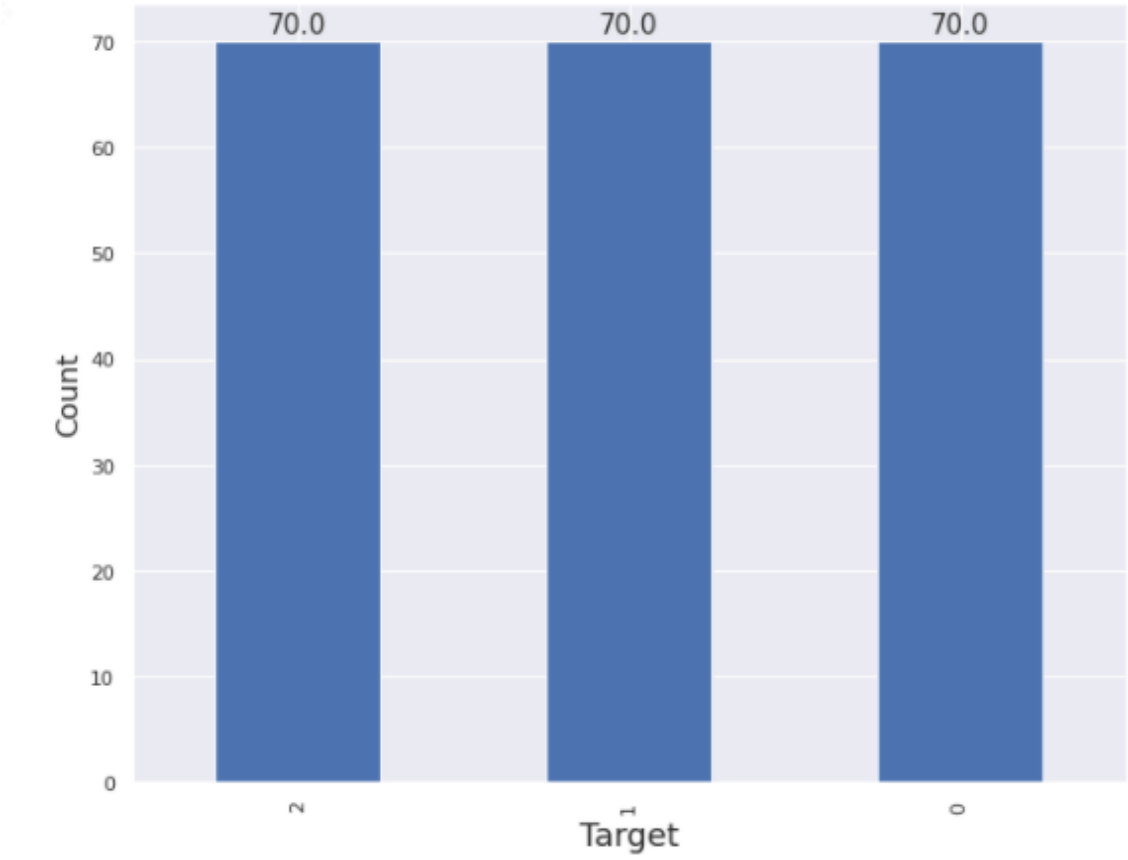
1.

We need to implement MLP from scratch for the wheat seeds dataset.
The dataset consists of 7 feature columns and 1 target column
The number of datapoints in the dataset are 210

Firstly we normalised the data using the code :

```
i=0
for col in df:
  if(i<7):
   df[col]=
df[col]-df[col].min()/(df[col].max()-df[col].min())
   i+=1
```

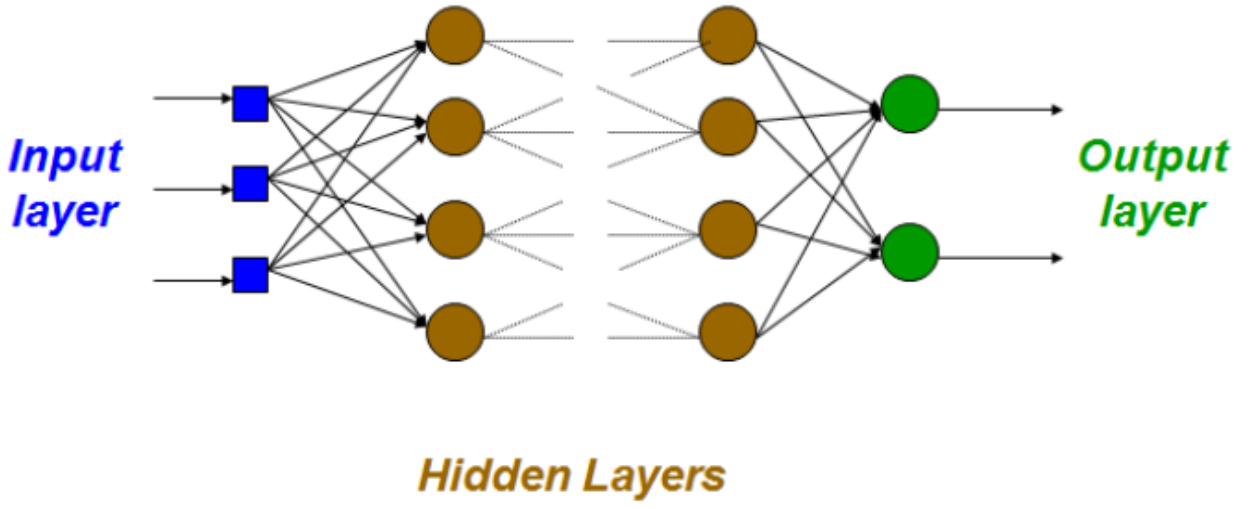
The plot showing the distribution of target variable is:



From the plot , it is clear that the data is uniformly distributed.

Now, we split the data into train-test using sklearn's train_test_split function. The code can be seen in the notebook

A neural network has one input layer, one output layer and hidden layers. The structure of a neural network is as follows . For our dataset , we have 7 inputs and 3 classes as output. So the neural network will consist of 7 neurons in the input layer and 3 neurons in the output layer.



To implement the neural network , we first need to initialise weights for each neuron . This can be done using :

```
.def init_net(n_inputs, n_outputs, hidden):
  network=[]
  feed= n_inputs
  for neuron in hidden:
    hid_layer = [{'wts':[random() for i in
range(feed+1)]} for i in range(neuron)]
    network.append(hid_layer)
```

```
    feed= neuron
  out_layer = [{'wts':[random() for i in
range(feed+1)]} for i in range(n_outputs)]
  network.append(out_layer)
  return network
```

The above function will store the weights and bias for each neuron in a dictionary with key name as `'wts'`

The output from the function will be of the type as given below depending on the parameters of the function.

```
[{'weights': [0.2550690257394217,
0.49543508709194095]}, {'weights':
[0.4494910647887381, 0.651592972722763]}]
```

After getting the weights, we need to multiply each weight with its corresponding feature and add it to the bias to get the output .
The below code does this:

```
def activate(wts, input):
 out= wts[-1]
 for i in range (len(wts)-1):
   out= out + wts[i]* input[i]
 return out
```

To introduce non-linearity to the output of neuron, we can use any activation function . Here we have used sigmoid activation function but we can also use relu or tanh as our activation functions

```
def sigmoid(out):
  sigm_out= 1.0 / (1.0 +exp((-1) * out))
  return sigm_out
```

Now since we have the weights corresponding to neurons for hidden and output layers , we need to get the output from our neural network which can be found using the below function:

```
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inp=[]
    for neuron in layer:
      out=activate(neuron['wts'],inputs)
      neuron['out']= sigmoid(out)
      new_inp.append(neuron['out'])
    inputs= new_inp
  return inputs
```

The output from the code above will be a list of outputs having same length as that of output layer

```
[0.6629970129852887, 0.7253160725279748]
```

Back-propagation is a way of propagating the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updating the weights in such a way that minimizes the loss.

```
def back_prop_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 neuron in network[i+1]:
          error= error+neuron['wts'][j]*
neuron['del']
        errors.append(error)
    else :
      for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j]-neuron['out'])
    for j in range(len(layer)):
      neuron = layer[j]
      neuron['del']=
errors[j]*sigm_der(neuron['out'])
```

To update the weights , we use :

```
  def update_weights(network, row, l_rate):
    for i in range(len(network)):
     inputs = row[:-1]
     if i != 0:
       inputs = [neuron['out'] for neuron in network[i -
  1]]
     for neuron in network[i]:
       for j in range(len(inputs)):
         neuron['wts'][j] = neuron['wts'][j]+ l_rate *
  neuron['del'] * inputs[j]
       neuron['wts'][-1] = neuron['wts'][-1]+ l_rate *
  neuron['del']
```

Given the feature values , to predict the output , we use :

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

The above function will forward_propagate the feature values to get output.

To train the neural network on given training data , we use the following function:

```
def train_network(network, train, l_rate, n_epoch,
n_outputs):
 for epoch in range(n_epoch):
   sum_error = 0
   for row in train:
     outputs = forward_propagate(network, row)
     expected = [0 for i in range(n_outputs)]
     expected[row[-1].astype(int)-1] = 1
     sum_error += sum([(expected[i]-outputs[i])**2 for
i in range(len(expected))])
     back_prop_error(network, expected)
     update_weights(network, row, l_rate)
   loss_value_scratch.append(sum_error/len(train))
   print('epoch=%d, lrate=%.3f, error=%.3f' % (epoch,
l_rate, sum_error/len(train)))
```

For each epoch, the function updates the weights after every subsequent steps of forward and backward propagation for each training data point.

To test the neural network on test data, we made a function called `test_network` which returns the accuracy after applying the neural network on test data .

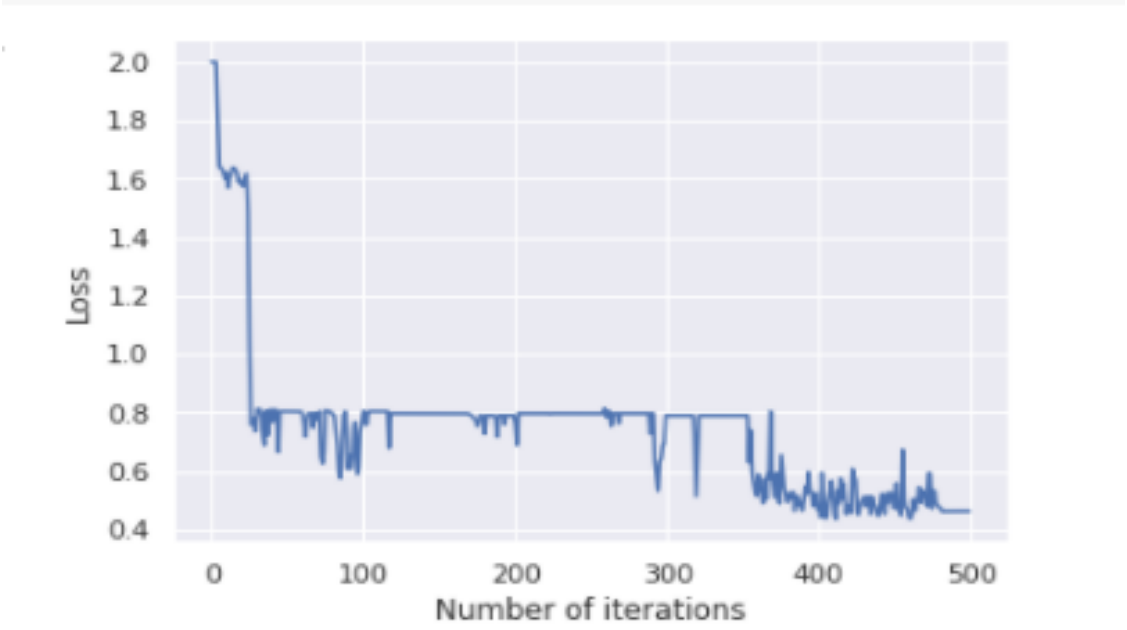
```
def test_network(train_data, test_data, n_epoch,
l_rate, hidden):
  predicted= back_propagation(train_data, test_data,
l_rate, n_epoch, hidden)
  actual = [row[-1] for row in test_data]
  accuracy = find_accuracy(actual, predicted)
  return accuracy
```

The above function takes train_data as one of the parameters. Using backpropagation algorithm, the neural network gets trained . Then we use the find_accuracy function to test the neural network on test_data and get the accuracy.

```
acc= test_network(train_df_ar, test_df_ar, 500, 0.5,
[20])
print(acc)
```

The above function will print accuracy for the test data . The number of epochs are 500 . The learning rate is 0.5 and there is a single hidden layer which consist of 20 neurons.

We realise that on using lesser learning rate such as 0.0001 , the loss decreases very slowly . On using high learning rates, the loss first decreases and then again starts increasing . This can be seen from the loss curve for learning rate of 0.5:



From above graph , we can see that the loss curve is not smooth. For example: we can see that there are points where first the loss decreases and then starts increasing again

The accuracy of the neural network built from scratch comes out to be `34.920 %`

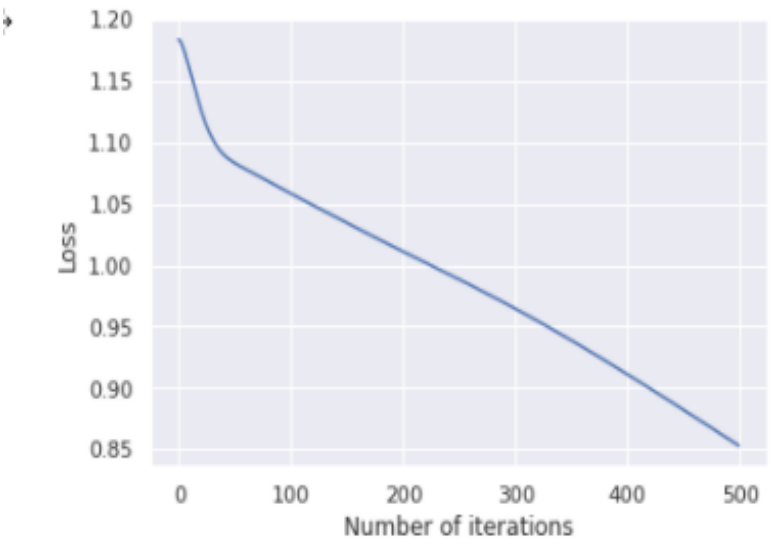
Whereas using the sklearn's built in neural network

```
clf = MLPClassifier(hidden_layer_sizes=(20,),
random_state=1,max_iter=500,activation='logistic',solv
er='sgd',power_t=0, tol=0).fit(X_train, y_train)
```

we get an accuracy of

`0.9523809523809523`

The loss curve for the function looks like this:



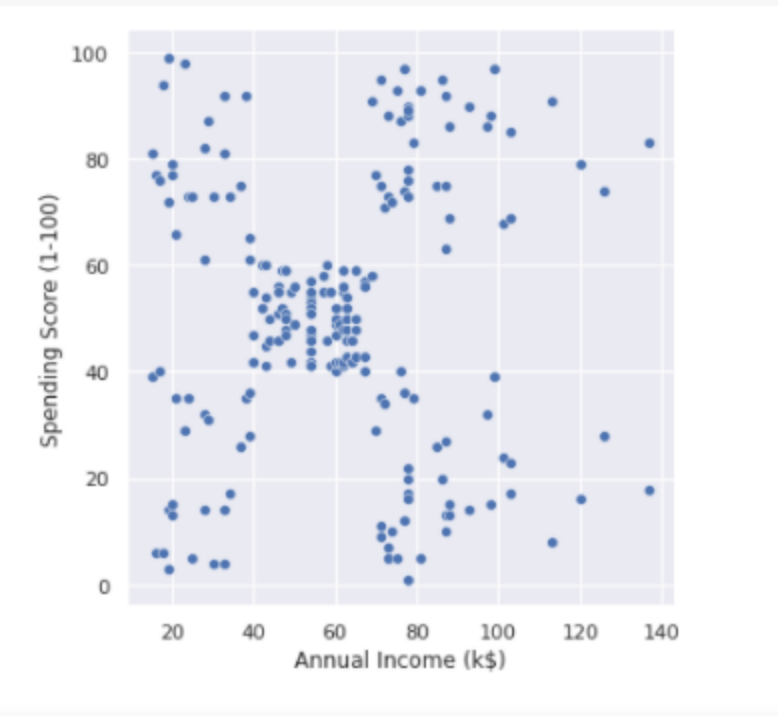
The curve is smooth and the loss keeps on decreasing even after 500 epochs

2.

The dataset used for this question can be found at :

https://www.kaggle.com/shwetabh123/mall-customers

We use 2 columns of this dataset for ease of plotting in clustering . Those columns are `"Annual Income (k$)","Spending Score (1-100)"`



The dataset consists of 200 data points .

The steps to perform k-means clustering are:

- Given *k*, the *k–means* algorithm works as follows:
  1) Randomly choose *k* data points (seeds) to be the initial centroids, cluster centers
  2) Assign each data point to the closest centroid
  3) Re–compute the centroids using the current cluster memberships.
  4) If a convergence criterion is not met, go to 2).

1.) To initialise centroids randomly :

```
def init_centroids(n_clusters):
 ind = random.sample(range(0, len(data)), n_clusters)
 centroids = []
 for i in ind:
    centroids.append(data.loc[i])
 centroids = np.array(centroids)
 return centroids
```

The above code gives an array of length n_clusters, consisting of centroids corresponding to each cluster .

2.) To find the nearest point corresponding to each cluster :

```
def findClosestCentroids(centroid, X):
    assigned_centroid = []
    for i in X:
        distance=[]
        for j in centroid:
            distance.append(calc_distance(i, j))
        assigned_centroid.append(np.argmin(distance))
    return assigned_centroid
```

The above code assigns each datapoint to the cluster with nearest distance from centroid.

3.) Once the clusters are formed , the following code will recompute the cluster centroids

```
def calc_centroids(clusters, X):
    new_centroids = []
```

```
        new_df = pd.concat([pd.DataFrame(X),
    pd.DataFrame(clusters, columns=['cluster'])],axis=1)
        for c in set(new_df['cluster']):
            current_cluster = new_df[new_df['cluster'] ==
    c][new_df.columns[:-1]]
            cluster_mean = current_cluster.mean(axis=0)
            new_centroids.append(cluster_mean)
        return new_centroids
```

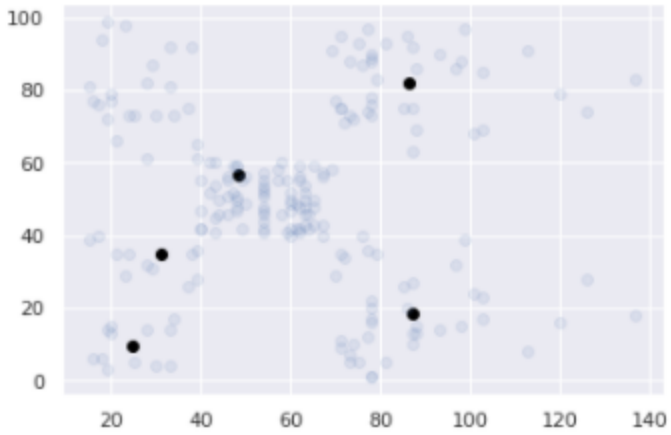
4.) **Here we will perform step 2 for a fixed number of times to get convergence**

The below code plots centroid of clusters formed for each iteration :

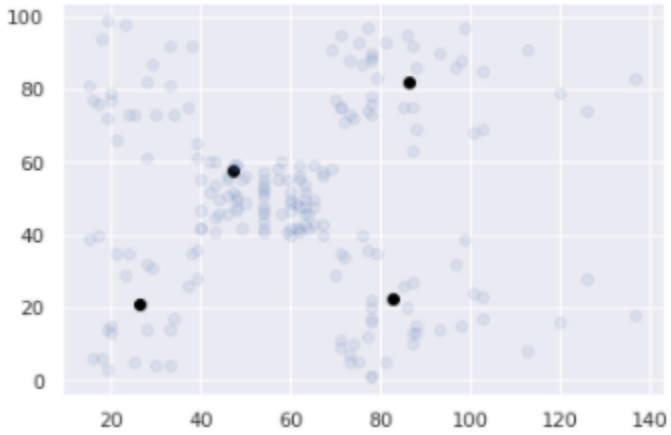
```
def show_clusters(n_clusters):
 centroids= init_centroids(n_clusters)
 for i in range(10):
    get_centroids = findClosestCentroids(centroids,
data_ar)
    centroids = calc_centroids(get_centroids, data_ar)
    plt.figure()
    plt.scatter(np.array(centroids)[:, 0],
np.array(centroids)[:, 1], color='black')
    plt.scatter(data_ar[:, 0], data_ar[:, 1], alpha=0.1)
    plt.show()
```

For num_clusters=5,4,3, the plots we are getting are :

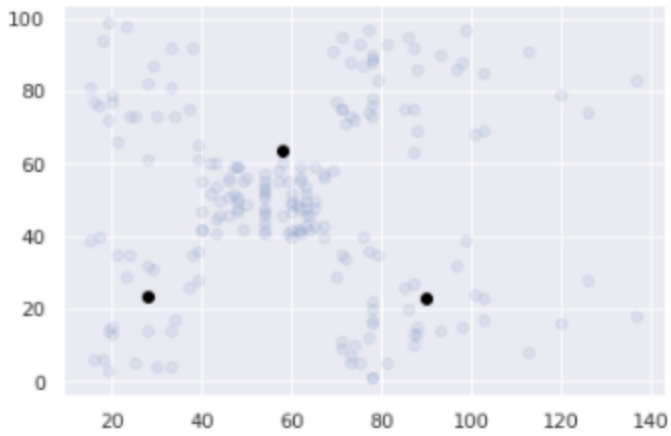
For num_clusters = 5:



For num_clusters= 4:

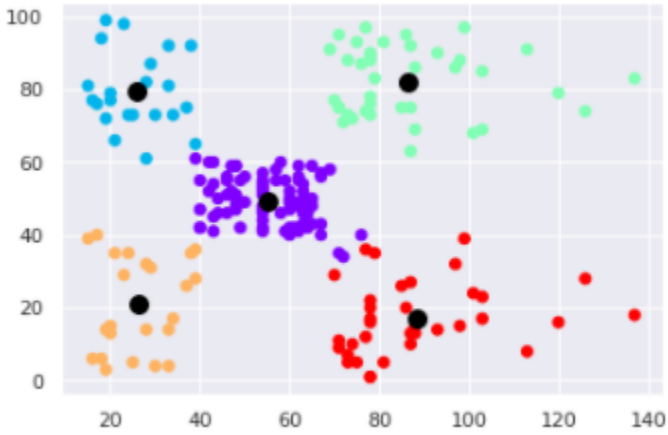


For num_clusters=3:

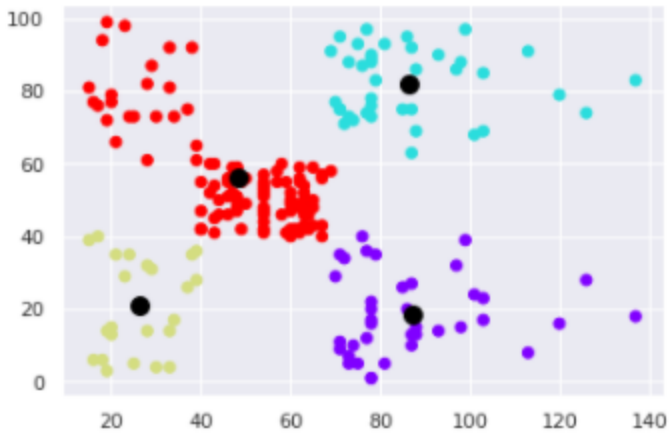


Now using sklearn's inbuilt routine :

For num_clusters=5:



For num_clusters=4:



For num_clusters=3: