Coding Assignment 2
Machine Learning
Dr. M. Shamsollahi

Mohammad Hossein Shafizadegan
99104781

January 3, 2024

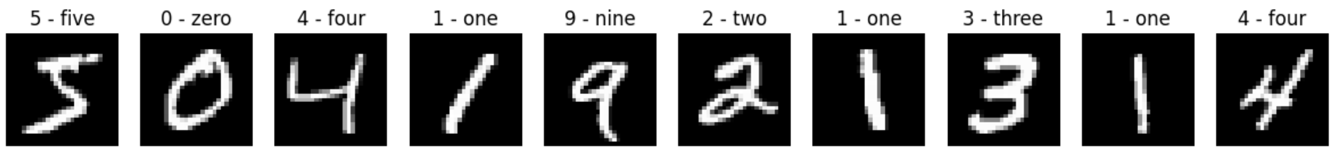
Contents

1 Question 1

1.1 Part1

First we load the MNIST dataset and visualize a sample of each class using the following code:

```python
# Step 1: Load the MNIST dataset and show an image from each class
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = MNIST(root='./data', train=False, download=True, transform=transform)

# Show an image from each class
classes = train_dataset.classes
figure, axes = plt.subplots(1, len(classes), figsize=(15, 15))
for i, ax in enumerate(axes):
  img, label = train_dataset[i]
  ax.imshow(img.squeeze().numpy(), cmap='gray')
  ax.set_title(classes[label])
  ax.axis('off')
plt.show()
```



Then we define the dataloaders for the train and test set as below:

```python
# Step 2: Define Dataloaders for the train and test sets
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Here we define our fully connected neural network including 4 layers by completing the ”FC_Neural_network” class. We note that for the ”forward” function, there is an argument which defines the type of the activation function used for the layers.

```python
class FC_Neural_network(nn.Module):
  def __init__(self):

    super(FC_Neural_network, self).__init__()
    self.fc1 = nn.Linear(784, 256)
    self.fc2 = nn.Linear(256, 128)
    self.fc3 = nn.Linear(128, 64)
    self.fc4 = nn.Linear(64, 10)

  def forward(self, x, act_func):

    x = torch.flatten(x, 1)
    x = self.fc1(x)

    if act_func == "sigmoid":
    x = torch.sigmoid(x)
    elif act_func == "relu":
    x = torch.relu(x)
    elif act_func == "Leaky relu":
    x = nn.functional.leaky_relu(x, negative_slope=0.01)

    x = self.fc2(x)
    x = self.fc3(x)
    x = self.fc4(x)

    return x
```

In order to calculate the accuracy for the sets, we have developed the following function:

```
# Accuracy calculation template
def calculate_accuracy(network, data_loader, device, act_func, criterion):
  network.eval()
  correct = 0
  total_loss = 0.0
  with torch.no_grad():
    for inputs, labels in data_loader:
      inputs, labels = inputs.to(device), labels.to(device)
      outputs = network(inputs, act_func)
      total_loss += criterion(outputs, labels).item() * inputs.size(0)
      _, predicted = torch.max(outputs.data, 1)
      correct += (predicted == labels).sum().item()
    loss = total_loss / len(data_loader.dataset)
    accuracy = correct / len(data_loader.dataset) * 100
    return loss, accuracy
```

The ‘torch.no_grad()‘ context manager is used here to optimize memory and computational efficiency during the evaluation phase of a neural network. It indicates that gradient calculations and storage are not needed because we are not training the network at this stage. By disabling gradient computation, we can save resources and make the evaluation process faster. This context manager ensures that no gradients are computed or stored during the evaluation, allowing for efficient execution of the code.

Then we have to complete the ‘train()‘ function. It can be fully observed in the attached notebook. Then we use this function and train neural networks with different activation functions and print the results as follows:

```
activation_functions = ["sigmoid", "relu", "Leaky relu"]

for act_func in activation_functions:

  print(f"*** NN with {act_func} activation function: ***")
  network = FC_Neural_network().to(device)
  optimizer = torch.optim.SGD(network.parameters(), lr=0.01, momentum=0.9)
  criterion = nn.CrossEntropyLoss()

  train(network, train_loader, test_loader, optimizer, criterion, device, act_func)
```

In the jupyter notebook, It can be seen that our networks have achieved good accuracy.

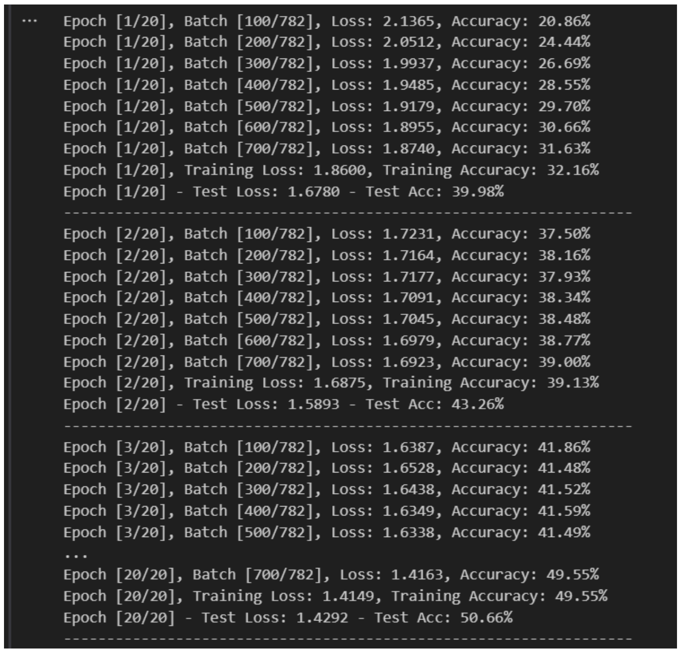
CIFAR10 Dataset

Now we train new fully connected neural network using the CIFAR10 dataset. First we have to properly load it.

```
# Set random seed for reproducibility
torch.manual_seed(42)
```

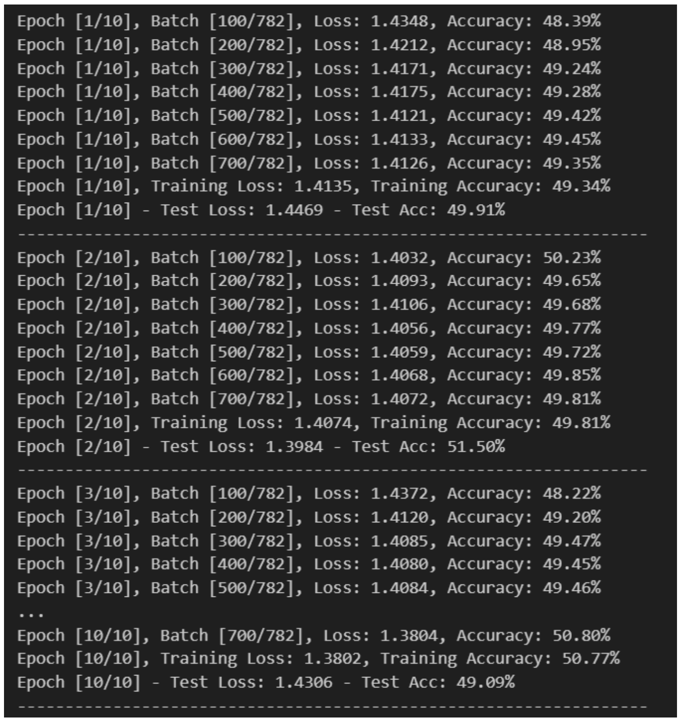
```python

# Data preprocessing and augmentation
transform_train = transforms.Compose([
  transforms.RandomCrop(32, padding=4),
  transforms.RandomHorizontalFlip(),
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_val = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
trainset = CIFAR10(root='./data', train=True, download=True, transform=transform_train)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

valset = CIFAR10(root='./data', train=False, download=True, transform=transform_val)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=False)
```

First we train the network for 20 epochs. It can be seen that the accuracy of the model after the first epoch is 39.98% which is relatively low. The point is that even after 20 epochs the performance of the model is still quite poor. The accuracy after 20 epochs is about 50 %.



```
...  Epoch [1/20], Batch [100/782], Loss: 2.1365, Accuracy: 20.86%
     Epoch [1/20], Batch [200/782], Loss: 2.0512, Accuracy: 24.44%
     Epoch [1/20], Batch [300/782], Loss: 1.9937, Accuracy: 26.69%
     Epoch [1/20], Batch [400/782], Loss: 1.9485, Accuracy: 28.55%
     Epoch [1/20], Batch [500/782], Loss: 1.9179, Accuracy: 29.70%
     Epoch [1/20], Batch [600/782], Loss: 1.8955, Accuracy: 30.66%
     Epoch [1/20], Batch [700/782], Loss: 1.8740, Accuracy: 31.63%
     Epoch [1/20], Training Loss: 1.8600, Training Accuracy: 32.16%
     Epoch [1/20] - Test Loss: 1.6780 - Test Acc: 39.98%
     ------------------------------------------------------------
     Epoch [2/20], Batch [100/782], Loss: 1.7231, Accuracy: 37.50%
     Epoch [2/20], Batch [200/782], Loss: 1.7164, Accuracy: 38.16%
     Epoch [2/20], Batch [300/782], Loss: 1.7177, Accuracy: 37.93%
     Epoch [2/20], Batch [400/782], Loss: 1.7091, Accuracy: 38.34%
     Epoch [2/20], Batch [500/782], Loss: 1.7045, Accuracy: 38.48%
     Epoch [2/20], Batch [600/782], Loss: 1.6979, Accuracy: 38.77%
     Epoch [2/20], Batch [700/782], Loss: 1.6923, Accuracy: 39.00%
     Epoch [2/20], Training Loss: 1.6875, Training Accuracy: 39.13%
     Epoch [2/20] - Test Loss: 1.5893 - Test Acc: 43.26%
     ------------------------------------------------------------
     Epoch [3/20], Batch [100/782], Loss: 1.6387, Accuracy: 41.86%
     Epoch [3/20], Batch [200/782], Loss: 1.6528, Accuracy: 41.48%
     Epoch [3/20], Batch [300/782], Loss: 1.6438, Accuracy: 41.52%
     Epoch [3/20], Batch [400/782], Loss: 1.6349, Accuracy: 41.59%
     Epoch [3/20], Batch [500/782], Loss: 1.6338, Accuracy: 41.49%
     ...
     Epoch [20/20], Batch [700/782], Loss: 1.4163, Accuracy: 49.55%
     Epoch [20/20], Training Loss: 1.4149, Training Accuracy: 49.55%
     Epoch [20/20] - Test Loss: 1.4292 - Test Acc: 50.66%
     ------------------------------------------------------------
```

Now we continue training the model for 10 more epochs. Based on the figure below, There is no enhancement in the accuracy of the model even the fact that the model has experienced learning following new epochs.



```
Epoch [1/10], Batch [100/782], Loss: 1.4348, Accuracy: 48.39%
Epoch [1/10], Batch [200/782], Loss: 1.4212, Accuracy: 48.95%
Epoch [1/10], Batch [300/782], Loss: 1.4171, Accuracy: 49.24%
Epoch [1/10], Batch [400/782], Loss: 1.4175, Accuracy: 49.28%
Epoch [1/10], Batch [500/782], Loss: 1.4121, Accuracy: 49.42%
Epoch [1/10], Batch [600/782], Loss: 1.4133, Accuracy: 49.45%
Epoch [1/10], Batch [700/782], Loss: 1.4126, Accuracy: 49.35%
Epoch [1/10], Training Loss: 1.4135, Training Accuracy: 49.34%
Epoch [1/10] - Test Loss: 1.4469 - Test Acc: 49.91%
------------------------------------------------------------------
Epoch [2/10], Batch [100/782], Loss: 1.4032, Accuracy: 50.23%
Epoch [2/10], Batch [200/782], Loss: 1.4093, Accuracy: 49.65%
Epoch [2/10], Batch [300/782], Loss: 1.4106, Accuracy: 49.68%
Epoch [2/10], Batch [400/782], Loss: 1.4056, Accuracy: 49.77%
Epoch [2/10], Batch [500/782], Loss: 1.4059, Accuracy: 49.72%
Epoch [2/10], Batch [600/782], Loss: 1.4068, Accuracy: 49.85%
Epoch [2/10], Batch [700/782], Loss: 1.4072, Accuracy: 49.81%
Epoch [2/10], Training Loss: 1.4074, Training Accuracy: 49.81%
Epoch [2/10] - Test Loss: 1.3984 - Test Acc: 51.50%
------------------------------------------------------------------
Epoch [3/10], Batch [100/782], Loss: 1.4372, Accuracy: 48.22%
Epoch [3/10], Batch [200/782], Loss: 1.4120, Accuracy: 49.20%
Epoch [3/10], Batch [300/782], Loss: 1.4085, Accuracy: 49.47%
Epoch [3/10], Batch [400/782], Loss: 1.4080, Accuracy: 49.45%
Epoch [3/10], Batch [500/782], Loss: 1.4084, Accuracy: 49.46%
...
Epoch [10/10], Batch [700/782], Loss: 1.3804, Accuracy: 50.80%
Epoch [10/10], Training Loss: 1.3802, Training Accuracy: 50.77%
Epoch [10/10] - Test Loss: 1.4306 - Test Acc: 49.09%
------------------------------------------------------------------
```

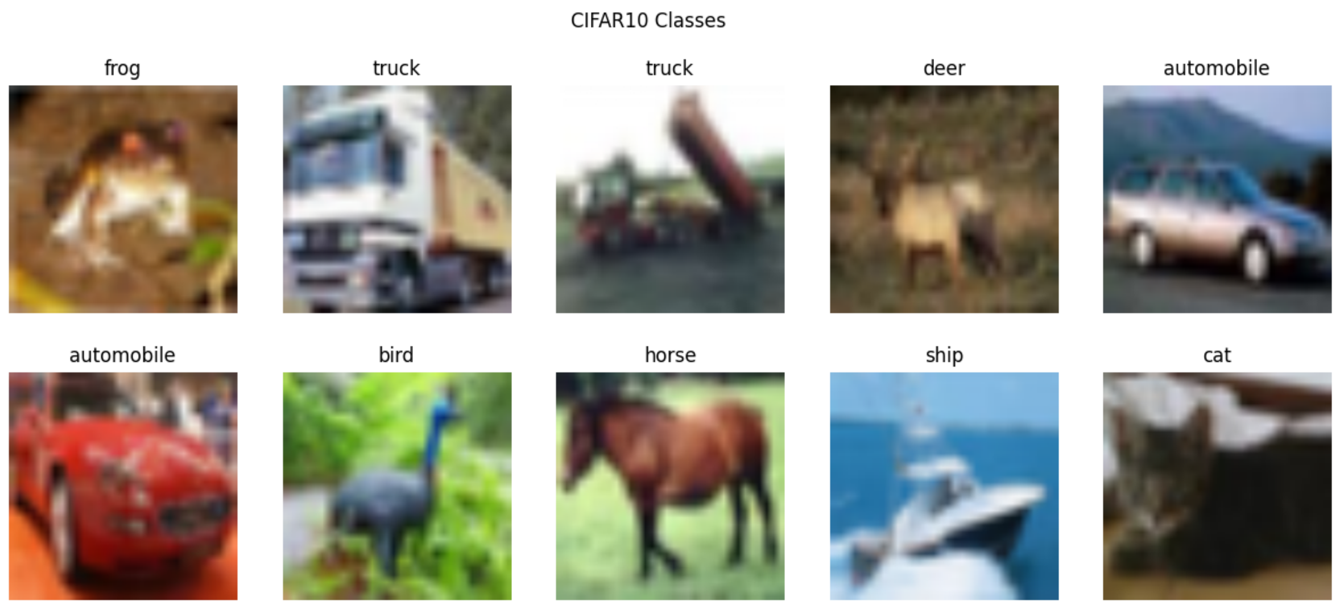
Indeed this section prove that the fully connected neural networks don't work well all the times and for some tasks and data, the convolutional neural networks are the solution.

1.2 Part 2: Transfer learning for image classification

Firs we load the pretrained alexnet model and the CIFAR10 dataset properly

```
    # Step 1: Load pre-trained AlexNet with predefined weights
    model = alexnet(pretrained=True)

    # Step 2: Load CIFAR10 dataset and plot an image of each class
    transform = transforms.Compose([
      transforms.Resize((224, 224)),
      transforms.ToTensor(),
    ])

    trainset = CIFAR10(root='./data', train=True, download=True, transform=transform)
    testset = CIFAR10(root='./data', train=False, download=True, transform=transform)
```

Then we visualize a sample data of each class:



In order to reduce the training load and required time, we only use two classes of the dataset for the training process. This can be done using the following code:

```
    # Use only two classes, to reduce the computational load
    class_labels = [0, 4] # Replace with the indices of the classes you want to use

    # Filter the dataset to include only the specified classes
    train_indices = [i for i in range(len(trainset)) if trainset.targets[i] in class_labels]
    test_indices = [i for i in range(len(testset)) if testset.targets[i] in class_labels]

    # Create custom datasets with only the selected classes
    filtered_train_dataset = torch.utils.data.Subset(trainset, train_indices)
    filtered_test_dataset = torch.utils.data.Subset(testset, test_indices)
```

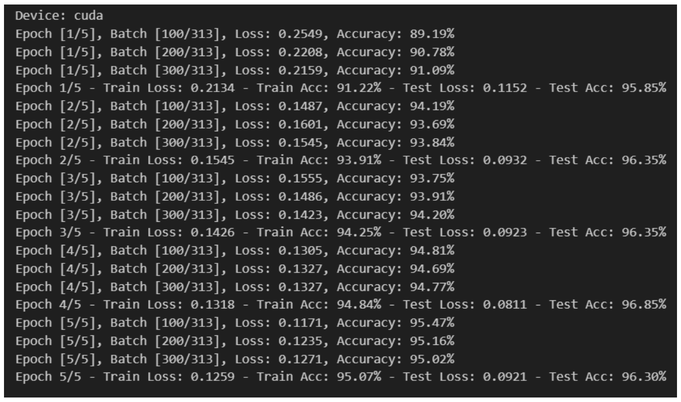
Then using the code below, in order to fine tune the model for our classification task, we change the fully connected layer:

```python
    # Step 4: Change fully connected layers for the classification task
    num_features = model.classifier[6].in_features
    model.classifier[6] = nn.Linear(num_features, 2)
```

We remember to freeze the convolutional layer using the code below:

```python
    # Reset the weights of the fully-connected layers, but not the convolutional layers
    for param in model.features.parameters():
      param.requires_grad = False
```

Using a quite same process, we train the model, define the loss function and print the accuracy and results. The code can be seen vividly in the jupyter notebook.



```
Device: cuda
Epoch [1/5], Batch [100/313], Loss: 0.2549, Accuracy: 89.19%
Epoch [1/5], Batch [200/313], Loss: 0.2208, Accuracy: 90.78%
Epoch [1/5], Batch [300/313], Loss: 0.2159, Accuracy: 91.09%
Epoch 1/5 - Train Loss: 0.2134 - Train Acc: 91.22% - Test Loss: 0.1152 - Test Acc: 95.85%
Epoch [2/5], Batch [100/313], Loss: 0.1487, Accuracy: 94.19%
Epoch [2/5], Batch [200/313], Loss: 0.1601, Accuracy: 93.69%
Epoch [2/5], Batch [300/313], Loss: 0.1545, Accuracy: 93.84%
Epoch 2/5 - Train Loss: 0.1545 - Train Acc: 93.91% - Test Loss: 0.0932 - Test Acc: 96.35%
Epoch [3/5], Batch [100/313], Loss: 0.1555, Accuracy: 93.75%
Epoch [3/5], Batch [200/313], Loss: 0.1486, Accuracy: 93.91%
Epoch [3/5], Batch [300/313], Loss: 0.1423, Accuracy: 94.20%
Epoch 3/5 - Train Loss: 0.1426 - Train Acc: 94.25% - Test Loss: 0.0923 - Test Acc: 96.35%
Epoch [4/5], Batch [100/313], Loss: 0.1305, Accuracy: 94.81%
Epoch [4/5], Batch [200/313], Loss: 0.1327, Accuracy: 94.69%
Epoch [4/5], Batch [300/313], Loss: 0.1327, Accuracy: 94.77%
Epoch 4/5 - Train Loss: 0.1318 - Train Acc: 94.84% - Test Loss: 0.0811 - Test Acc: 96.85%
Epoch [5/5], Batch [100/313], Loss: 0.1171, Accuracy: 95.47%
Epoch [5/5], Batch [200/313], Loss: 0.1235, Accuracy: 95.16%
Epoch [5/5], Batch [300/313], Loss: 0.1271, Accuracy: 95.02%
Epoch 5/5 - Train Loss: 0.1259 - Train Acc: 95.07% - Test Loss: 0.0921 - Test Acc: 96.30%
```

Answer to the questions

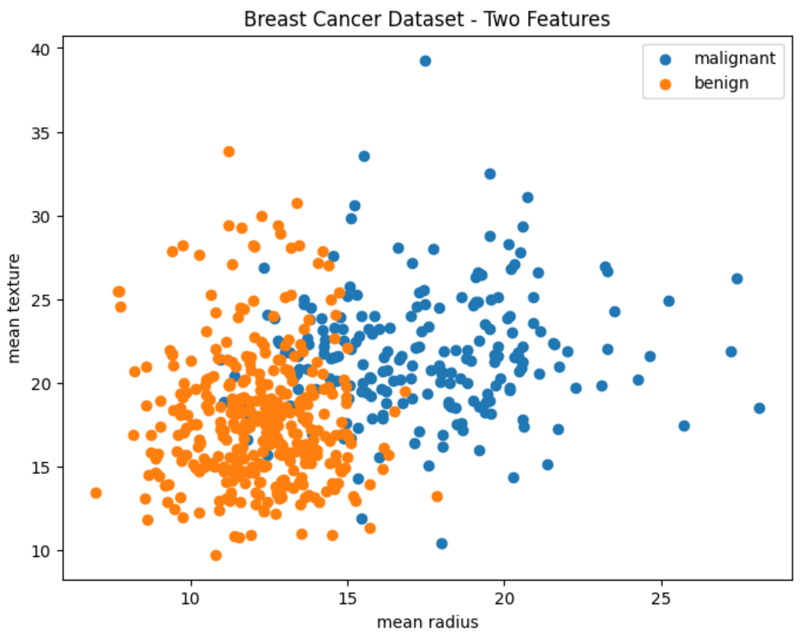
- The '.train()' function sets the model in training mode, enabling gradient computation and updating of trainable parameters.
- The '.eval()' function sets the model in evaluation mode, disabling gradient computation and freezing the parameters.
- In this example, it is important to use '.eval()' during evaluation to ensure that the model behaves consistently and does not update its parameters based on the evaluation data.
- Layers affected by '.train()' and '.eval()' functions in AlexNet:
  - Dropout layers: Affected by '.train()' to apply dropout regularization during training.
  - Batch Normalization layers: Affected by '.train()' to track running statistics and apply normalization during training.

2 Question 2

First we load the breast cancer dataset and visualize the data considering features one and two. Here is code developed for this section:

```python
    # Load the Breast Cancer dataset
    data = load_breast_cancer()

    # Get the selected features and labels
    X = data.data[:, [feature1_index, feature2_index]]
    y = data.target

    # Get the class labels
    target_names = data.target_names

    for class_value in np.unique(y):
      # Get the indices of samples belonging to the current class
      indices = np.where(y == class_value)

      # Plot the samples of the current class
      plt.scatter(X[indices, 0], X[indices, 1], label=target_names[class_value])
```

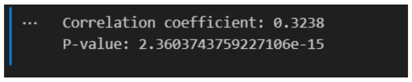
The resulted figure can be seen below:



In order to check whether and how much do these features are correlated to each other, we have calculated the correlation coefficients and the corresponding p value using the built in function ”pearsonr” from scipy.stats. Here is the code:

```
    from scipy.stats import pearsonr

    # Calculate the correlation coefficient
    correlation_coef, p_value = pearsonr(X[:, 0], X[:, 1])

    print(f"Correlation coefficient: {correlation_coef:.4f}")
    print(f"P-value: {p_value}")
```

It can be seen that the correlation coefficient for features 1 and 2 are relatively small therefor, these features are not well correlated. We also note the significantly small p value which verify our result quite well.



```
Correlation coefficient: 0.3238
P-value: 2.3603743759227106e-15
```

After splitting data into train and test set using ”train_test_split()” builtin function, we will design our decision tree model using the following code:

```python
from sklearn.tree import DecisionTreeClassifier

# Create a decision tree classifier
clf = DecisionTreeClassifier(criterion=splitter, max_depth=depth, random_state=42)

# Fit the model on the training data
clf.fit(X_train, y_train)
```

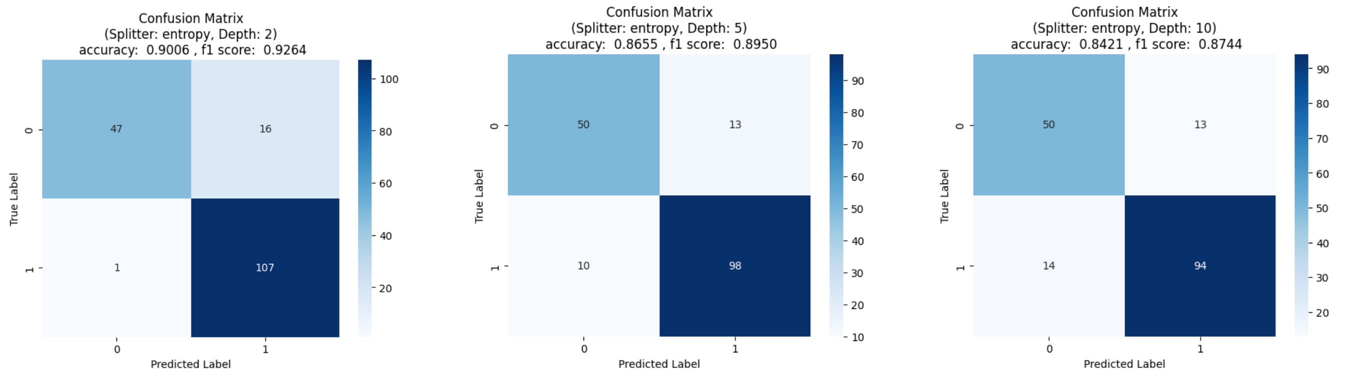
Then using this model, we simply predict the outputs for the test data. For assessing the performance of the model, we will report the f1 score, accuracy and the corresponding confusion matrix:

```python
# Predict the labels for the test data
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

Finally in a loop, we use this code for different tree depth, then we print and visualize the results as well.

```python
splitter = 'entropy'
depths = [2, 5, 10]

# Fit and evaluate the decision tree for each depth
for depth in depths:
  # Create a decision tree classifier
  clf = DecisionTreeClassifier(criterion=splitter, max_depth=depth, random_state=42)
```

The results can be seen below:



It can be seen that as we increase the depth of the tree the accuracy will decrease regarding the concept of overfitting.

The accuracy for other algorithms we used before is a bit higher as KNN and SVM classifiers achieved accuracy of 96.5 % but the best accuracy for decision tree is 90 %. We should note that we have to assess other measures like f1 score or sensitivity too.