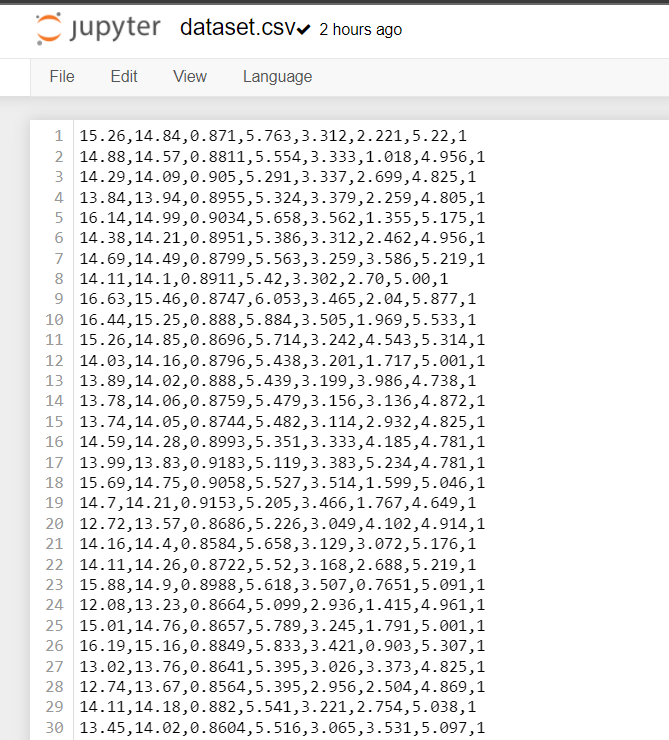
**MACHINE LEARNING**

**LAB ASSIGNMENT 2**

**GITHUB LINK:**

[**https://github.com/Vishakha1909/ML-Lab-Assignments/tree/main/Assignment%202**](https://github.com/Vishakha1909/ML-Lab-Assignments/tree/main/Assignment%202)

**DATASET:**

****

**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

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()

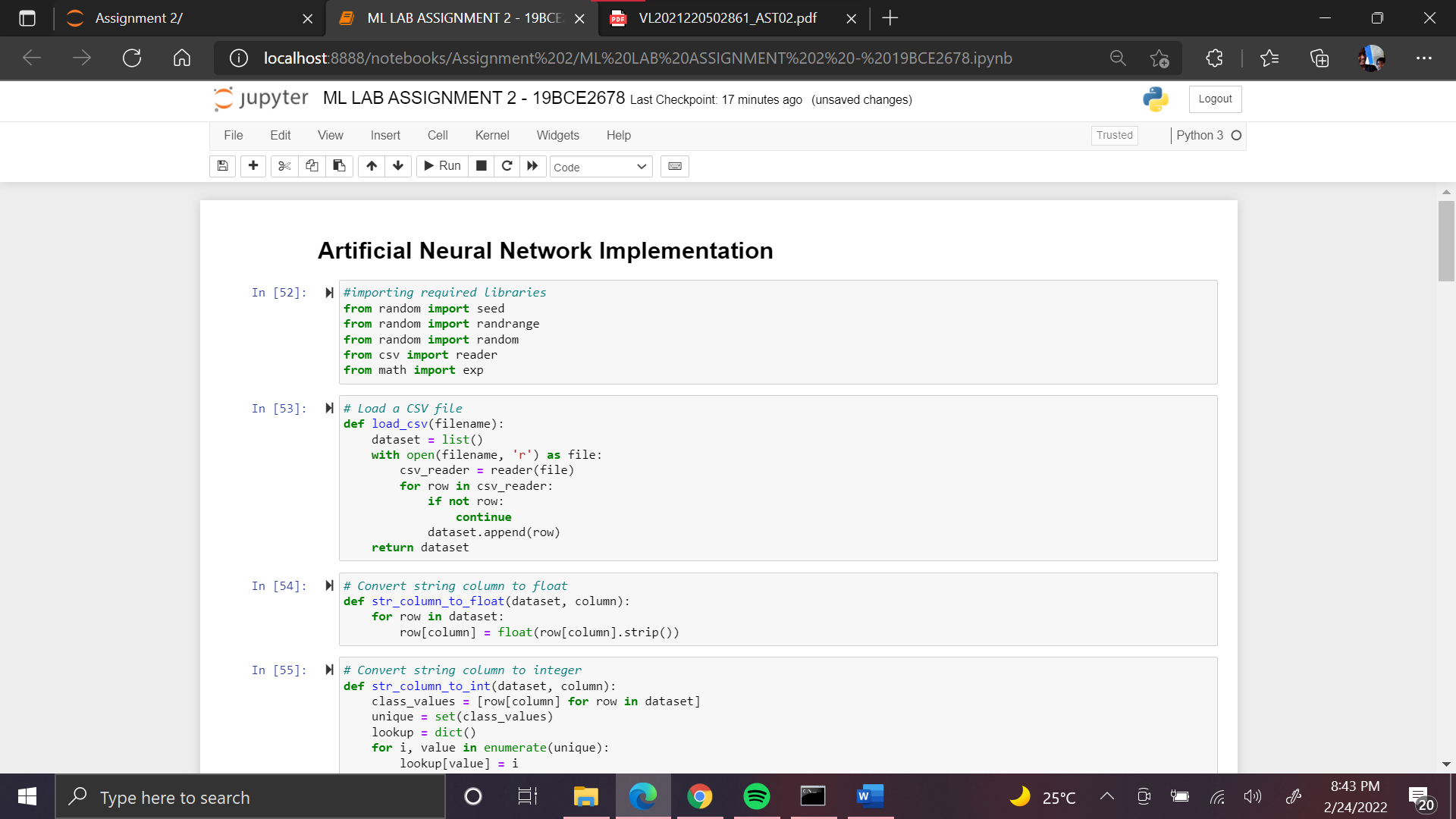
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

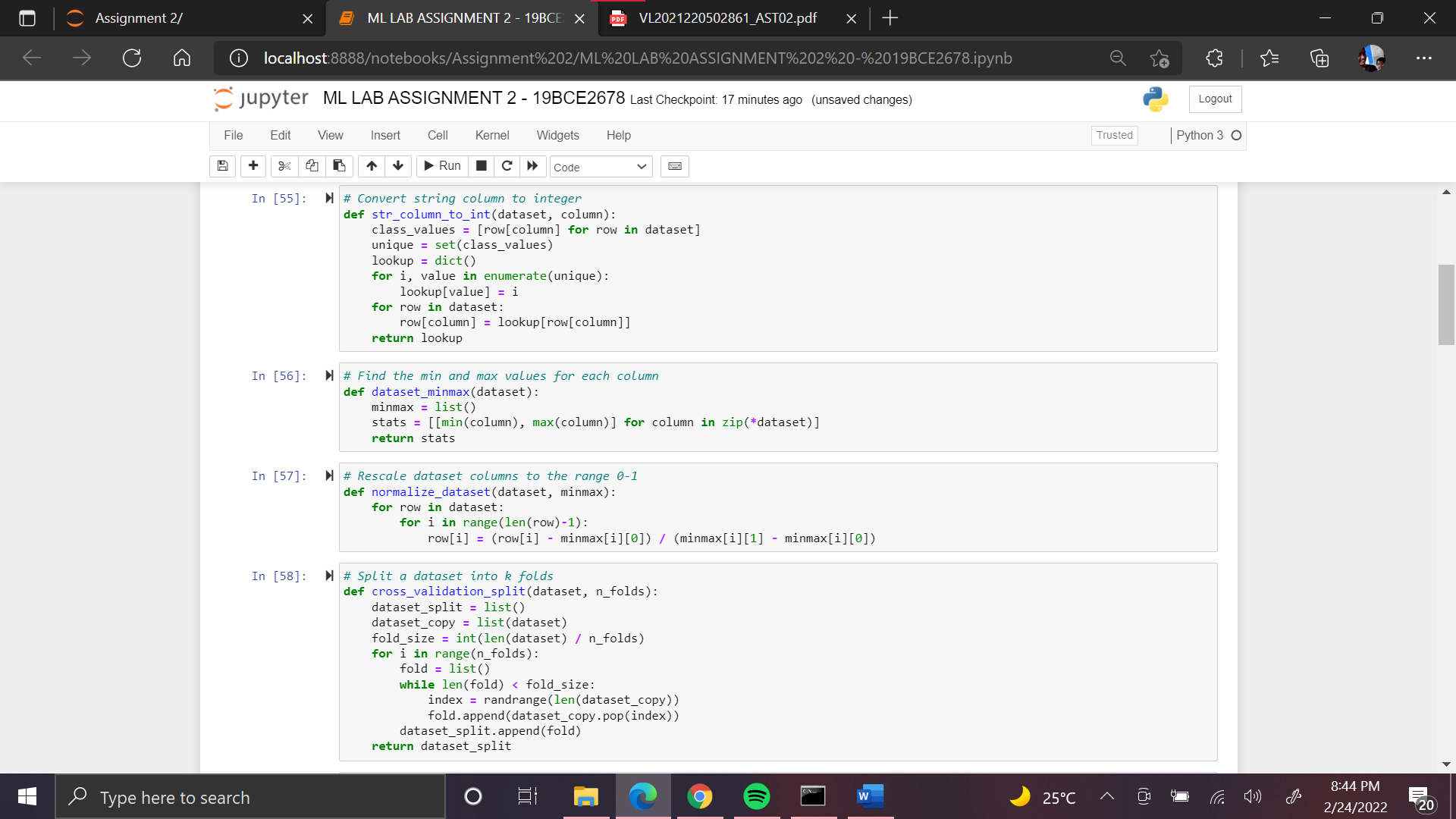
# printing error values of each epoch

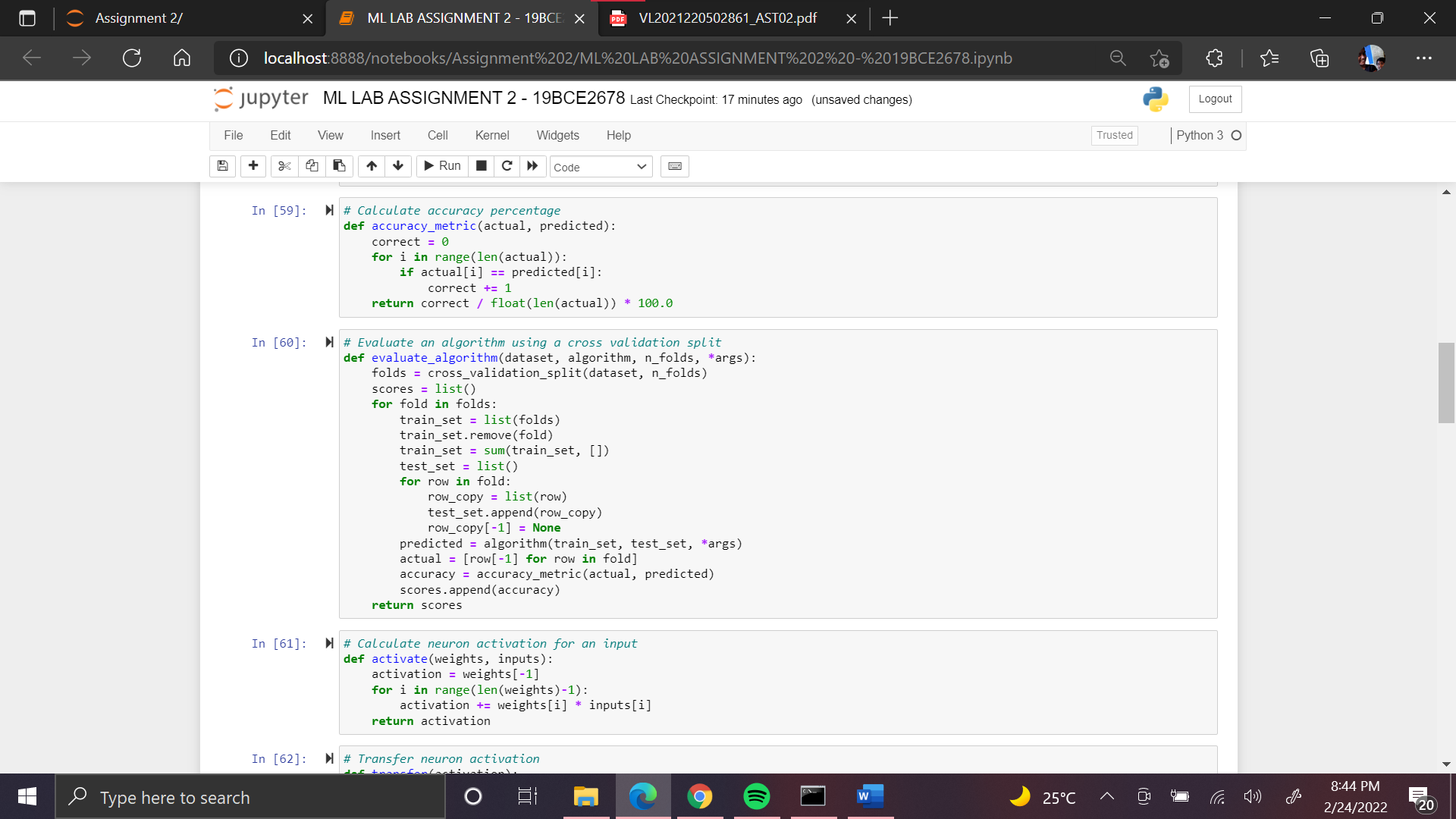
for i in range(len(er)):

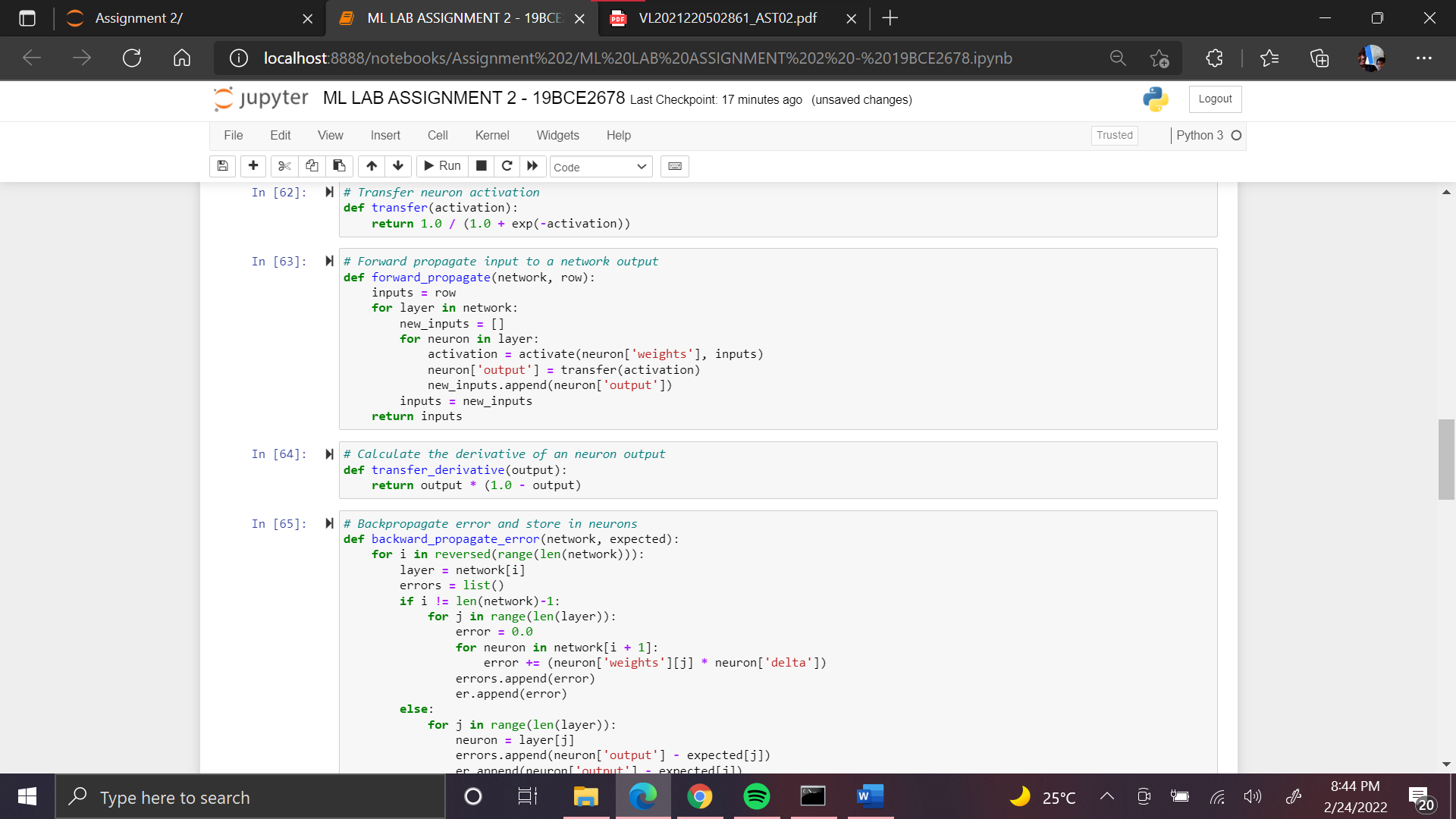
print(er[i])

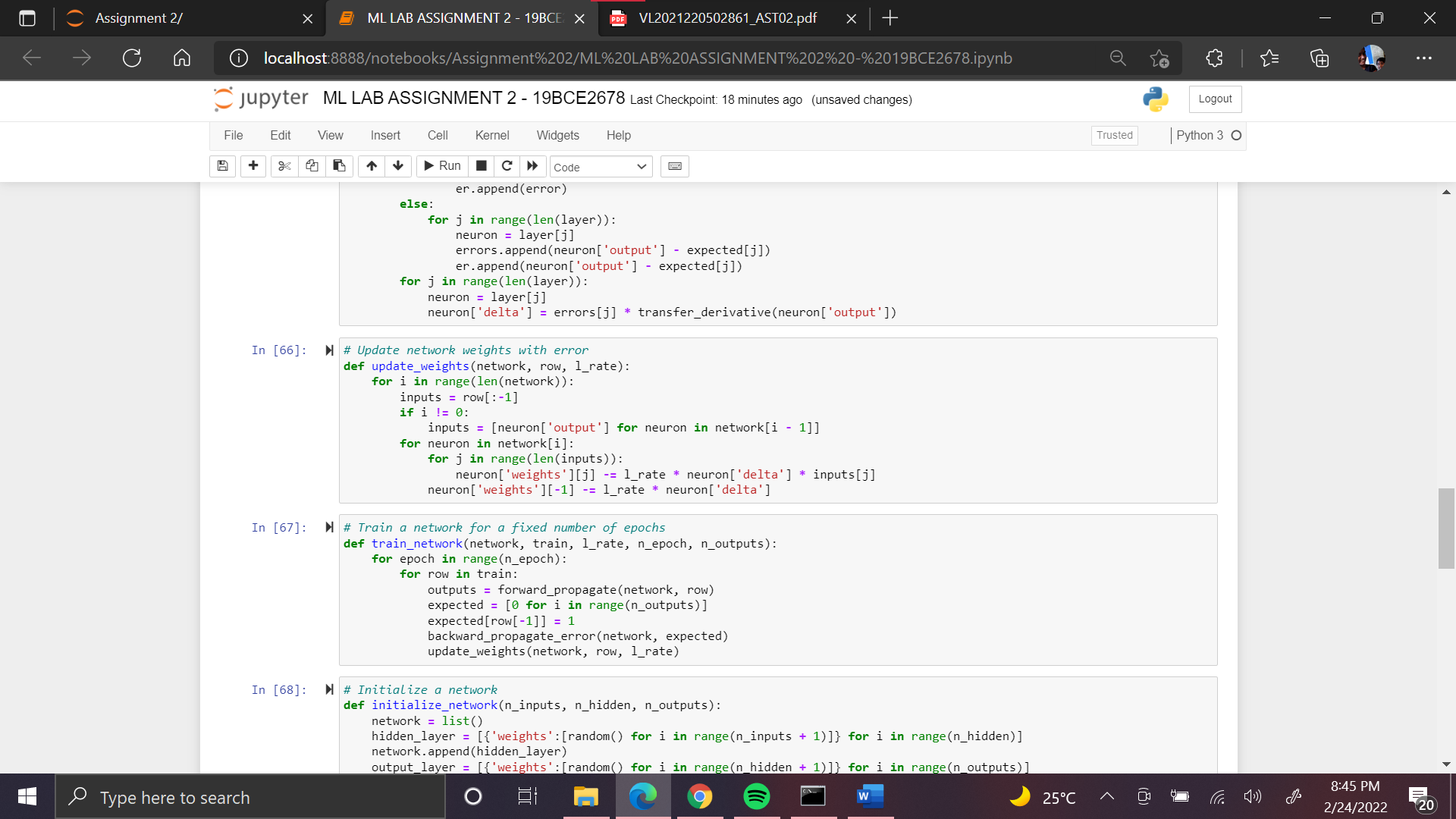
**SCREENSHOTS:**

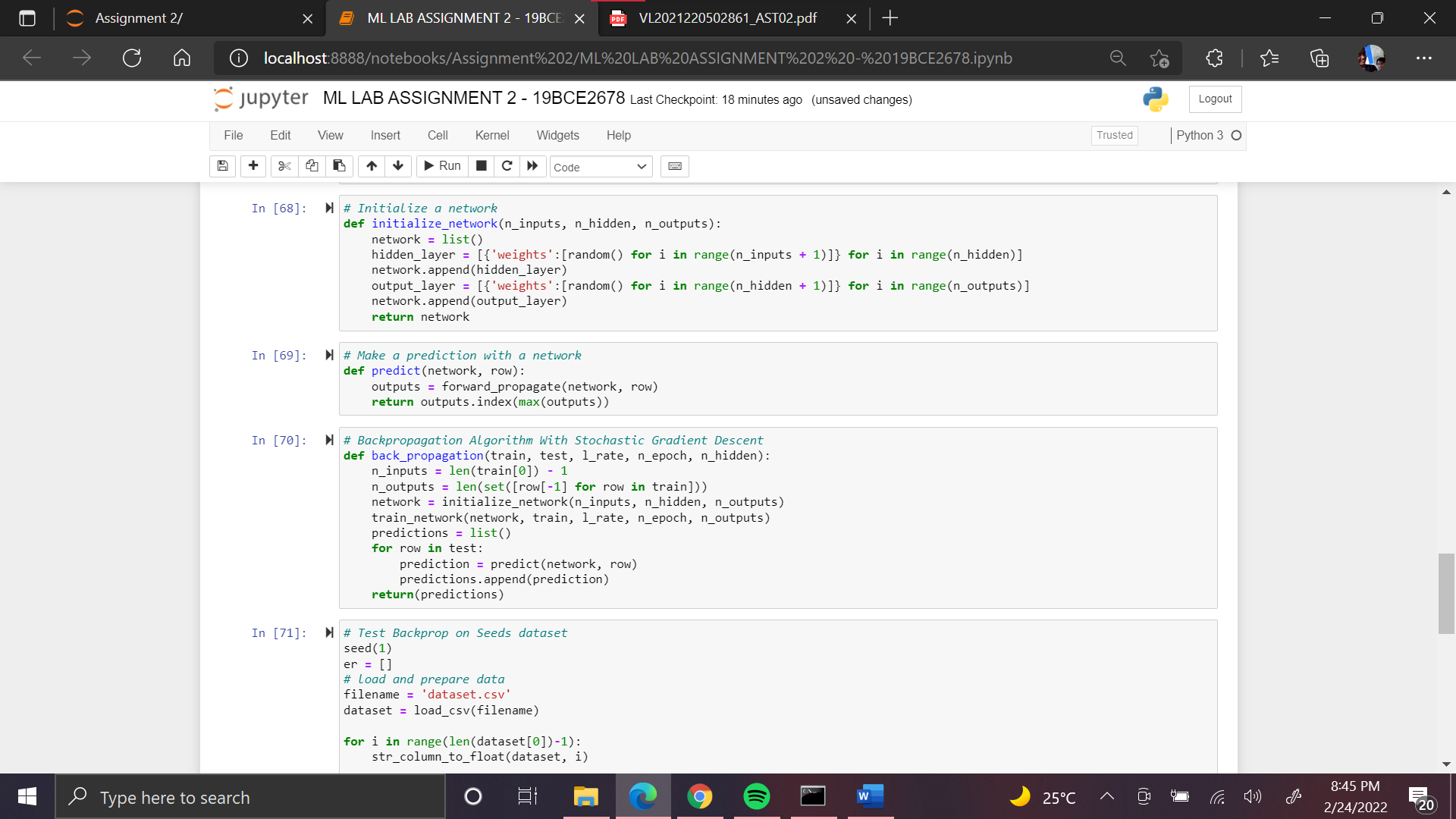
****

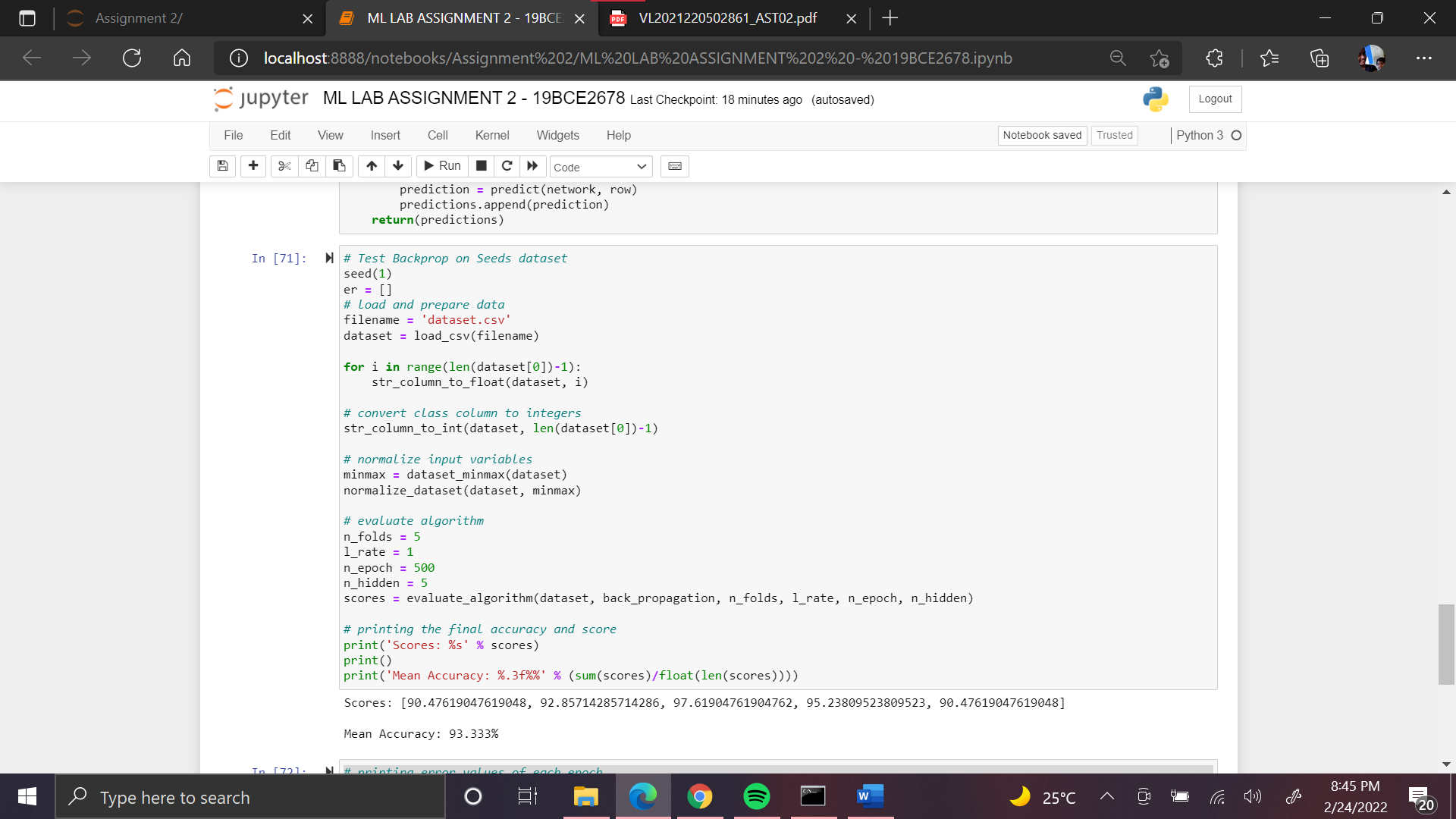
****

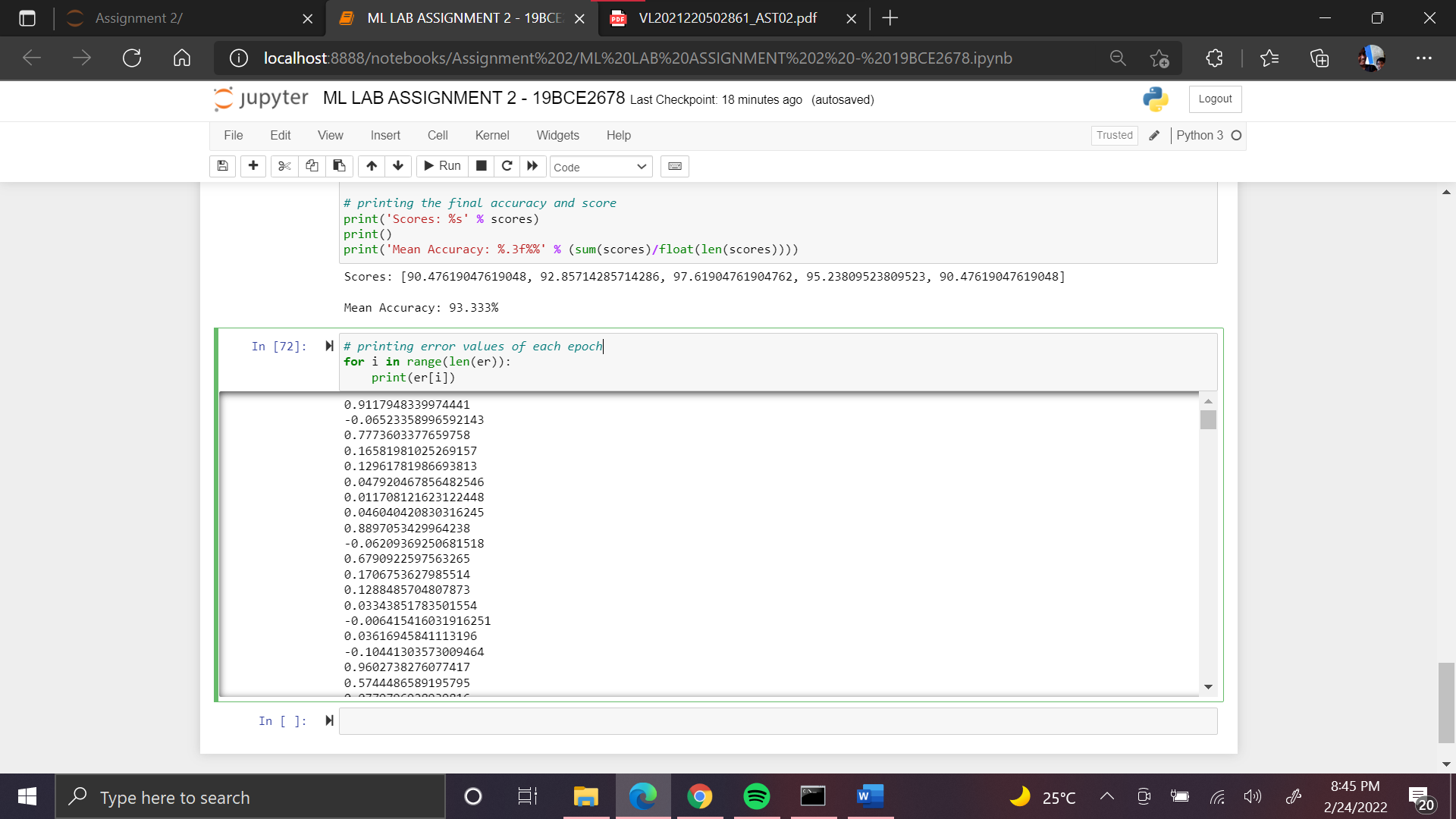
****

****

****

****

****

****