 4**

**MACHINE LEARNING**

**PRIYANSHU SHARMA**

**15BCE1282**

**SINGLE LAYER PERCEPTRON**

**CODE**

#SINGLE LAYER PERCEPTRON

import pandas as pd

import sklearn

import matplotlib.pyplot as plt

import numpy as np

from scipy.special import expit

import sys

class NeuralNetMLP(object):

def \_\_init\_\_(self, n\_output, n\_features, n\_hidden=30,

l1=0.0, l2=0.0, epochs=500, eta=0.001,

alpha=0.0, decrease\_const=0.0, shuffle=True,

minibatches=1, random\_state=None):

np.random.seed(random\_state)

self.n\_output = n\_output

self.n\_features = n\_features

self.n\_hidden = n\_hidden

self.w1, self.w2 = self.\_initialize\_weights()

self.l1 = l1

self.l2 = l2

self.epochs = epochs

self.eta = eta

self.alpha = alpha

self.decrease\_const = decrease\_const

self.shuffle = shuffle

self.minibatches = minibatches

def \_encode\_labels(self, y, k):

onehot = np.zeros((k, y.shape[0]))

for idx, val in enumerate(y):

onehot[val, idx] = 1.0

return onehot

def \_initialize\_weights(self):

w1 = np.random.uniform(-1.0, 1.0,

size=self.n\_hidden\*(self.n\_features + 1))

w1 = w1.reshape(self.n\_hidden, self.n\_features + 1)

w2 = np.random.uniform(-1.0, 1.0,

size=self.n\_output\*(self.n\_hidden + 1))

w2 = w2.reshape(self.n\_output, self.n\_hidden + 1)

return w1, w2

def \_sigmoid(self, z):

# expit is equivalent to 1.0/(1.0 + np.exp(-z))

return expit(z)

def \_sigmoid\_gradient(self, z):

sg = self.\_sigmoid(z)

return sg \* (1 - sg)

def \_add\_bias\_unit(self, X, how='column'):

if how == 'column':

X\_new = np.ones((X.shape[0], X.shape[1]+1))

X\_new[:, 1:] = X

elif how == 'row':

X\_new = np.ones((X.shape[0]+1, X.shape[1]))

X\_new[1:, :] = X

else:

raise AttributeError('`how` must be `column` or `row`')

return X\_new

def \_feedforward(self, X, w1, w2):

a1 = self.\_add\_bias\_unit(X, how='column')

z2 = w1.dot(a1.T)

a2 = self.\_sigmoid(z2)

a2 = self.\_add\_bias\_unit(a2, how='row')

z3 = w2.dot(a2)

a3 = self.\_sigmoid(z3)

return a1, z2, a2, z3, a3

def \_L2\_reg(self, lambda\_, w1, w2):

return (lambda\_/2.0) \* (np.sum(w1[:, 1:] \*\* 2) + np.sum(w2[:, 1:] \*\* 2))

def \_L1\_reg(self, lambda\_, w1, w2):

return (lambda\_/2.0) \* (np.abs(w1[:, 1:]).sum() + np.abs(w2[:, 1:]).sum())

def \_get\_cost(self, y\_enc, output, w1, w2):

term1 = -y\_enc \* (np.log(output))

term2 = (1 - y\_enc) \* np.log(1 - output)

cost = np.sum(term1 - term2)

L1\_term = self.\_L1\_reg(self.l1, w1, w2)

L2\_term = self.\_L2\_reg(self.l2, w1, w2)

cost = cost + L1\_term + L2\_term

return cost

def \_get\_gradient(self, a1, a2, a3, z2, y\_enc, w1, w2):

# backpropagation

sigma3 = a3 - y\_enc

z2 = self.\_add\_bias\_unit(z2, how='row')

sigma2 = w2.T.dot(sigma3) \* self.\_sigmoid\_gradient(z2)

sigma2 = sigma2[1:, :]

grad1 = sigma2.dot(a1)

grad2 = sigma3.dot(a2.T)

# regularize

grad1[:, 1:] += (w1[:, 1:] \* (self.l1 + self.l2))

grad2[:, 1:] += (w2[:, 1:] \* (self.l1 + self.l2))

return grad1, grad2

def predict(self, X):

a1, z2, a2, z3, a3 = self.\_feedforward(X, self.w1, self.w2)

y\_pred = np.argmax(z3, axis=0)

return y\_pred

def fit(self, X, y, print\_progress=False):

self.cost\_ = []

X\_data, y\_data = X.copy(), y.copy()

y\_enc = self.\_encode\_labels(y, self.n\_output)

delta\_w1\_prev = np.zeros(self.w1.shape)

delta\_w2\_prev = np.zeros(self.w2.shape)

for i in range(self.epochs):

# adaptive learning rate

self.eta /= (1 + self.decrease\_const\*i)

if print\_progress:

sys.stderr.write(

'\rEpoch: %d/%d' % (i+1, self.epochs))

sys.stderr.flush()

if self.shuffle:

idx = np.random.permutation(y\_data.shape[0])

X\_data, y\_data = X\_data[idx], y\_data[idx]

mini = np.array\_split(range(

y\_data.shape[0]), self.minibatches)

for idx in mini:

# feedforward

a1, z2, a2, z3, a3 = self.\_feedforward(

X[idx], self.w1, self.w2)

cost = self.\_get\_cost(y\_enc=y\_enc[:, idx],

output=a3,

w1=self.w1,

w2=self.w2)

self.cost\_.append(cost)

# compute gradient via backpropagation

grad1, grad2 = self.\_get\_gradient(a1=a1, a2=a2,

a3=a3, z2=z2,

y\_enc=y\_enc[:, idx],

w1=self.w1,

w2=self.w2)

# update weights

delta\_w1, delta\_w2 = self.eta \* grad1, self.eta \* grad2

self.w1 -= (delta\_w1 + (self.alpha \* delta\_w1\_prev))

self.w2 -= (delta\_w2 + (self.alpha \* delta\_w2\_prev))

delta\_w1\_prev, delta\_w2\_prev = delta\_w1, delta\_w2

return self

data = pd.read\_csv('C:/Users/PRIYANSHU SHARMA/Desktop/PRIYANSHU/6 STUDY/6 SEMSTER/MACHINE LEARNING/LAB/breast.csv')

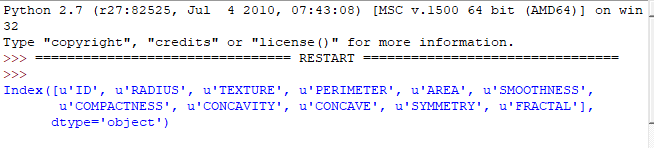
data.head()

colnames=['ID', 'RADIUS', 'TEXTURE', 'PERIMETER', 'AREA', 'SMOOTHNESS', 'COMPACTNESS', 'CONCAVITY', 'CONCAVE', 'SYMMETRY', 'FRACTAL']

data = pd.read\_csv('C:/Users/PRIYANSHU SHARMA/Desktop/PRIYANSHU/6 STUDY/6 SEMSTER/MACHINE LEARNING/LAB/breast.csv', names=colnames, header=None)

data.head()

print(data.columns)



data.describe()

from sklearn.cross\_validation import train\_test\_split

X = data.iloc[0:, [1,2,3,4,5,6,7]].values

X\_train,X\_test,Y\_train,Y\_test = train\_test\_split(X,data['FRACTAL'], test\_size=0.3, random\_state=0)

nn = NeuralNetMLP(n\_output=10,

n\_features=X\_train.shape[1],

n\_hidden=25,

l2=0.1,

l1=0.0,

epochs=1000,

eta=0.001,

alpha=0.001,

decrease\_const=0.00001,

shuffle=True,

minibatches=50,

random\_state=1)

nn.fit(X\_train, Y\_train, print\_progress=True)

plt.plot(range(len(nn.cost\_)), nn.cost\_)

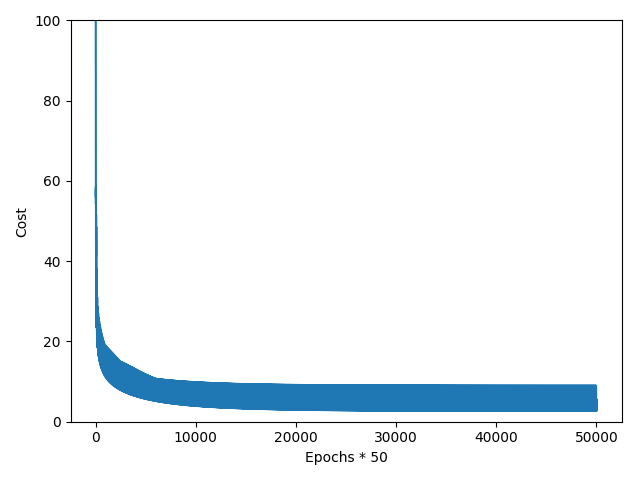
plt.ylim([0, 100])

plt.ylabel('Cost')

plt.xlabel('Epochs \* 50')

plt.tight\_layout()

plt.show()



batches = np.array\_split(range(len(nn.cost\_)), 1000)

cost\_ary = np.array(nn.cost\_)

cost\_avgs = [np.mean(cost\_ary[i]) for i in batches]

plt.plot(range(len(cost\_avgs)),

cost\_avgs,

color='red')

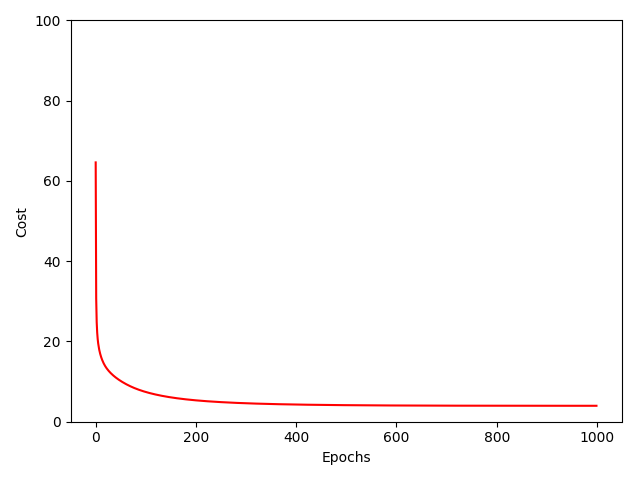
plt.ylim([0, 100])

plt.ylabel('Cost')

plt.xlabel('Epochs')

plt.tight\_layout()

plt.show()



Y\_train\_pred = nn.predict(X\_train)

acc = np.sum(Y\_train == Y\_train\_pred, axis=0) / X\_train.shape[0]

print('Training accuracy: %.2f%%' % (acc \* 100))

Y\_test\_pred = nn.predict(X\_test)

acc = np.sum(Y\_test == Y\_test\_pred, axis=0) / X\_test.shape[0]

print('Testing accuracy: %.2f%%' % (acc \* 100))

