In [2]:
```
import numpy as np
import pandas as pd
from math import exp
from sklearn.metrics import confusion_matrix
from sklearn.metrics import cohen_kappa_score
import numpy as np
import csv
import matplotlib.pyplot as plt
```

In [3]:
```
cd /Users/jithu/Documents/Notes
```

```
/Users/jithu/Documents/Notes
```

In [4]:
```
dataset = pd.read_csv('data.csv')
```

In [5]:
```
# Backprop on the Vowel Dataset
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp
from sklearn.metrics import confusion_matrix
from sklearn.metrics import cohen_kappa_score
import numpy as np
import csv

# Load a CSV file
def loadCsv(filename):
        trainSet = []

        lines = csv.reader(open(filename, 'r'))
        dataset = list(lines)
        for i in range(len(dataset)):
                for j in range(4):
                        #print("DATA {}".format(dataset[i]))
                        dataset[i][j] = float(dataset[i][j])
                trainSet.append(dataset[i])
        return trainSet

def minmax(dataset):
        minmax = list()
        stats = [[min(column), max(column)] for column in zip(*dataset)]
        return stats

# Rescale dataset columns to the range 0-1
def normalize(dataset, minmax):
        for row in dataset:
                for i in range(len(row)-1):
                        row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] -

# Convert string column to float
def column_to_float(dataset, column):
        for row in dataset:
                try:
                        row[column] = float(row[column])
```

```
                except ValueError:
                        print("Error with row",column,":",row[column])
                        pass

# Convert string column to integer
def column_to_int(dataset, column):
        class_values = [row[column] for row in dataset]
        unique = set(class_values)
        lookup = dict()
        for i, value in enumerate(unique):
                lookup[value] = i
        for row in dataset:
                row[column] = lookup[row[column]]
        return lookup

# Find the min and max values for each column


# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
        dataset_split = list()
        dataset_copy = list(dataset)
        fold_size = int(len(dataset) / n_folds)
        for i in range(n_folds):
                fold = list()
                while len(fold) < fold_size:
                        index = randrange(len(dataset_copy))
                        fold.append(dataset_copy.pop(index))
                dataset_split.append(fold)
        return dataset_split

# Calculate accuracy percentage
def accuracy_met(actual, predicted):
        correct = 0
        for i in range(len(actual)):
                if actual[i] == predicted[i]:
                        correct += 1
        return correct / float(len(actual)) * 100.0

# Evaluate an algorithm usimport pandas as pd

def run_algorithm(dataset, algorithm, n_folds, *args):

        folds = cross_validation_split(dataset, n_folds)
        #for fold in folds:
                #print("Fold {} \n \n".format(fold))
        scores = list()
        for fold in folds:
                #print("Test Fold {} \n \n".format(fold))
                train_set = list(folds)
                train_set.remove(fold)
                train_set = sum(train_set, [])
                test_set = list()
                for row in fold:
                        row_copy = list(row)
                        test_set.append(row_copy)
                        row_copy[-1] = None
```

```
                predicted = algorithm(train_set, test_set, *args)
                actual = [row[-1] for row in fold]
                accuracy = accuracy_met(actual, predicted)
                cm = confusion_matrix(actual, predicted)
                print('\n'.join([''.join(['{:4}'.format(item) for item in
                #confusionmatrix = np.matrix(cm)
                FP = cm.sum(axis=0) - np.diag(cm)
                FN = cm.sum(axis=1) - np.diag(cm)
                TP = np.diag(cm)
                TN = cm.sum() - (FP + FN + TP)
                print('False Positives\n {}'.format(FP))
                print('False Negetives\n {}'.format(FN))
                print('True Positives\n {}'.format(TP))
                print('True Negetives\n {}'.format(TN))
                TPR = TP/(TP+FN)
                print('Sensitivity \n {}'.format(TPR))
                TNR = TN/(TN+FP)
                print('Specificity \n {}'.format(TNR))
                Precision = TP/(TP+FP)
                print('Precision \n {}'.format(Precision))
                Recall = TP/(TP+FN)
                print('Recall \n {}'.format(Recall))
                Acc = (TP+TN)/(TP+TN+FP+FN)
                print('Áccuracy \n{}'.format(Acc))
                Fscore = 2*(Precision*Recall)/(Precision+Recall)
                print('FScore \n{}'.format(Fscore))
                k=cohen_kappa_score(actual, predicted)
                print('Çohen Kappa \n{}'.format(k))
                scores.append(accuracy)
        return scores


# Calculate neuron activation for an input
def activate(weights, inputs):
        activation = weights[-1]
        for i in range(len(weights)-1):
                activation += weights[i] * inputs[i]
        return activation

# Transfer neuron activation
def transfer(activation):
        return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
        inputs = row
        for layer in network:
                new_inputs = []
                for neuron in layer:
                        activation = activate(neuron['weights'], inputs)
                        neuron['output'] = transfer(activation)
                        new_inputs.append(neuron['output'])
                inputs = new_inputs
        return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
```

```
        return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
        for i in reversed(range(len(network))):
                layer = network[i]
                errors = list()
                if i != len(network)-1:
                        for j in range(len(layer)):
                                error = 0.0
                                for neuron in network[i + 1]:
                                        error += (neuron['weights'][j] * ne
                                errors.append(error)
                else:
                        for j in range(len(layer)):
                                neuron = layer[j]
                                errors.append(expected[j] - neuron['output
                for j in range(len(layer)):
                        neuron = layer[j]
                        neuron['delta'] = errors[j] * transfer_derivative(r

# Update network weights with error
def update_weights(network, row, l_rate):
        for i in range(len(network)):
                inputs = row[:-1]
                if i != 0:
                        inputs = [neuron['output'] for neuron in network[i
                for neuron in network[i]:
                        for j in range(len(inputs)):
                                temp = l_rate * neuron['delta'] * inputs[j

                                neuron['weights'][j] += temp
                                #print("neuron weight{} \n".format(neuron[
                                neuron['prev'][j] = temp
                        temp = l_rate * neuron['delta'] + mu * neuron['prev
                        neuron['weights'][-1] += temp
                        neuron['prev'][-1] = temp


# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
        for epoch in range(n_epoch):
                for row in train:
                        outputs = forward_propagate(network, row)
                        #print(network)
                        expected = [0 for i in range(n_outputs)]
                        expected[row[-1]] = 1
                        #print("expected row{}\n".format(expected))
                        backward_propagate_error(network, expected)
                        update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
        network = list()
        hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]
        network.append(hidden_layer)
        hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]
```

```
        network.append(hidden_layer)
        output_layer = [{'weights':[random() for i in range(n_hidden + 1)]
        network.append(output_layer)
        #print(network)
        return network

# Make a prediction with a network
def predict(network, row):
        outputs = forward_propagate(network, row)
        return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
        n_inputs = len(train[0]) - 1
        n_outputs = len(set([row[-1] for row in train]))
        network = initialize_network(n_inputs, n_hidden, n_outputs)
        train_network(network, train, l_rate, n_epoch, n_outputs)
        #print("network {}\n".format(network))
        predictions = list()
        for row in test:
                prediction = predict(network, row)
                predictions.append(prediction)
        return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
filename = 'data.csv'
dataset = loadCsv(filename)
for i in range(len(dataset[0])-1):
        column_to_float(dataset, i)
# convert class column to integers
column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = minmax(dataset)
normalize(dataset, minmax)
# evaluate algorithm
n_folds = 5
l_rate = 0.1
mu=0.001
n_epoch = 20
n_hidden = 4
scores = run_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch

print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
   0   0   0   0  17   0
   0   0   0   0   9   0
   0   0   0   0  27   0
   0   0   0   0  31   0
   0   0   0   0  47   0
   0   0   0   0  43   0
False Positives
 [  0   0   0   0 127   0]
False Negetives
 [17  9 27 31  0 43]
True Positives
```

```
 [ 0  0  0  0 47  0]
True Negetives
 [157 165 147 143   0 131]
Sensitivity
 [0. 0. 0. 0. 1. 0.]
Specificity
 [1. 1. 1. 1. 0. 1.]
Precision
 [       nan        nan        nan        nan 0.27011494        nan]
Recall
 [0. 0. 0. 0. 1. 0.]
Áccuracy
[0.90229885 0.94827586 0.84482759 0.82183908 0.27011494 0.75287356]
FScore
[       nan        nan        nan        nan 0.42533937        nan]
Çohen Kappa
0.0
<ipython-input-5-70463cd331ec>:116: RuntimeWarning: invalid value encounter
ed in true_divide
  Precision = TP/(TP+FP)
   0   0   0   0  17   0
   0   0   0   0  21   0
   0   0   0   0  40   0
   0   0   0   0  28   0
   0   0   0   0  39   0
   0   0   0   0  29   0
False Positives
 [  0   0   0   0 135   0]
False Negetives
 [17 21 40 28  0 29]
True Positives
 [ 0  0  0  0 39  0]
True Negetives
 [157 153 134 146   0 145]
Sensitivity
 [0. 0. 0. 0. 1. 0.]
Specificity
 [1. 1. 1. 1. 0. 1.]
Precision
 [       nan        nan        nan        nan 0.22413793        nan]
Recall
 [0. 0. 0. 0. 1. 0.]
Áccuracy
[0.90229885 0.87931034 0.77011494 0.83908046 0.22413793 0.83333333]
FScore
[       nan        nan        nan        nan 0.36619718        nan]
Çohen Kappa
0.0
   0   0   0   0  12   0
   0   0   0   0  18   0
   0   0   0   0  45   0
   0   0   0   0  28   0
   0   0   0   0  37   0
   0   0   0   0  34   0
False Positives
 [  0   0   0   0 137   0]
False Negetives
 [12 18 45 28  0 34]
True Positives
 [ 0  0  0  0 37  0]
```

```
True Negetives
 [162 156 129 146   0 140]
Sensitivity
 [0. 0. 0. 0. 1. 0.]
Specificity
 [1. 1. 1. 1. 0. 1.]
Precision
 [       nan        nan        nan        nan 0.21264368        nan]
Recall
 [0. 0. 0. 0. 1. 0.]
Áccuracy
[0.93103448 0.89655172 0.74137931 0.83908046 0.21264368 0.8045977 ]
FScore
[      nan       nan       nan       nan 0.3507109       nan]
Çohen Kappa
0.0
   0   0   0   0  10   0
   0   0   0   0  22   0
   0   0   0   0  31   0
   0   0   0   0  35   0
   0   0   0   0  40   0
   0   0   0   0  36   0
False Positives
 [  0   0   0   0 134   0]
False Negetives
 [10 22 31 35  0 36]
True Positives
 [ 0  0  0  0 40  0]
True Negetives
 [164 152 143 139   0 138]
Sensitivity
 [0. 0. 0. 0. 1. 0.]
Specificity
 [1. 1. 1. 1. 0. 1.]
Precision
 [       nan        nan        nan        nan 0.22988506        nan]
Recall
 [0. 0. 0. 0. 1. 0.]
Áccuracy
[0.94252874 0.87356322 0.82183908 0.79885057 0.22988506 0.79310345]
FScore
[       nan        nan        nan        nan 0.37383178        nan]
Çohen Kappa
0.0
   0   0   0   0  16   0
   0   0   0   0  19   0
   0   0   0   0  29   0
   0   0   0   0  29   0
   0   0   0   0  43   0
   0   0   0   0  38   0
False Positives
 [  0   0   0   0 131   0]
False Negetives
 [16 19 29 29  0 38]
True Positives
 [ 0  0  0  0 43  0]
True Negetives
 [158 155 145 145   0 136]
Sensitivity
 [0. 0. 0. 0. 1. 0.]
Specificity
```

```
 [1. 1. 1. 1. 0. 1.]
Precision
 [       nan        nan        nan        nan 0.24712644        nan]
Recall
 [0. 0. 0. 0. 1. 0.]
Áccuracy
[0.90804598 0.8908046  0.83333333 0.83333333 0.24712644 0.7816092 ]
FScore
[       nan        nan        nan        nan 0.39631336        nan]
Çohen Kappa
0.0
Scores: [27.011494252873565, 22.413793103448278, 21.26436781609195, 22.9885
05747126435, 24.71264367816092]
Mean Accuracy: 23.678%
```

In [6]:

```
#df = pd.DataFrame(dataset)
```

In [7]:

```
#accuracy = history.history['accuracy']
#val_accuracy = history.history['val_accuracy']
#loss = history.history['loss']
#val_loss = history.history['val_loss']
```

In [9]:

```
W1 = np.random.normal(0.1,0.2, size=(6,3))
W2 = np.random.normal(0.1,0.2, size=(3,1))
epoch = 10
learning_rate = 0.001
loss_sgd_tr = []
loss_sgd_te = []
epochs = []

for i in range (epoch):
    loss = 0
    loss1 = 0
    for j in range (len(train)):
        #forward pass:
        forward = forward_prop1(train[j],train[j],W1,W2)
        loss += forward['loss']
        #back pass:
        w11 , w22 = back_prop1(train[j],W1,W2,forward)
        #weight updates
        W1 = W1 - (learning_rate * w11)
        W2 = W2 - (learning_rate * w22)

    for k in range(len(X_test)):
        forward = forward_prop1(test[k],test[k],W1,W2)
        loss1 += forward['loss']

    loss_sgd_tr.append(loss/len(train))
    loss_sgd_te.append(loss1/len(test))
    epochs.append(i)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-ce7298dc1969> in <module>
     10     loss = 0
     11     loss1 = 0
---> 12     for j in range (len(train)):
     13         #forward pass:
     14         forward = forward_prop1(train[j],train[j],W1,W2)

NameError: name 'train' is not defined
```

In [ ]:

In [ ]:

In [ ]: