Project Report

COMP 472 - Introduction to Artificial Intelligence - Fall 2024
Concordia University, Montreal QC

Instructor:	Ali Ayub
Due:	November 25th, 2024
Prepared by:	Matei Razvan Garila - 40131709
	Ralph Elhage - 40131981

We certify that this submission is our original work and meets the Faculty's Expectations of Originality

TABLE OF CONTENTS

Project Introduction

For this project, we are tasked to perform image classification on the CIFAR-10 dataset using some of the models we have learned in class; these models are: Naive Bayes, Decision Tree, Multi-Layer Perceptron (MLP), and Convolutional Neural Network (CNN). Our goals are to detect the 10 object classes in the CIFAR-10 dataset, then to apply basic evaluation metrics to these models (accuracy, confusion matrix, precision, and recall).

Dataset Overview

As stated in the introduction, we will be using the CIFAR-10 dataset for this project. The dataset contains 50,000 training and 10,000 test RGB images belonging to 10 classes; the images are of size 32 x 32 x 3. For our purposes, we will only be using 500 training and 100 test images. Since Naive Bayes, Decision Trees, and MLPs are not well-suited for this dataset, we will be using a pre-trained ResNet-18 CNN to extract the necessary feature vectors from the RGB images. Lastly, we have to use scikit's PCA to further reduce the size of the feature vector. Here is how we extracted the data we are going to use for the first 3 models:

Step 0

The initial step is to define the device we are going to use. In our case we are using torch.cuda if it is available, else we will be doing our calculations with the CPU

```
10  # Step 0: Define the device you are going to use
11  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
12  print(f"Using device: {device}")
```

Step 1

The first step is to define the transformation we are going to use. We need to resize the images to 224 x 224, followed by converting the images to PyTorch, then normalize them as per ImageNet.

```
14  # Step 1: Define the transformations
15  # Resize images to 224x224 (as required for ResNet-18) and normalize them
16  transform = transforms.Compose([
17      transforms.Resize((224, 224)),  # Resize images to 224x224
18      transforms.ToTensor(),  # Convert images to PyTorch tensors
19      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))  # Normalize images as per ImageNet
20  ])
21  print("Done defining transformation")
```

Step 2

The second step is to load the CIFAR-10 dataset with the specified transformations.

```python
# Step 2: Load the CIFAR-10 dataset with specified transformations
train_data = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform
)
test_data = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform
)
print("Done loading initial data")
```

Step 3

Following that, we then have to create a subset of 500 training and 100 test images.

```python
# Step 3: Create 500 training and 100 test image subsets
def subset_cifar10(dataset, num_per_class):
    class_counts = {i: 0 for i in range(10)}  # Counter for each class
    indices = []

    for idx, (_, label) in enumerate(dataset):
        if class_counts[label] < num_per_class:
            indices.append(idx)
            class_counts[label] += 1
        if all(count >= num_per_class for count in class_counts.values()):
            break

    # Return a subset of the original dataset
    subset = torch.utils.data.Subset(dataset, indices)
    return subset


# Subset the training and testing datasets
train_subset = subset_cifar10(train_data, 500)  # 500 images per class for training
test_subset = subset_cifar10(test_data, 100)    # 100 images per class for testing
print("Done subset training and test data")
```

Step 4

After creating the subsets, we then have to load the pre-trained ResNet-18 model, set it to eval mode, and move the model to the GPU. We use the GPU because they are designed for parallel processing, and their architecture efficiently manages the heavy computational loads that are required.

```
# Step 4: Load pre-trained ResNet-18 model
resnet18 = models.resnet18(pretrained=True)
# Remove the last layer to use ResNet as a feature extractor
resnet18 = nn.Sequential(*list(resnet18.children())[:-1])
resnet18.eval()  # Set to evaluation mode

# Move the model to GPU
resnet18 = resnet18.to(device)
print("Done loading and setting pre-trained model")
```

Step 5

For step 5, we use DataLoader to manage batching and shuffling of the data, which makes it efficient for training.

```
# Step 5: Create DataLoaders for batch processing
train_loader = DataLoader(train_subset, batch_size=16, shuffle=False)
test_loader = DataLoader(test_subset, batch_size=16, shuffle=False)
print("Done creating loaders")
```

Step 6

Step 6 is all about extracting the features vectors from the DataLoader.

```
# Function to extract feature vectors
def extract_features(loader):
    features = []
    labels = []
    with torch.no_grad():  # Don't calculate gradients, we just want features
        for images, label in loader:
            # Move images and labels to GPU
            images = images.to(device)
            output = resnet18(images)
            output = output.view(output.size(0), -1)  # Flatten the output

            # Move output and labels back to CPU before converting to numpy
            features.append(output.cpu().numpy())
            labels.extend(label.numpy())
    return np.concatenate(features), np.array(labels)


# Step 6: Extract the Features from the loaders
training_features, training_labels = extract_features(train_loader)
print("Done extracting training features")

testing_features, testing_labels = extract_features(test_loader)
print("Done extracting testing features")
```

Step 7

The final step is to apply PCA to reduce the feature dimensions from 512 to 50.

```
101  # Step 7: Apply PCA to reduce the feature dimensions from 512 to 50
102  pca = PCA(n_components=50)  # Set number of components to 50
103  training_features_reduced = pca.fit_transform(training_features)
104  print("Done applying PCA to training features")
```

A source of error that needs mentioning, since it created a lot of problems for us. When extracting the features, initially, we forgot to update the labels we extracted along with the data. Therefore, when we were training our models, we were using labels that were meant for a different subset. This explained why, initially, we were getting low accuracy.

Naive Bayes

Custom Implementation

The custom implementation manually calculates means, variance, and priors for each class during the *fit* phase. It predicts labels based on log-likelihood computed using Gaussian probability density. It also includes a small ε (1e-9) to avoid dividing by 0. For the training methodology, there are no epochs since it is not iterative; it relies on analytical computation of class statistics. The learning rate is also not applicable since no gradient descent is involved. Gaussian Naive Bayes do not use a loss function for training; and since there is no iterative optimization, it relies on direct computation

```
Gaussian Naive Bayes Accuracy: 79.20%
Custom Accuracy: 0.79
Custom Precision: 0.80
Custom Recall: 0.79
Custom F1-Score: 0.79
```

We managed to achieve an accuracy of 79.20% with this implementation. The “Custom Accuracy”, “Custom Precision”, “Custom Recall”, and “Custom F1-Score” were all obtained from the Sklearn library functions. The results obtained are similar to what we calculated from our custom implementation.

Scikit’s Gaussian Naive Bayes (Sklearn)

When using a pre-written version of the model, all we have to do is: initialize the model (GaussianNB()), train the model (fit method), and predict the labels (predict method).

After executing the code here are the results:

```
Sklearn model Accuracy: 79.20%
Sklearn Accuracy: 0.79
Sklearn Precision: 0.80
Sklearn Recall: 0.79
Sklearn F1-Score: 0.79
```

Comparing the results to the custom implementation of Naive Bayes, we can conclude that they are identical. This is within’ norms as they both follow the same algorithm, therefore they should produce identical outputs. Once again, “Sklearn Accuracy”, “Sklearn Precision”, “Sklearn recall”, and “Sklearn F1-Score” were all computed from the Sklearn library.

Decision Tree

Custom Implementation

For the custom implementation of the Decision Tree, we are implementing a decision tree from scratch using recursive tree construction. It splits nodes based on the Gini impurity, calculated by the *_gini* function. It finds the best split by iterating over all features and thresholds. The recursion is stopped only when the maximum depth is reached, the node contains samples of only one class (pure node), or no further splits are possible.

For the training methodology, the criterion used is the Gini impurity. Epochs are not used since a decision tree is not iterative; it's built recursively. There is also no learning rate (no gradient-based optimization) or loss function. The stopping conditions are as follows: the maximum depth (5, 10, 20, 50) and node purity or insufficient samples to split.

The custom tree construction is as follows: *_build_tree* to recursively create subtrees until stopping conditions are met, splits using *_split*, which partitions the dataset into left and right subsets, and *_find_best_split* evaluates all features and thresholds to minimize the weighted Gini impurity.

```
Loaded the pre-trained model
Finished predicting test features
Decision Tree Accuracy: 61.00%
Custom Accuracy: 0.61
Custom Precision: 0.62
Custom Recall: 0.61
Custom F1-Score: 0.61
```

We managed to achieve an accuracy of 61.00% with this implementation. The "Custom Accuracy", "Custom Precision", "Custom Recall", and "Custom F1-Score" were all obtained from the Sklearn library functions. The results obtained are similar to what we calculated from our custom implementation.

Scikit's Decision Tree Classifier(Sklearn)

When using a pre-written version of the model, all we have to do is: initialize the model (DecisionTreeClassifier() specify the criterion, max_depth, and a random_state), train the model (fit method), and predict the labels (predict method).

After executing the code here are the results:

```
Scikit-learn Decision Tree Accuracy: 61.90%
Sklearn Accuracy: 0.62
Sklearn Precision: 0.63
Sklearn Recall: 0.62
Sklearn F1-Score: 0.62
```

Comparing the results to the custom implementation of Naive Bayes, we can conclude that the SKlearn version got 0.90% better accuracy. This also influenced all 4 metrics, gaining an extra percent point. Once again, “Sklearn Accuracy”, “Sklearn Precision”, “Sklearn recall”, and “Sklearn F1-Score” were all computed from the Sklearn library

Multi-Layer Perceptron (MLP)

Output

```
Using device: cpu
Epoch [1/20], Loss: 0.7365
Epoch [2/20], Loss: 0.4171
Epoch [3/20], Loss: 0.3457
Epoch [4/20], Loss: 0.2788
Epoch [5/20], Loss: 0.2166
Epoch [6/20], Loss: 0.1760
Epoch [7/20], Loss: 0.1352
Epoch [8/20], Loss: 0.1165
Epoch [9/20], Loss: 0.0893
Epoch [10/20], Loss: 0.0863
Epoch [11/20], Loss: 0.0802
Epoch [12/20], Loss: 0.0649
Epoch [13/20], Loss: 0.0506
Epoch [14/20], Loss: 0.0508
Epoch [15/20], Loss: 0.0278
Epoch [16/20], Loss: 0.0246
Epoch [17/20], Loss: 0.0181
Epoch [18/20], Loss: 0.0201
Epoch [19/20], Loss: 0.0413
Epoch [20/20], Loss: 0.0325
Model saved to ./mlp\mlp_model.pth
Accuracy: 0.82
Precision: 0.82
Recall: 0.82
F1-Score: 0.82
```

The main architecture of the MLP consists of three fully connected layers with the following structure:

- Linear(50, 512) followed by ReLU activation.
- Linear(512, 512) followed by Batch Normalization and ReLU activation.
- Linear(512, 10) as the output layer.

Model Variants

Our code includes the baseline model: as stated above, a three-layer network with specific input, hidden, and output dimensions. Potential variations may imply adding or reducing layers. Such modifications surely impact the output of the model and would have to be tested thoroughly.

Training Methodology

The MLP was trained for 20 epochs using the following hyperparameters:

- Learning Rate: 0.01.
- Loss Function: Cross-Entropy Loss (torch.nn.CrossEntropyLoss), appropriate for multi-class classification.
- Batch Size: 64, ensuring efficient updates while maintaining stability.
- Optimizer: Stochastic Gradient Descent (SGD) with momentum set to 0.9 to accelerate convergence and avoid local minima.

Optimization Techniques

MLP utilizes an SGD Optimizer with Momentum, implemented as:

```
90      # Define loss function and optimizer
91      criterion = nn.CrossEntropyLoss()
92      optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Finally, the model was saved in the *./mlp* folder under *mlp_model.pth* to avoid the need for retraining.

Convolutional Neural Network (CNN)

Output

```
Using device: cpu
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/20], Loss: 1.2975
Epoch [2/20], Loss: 0.8343
Epoch [3/20], Loss: 0.6447
Epoch [4/20], Loss: 0.5205
Epoch [5/20], Loss: 0.4189
Epoch [6/20], Loss: 0.3366
Epoch [7/20], Loss: 0.2704
Epoch [8/20], Loss: 0.2198
Epoch [9/20], Loss: 0.1761
Epoch [10/20], Loss: 0.1433
Epoch [11/20], Loss: 0.1120
Epoch [12/20], Loss: 0.0961
Epoch [13/20], Loss: 0.0809
Epoch [14/20], Loss: 0.0630
Epoch [15/20], Loss: 0.0600
Epoch [16/20], Loss: 0.0509
Epoch [17/20], Loss: 0.0403
Epoch [18/20], Loss: 0.0366
Epoch [19/20], Loss: 0.0331
Epoch [20/20], Loss: 0.0335
Model saved to ./cnn\cnn_model.pth
Accuracy: 0.84
Precision: 0.83
Recall: 0.84
F1-Score: 0.83
```

The primary CNN model implemented follows the VGG11 architecture. This implies 5 convolutional blocks, each containing the following:

- Convolutional layers with kernel size 3, stride 1, and padding 1.
- Batch Normalization after each convolutional layer.
- ReLU activation for non-linearity.
- Max Pooling (kernel size 2, stride 2) at the end of each block.

It also includes a fully connected classifier with:

- Linear(512, 4096) followed by ReLU and Dropout (0.5).
- Linear(4096, 4096) followed by ReLU and Dropout (0.5).
- Linear(4096, 10) as the output layer.

Model Variants

Our code contains the baseline VGG11 architecture stated in the previous section. Potential variations may include depth modification, kernel size variation, and varying pooling strategies (e.g. average pooling).

Training Methodology

The CNN was trained for 20 epochs with the following configuration:

- LP.earning Rate: 0.01.
- Loss Function: Cross-Entropy Loss (torch.nn.CrossEntropyLoss).
- Batch Size: 64, balancing computational efficiency and model performance.
- Optimizer: SGD with momentum set to 0.9, mirroring the methodology used for the ML

Optimization Techniques

Similarly to MLP, CNN uses an SGD optimizer with momentum, implemented as:

```
133     # Define loss function and optimizer
134     criterion = nn.CrossEntropyLoss()
135     optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Finally, the model is saved as *./cnn/cnn_model.pth* after training, to reproduce the same results faster on consequent runs.

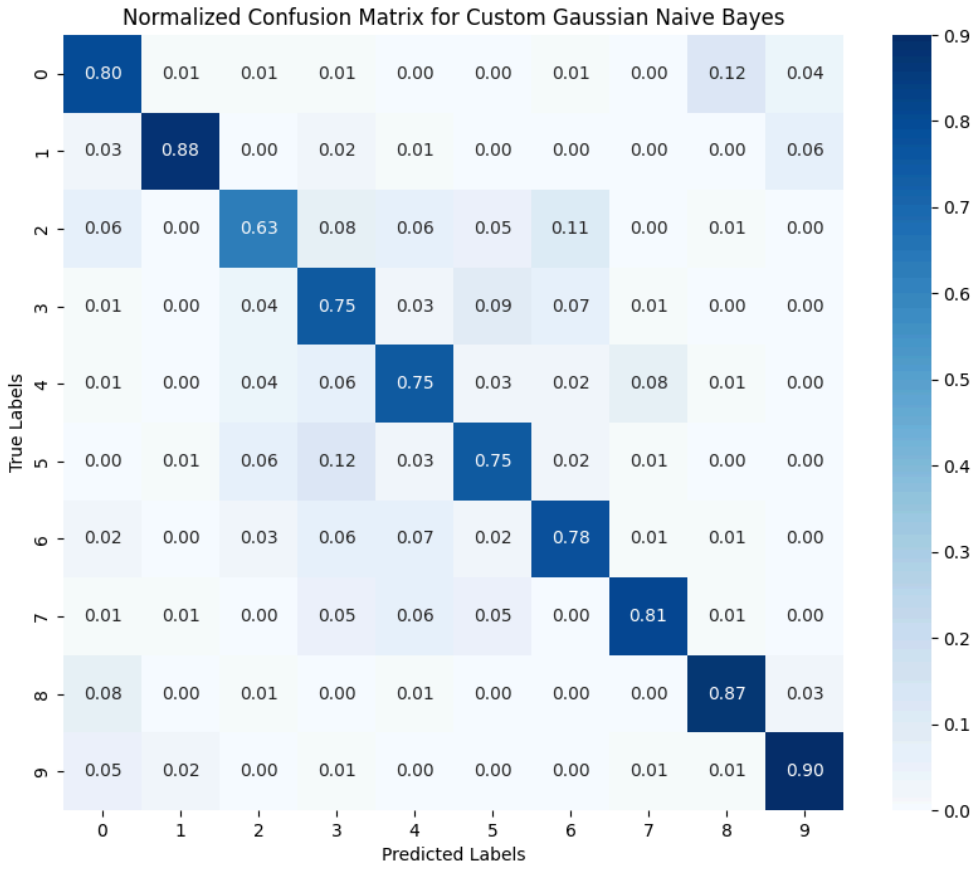
Evaluation

Gaussian Naive Bayes		
	Custom Implementation	SKlearn Implementation
Accuracy	79.20%	79.20%
Precision	80%	80%
Recall	79%	79%
F1-Score	79%	79%
Decision Tree Classifier (Max Depth = 10 - this got the best results)		
	Custom Implementation	SKlearn Implementation
Accuracy	61%	61.90%
Precision	62%	63%
Recall	61%	62%
F1-Score	61%	62%
Multi-Layer Perceptron (MLP)		
	Custom Implementation	SKlearn Implementation
Accuracy	82%	NA
Precision	82%	NA
Recall	82%	NA
F1-Score	82%	NA
Convolutional Neural Network (CNN)		
	Custom Implementation	SKlearn Implementation
Accuracy	84%	NA
Precision	83%	NA
Recall	84%	NA
F1-Score	83%	NA

Model Performance

Gaussian Naive Bayes Custom Confusion Matrix

From what we can observe the model had a hard time identifying images from class 2, but exhibited the greatest accuracy when predicting images from class 9. The model had an easier time identifying edge classes (class 0, 1, 7, 8, 9).

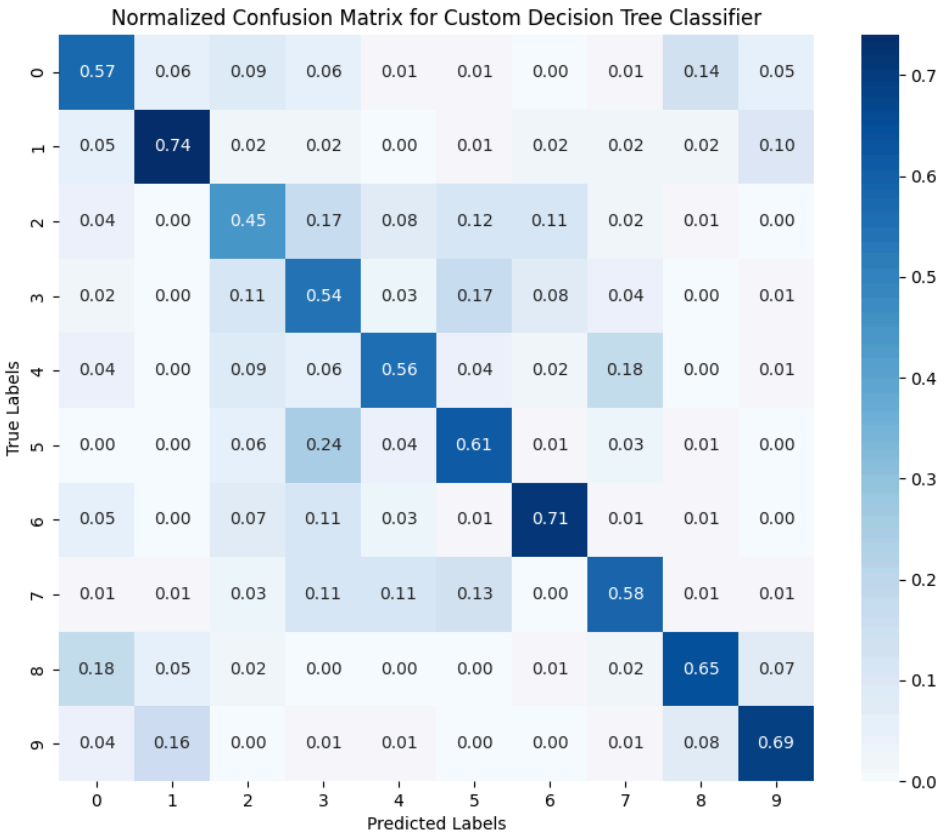


Gaussian Naive Bayes SKlearn Confusion Matrix

When generating the confusion matrix for the Sklearn Gaussian Naive Bayes, the matrix is identical to the custom confusion matrix, down to the percentages. Once again, the model has trouble identifying images from class 2, but easily identified images from class 9. Just like for the custom implementation, the edge classes (class 0, 1, 7, 8, 9) have higher identification percentages.

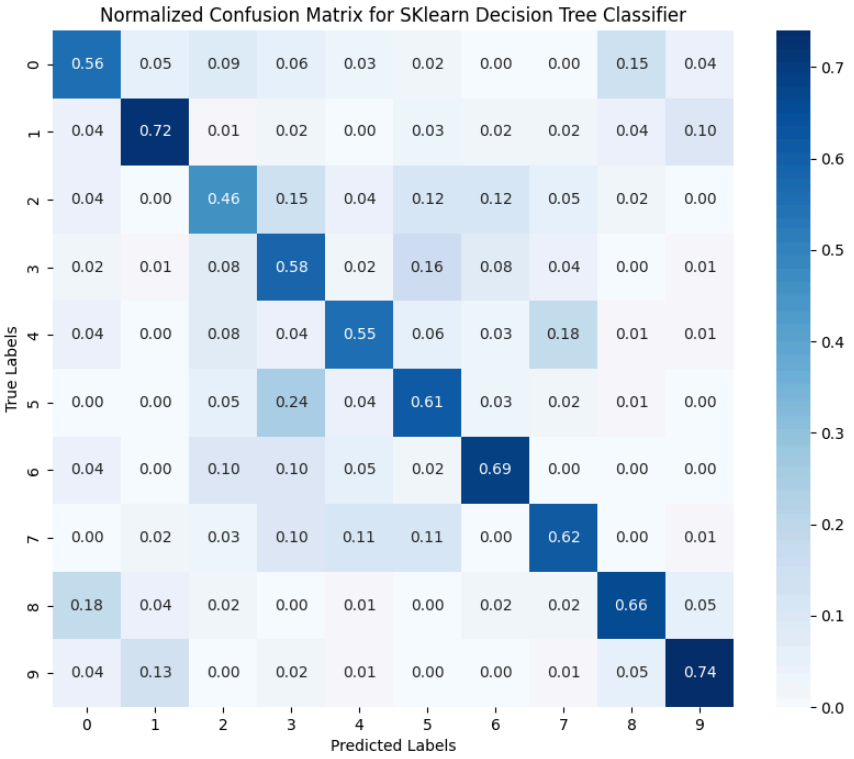
Decision Tree Classifier Custom Confusion Matrix

From what we can observe the model had a hard time identifying images from class 2, but exhibited the greatest accuracy when predicting images from class 1. In fact, the model had an easier time identifying classes 1, 6, 8, 9.

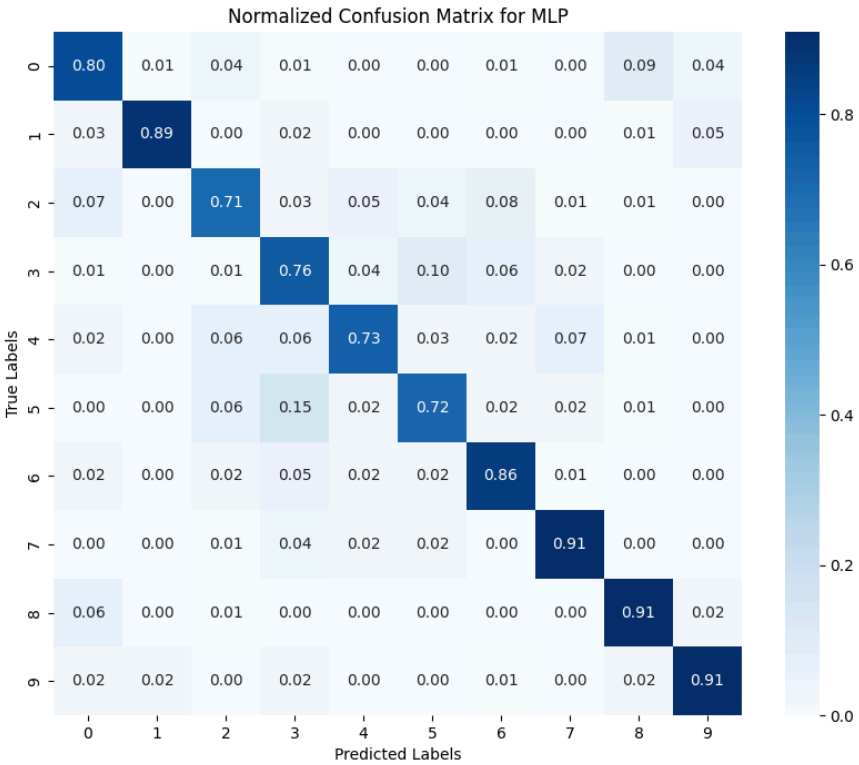


Decision Tree Classifier Custom Confusion Matrix

When generating the confusion matrix for the Sklearn Decision Tree Classifier, the matrix differs from the custom confusion matrix. The best class for SKlearn is class 9, while the worst class is 2. The best identified classes are class 1, 6, 8, 9.

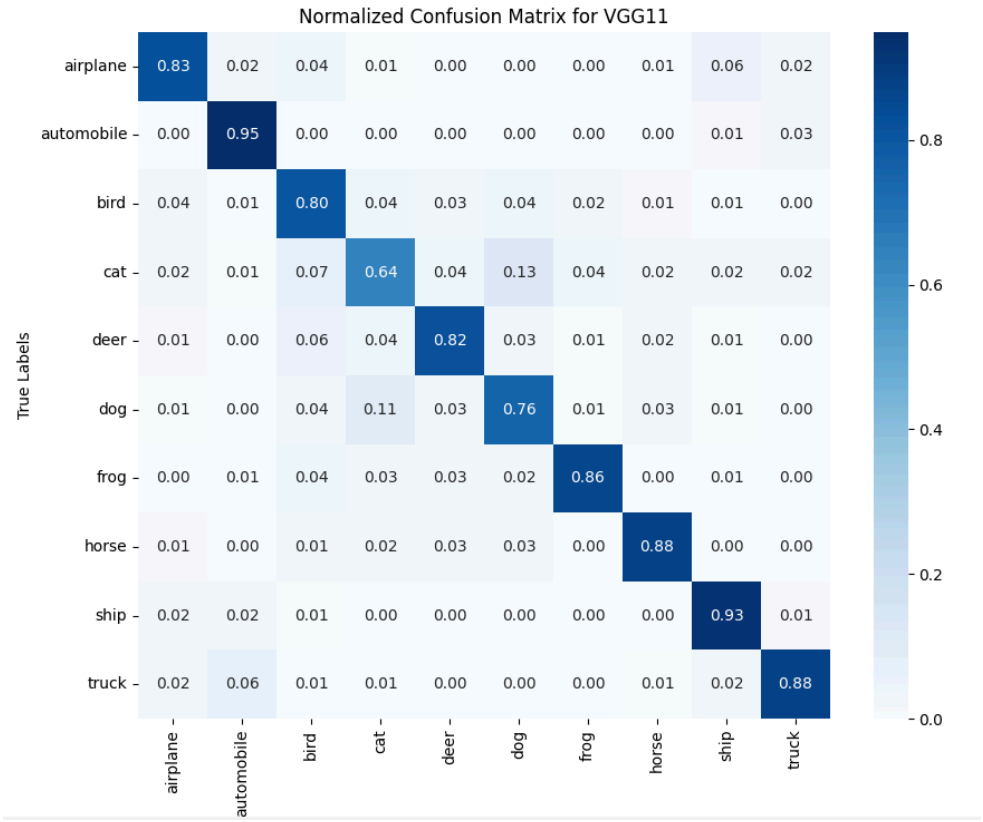


MLP Custom Confusion Matrix



The MLP performs relatively well, as most classes have high classification rates along the diagonal. This suggests the model is learning useful patterns from the data and generalizing reasonably well.
Classes 7, 8, and 9 stand out as the best-predicted, with over 90% accuracy in each case.
Class 5 shows the most confusion, with 15% misclassification into Class 6, and 10% or more spread across other classes (e.g., 2, 3, and 4).
Class 2 also struggles, with significant misclassifications into neighboring classes like 3 and 4. This indicates shared feature patterns or poor separation in the feature space.

CNN Custom Confusion Matrix



The model performs better than the MLP model, as evidenced by the higher diagonal values (correct classifications) across most classes.
Classes like automobile, frog, ship, and truck have high classification rates (95%, 86%, 93%, and 88%, respectively).
Bird, cat, dog, and deer show overlapping confusion, which could indicate that these animal classes share similar textures or features that the model finds hard to separate.

Depth Impact on Models

Since Decision Tree is the only depth-related algorithm for which we have implemented different depths, we will be using it as a reference to dress the following depth impact analysis.

For max depth = 5 here are the results (custom implementation | SKlearn implementation): accuracy = 54.90% | 54.90%, precision = 56% | 56%, recall = 55%| 55%, and F1-score= 54% | 54%.

For max depth = 10 here are the results (custom implementation | SKlearn implementation): accuracy = 61.00% | 61.90%, precision = 62% | 63%, recall = 61%| 62%, and F1-score= 61% | 62%.

For max depth = 20 here are the results (custom implementation | SKlearn implementation): accuracy = 59.30% | 59.40%, precision = 59% | 60%, recall = 59%| 59%, and F1-score= 59% | 60%.

For max depth = 50 here are the results (custom implementation | SKlearn implementation): accuracy = 59.30% | 59.70%, precision = 59% | 60%, recall = 59%| 60%, and F1-score= 59% | 60%.

From testing different trees with varying depths, we can conclude that the “best” results come from depth 10. Not only does the SKlearn version have the best accuracy, but SKlearn at depth 10 has the best percentages overall for the remaining 3 metrics. The next best depth is 50; the percentages are only a few points behind. Our custom implementation is also not that far behind; being only at most 1% lower than the SKlearn version.

Summary

Here are the models ranked from best to worst:

1. CNN: Highest accuracy and balanced metrics, but computationally expensive.
2. MLP: Strong performance and flexibility with less computational cost than CNN.
3. Gaussian Naive Bayes: A strong baseline model with consistent performance across metrics.
4. Decision Tree: Limited generalization and accuracy but interpretable and easy to implement.

Naturally, there are some trade-offs between models. CNN and MLP excel in learning complex relationships but require more resources. Simpler models like Naive Bayes and Decision Trees are computationally efficient but have lower accuracy and recall.

How to Run

Before delving into how to run the code, we'll go over the python files in our repo:

- main.py
  This file is responsible for extracting the 500 training and 100 test images from the CIFAR-10 dataset (for more details please view the "Dataset Overview" section). This is also where the numpy files are saved (files ending in *npy*); here is a list of the files:

  test_features.npy
  test_features_pca.npy
  testing_labels.npy
  train_features.npy
  train_features_pca.npy
  training_labels.npy

  These files will be used in the files containing the models and their logic.

- gaussian_naive_bayes.py
  This is the file where the logic of the custom and Sklearn's implementation of gaussian naive bayes. This file also saves the custom and Sklearn's trained models of naive bayes in the files named: "gaussian_naive_bayes_model.pkl" and "gaussian_naive_bayes_modelSK.joblib". These files can be used instead of training the model on every execution. The required files to execute naive bayes are: "train_features_pca.npy", "training_labels.npy", "test_features_pca.npy", and "testing_labels.npy".

- Decision_tree_classifier.py
  This is the file where the logic of the custom and Sklearn's implementation of the decision tree classifier. This file also saves the custom and Sklearn's trained models of tree classifiers in the files named: "decision_tree_classifier.pkl", "decision_tree_classifier_SK[5, 10, 20, 50].joblib". The number after SK represents the max depth of the tree, for the custom implementation only the 10 max depth tree is saved. These files can be used instead of training the model on every execution. The required files to execute naive bayes are: "train_features_pca.npy", "training_labels.npy", "test_features_pca.npy", and "testing_labels.npy".

- mlp.py
  This is the file that implements our custom implementation of the MLP algorithm. The saved model is produced as *./mlp/mlp_model.pth .*

- cnn.py
  This is the file that implements our custom implementation of the CNN algorithm. The saved model is produced as *./mlp/cnn_model.pth* .

Generating Dataset Files

If the *npy* files are not present in the directory, you need to first run *main.py*. This will save the files outlined above. After the file is done executing, the files outlined above will be updated to the most recent execution.

Running gaussian_naive_bayes.py

On every execution of the file, you will run the custom implementation of the model, followed by SKlearn's version. The program will check if a saved model exists before training a new one. If you want to have the custom model trained again please delete the "gaussian_naive_bayes_model.pkl" file and a new model will be trained on the most recent training and test images.

It is the same for SKlearn's implementation of the model, it will run the saved model. If you want to train a new model you will need to make these changes:

```
133     # Initialize the Gaussian Naive Bayes classifier
134     # To use saved trained model comment out the training and saving lines
135     gnb = GaussianNB()
136
137     # Train the model
138     gnb.fit(train_features, train_labels)
139
140     # Save the model - to use saved model comment until this line (123)
141     dump(gnb, "GaussianNaiveBayes/gaussian_naive_bayes_modelSK.joblib")
142
143     # This is the saved trained model
144     # gnb = load("GaussianNaiveBayes/gaussian_naive_bayes_modelSK.joblib")
145
146     # Predict the labels for the test set
147     y_pred = gnb.predict(test_features)
148
149     # Calculate the accuracy
150     accuracy = (accuracy_score(test_labels, y_pred)) * 100
151     print(f'Sklearn model Accuracy: {accuracy:.2f}%')
```

Running decision_tree_classifier.py

The same can be said for the decision tree classifier, both files are set up identically. One thing to note, there are 4 SKlearn models saved, each of varying depth. Therefore, please be aware of which model you are loading. If you want to train a model of different lengths, make these changes:

```
185
186     # Train and test the Decision Tree classifier with max_depth=50
187     # To use saved trained model comment out the training and saving lines
188     clf = DecisionTreeClassifier(criterion='gini', max_depth=50, random_state=42)
189     clf.fit(train_features, train_labels)
190
191     # Save the model - to use saved model comment until this line
192     dump(clf, "DecisionTreeClassifier/decision_tree_classifier_SK50.joblib")
193     #
194     # This is the saved trained model
195     # clf = load("DecisionTreeClassifier/decision_tree_classifier_SK10.joblib")
196
197     # Predict on the test set
198     predictions = clf.predict(test_features)
```

Running mlp.py

Like all other algorithms, make sure to run main.py first to obtain the data needed to train the algorithm.
Simply run mlp.py to train the model and get its output on the terminal. Note that running mlp.py also saves the current model in the ./mlp/ folder.

Running cnn.py

Like all other algorithms, make sure to run main.py first to obtain the data needed to train the algorithm.
Simply run cnn.py to train the model and get its output on the terminal. Note that running cnn.py also saves the current model in the ./cnn/ folder.