VISHAKHA KUMARESAN 19BCE2678

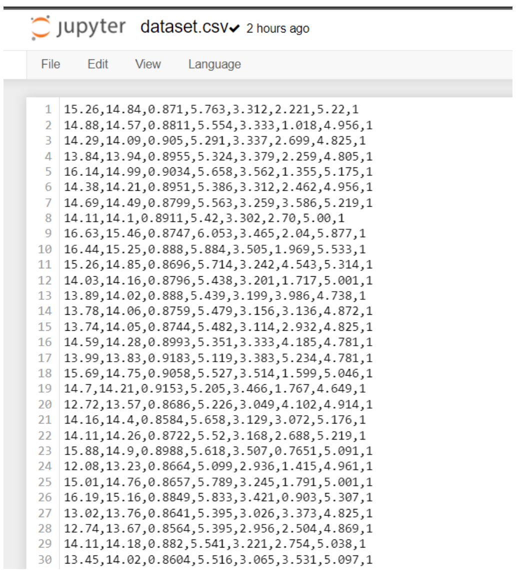
MACHINE LEARNING

LAB ASSIGNMENT 2

GITHUB LINK:

https://github.com/Vishakha1909/ML-Lab-Assignments/tree/main/Assignment%202

DATASET:




```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
14.38,14.21,0.8951,5.386,3.312,2.462,4.956,1
14.69,14.49,0.8799,5.563,3.259,3.586,5.219,1
14.11,14.1,0.8911,5.42,3.302,2.70,5.00,1
16.63,15.46,0.8747,6.053,3.465,2.04,5.877,1
16.44,15.25,0.888,5.884,3.505,1.969,5.533,1
15.26,14.85,0.8696,5.714,3.242,4.543,5.314,1
14.03,14.16,0.8796,5.438,3.201,1.717,5.001,1
13.89,14.02,0.888,5.439,3.199,3.986,4.738,1
13.78,14.06,0.8759,5.479,3.156,3.136,4.872,1
13.74,14.05,0.8744,5.482,3.114,2.932,4.825,1
14.59,14.28,0.8993,5.351,3.333,4.185,4.781,1
13.99,13.83,0.9183,5.119,3.383,5.234,4.781,1
15.69,14.75,0.9058,5.527,3.514,1.599,5.046,1
14.7,14.21,0.9153,5.205,3.466,1.767,4.649,1
12.72,13.57,0.8686,5.226,3.049,4.102,4.914,1
14.16,14.4,0.8584,5.658,3.129,3.072,5.176,1
14.11,14.26,0.8722,5.52,3.168,2.688,5.219,1
15.88,14.9,0.8988,5.618,3.507,0.7651,5.091,1
12.08,13.23,0.8664,5.099,2.936,1.415,4.961,1
15.01,14.76,0.8657,5.789,3.245,1.791,5.001,1
16.19,15.16,0.8849,5.833,3.421,0.903,5.307,1
13.02,13.76,0.8641,5.395,3.026,3.373,4.825,1
12.74,13.67,0.8564,5.395,2.956,2.504,4.869,1
14.11,14.18,0.882,5.541,3.221,2.754,5.038,1
13.45,14.02,0.8604,5.516,3.065,3.531,5.097,1
```

CODE:

```
#importing required libraries
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp


# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
```

```
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
def str_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
def dataset_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(dataset, minmax):
    for row in dataset:
        for i in range(len(row)-1):
```

```
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# 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_metric(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 using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        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_metric(actual, predicted)
        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] * neuron['delta'])
                errors.append(error)
                er.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(neuron['output'] - expected[j])
                er.append(neuron['output'] - expected[j])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# 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 - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] -= l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] -= l_rate * neuron['delta']

# 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)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            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)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    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)
    predictions = list()
    for row in test:
        prediction = predict(network, row)
        predictions.append(prediction)
    return(predictions)

# Test Backprop on Seeds dataset
seed(1)
er = []
# load and prepare data
filename = 'dataset.csv'
dataset = load_csv(filename)

for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)

# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)

# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)

# evaluate algorithm
```

VISHAKHA KUMARESAN 19BCE2678

n_folds = 5

l_rate = 1

n_epoch = 500

n_hidden = 5

scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)

printing the final accuracy and score

print('Scores: %s' % scores)

print()

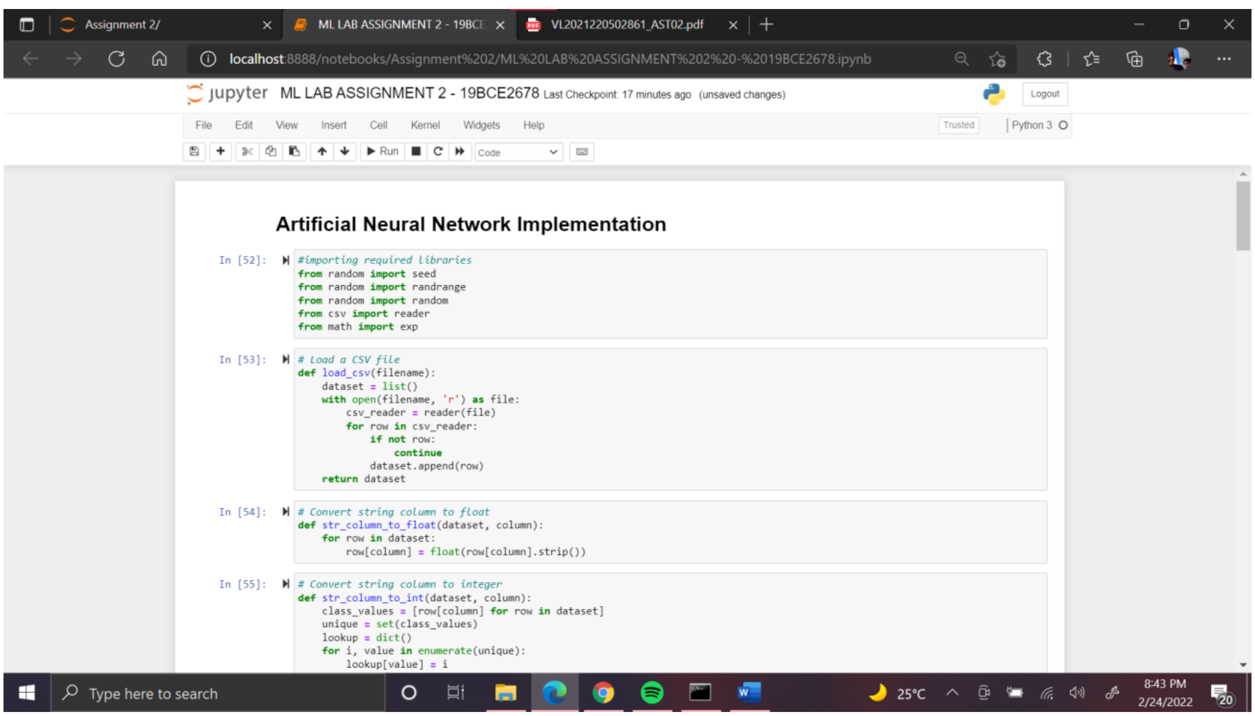
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

printing error values of each epoch

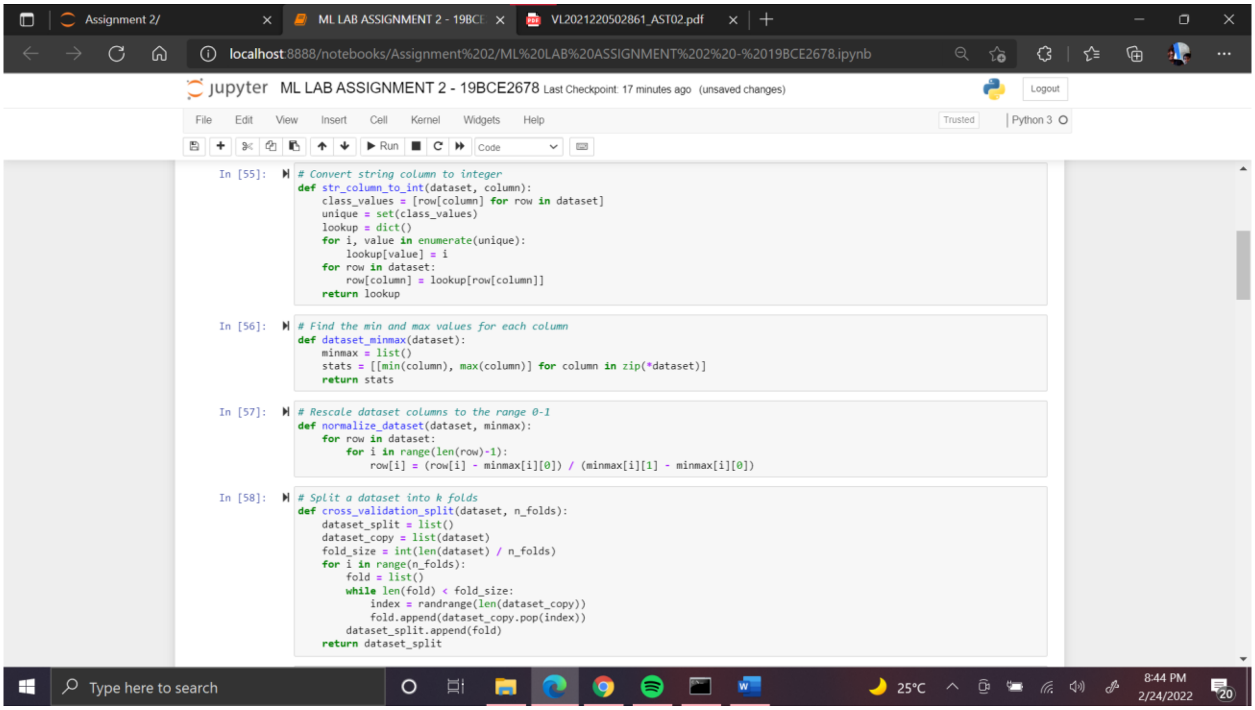
for i in range(len(er)):

    print(er[i])

SCREENSHOTS:



VISHAKHA KUMARESAN 19BCE2678

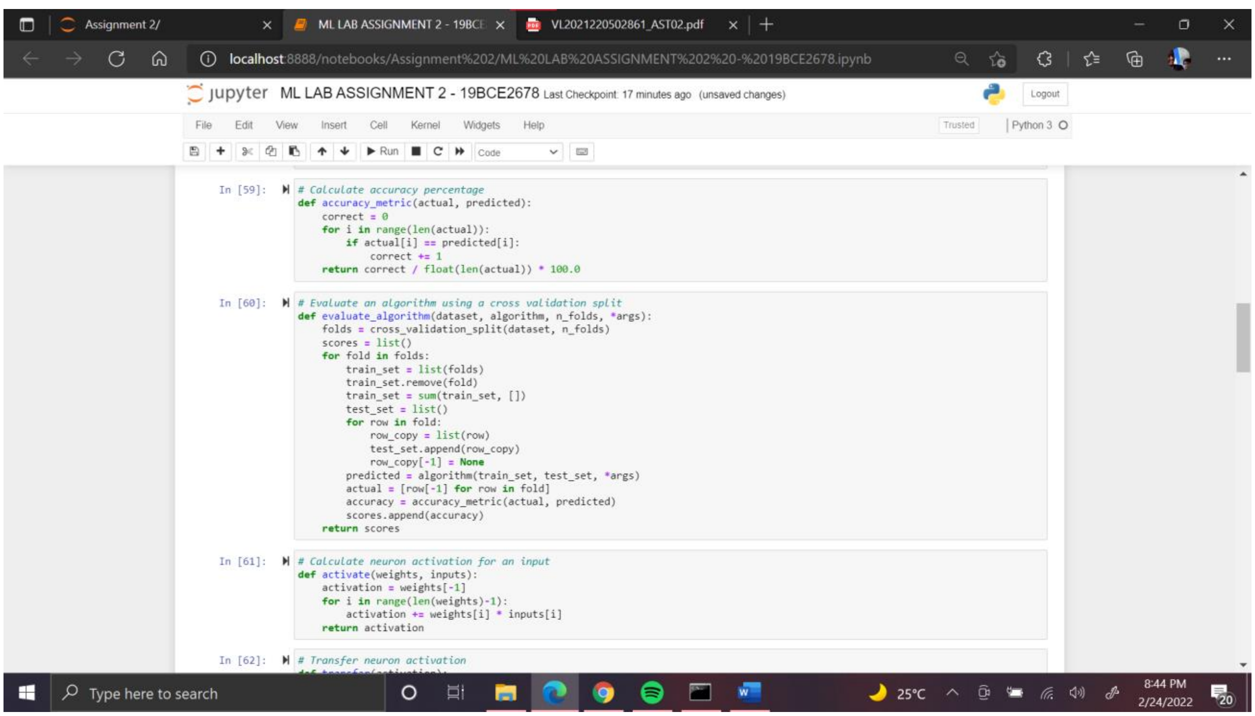


```python
# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
    for row in dataset:
        row[column] = lookup[row[column]]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

```python
# Find the min and max values for each column
def dataset_minmax(dataset):
    minmax = list()
    stats = [[min(column), max(column)] for column in zip(*dataset)]
    return stats
```

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

```python
# 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
```



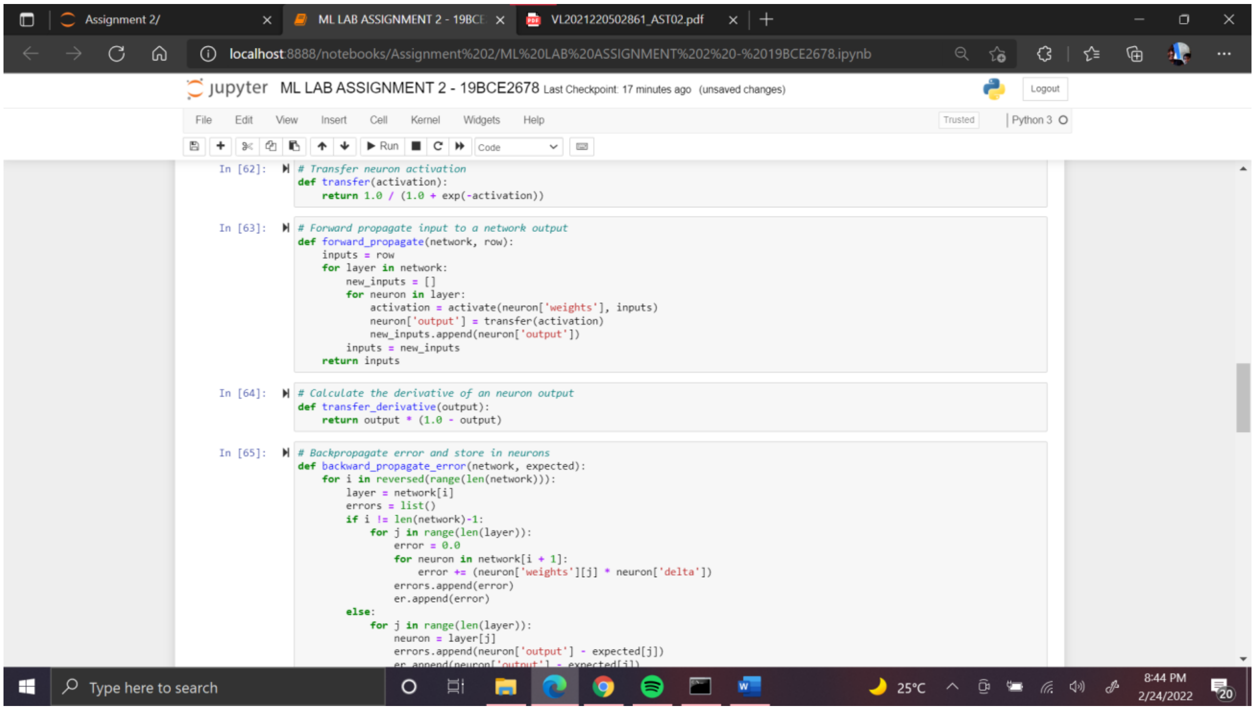
```python
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

```python
# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        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_metric(actual, predicted)
        scores.append(accuracy)
    return scores
```

```python
# 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
```

```python
# Transfer neuron activation
```

VISHAKHA KUMARESAN 19BCE2678



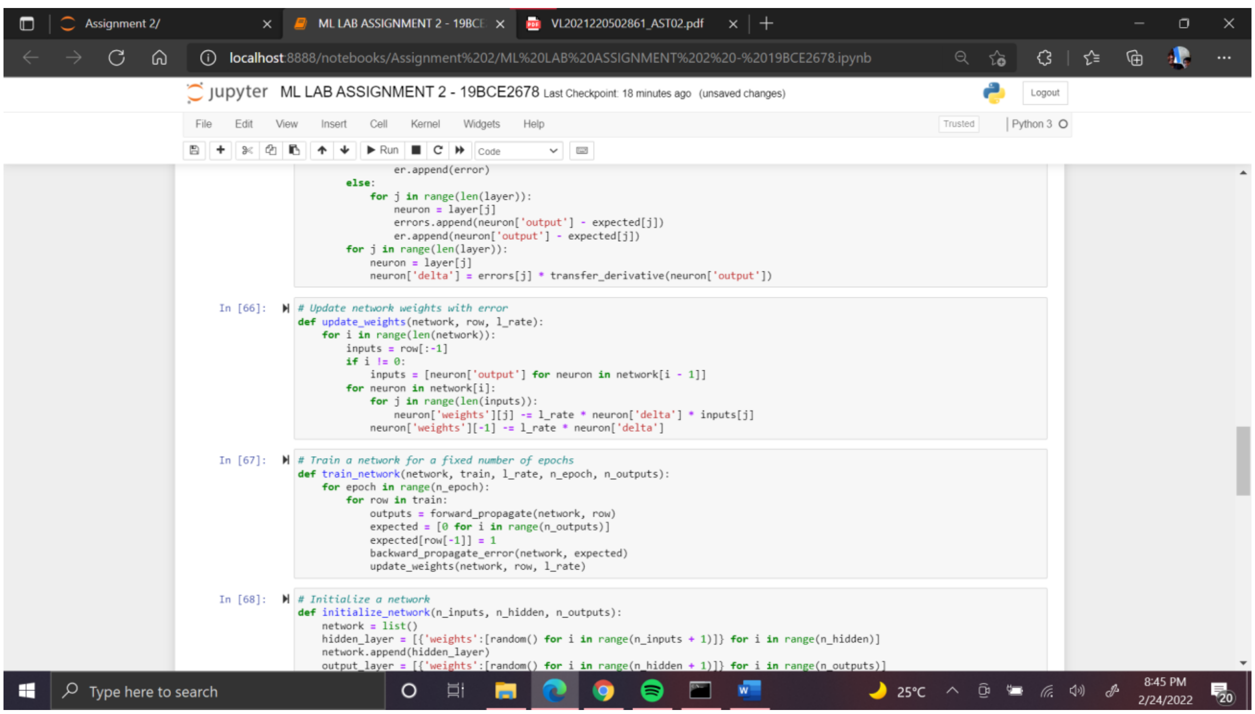
ML LAB ASSIGNMENT 2 - 19BCE2678 Last Checkpoint: 17 minutes ago (unsaved changes)

In [62]:
```python
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

In [63]:
```python
# 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
```

In [64]:
```python
# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)
```

In [65]:
```python
# 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] * neuron['delta'])
                errors.append(error)
                er.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(neuron['output'] - expected[j])
                er.append(neuron['output'] - expected[j])
```



ML LAB ASSIGNMENT 2 - 19BCE2678 Last Checkpoint: 18 minutes ago (unsaved changes)

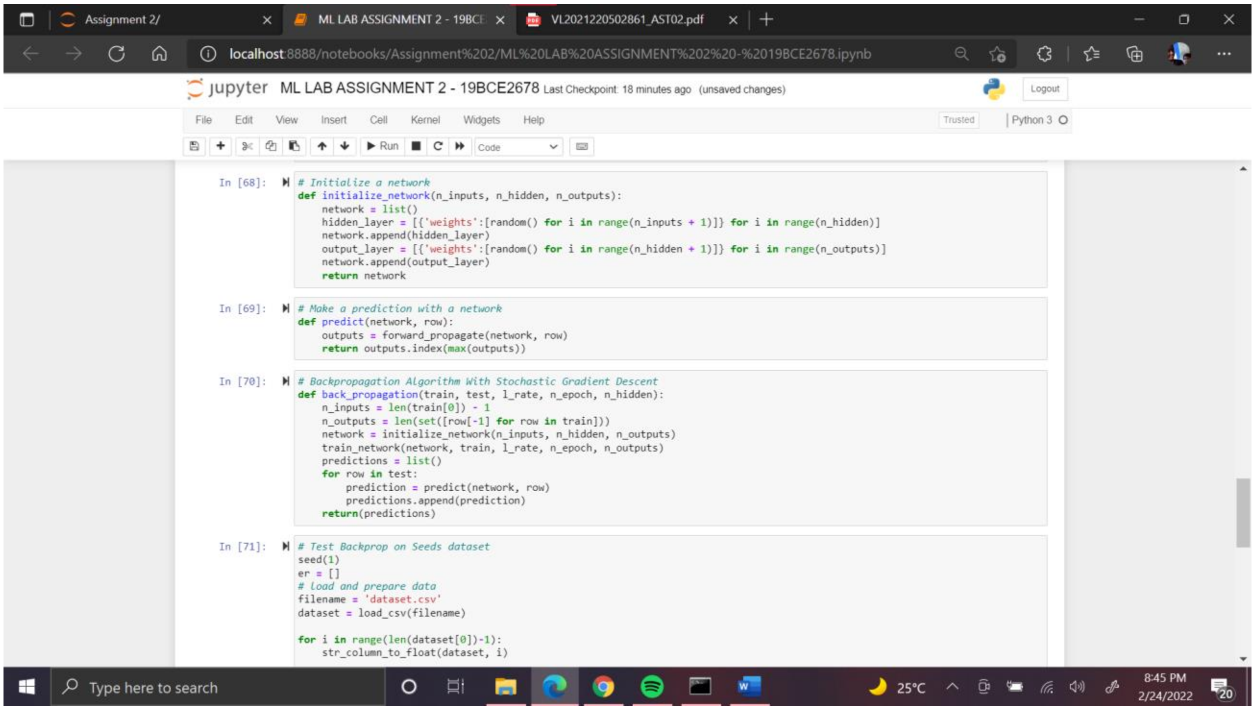
```python
                er.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(neuron['output'] - expected[j])
                er.append(neuron['output'] - expected[j])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

In [66]:
```python
# 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 - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] -= l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] -= l_rate * neuron['delta']
```

In [67]:
```python
# 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)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
```

In [68]:
```python
# 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)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
```

VISHAKHA KUMARESAN 19BCE2678



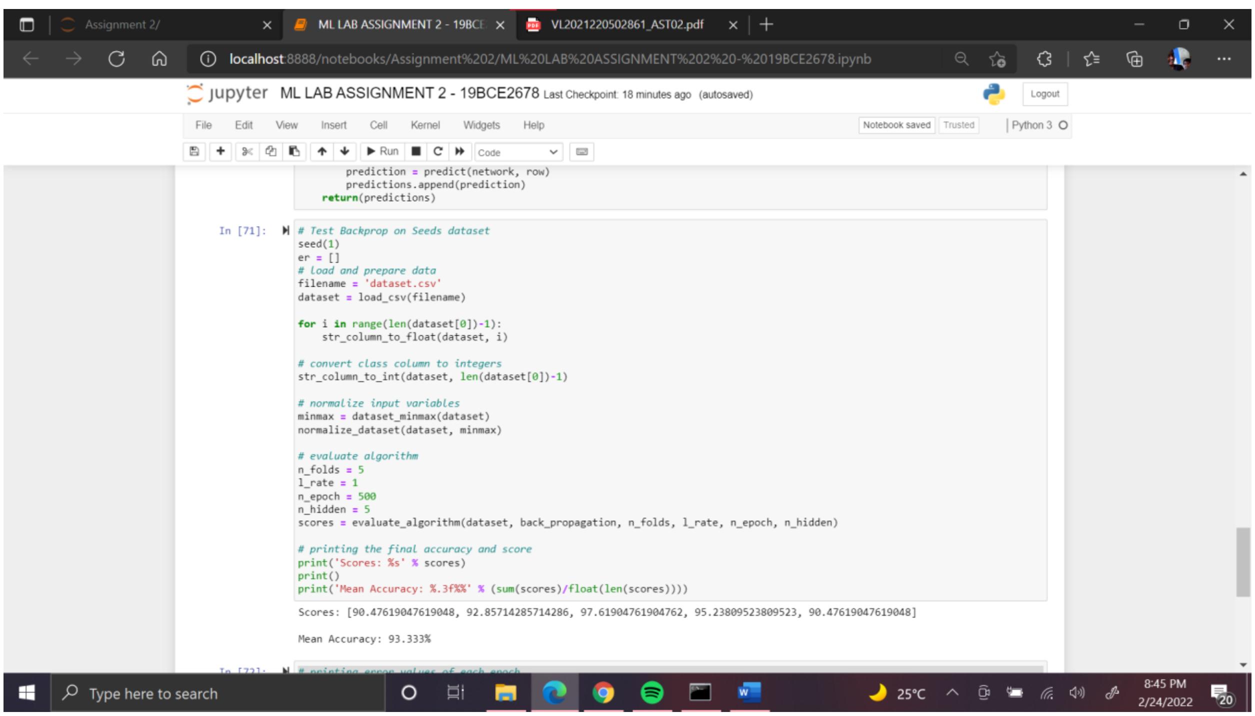
```python
In [68]: # Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network

In [69]: # Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

In [70]: # 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)
    predictions = list()
    for row in test:
        prediction = predict(network, row)
        predictions.append(prediction)
    return(predictions)

In [71]: # Test Backprop on Seeds dataset
seed(1)
er = []
# load and prepare data
filename = 'dataset.csv'
dataset = load_csv(filename)

for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)
```



```python
        prediction = predict(network, row)
        predictions.append(prediction)
    return(predictions)

In [71]: # Test Backprop on Seeds dataset
seed(1)
er = []
# load and prepare data
filename = 'dataset.csv'
dataset = load_csv(filename)

for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)

# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)

# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)

# evaluate algorithm
n_folds = 5
l_rate = 1
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)

# printing the final accuracy and score
print('Scores: %s' % scores)
print()
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

Scores: [90.47619047619048, 92.85714285714286, 97.61904761904762, 95.23809523809523, 90.47619047619048]

Mean Accuracy: 93.333%
```

VISHAKHA KUMARESAN 19BCE2678