Neural Networks Report

1. Multilayer Perceptron

```
import numpy as np
import pandas as pd
from scipy.io import loadmat
from preprocessing import NormalScaler

class MLP:

    def __init__(self, layers_arr, activ_arr):
        '''
        This is the constructor for the class MLP.
        The following attributes are initialized in this constructor
        W_list: a list of weight matrices. Each weight matrix
                connects one layer to the next one.
        b_list: a list of bias weights. Each layer has a bias weight.
        grad_W: list of gradient matrices. Each matrix has gradients
                of the J wrt weights in the corresponding weight matrix in W_list.
        b_grad: list of gradients of J wrt biases.
        A: a list of vectors. Each vector represents a layer of neurons and their values.
        derr: a list of vectors containing errors generated using backpropogation.
        L: number of layers excluding the input layer
        activ_arr: contains activation function to be used for a particular layer

        '''
        self.W_list = []
        self.b_list = []
        self.grad_W = []
        self.grad_b = []
        self.A = []
        self.derr = []
        self.L = len(layers_arr)-1
        self.activ_arr = activ_arr
        self.n = layers_arr[0]
        for l in range(self.L):
            self.b_list.append(np.random.randn(1,).astype(np.float64)[0])
            self.grad_b.append(np.random.randn(1,).astype(np.float64)[0])
            self.W_list.append(np.random.randn( layers_arr[l+1], layers_arr[l]).astype(np.float64))
            self.grad_W.append(np.random.randn( layers_arr[l+1], layers_arr[l]).astype(np.float64))
        for l in range(self.L+1):
            self.A.append(np.ndarray(shape=( layers_arr[l], 1 )).astype(np.float64))
            self.derr.append(np.ndarray(shape=( layers_arr[l], 1 )).astype(np.float64))

        # vectorized versions of activation functions
        self.relu_v = np.vectorize(self.relu)
        self.sigmoid_v = np.vectorize(self.sigmoid)
```

```python
def compute_Z(self, l):
    '''
    This function computes the output of a layer of neurons before activation.
    '''
    return np.matmul(self.W_list[l-1], self.A[l-1]) + self.b_list[l-1]

def activation(self, Z, activ, deriv = 0):
    '''
    This function returns the activated output of a layer of neurons.
    '''
    if(activ=='sigmoid'):
        return self.sigmoid_v(Z, deriv)
    elif(activ=='relu'):
        return self.relu_v(Z, deriv)

def relu(self, x, deriv = 0):
     '''
    This function returns the relu activated value.
                    f(x) = x^+ = max(0,x)
                    f'(x) = 1 if x ≥ 0 else 0
    '''
    if deriv==1:
        return 0 if x<0 else 1
    return 0 if x<0 else x

def sigmoid(self, x, deriv = 0):
     '''
    This function returns the sigmoid activated value.
                    f(x) = 1/(1+e^-x)
                    f'(x) = f(x)(1 - f(x))
   '''
    if deriv==1:
        return self.sigmoid(x)*(1-self.sigmoid(x))
    return 1/(1+np.exp(-x))

def forward_prop(self, X_i):
    '''
    This function takes ith data vector and propogates
    it forward in the neural network.
                    a^(l) = σ(z^(l))
   '''
    self.A[0] = X_i.reshape(-1,1)
    for l in range(1,self.L+1):
        self.A[l] = self.activation(self.compute_Z(l), self.activ_arr[l-1])

def train(self, X, y, alpha, batch_size, max_iter):
    '''
    This function takes the training data and target values,
    performs forward propogation, then applies backward propogation
    algorithm to update the weight matrices.
    mini-batch gradient descent has been used to update weights.
    '''
    m = y.shape[0]
    for iteration in range(max_iter):
```

```
        for i in range(0,m-batch_size+1,batch_size):
            for l in range(self.L): self.grad_b[l]=0
            for l in range(self.L): self.grad_W[l].fill(0)

            for j in range(i,i+batch_size):
                # forward propogation
                self.forward_prop(X[j])

                # Backpropogation of errors
                self.derr[self.L] = (self.A[self.L]-y[j].reshape(
1,1)) * self.activation(self.compute_Z(self.L), self.activ_arr[self.L-1], 1)
                for l in range(self.L-1, 0,-1):
                    self.derr[l] = self.activation(self.compute_Z(l), self.activ_arr[l-
1], 1)*np.matmul(self.W_list[l].T, self.derr[l+1])

                for l in range(self.L, 0,-1):
                    self.grad_b[l-1] += np.mean(self.derr[l])
                    self.grad_W[l-1] += np.matmul(self.derr[l], self.A[l-1].T)

            # weight update after backpropogating each batch
            for l in range(self.L, 0,-1):
                self.b_list[l-1] -= (alpha/batch_size)*self.grad_b[l-1]
                self.W_list[l-1] -= (alpha/batch_size)*self.grad_W[l-1]

        print("iteration: {0} ".format(iteration),end="  ")
        print(" ",self.eval_cost(X,y),end=" ")
        print("  ",self.accuracy(X,y)," ")

    def eval_cost(self, X, y):
        '''
        This function computes the total cost and returns it.
        '''
        cost = 0

        for i in range(y.shape[0]):
            # forward propogation
            self.forward_prop(X[i])
            cost += np.sum((self.A[self.L]-y[i].reshape(-1,1))**2)
        return cost/(2*X.shape[0])

    def accuracy(self, X, y):
        '''
        This function finds the accuracy predictions on given data
        '''
        acc = 0
        for i in range(y.shape[0]):
            # forward propogation
            self.forward_prop(X[i])
            t1 = 0 if self.A[self.L][0][0]<0.5 else 1
            t2 = 0 if self.A[self.L][1][0]<0.5 else 1

            acc += ((t1==y[i][0]) and (t2==y[i][1]))
        return acc/y.shape[0]
```

```python
    def conf_mat(self, X, y):
        '''
        This function constructs a confusion matrix and returns it
        '''
        conf_mat = np.zeros((y.shape[1],y.shape[1]))
        y_p = self.predict(X)
        for i in range(y.shape[0]):
            # forward propogation
            conf_mat[int(np.argmax(y[i]))][int(y_p[i])] += 1
        return conf_mat

    def predict(self, X):
        y_pred = np.ndarray(X.shape[0])
        for i in range(X.shape[0]):
            # forward propogation
            self.forward_prop(X[i])
            y_pred[i] = np.argmax(self.A[self.L])
        return y_pred


def start_run():
    # m = number of feature vectors
    m = X_train.shape[0]
    # n = number of features
    n = X_train.shape[1]

    alpha = 0.5
    max_iter = 25

    # number of neurons in each layer
    Layers = [n,16,8,2]
    activations = ['sigmoid','sigmoid','sigmoid']
    batch_size = 32

    model = MLP(Layers, activations)
    model.train(X_train, y_train, alpha, batch_size, max_iter)

    # print("Test Accuracy", model.accuracy(X_test,y_test))
    conf = model.conf_mat(X_test,y_test)
    print("Confusion matrix", conf)
    print("Test accuracy: ", model.accuracy(X_test,y_test))

    # Results Visualization
    plt.figure()
    plt.title(f'Cost Function vs iteration plot {Layers}\n alpha={alpha} max_iter={max_iter} batch_size={batch_size}')
    plt.xlabel("iteration")
    plt.ylabel("cost")
    plt.plot(model.cost_arr['train'],c='c',label='training set avg cost')
    plt.plot(model.cost_arr['test'], c='r',label='testing set avg cost')
    plt.legend(loc='upper right')
    plt.savefig(f"./Results/mlp/{alpha}_{max_iter}_{batch_size}_{Layers[1:3]}_cost_iter.png")
    plt.show()

    plt.figure()
```

```python
    plt.title(f"Accuracy vs iteration plot {Layers} \n alpha={alpha} max_iter={max_iter} batch_size={batch_size}")
    plt.xlabel("iteration")
    plt.ylabel("accuracy")
    plt.plot(model.acc_arr['train'],c='c',label='training set accuracy')
    plt.plot(model.acc_arr['test'], c='r',label='testing set accuracy')
    plt.legend(loc='upper left')
    plt.savefig(f"./Results/mlp/{alpha}_{max_iter}_{batch_size}_{Layers[1:3]}_acc_iter.png")
    plt.show()

    return model

if __name__=='__main__':

    # data input
    data = pd.DataFrame(loadmat('./data5.mat')['x'])
    data = data.sample(frac=1).reset_index(drop=True)
    X = data.loc[:,:71].values
    y = data.loc[:,72:73].values
    y_cat = np.zeros((y.shape[0],2)).astype(np.int)
    for i in range(y.shape[0]):
        y_cat[i][int(y[i])] = 1

    # data preprocessing
    scaler = NormalScaler()
    for j in range(X.shape[1]):
        scaler.fit(X[:,j])
        X[:,j] = scaler.transform(X[:,j])

    # give 'holdout' for hold-out cross validation split
    # or 'kfold' for k-fold cross validation split.
    split = 'holdout'
    if split=='holdout':
        train_percent = 0.6
        X_train = X[:int(train_percent*X.shape[0])]
        y_train = y_cat[:int(train_percent*X.shape[0])]
        X_test = X[int(train_percent*X.shape[0]):]
        y_test = y_cat[int(train_percent*X.shape[0]):]
        start_run()
    elif split=='kfold':
        k_fold = 4
        Nk = X.shape[0]//k_fold
        models = []
        acc = []
        for i in range(0, X.shape[0], Nk):
            X_test = X[i:i+Nk,:]
            X_train = np.delete(X,range(i,i+Nk),0)
            y_test = y_cat[i:i+Nk]
            y_train = np.delete(y_cat,range(i,i+Nk),0)
            models.append(start_run())
            acc.append(models[-1].accuracy(X_test,y_test))

        print("Average Accuracy: ", np.mean(acc))
```

Results:

Hold-out cross validation 60-40 split:

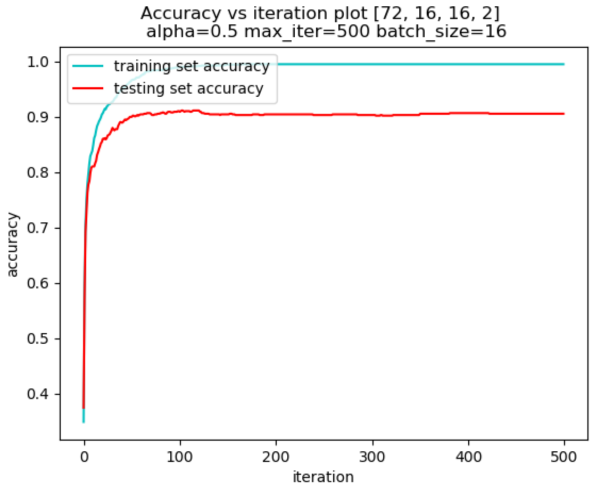
alpha = 0.5   max_iter = 500   batch_size = 12   layers = [72, 16, 16, 2]

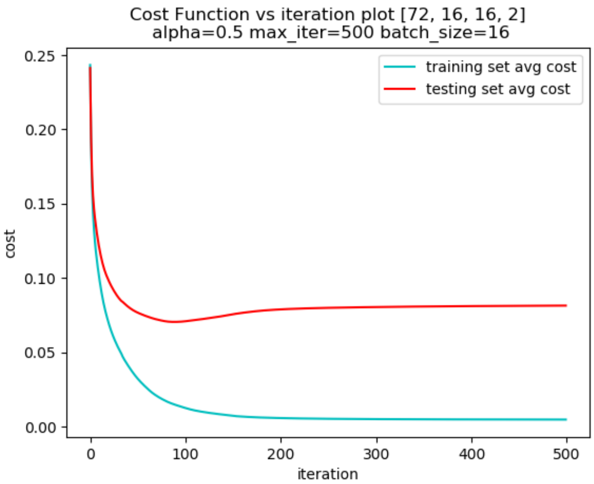
[[265. 21.]

[22. 229.]]

test_accuracy = 0.9199255121042831

time taken = 404.65393233299255





K-fold cross validation split (k=5):

Split-1:

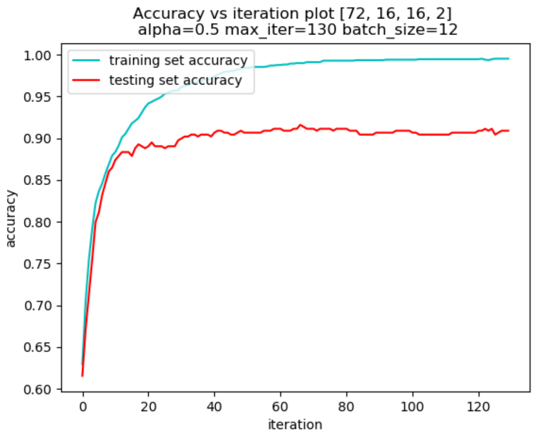
alpha = 0.5   max_iter = 130   batch_size = 12   layers = [72, 16, 16, 2]

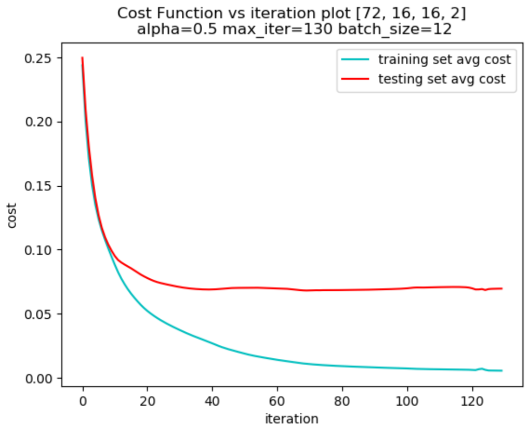
confusion matrix

[[192. 22.]

[13. 202.]]

test_accuracy = 0.9184149184149184





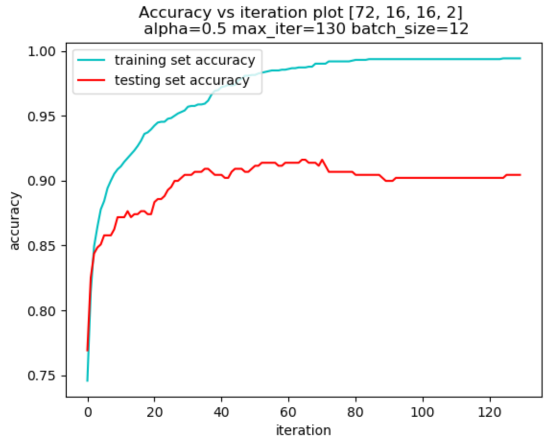
Split-2:

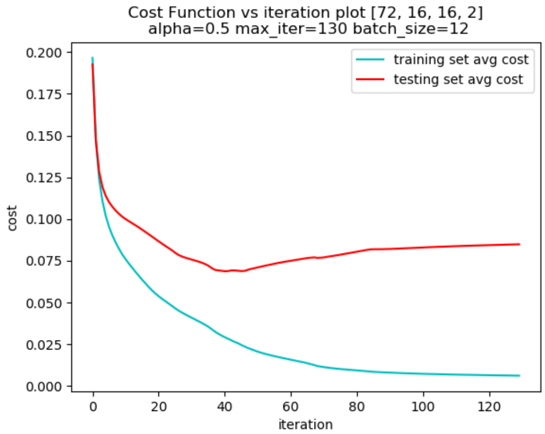
confusion matrix:

[[247. 28.]

[30. 232.]]

test_accuracy = 0.8919925512104283





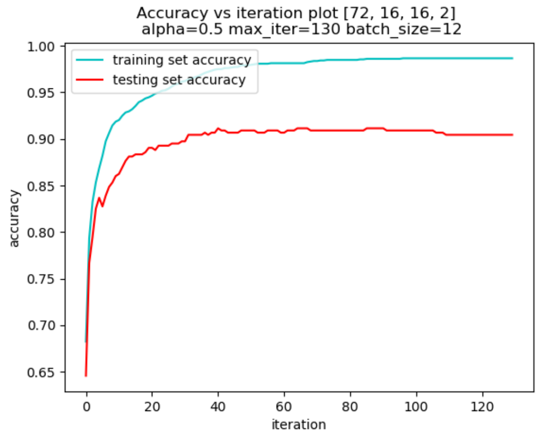
Split-3:

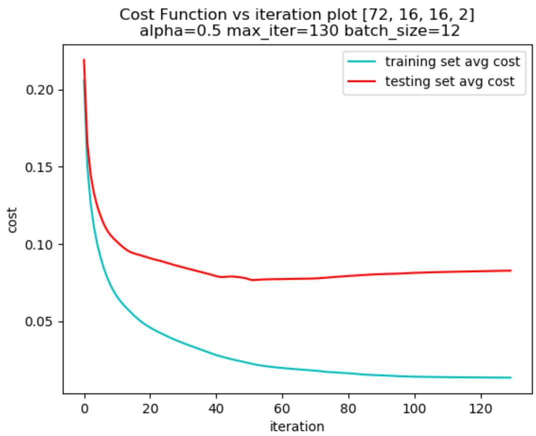
confusion matrix:

[[238. 32.]

 [29. 238.]]

 test_accuracy = 0.8864059590316573





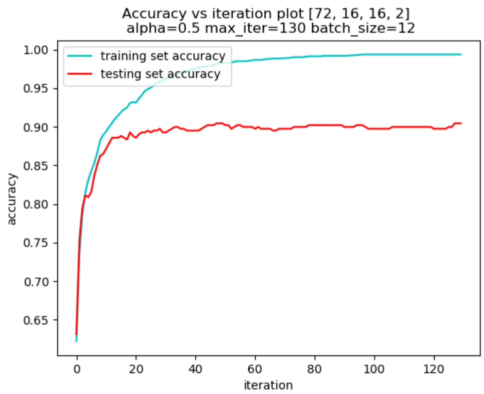
Split-4:

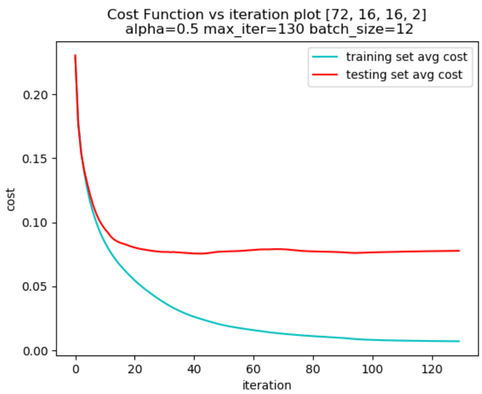
confusion matrix:

[[239. 33.]

 [21. 244.]]

 test_accuracy = 0.8994413407821229





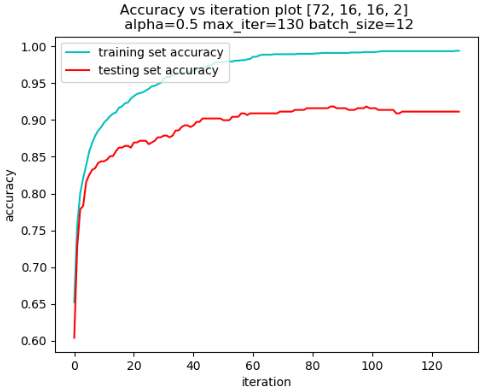
Split-5:

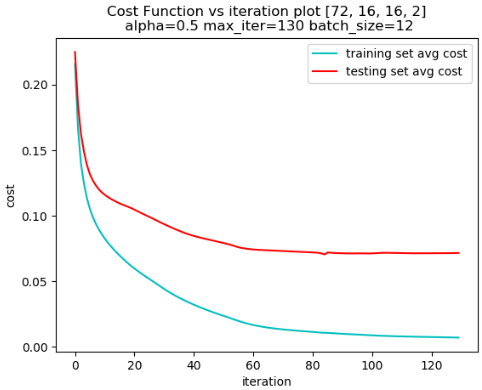
confusion matrix:

[[239. 33.]

[21. 244.]]

test_accuracy = 0.8994413407821229





Average Accuracy: 0.9067599067599067

2. Radial Basis Feed Forward Neural Network

```
import pandas as pd
import numpy as np
from scipy.io import loadmat
from sklearn.cluster import KMeans
from preprocessing import NormalScaler

def gaussian(x, mu, std):
    '''
        ϕ(x,β,μ) = e^(-1/(2σ^2) ||x−u||^2)
    '''
    return np.exp((-1/(2*(std**2))) * np.sum((x - mu)**2))

def multiquadric(x, mu, std):
    '''
        ϕ(x,σ,μ) = (||x − u||^2 + σ^2)^(1/2)
    '''
    return np.sum(np.sqrt(np.sum(np.square(x-mu)) + (std**2)))

def linear(x, mu, std):
    '''
        ϕ(x,μ) = ||x − μ||
     '''
    return np.sum(abs(x - mu))

def train(X_train, y_train):
    # m = number of feature vectors
    m = X_train.shape[0]
```

```python
    # n = number of features
    n = X.shape[1]

    # running k means clustering
    kmeans = KMeans(n_clusters=k, max_iter=max_iter , random_state=0).fit(X_train)
    means = kmeans.cluster_centers_
    assignments = kmeans.predict(X_train)

    stds = []
    for i in range(k):
        temp = X_train[(assignments==i)]
        stds.append((1/temp.shape[0])*np.sum(abs(temp-means[i])))
    stds = np.array(stds)

    # Hidden layer matrix
    H = np.ndarray((m,k))
    for i in range(m):
        for j in range(k):
            H[i][j] = kernel(X_train[i], means[j], stds[j])

    # weight matrix
    W = np.dot(np.linalg.pinv(H),y_train)
    return {'W': W, 'means':means, 'stds':stds}

def test(X_test, y_test, W, means, stds):
    # testing data
    mt = X_test.shape[0]
    Ht = np.ndarray((mt,k))
    for i in range(mt):
        for j in range(k):
            Ht[i][j] = kernel(X_test[i], means[j], stds[j])
    yt = np.dot(Ht,W)

    # computing confusion matrix
    conf_mat = np.zeros((y_test.shape[1],y_test.shape[1]))
    for i in range(y_test.shape[0]):
        a = int(np.argmax(y_test[i]))
        b = int(np.argmax(yt[i]))
        conf_mat[a][b] += 1

    cost = np.mean((yt-y_test)**2)
    print(f'cost: {cost}')
    acc = 0
    for i in range(y_test.shape[0]):
        acc += (np.argmax(yt[i]) == np.argmax(y_test[i]))
    acc /=y_test.shape[0]
    print(f'accuracy = {acc}')
    print(f'confusion matrix\n {conf_mat}')
    return (cost, acc)

if __name__=='__main__':
    # data input
    data = pd.DataFrame(loadmat('./data5.mat')['x'])
    data = data.sample(frac=1).reset_index(drop=True)
```

```
X = data.loc[:,[i for i in range(72)]].values
y = data.loc[:,72:73].values
y_cat = np.zeros((y.shape[0],2))
for i in range(y.shape[0]):
    y_cat[i][int(y[i])] = 1

# data preprocessing
scaler = NormalScaler()
for j in range(X.shape[1]):
    scaler.fit(X[:,j])
    X[:,j] = scaler.transform(X[:,j])

kernel = gaussian
# kernel = multiquadric
# kernel = linear

# splitting data
max_iter = 30
k = 50
split = 'kfold'

if split=='holdout':
    train_percent = 0.6
    X_train = X[:int(train_percent*X.shape[0]),:]
    y_train = y_cat[:int(train_percent*X.shape[0])]
    X_test = X[int(train_percent*X.shape[0]):, :]
    y_test = y_cat[int(train_percent*X.shape[0]):]

    params = train(X_train, y_train)
    test(X_test, y_test, params['W'], params['means'], params['stds'])

elif split=='kfold':
    k_fold = 4
    Nk = X.shape[0]//k_fold
    accs = []
    iteration = 1
    for i in range(0, X.shape[0], Nk):
        print("K-fold: ", iteration)
        X_test = X[i:i+Nk,:]
        X_train = np.delete(X,range(i,i+Nk),0)
        y_test = y_cat[i:i+Nk]
        y_train = np.delete(y_cat,range(i,i+Nk),0)

        params = train(X_train, y_train)
        cost, acc = test(X_test, y_test, params['W'], params['means'], params['stds'])
        accs.append(acc)
        iteration+=1
    print("\nAvg Accuracy: ", np.mean(accs),'\n')
```

Results:

Holdout-fold cross validation split (60% train split):

Hidden neurons, k = 50 max_iter = 30

Gaussian	Multiquadratic	Linear
accuracy = 0.827906976744186 cost: 0.13094991361528405 confusion matrix [[342. 89.] [59. 370.]]	accuracy = 0.8406976744186047 cost: 0.15861561066805005 confusion matrix [[382. 54.] [83. 341.]]	accuracy = 0.872093023255814 cost = 0.10932858771888981 conf_mat= [[347. 51.] [59. 403.]]

K-fold cross validation split (5-fold):

Hidden neurons, k = 50 max_iter = 30

K-fold	Gaussian	Multiquadratic	Linear
1	cost = 0.1331342322989426 accuracy = 0.8438228438228438 conf_mat= [[177. 35.] [32. 185.]]	cost = 0.1281010587175988 accuracy = 0.8578088578088578 conf_mat= [[182. 30.] [31. 186.]]	cost = 0.11550470545582738 accuracy = 0.8648018648018648 conf_mat= [[179. 36.] [22. 192.]]
2	cost = 0.13447119041893205 accuracy = 0.8275058275058275 conf_mat= [[167. 35.] [39. 188.]]	cost = 0.13504372491380148 accuracy = 0.8391608391608392 conf_mat= [[180. 34.] [35. 180.]]	cost = 0.1306211602503553 accuracy = 0.8321678321678322 conf_mat= [[169. 35.] [37. 188.]]
3	cost = 0.11782207618134423 accuracy = 0.8508158508158508 conf_mat= [[187. 32.] [32. 178.]]	cost = 0.14094327191240583 accuracy = 0.8041958041958042 conf_mat= [[175. 45.] [39. 170.]]	cost = 0.10703102909519147 accuracy = 0.8717948717948718 conf_mat= [[212. 30.] [25. 162.]]
4	cost = 0.12811693124021806 accuracy = 0.8391608391608392 conf_mat= [[190. 40.] [29. 170.]]	cost = 0.11177006006949651 accuracy = 0.8624708624708625 conf_mat= [[186. 26.] [33. 184.]]	cost = 0.11502803725230505 accuracy = 0.8601398601398601 conf_mat= [[170. 36.] [24. 199.]]
5	cost = 0.1417911632921012 accuracy = 0.8368298368298368 conf_mat= [[182. 29.] [41. 177.]]	cost = 0.11878027395114374 accuracy = 0.8554778554778555 conf_mat= [[184. 31.] [31. 183.]]	cost = 0.11306227811837886 accuracy = 0.8508158508158508 conf_mat= [[179. 27.] [37. 186.]]
Average Accuracy:	0.8396270396270396	0.8438228438228439	0.8559440559440559

3. Stacked Autoencoder based classifier

```
import numpy as np
import pandas as pd
from scipy.io import loadmat
from preprocessing import NormalScaler
from MLP_auto import MLP as MLP_auto
from MLP import MLP

if __name__=='__main__':
    data = pd.DataFrame(loadmat('./data5.mat')['x'])
    data = data.sample(frac=1).reset_index(drop=True)

    X = data.loc[:,:71].values
    y = data.loc[:,72:73].values
    y_cat = np.zeros((y.shape[0],2))
```

```python
for i in range(y.shape[0]):
    y_cat[i][int(y[i])] = 1

# data preprocessing
scaler = NormalScaler()
for j in range(X.shape[1]):
    scaler.fit(X[:,j])
    X[:,j] = scaler.transform(X[:,j])

# m = number of feature vectors
m = X.shape[0]
# n = number of features
n = X.shape[1]

train_percent = 0.6
X_train = X[:int(train_percent*X.shape[0]),:]
y_train = y_cat[:int(train_percent*X.shape[0]),:]
X_test = X[int(train_percent*X.shape[0]):,:]
y_test = y_cat[int(train_percent*X.shape[0]):,:]

# hidden layers array
Layers = [42,24,12]
alpha = 0.5
max_iter = 30

# pretraining 3 autoencoders
model11 = MLP_auto([n, Layers[0]], ['sigmoid'])
print("pre-training autoencoder 1")
model11.train(X_train,X_train, alpha, 12, max_iter)

out1 = model11.output_hidden(X_train)

model12 = MLP_auto([Layers[0], Layers[1]], ['sigmoid'])
print("pre-training autoencoder 2")
model12.train(out1, out1, alpha, 12, max_iter)

out2 = model12.output_hidden(out1)

model13 = MLP_auto([Layers[1], Layers[2]], ['sigmoid'])
print("pre-training autoencoder 3")
model13.train(out2, out2, alpha, 12, max_iter)

# finetuning the stacked autoencoder
print("fine tuning stacked autoencoder")

final_model = MLP([n, *Layers, 2], ['sigmoid','sigmoid','sigmoid','sigmoid'])
# final_model.W_list[0:3] = model.W_list[0:3]
final_model.W_list[0] = model11.W_list[0]
final_model.W_list[1] = model12.W_list[0]
final_model.W_list[2] = model13.W_list[0]



# training deep neural network
```

```
    alpha = 0.5
    batch_size = 12
    max_iter = 200
    final_model.train(X_train, y_train,X_test,y_test, alpha, batch_size, max_iter)
    print(final_model.accuracy(X_test,y_test))
    print(final_model.conf_mat(X_test, y_test))
```

Results:

Number of Hidden Layer Neurons: 42, 24,12

Holdout cross validation split: 60%-40%

Pretraining:

Learning rate = 0.5   max_iter = 30   batch size = 12

Fine tuning:

Learning rate = 0.5   max_iter = 200   batch size = 12

Accuracy: 0.9162790697674419

Confusion Matrix:

[[403. 41.]

 [31. 385.]]

4. Extreme Learning Machine

```
import numpy as np
import pandas as pd
from scipy.io import loadmat
from preprocessing import NormalScaler

class ELM:
    def __init__(self, L, X, Y, activation="tanh"):
        '''
        In this constructor the hidden layer matrix
        H is made using random weights a and b.
        m: number of samples
        n: number of features
        L: number of neurons in the hidden layer
        '''
        m = X.shape[0]
        n = X.shape[1]
        self.L = L
        H = np.ndarray((m, L), dtype=np.float64)
        self.a = np.random.randn(L, n)
```

```python
        self.b = np.random.randn(L)
        if(activation=="gaussian"): self.activate = self.gaussian
        elif(activation=="tanh"): self.activate = self.tanh

        for i in range(m):
            for j in range(L):
                H[i][j] = self.activate(X[i], self.a[j], self.b[j])

        self.train(H,Y)

    def train(self, H, Y):
        '''
        This function uses the following vectorized formula
        to compute and return the weight matrix between
        hidden and output layer.
        '''
        self.W = np.dot(np.linalg.pinv(H), Y)
        return self.W

    def test(self, X_test, y_test):
        '''
        This function computes the predicted values with
        the given test feature vectors.
        '''
        y_pred = y_test.copy()
        acc = 0
        conf_mat = np.zeros((y_test.shape[1], y_test.shape[1]))
        for i in range(X_test.shape[0]):
            x = []
            for j in range(self.L):
                x.append(self.activate(X_test[i], self.a[j], self.b[j]))
            x = np.array(x)
            max_i = np.argmax(np.dot(x.reshape(1,-1), self.W))

            for j in range(y_pred.shape[1]):
                if j==max_i:
                    y_pred[i][j] = 1
                    if max_i==np.argmax(y_test[i]):
                        acc+=1
                else: y_pred[i][j] = 0

            conf_mat[np.argmax(y_test[i])][np.argmax(y_pred[i])] +=1

        acc = acc/y_pred.shape[0]
        print(acc)
        print(conf_mat)
        return (acc, y_pred)



    def gaussian(self,x,a,b):
        '''
        This function returns the gaussian activated output
        Of a neuron.
```

```
            G(a,b,x) = exp(-b||x - a||^2)
        '''

        t = -b*np.sum(np.square(x.reshape(-1,1)-a.reshape(-1,1)))
        r = np.exp(t)
        return r

    def tanh(self,x,a,b):
        '''
        This function returns the tanh activated output
        Of a neuron.
            G(a,b,x) = (1-exp(-(ax+b)))/(1+exp(-(ax+b)))
        '''
        tmp = np.exp(-(np.dot(x.reshape(1,-1), a.reshape(1,-1).T)[0][0] + b))
        return (1-tmp)/(1+tmp)

if __name__=='__main__':

    # data input
    data = pd.DataFrame(loadmat('./data5.mat')['x'])
    data = data.sample(frac=1).reset_index(drop=True)
    X = data.loc[:,:71].values
    y = data.loc[:,72:73].values
    y_cat = np.zeros((y.shape[0],2))
    for i in range(y.shape[0]):
        y_cat[i][int(y[i])] = 1

    # data preprocessing
    scaler = NormalScaler()
    for j in range(X.shape[1]):
        scaler.fit(X[:,j])
        X[:,j] = scaler.transform(X[:,j])

    k_fold = 5
    Nk = X.shape[0]//k_fold
    models = []
    acc = []
    iterat = 1
    for i in range(0, X.shape[0]-Nk+1, Nk):
        print("\n\nk fold iteration: ", iterat)
        X_test = X[i:i+Nk,:]
        X_train = np.delete(X,range(i,i+Nk),0)
        y_test = y_cat[i:i+Nk,:]
        y_train = np.delete(y_cat,range(i,i+Nk),0)

        # m = number of feature vectors
        m = X_train.shape[0]
        # n = number of features
        n = X_train.shape[1]
        L = 128
        elm = ELM(L, X_train, y_train, "gaussian")
        models.append(elm)
        acc.append(elm.test(X_test,y_test)[0])
```

```
        iterat+=1

    print("Average Accuracy: ", np.mean(acc))
```

Results:

K-fold cross validation split (5-fold):

Tanh Hidden neurons, L = 128

Gaussian hidden neurons, L = 300

K-fold	Tanh activation	Gaussian activation
1	accuracy = 0.8344988344988346 conf_mat= [[190. 33.] [38. 168.]]	accuracy = 0.7808857808857809 conf_mat= [[169. 44.] [50. 166.]]
2	accuracy = 0.8741258741258742 conf_mat= [[184. 25.] [29. 191.]]	accuracy = 0.8018648018648019 conf_mat= [[171. 43.] [42. 173.]]
3	accuracy = 0.8275058275058275 conf_mat= [[181. 40.] [34. 174.]]	accuracy = 0.7878787878787878 conf_mat= [[159. 50.] [41. 179.]]
4	accuracy = 0.8275058275058275 conf_mat= [[174. 32.] [42. 181.]]	accuracy = 0.7972027972027972 conf_mat= [[167. 43.] [44. 175.]]
5	accuracy = 0.8671328671328671 conf_mat= [[187. 28.] [29. 185.]]	accuracy = 0.7808857808857809 conf_mat= [[175. 51.] [43. 160.]]
Average Accuracy:	0.846153846153846	0.7897435897435898

5. Stacked autoencoder based ELM classifier

```
from preprocessing import NormalScaler
import numpy as np
import pandas as pd
from scipy.io import loadmat
from elm import ELM
from MLP_auto import MLP


if __name__=='__main__':
    data = pd.DataFrame(loadmat('./data5.mat')['x'])
    data = data.sample(frac=1).reset_index(drop=True)
    X = data.loc[:,:71].values
    y = data.loc[:,72:73].values
    y_cat = np.zeros((y.shape[0],2))
    for i in range(y.shape[0]):
        y_cat[i][int(y[i])] = 1

    # data preprocessing
    scaler = NormalScaler()
```

```
    for j in range(X.shape[1]):
        scaler.fit(X[:,j])
        X[:,j] = scaler.transform(X[:,j])

    # m = number of feature vectors
    m = X.shape[0]
    # n = number of features
    n = X.shape[1]

    train_percent = 0.6
    X_train = X[:int(train_percent*X.shape[0]),:]
    y_train = y_cat[:int(train_percent*X.shape[0]),:]
    X_test = X[int(train_percent*X.shape[0]):,:]
    y_test = y_cat[int(train_percent*X.shape[0]):,:]

    alpha = 0.6
    max_iter = 25
    batch_size = 12

    # pretraining 3 autoencoders
    model11 = MLP([n, 42], ['sigmoid'])
    print("pre training autoencoder 1")
    model11.train(X_train,X_train, alpha, batch_size, max_iter)

    out1 = model11.output_hidden(X_train)

    model12 = MLP([42, 24], ['sigmoid'])
    print("pre training autoencoder 2")
    model12.train(out1, out1, alpha, batch_size, max_iter)

    # stacking the pretrained autoencoders
    model = MLP([n, 42, 24], ['sigmoid','sigmoid'])

    # initializing pretrained weights
    model.W_list[0] = model11.W_list[0]
    model.W_list[-1] = model11.W_list[0].T

    model.W_list[1] = model12.W_list[0]
    model.W_list[-2] = model12.W_list[0].T

    # finetuning the stacked autoencoder
    # print("training stacked autoencoder")
    # model.train(X_train, X_train, alpha, batch_size, 50)

    print("\nELM part of the neural network\n")

    elm_X_train = np.ndarray((X_train.shape[0], model.A[2].shape[0]))
    elm_X_test = np.ndarray((X_test.shape[0], model.A[2].shape[0]))

    for i in range(X_train.shape[0]):
        model.forward_prop(X_train[i])
        elm_X_train[i] = model.A[2].reshape(-1,)
    for i in range(X_test.shape[0]):
        model.forward_prop(X_test[i])
```

```
        elm_X_test[i] = model.A[2].reshape(-1,)

    # using tanh activation for elm
    elm_model = ELM(128, elm_X_train, y_train, 'tanh')
    elm_model.test(elm_X_test,y_test)
    elm_model.test(elm_X_train,y_train)
```

Results:

Holdout cross validation split: 60%-40%

Autoencoder:

Number of Hidden Layer Neurons: 42, 24

Learning rate = 0.6   max_iter = 25   batch size = 12

ELM part:

Hidden neurons, L = 128

Activation = tanh

Test set Accuracy: 0.8023255813953488

Confusion matrix:

[[332. 103.]

 [67. 358.]]

Train set Accuracy: 0.8454968944099379

Confusion Matrix

[[533. 107.]

 [92. 556.]]

Name: S Rohith
Id: 2017A7PS0034H