Lab -7

PRML
AY 2020-21 Trimester - III

Neural Network and Clustering **J Jahnavi (B19CSE109)**

Multi-Layer Perceptron using Backpropagation Algorithm

From Scratch

The Algorithm:

- Exploratory Data Analysis
  - Checking for missing values:

The number of missing values for each feature/column is as follows:

```
df.isnull().sum()
```

```
15.26    0
14.84    0
0.871    0
5.763    0
3.312    0
2.221    0
5.22     0
1        0
dtype: int64
```

The plot for missing data:

Missing value in the dataset

The above information implies that there are no missing values in the dataset.

- Distribution of the 3 output classes:

- Initialize Network

A function named initialize_network() creates a new neural network ready for training. It accepts three parameters: the number of inputs, the number of neurons to have in the hidden layer, and the number of outputs. The function is defined as follows:

```
def initialize_network(n_inputs, n_hidden, n_outputs):
  network = []
  hidden_lay = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
  network.append(hidden_lay)
  output_lay = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
  network.append(output_lay)
  return network
```

- Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. This is forward propagation which can be broken into the following parts:

1. Neuron Activation: to calculate the activation of one neuron given an input
2. Neuron Transfer: once a neuron is activated, we need to transfer the activation to see the neuron output
3. Forward Propagation: work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer

Calculate neuron activation for an input

```
[636] def activate(weights, inputs):
        activation = weights[-1]
        for i in range(len(weights)-1):
          activation += weights[i] * inputs[i]
        return activation
```

Transfer neuron activation

```
[637] def transfer(activation):
        return 1.0 / (1.0 + exp(-activation))
```

Forward propagate input to a network output

```
[638] def forward_propagate(network, row):
        inputs = row
        for layer in network:
          new_inputs = []
          for n in layer:
            activation = activate(n['weights'], inputs)
            n['output'] = transfer(activation)
            new_inputs.append(n['output'])
          inputs = new_inputs
        return inputs
```

- Back Propagate Error

This part is broken down into two sections.

1. Transfer Derivative
2. Error Backpropagation

Calculate the derivative of an neuron output

```
[639] def transfer_derivative(output):
        return output * (1.0 - output)
```

Backpropagate error and store in neurons

```
[640] def backward_propagate_error(network, expected):
        for i in reversed(range(len(network))):
          layer = network[i]
          errors = []
          if i != len(network)-1:
            for j in range(len(layer)):
              error = 0.0
              for n in network[i + 1]:
                error += (n['weights'][j] * n['delta'])
              errors.append(error)
          else:
            for j in range(len(layer)):
              n = layer[j]
              errors.append(expected[j] - n['output'])
          for j in range(len(layer)):
            n = layer[j]
            n['delta'] = errors[j] * transfer_derivative(n['output'])
```

- Train Network

This part is broken down into two sections:

1. Update Weights.
2. Train Network.

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 = [n['output'] for n in network[i - 1]]
    for n in network[i]:
      for j in range(len(inputs)):
        n['weights'][j] += l_rate * n['delta'] * inputs[j]
      n['weights'][-1] += l_rate * n['delta']
```

Train a network for a fixed number of epochs

```
[642] 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)
```

- Predict

Make a prediction with a network

```
[644] def predict(network, row):
          outputs = forward_propagate(network, row)
          return outputs.index(max(outputs))
```

- Backpropagation Algorithm With Stochastic Gradient Descent

Backpropagation Algorithm With Stochastic Gradient Descent

```
[645] 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 = []
        for row in test:
          prediction = predict(network, row)
          predictions.append(prediction)
        return (predictions)
```

- Evaluate the algorithm

Evaluate algorithm

```
[653] n_folds = 5
      l_rate = 0.1
      n_epoch = 500
      n_hidden = 5
      scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)
      mean_accuracy = sum(scores)/float(len(scores))
      print(f'Scores: {scores}')
      print(f'Mean Accuracy: {(mean_accuracy):.2f}%')

      Scores: [90.47619047619048, 92.85714285714286, 97.61904761904762, 95.23809523809523, 88.09523809523809]
      Mean Accuracy: 92.86%
```

Using Scikit Learn's in-built implementation

Using Scikit Learn's in-built implementation

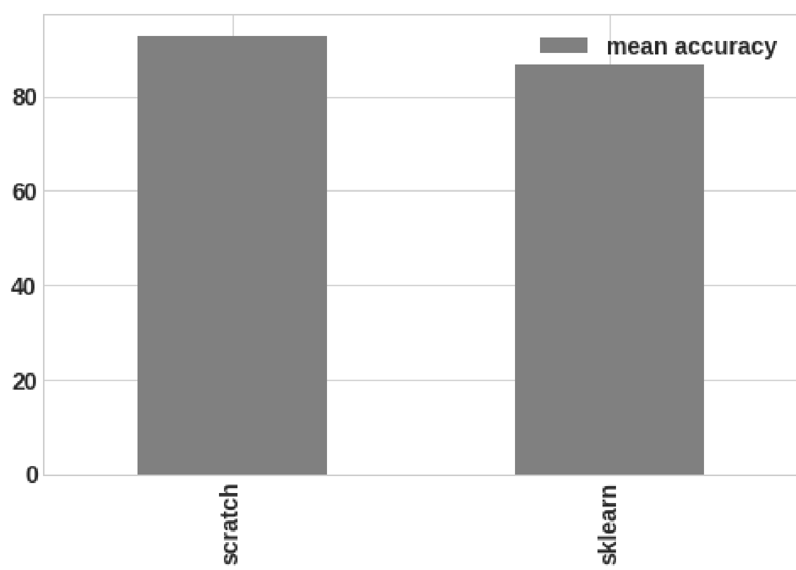
```
[654] from sklearn.neural_network import MLPClassifier
      from sklearn.datasets import make_classification
      from sklearn.model_selection import train_test_split
      X = df.drop(df.columns[-1],axis=1)
      y = df[df.columns[-1]]
      X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
      clf = MLPClassifier(random_state=42 , max_iter=500,learning_rate_init=0.1, hidden_layer_sizes=5).fit(X_train, y_train)
```

```
[655] from sklearn.model_selection import cross_val_score
      scores_scikit = cross_val_score(clf, X, y, cv=5)
      scores_scikit

      array([0.9047619 , 0.97619048, 0.80952381, 0.95238095, 0.53658537])
```

```
[656] mean_acc_scikit = clf.score(X_test, y_test)
      print(f'Mean Accuracy: {(mean_acc_scikit*100):.2f}%')

      Mean Accuracy: 86.79%
```

Comparison:

Model	Cross-validation scores	Mean Accuracy
From Scratch	[90.47619047619048, 92.85714285714286, 97.61904761904762, 95.23809523809523, 88.09523809523809]	92.86%
Using Scikit Learn	[90.47619 , 97.619048, 80.952381, 95.238095, 53.658537]	86.79%



K-Means Clustering

From Scratch

- Exploratory Data Analysis
    - Checking for missing values:

The number of missing values for each feature/column is as follows:

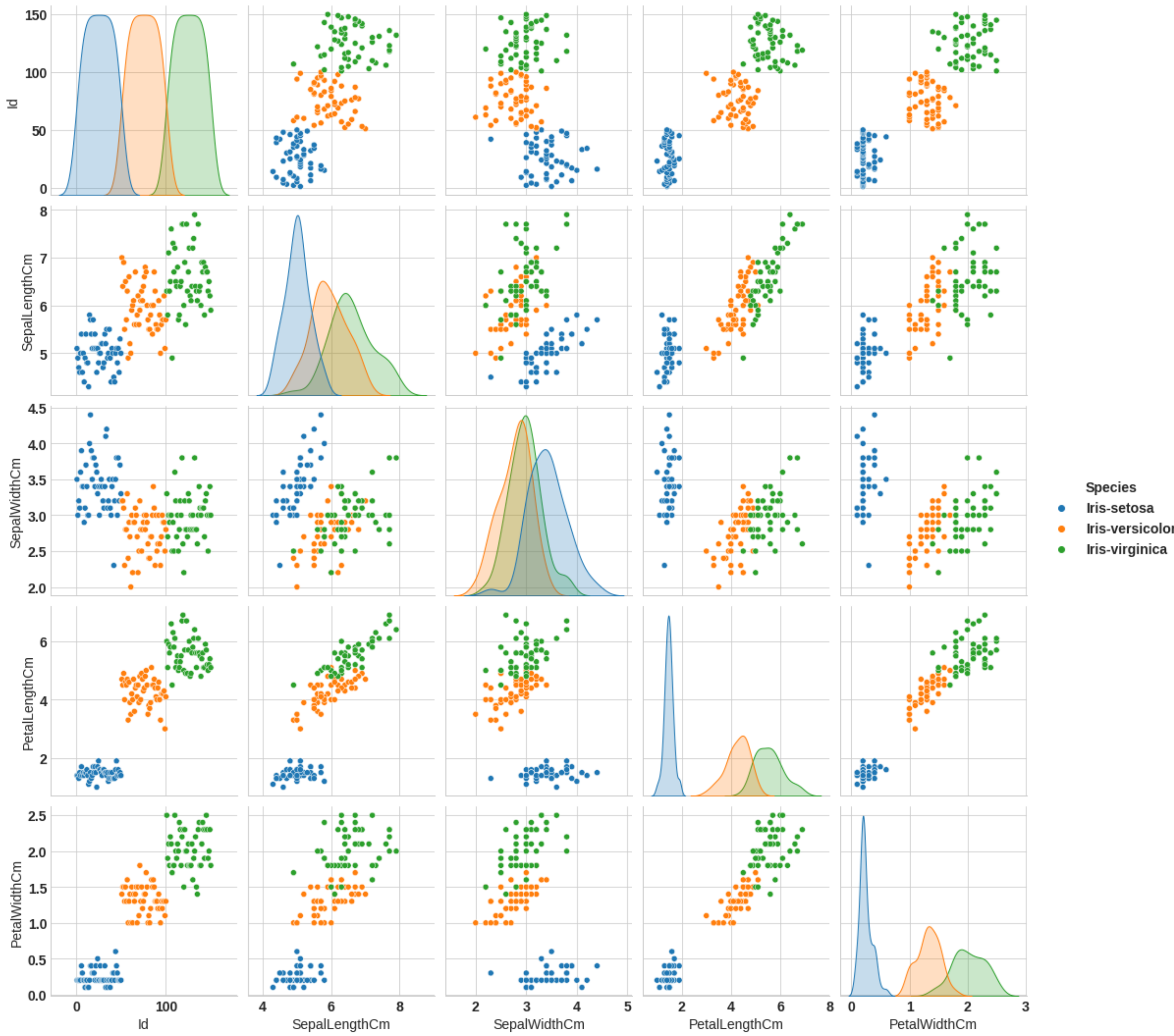
```
data.isnull().sum()

Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

The plot for missing data:

Missing value in the dataset

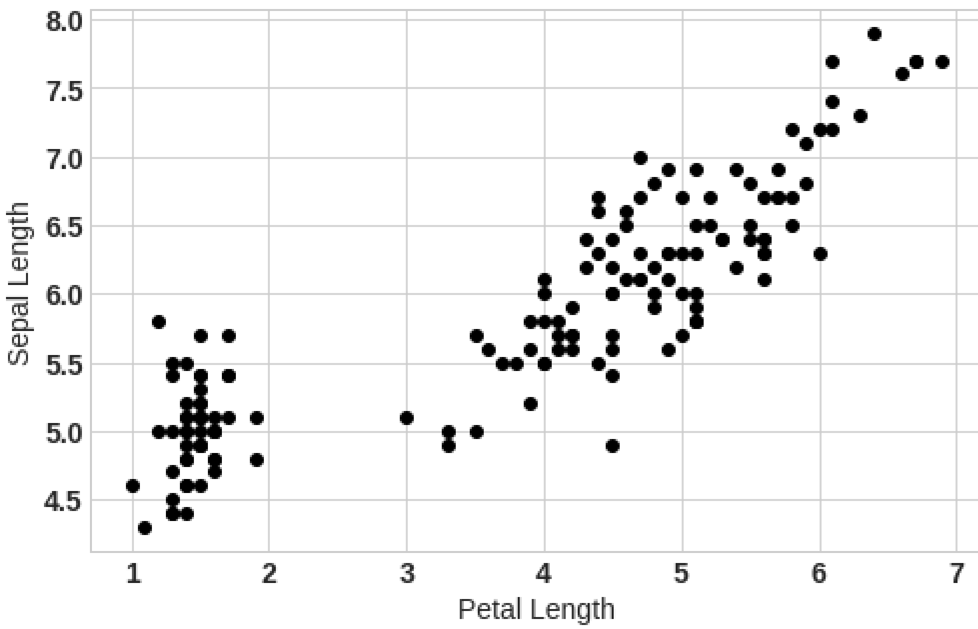
The above information implies that there are no missing values in the dataset.

    - Distribution of the 3 output classes:



- Visualize data points

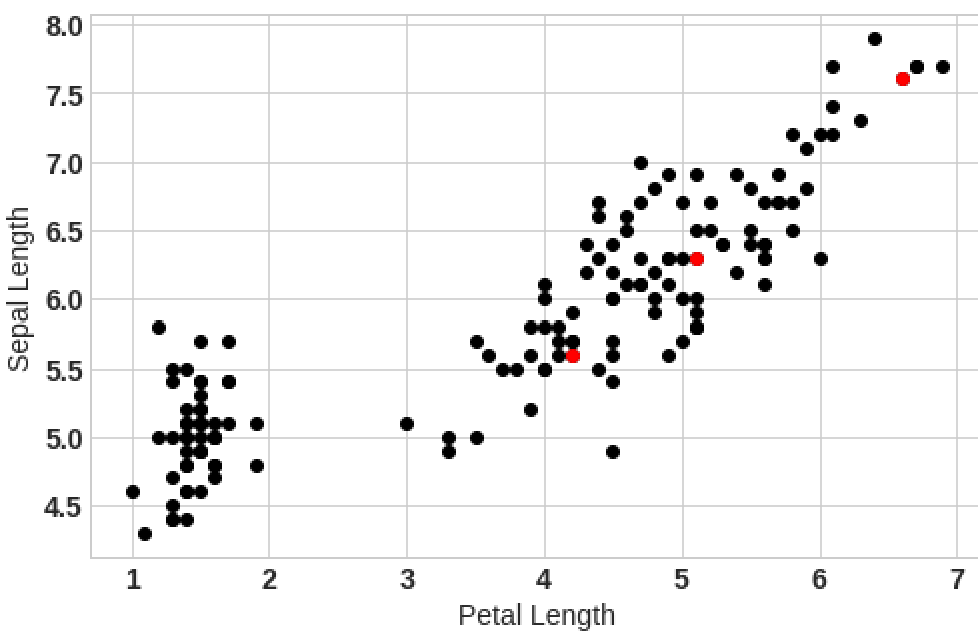
By taking only two variables from the data – “SepalLengthCm” and “PetalLengthCm”, we make it easier to visualize the steps as well



- Choose the number of clusters (k)

```
#number of clusters
K=3
```

- Selecting random centroid for each cluster



Here, the red points represent centroids for each cluster. We have chosen these points randomly

- Assign all the points to the closest cluster centroid and recompute centroids of newly formed clusters

  We do this on a loop until the centroids stop changing after each iteration.
- Visualize the clusters