MGMT 59000: Analyzing Unstructured Data

Team Assignment 3

Group 1

Part 1

Step 1

data = pd.read_excel('Assignment 3.xlsx')

Building the training and test datasets

# Filtering the first 400 restaurant reviews and the first 400 movie reviews
train_restaurant = data[(data['label'] == 'restaurant') & (data['id'] <= 400)]
train_movie = data[(data['label'] == 'movie') & (data['id'] >= 501) & (data['id'] <= 900)]

# Combining the two datasets to form the training dataset
train_dataset = pd.concat([train_restaurant, train_movie])

# The rest of the data will be used as the test dataset
test_dataset = data.drop(train_dataset.index)

# Checking the first few rows of each dataset to ensure they are correct
train_dataset.head(), test_dataset.head()

Part 1 - Step 1

In [2]:
```
data = pd.read_excel('Assignment 3.xlsx')

# Building the training and test datasets as specified

# Filtering the first 400 restaurant reviews and the first 400 movie reviews
train_restaurant = data[(data['label'] == 'restaurant') & (data['id'] <= 400)]
train_movie = data[(data['label'] == 'movie') & (data['id'] >= 501) & (data['id'] <= 900)]

# Combining the two datasets to form the training dataset
train_dataset = pd.concat([train_restaurant, train_movie])

# The rest of the data will be used as the test dataset
test_dataset = data.drop(train_dataset.index)

# Checking the first few rows of each dataset to ensure they are correct
train_dataset.head(), test_dataset.head()
```

Out[2]:
```
(   id                                             review       label
 0   1  About the shop: There is a restaurant in Soi L...  restaurant
 1   2  About the shop: Through this store for about t...  restaurant
 2   3  Roast Coffee &amp;amp; Eatery is a restaurant loca...  restaurant
 3   4  Eat from the children. The shop is opposite. P...  restaurant
 4   5  The Ak 1 shop at another branch tastes the sam...  restaurant,
       id                                             review       label
 400  401  This shop is one of my favorites. Not far from...  restaurant
 401  402  This is the first time to eat a crab cake or s...  restaurant
 402  403  two stars for good environment and quite ok fo...  restaurant
 403  404  The restaurant is decorated like a scaffolding...  restaurant
 404  405  Good fresh food, especially raw fish, plus aki...  restaurant)
```

Step 2

```
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import string

# Define a custom tokenizer function
def custom_tokenizer(text):
    # Initialize WordNetLemmatizer
    lemmatizer = WordNetLemmatizer()

    # Split the text into words
    words = text.split()

    # Lemmatize, remove stop words and punctuations
    tokens = [lemmatizer.lemmatize(word) for word in words
              if word.lower() not in stopwords.words('english')
              and word not in string.punctuation]

    return tokens

# Initialize TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(tokenizer=custom_tokenizer, min_df=5, ngram_range=(1, 2))

# Apply TF-IDF transformation to the training dataset
tfidf_matrix = tfidf_vectorizer.fit_transform(train_dataset['review'])

# To view the shape of the TF-IDF matrix and feature names
print(tfidf_matrix.shape)
print(tfidf_vectorizer.get_feature_names_out()[:10])
```

```
/Users/harsh/anaconda3/lib/python3.11/site-packages/sklearn/feature_extraction/text.py:525: UserWarning: The parame
ter 'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(

(800, 5218)
['!!' '!!!' '&amp;' '&quot;' '&quot;a' '&quot;all' '&quot;black'
 '&quot;can' '&quot;crazy' '&quot;da']
```

Step 3

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, train_dataset['label'],
test_size=0.3, random_state=42)

# Initialize models
naive_bayes_model = MultinomialNB()
logit_model = LogisticRegression()
random_forest_model = RandomForestClassifier(n_estimators=50)
svm_model = SVC()
ann_model = MLPClassifier(hidden_layer_sizes=(4,), max_iter=1000)

# Dictionary to store models
models = {
   "Naive Bayes": naive_bayes_model,
   "Logistic Regression": logit_model,
   "Random Forest": random_forest_model,
   "SVM": svm_model,
   "ANN": ann_model
}

# Train each model and calculate accuracy
for name, model in models.items():
   model.fit(X_train, y_train)
   predictions = model.predict(X_test)
   accuracy = accuracy_score(y_test, predictions)
   print(f"{name} Accuracy: {accuracy:.2f}")
```

```
Naive Bayes Accuracy: 1.00
Logistic Regression Accuracy: 0.99
Random Forest Accuracy: 0.97
SVM Accuracy: 0.98
ANN Accuracy: 0.97

/Users/harsh/anaconda3/lib/python3.11/site-packages/sklearn/neural_network/_multilayer_perceptron.py:691: Convergen
ceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  warnings.warn(
```

Inference: Naïve Bayes model performs the best as per the accuracy rate. Logistic Regression follows it by being the second best with 0.99 accuracy.

But a point to consider might be that the Naive Bayes model reaching an accuracy of 1.00 could suggest overfitting, where the model has learned to perform perfectly on the training data but might not generalize well to unseen data.

Step 4

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# Number of words to consider as features
max_features = 10000

# Maximum length of each document
maxlen = 100

# Initialize the tokenizer with a maximum number of words
tokenizer = Tokenizer(num_words=max_features)

# Fit the tokenizer on the training data
tokenizer.fit_on_texts(train_dataset['review'])

# Convert the texts to sequences
train_sequences = tokenizer.texts_to_sequences(train_dataset['review'])
test_sequences = tokenizer.texts_to_sequences(test_dataset['review'])

# Pad the sequences so they all have the same length
X_train = pad_sequences(train_sequences, maxlen=maxlen)
X_test = pad_sequences(test_sequences, maxlen=maxlen)

# To view the shape of the sequences
print(X_train.shape)
print(X_test.shape)
```

Part 1 - Step 4

In [5]:
```
#pip install keras
```

In [6]:
```
#pip install --upgrade keras tensorflow
```

In [7]:
```
#pip install tensorflow
```

In [8]:
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# Number of words to consider as features
max_features = 10000  # This is an arbitrary number, you can adjust it based on your vocabulary size

# Maximum length of each document
maxlen = 100

# Initialize the tokenizer with a maximum number of words
tokenizer = Tokenizer(num_words=max_features)

# Fit the tokenizer on the training data
tokenizer.fit_on_texts(train_dataset['review'])

# Convert the texts to sequences
train_sequences = tokenizer.texts_to_sequences(train_dataset['review'])
test_sequences = tokenizer.texts_to_sequences(test_dataset['review'])

# Pad the sequences so they all have the same length
X_train = pad_sequences(train_sequences, maxlen=maxlen)
X_test = pad_sequences(test_sequences, maxlen=maxlen)

# Example: To view the shape of the sequences
print(X_train.shape)
print(X_test.shape)
```

```
(800, 100)
(200, 100)
```

Step 5

```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dropout, Dense
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np

# Assuming max_features and maxlen are defined as before
max_features = 10000
maxlen = 100

# Initialize and fit the tokenizer
tokenizer = Tokenizer(num_words=max_features)
tokenizer.fit_on_texts(train_dataset['review'])

# Convert texts to sequences
train_sequences = tokenizer.texts_to_sequences(train_dataset['review'])
test_sequences = tokenizer.texts_to_sequences(test_dataset['review'])

# Pad sequences
X_train = pad_sequences(train_sequences, maxlen=maxlen)
```

```
X_test = pad_sequences(test_sequences, maxlen=maxlen)

# Convert labels to numeric
y_train = np.array(train_dataset['label'].apply(lambda x: 1 if x == 'movie' else 0))
y_test = np.array(test_dataset['label'].apply(lambda x: 1 if x == 'movie' else 0))

# Define the model
model = Sequential()
model.add(Embedding(max_features, 20, input_length=maxlen))
model.add(LSTM(40, dropout=0.2, recurrent_dropout=0.2))
model.add(Dropout(0.1))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, batch_size=100, epochs=10, validation_data=(X_test, y_test))


# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
```

```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
```

```
Epoch 1/10
8/8 [==============================] - 1s 61ms/step - loss: 0.6918 - accuracy: 0.5450 - val_loss: 0.6888 - val_accu
racy: 0.6550
Epoch 2/10
8/8 [==============================] - 0s 45ms/step - loss: 0.6836 - accuracy: 0.8012 - val_loss: 0.6771 - val_accu
racy: 0.8700
Epoch 3/10
8/8 [==============================] - 0s 45ms/step - loss: 0.6482 - accuracy: 0.9325 - val_loss: 0.5837 - val_accu
racy: 0.9850
Epoch 4/10
8/8 [==============================] - 0s 54ms/step - loss: 0.3952 - accuracy: 0.9688 - val_loss: 0.2372 - val_accu
racy: 0.9500
Epoch 5/10
8/8 [==============================] - 0s 46ms/step - loss: 0.2136 - accuracy: 0.9550 - val_loss: 0.2631 - val_accu
racy: 0.9250
Epoch 6/10
8/8 [==============================] - 0s 45ms/step - loss: 0.2391 - accuracy: 0.9287 - val_loss: 0.3491 - val_accu
racy: 0.8800
Epoch 7/10
8/8 [==============================] - 0s 47ms/step - loss: 0.2404 - accuracy: 0.9187 - val_loss: 0.3015 - val_accu
racy: 0.8900
Epoch 8/10
8/8 [==============================] - 0s 45ms/step - loss: 0.2058 - accuracy: 0.9325 - val_loss: 0.3021 - val_accu
racy: 0.9000
Epoch 9/10
8/8 [==============================] - 0s 40ms/step - loss: 0.1878 - accuracy: 0.9463 - val_loss: 0.2422 - val_accu
racy: 0.9500
Epoch 10/10
8/8 [==============================] - 0s 42ms/step - loss: 0.2763 - accuracy: 0.9425 - val_loss: 0.5415 - val_accu
racy: 0.9000
7/7 [==============================] - 0s 3ms/step - loss: 0.5415 - accuracy: 0.9000
Accuracy: 0.90
```

Inference:

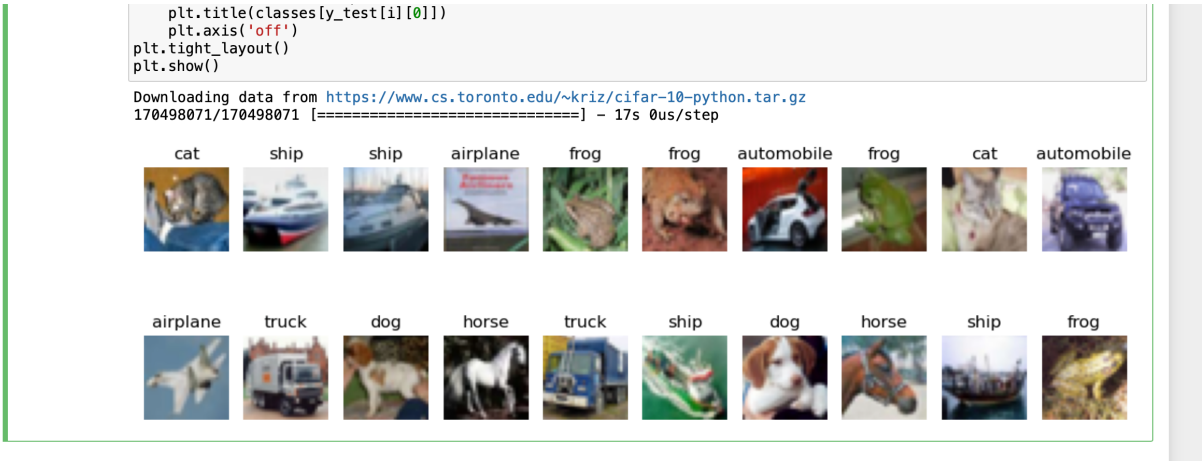
While comparing accuracy rates from the traditional machine learning models in step 3 with the deep learning model in step 5, we observe that the Naive Bayes, Logistic Regression, and SVM models all had higher accuracy rates than the deep learning model, with Naive Bayes even reaching perfect accuracy. The Random Forest and simple ANN models had accuracy rates that were equivalent to that of the deep learning model.

- While the traditional models showed higher accuracy rates on the training set; the deep learning model's performance is respectable and with further training, more data, and hyperparameter tuning, it could potentially outperform the traditional models. The LSTM's ability to understand sequence might make it a better choice for more complex classification tasks.
- The warning from the ANN model in step 3 suggests that the model had not converged, which might indicate that the deep learning model could outperform it with appropriate tuning.
- The traditional models might have performed exceptionally well due to the nature of the dataset. If the dataset is straightforward or if the distinctive features between restaurant and movie reviews are easily captured by simple statistical models, the additional complexity of a deep learning model might not offer a significant advantage.

Part 2

Step 1

```
from keras.datasets import cifar10
import matplotlib.pyplot as plt

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Define the CIFAR-10 classes
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']

# Visualize the first 20 images of the test set
plt.figure(figsize=(10, 4))
for i in range(20):
    plt.subplot(2, 10, i+1)
    plt.imshow(x_test[i])
    plt.title(classes[y_test[i][0]])
    plt.axis('off')
plt.tight_layout()
plt.show()
```



The output shows the first 20 images of the CIFAR-10 test set along with their corresponding labels. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images.

The labels visible in the image are:

- Cat
- Ship
- Ship
- Airplane
- Frog
- Frog
- Automobile
- Frog
- Cat
- Automobile
- Airplane
- Truck
- Dog
- Horse
- Truck
- Ship
- Dog
- Horse
- Ship
- Frog

These labels correspond to the CIFAR-10 classes defined. The dataset includes a variety of objects that are commonly used for computer vision tasks such as image classification. The classes include modes of transportation like airplanes, trucks, and ships; animals like cats, dogs, frogs, horses, and birds; and other categories such as automobiles and deer.

The visualization demonstrates the diversity and complexity of the dataset, with images varying in scale, pose, and background. For instance, some images may show the object up close and centered, while others might include the object in a more complex environment or from different angles. This variability makes CIFAR-10 a challenging dataset for developing and testing image classification algorithms.

Having a balanced dataset with an equal number of images for each class helps to ensure that a trained model does not become biased toward more frequently represented classes. It also provides a good test bed for evaluating how well a model can generalize from training data to new, unseen images.

Step 2

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=x_train.shape[1:]),  # a
    Dropout(0.2),                                                          # b
    Conv2D(32, (3, 3), activation='relu'),                                 # c
    MaxPooling2D(pool_size=(2, 2)),                                        # d
    Conv2D(64, (3, 3), activation='relu'),                                 # e
    Dropout(0.2),                                                          # f
    Conv2D(64, (3, 3), activation='relu'),                                 # g
    MaxPooling2D(pool_size=(2, 2)),                                        # h
    Flatten(),                                                             # i
    Dense(256, activation='relu'),                                         # j
    Dropout(0.2),                                                          # k
    Dense(10, activation='softmax')                                        # l
])

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
batch_size = 500  # Smaller than 500 to prevent overheating
epochs = 5
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test,
y_test), verbose=1)
```

# Evaluate the model
accuracy = model.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy: {accuracy * 100:.2f}%')

```
    Dense(10, activation='softmax')                                    # C
])

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
batch_size = 500  # Smaller than 500 to prevent overheating
epochs = 5
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), verbose=1)

# Evaluate the model
accuracy = model.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy: {accuracy * 100:.2f}%')
```

```
Epoch 1/5
100/100 [==============================] - 26s 252ms/step - loss: 1.8566 - accuracy: 0.3230 - val_loss: 1.7781 - va
l_accuracy: 0.3576
Epoch 2/5
100/100 [==============================] - 22s 225ms/step - loss: 1.4671 - accuracy: 0.4694 - val_loss: 1.4522 - va
l_accuracy: 0.4808
Epoch 3/5
100/100 [==============================] - 24s 243ms/step - loss: 1.3190 - accuracy: 0.5296 - val_loss: 1.2516 - va
l_accuracy: 0.5598
Epoch 4/5
100/100 [==============================] - 24s 236ms/step - loss: 1.2041 - accuracy: 0.5729 - val_loss: 1.1731 - va
l_accuracy: 0.5887
Epoch 5/5
100/100 [==============================] - 25s 245ms/step - loss: 1.1096 - accuracy: 0.6073 - val_loss: 1.0774 - va
l_accuracy: 0.6156
Accuracy: 61.56%
```

Here's an interpretation of the results:

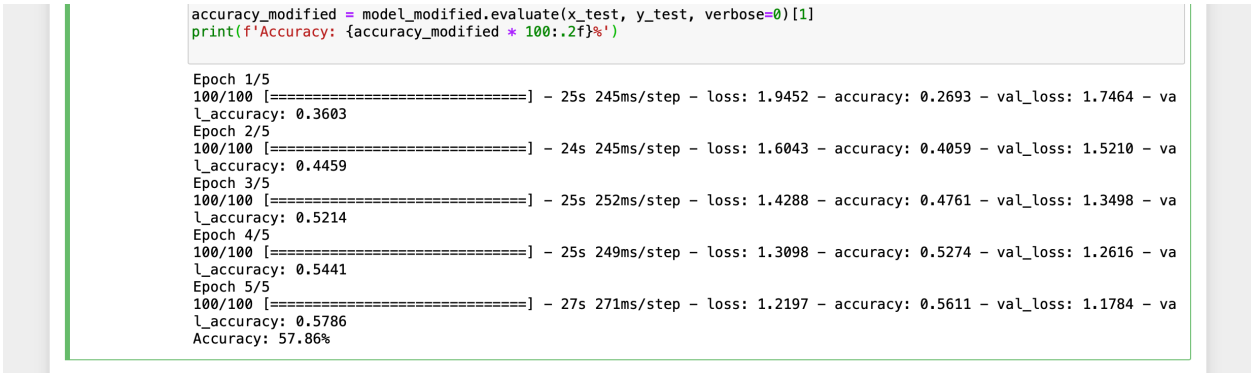
- Initial Low Accuracy: The model starts with an accuracy of 35.76% in the first epoch. This is expected as the model is just beginning to learn from the data.
- Steady Improvement: With each epoch, the model's accuracy improves, showing that it is learning from the training data. This is evidenced by the increase in accuracy from 35.76% in the first epoch to 61.56% by the fifth epoch.
- Learning Rate: The rate of improvement in accuracy appears to be diminishing with each epoch. This suggests that initial learning is rapid but begins to plateau as the model starts fitting the training data as much as it can with its given architecture.
- Validation Loss: The validation loss decreases with each epoch, which indicates that the model is getting better at generalizing to unseen data. The decrease from 1.7781 to 1.0774 suggests improvement in the model's ability to predict accurately on the test set.

From these results, we can infer that the CNN architecture with the specified layers and hyperparameters is capable of learning and improving its prediction accuracy over time. However, since the accuracy after 5 epochs is a bit over 60%, there is likely room for improvement either through further training, hyperparameter tuning, or architectural changes.

Step 3

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)


# Define the modified CNN model
model_modified = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=x_train.shape[1:]),  # a
    Dropout(0.2),                                                          # b
    Conv2D(32, (3, 3), activation='relu'),                                 # c
    MaxPooling2D(pool_size=(2, 2)),                                        # d
    Conv2D(64, (3, 3), activation='relu'),                                 # e
    Dropout(0.2),                                                          # f
    Conv2D(64, (3, 3), activation='relu'),                                 # g
    MaxPooling2D(pool_size=(2, 2)),                                        # h
    Conv2D(128, (3, 3), activation='relu'),                                # step 3 a
    Dropout(0.2),                                                          # step 3 b
    Conv2D(128, (3, 3), activation='relu'),                                # step 3 c
    Flatten(),                                                             # i
    Dense(256, activation='relu'),                                         # j
    Dropout(0.2),                                                          # k
    Dense(10, activation='softmax')                                        # l
])

# Compile the modified model
model_modified.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

```
# Train the modified model
batch_size = 500  # Smaller than 500 to prevent overheating
epochs = 5
model_modified.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_data=(x_test, y_test), verbose=1)

# Evaluate the modified model
accuracy_modified = model_modified.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy: {accuracy_modified * 100:.2f}%')
```



```
accuracy_modified = model_modified.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy: {accuracy_modified * 100:.2f}%')

Epoch 1/5
100/100 [==============================] - 25s 245ms/step - loss: 1.9452 - accuracy: 0.2693 - val_loss: 1.7464 - va
l_accuracy: 0.3603
Epoch 2/5
100/100 [==============================] - 24s 245ms/step - loss: 1.6043 - accuracy: 0.4059 - val_loss: 1.5210 - va
l_accuracy: 0.4459
Epoch 3/5
100/100 [==============================] - 25s 252ms/step - loss: 1.4288 - accuracy: 0.4761 - val_loss: 1.3498 - va
l_accuracy: 0.5214
Epoch 4/5
100/100 [==============================] - 25s 249ms/step - loss: 1.3098 - accuracy: 0.5274 - val_loss: 1.2616 - va
l_accuracy: 0.5441
Epoch 5/5
100/100 [==============================] - 27s 271ms/step - loss: 1.2197 - accuracy: 0.5611 - val_loss: 1.1784 - va
l_accuracy: 0.5786
Accuracy: 57.86%
```

Here's an analysis of these results:

- Starting Accuracy: The model begins with a similar accuracy to the previous model from step 2 at around 36%, which is expected since the model weights are initialized randomly.
- Accuracy Progression: The model's accuracy increases with each epoch, showing learning progress. However, the final accuracy after 5 epochs is 57.86%, which is lower than the 61.56% achieved in step 2.
- Comparing Model Performances: Despite the additional complexity added to the model in step 3, the accuracy has decreased compared to the model in step 2. This could be due to several factors, such as the model in step 3 potentially requiring more epochs to converge due to its increased complexity, or it could be beginning to overfit the training data.
- Validation Loss: The validation loss has not consistently decreased. It decreases initially but increases again at the 5th epoch, which might suggest that the model is not generalizing as well as the simpler model from step 2.

Step 4

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

# Load and preprocess the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define the modified CNN model (from step 3)
model_modified = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=x_train.shape[1:]),  # a
    Dropout(0.2),                                                          # b
    Conv2D(32, (3, 3), activation='relu'),                                 # c
    MaxPooling2D(pool_size=(2, 2)),                                        # d
    Conv2D(64, (3, 3), activation='relu'),                                 # e
    Dropout(0.2),                                                          # f
    Conv2D(64, (3, 3), activation='relu'),                                 # g
    MaxPooling2D(pool_size=(2, 2)),                                        # h
    Conv2D(128, (3, 3), activation='relu'),                                # step 3 a
    Dropout(0.2),                                                          # step 3 b
    Conv2D(128, (3, 3), activation='relu'),                                # step 3 c
    Flatten(),                                                             # i
    Dense(256, activation='relu'),                                         # j
    Dropout(0.2),                                                          # k
    Dense(10, activation='softmax')                                        # l
])

# Compile the model
model_modified.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model for 20 epochs
batch_size = 500  # Smaller than 500 to prevent overheating
epochs_extended = 20
model_modified.fit(x_train, y_train, batch_size=batch_size, epochs=epochs_extended,
validation_data=(x_test, y_test), verbose=1)
```

# Evaluate the model
accuracy_extended = model_modified.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy after 20 epochs: {accuracy_extended * 100:.2f}%')

```
# Train the model for 20 epochs
batch_size = 500  # Smaller than 500 to prevent overheating
epochs_extended = 20
model_modified.fit(x_train, y_train, batch_size=batch_size, epochs=epochs_extended, validation_data=(x_test, y_test))

# Evaluate the model
accuracy_extended = model_modified.evaluate(x_test, y_test, verbose=0)[1]
print(f'Accuracy after 20 epochs: {accuracy_extended * 100:.2f}%')
```

```
Epoch 1/20
100/100 [==============================] - 26s 258ms/step - loss: 1.9565 - accuracy: 0.2647 - val_loss: 1.7247 - va
l_accuracy: 0.3670
Epoch 2/20
100/100 [==============================] - 26s 264ms/step - loss: 1.6011 - accuracy: 0.4078 - val_loss: 1.4630 - va
l_accuracy: 0.4661
Epoch 3/20
100/100 [==============================] - 25s 254ms/step - loss: 1.4032 - accuracy: 0.4881 - val_loss: 1.2919 - va
l_accuracy: 0.5404
Epoch 4/20
100/100 [==============================] - 26s 260ms/step - loss: 1.2854 - accuracy: 0.5360 - val_loss: 1.2533 - va
l_accuracy: 0.5546
Epoch 5/20
100/100 [==============================] - 26s 259ms/step - loss: 1.1811 - accuracy: 0.5788 - val_loss: 1.1211 - va
l_accuracy: 0.6055
Epoch 6/20
100/100 [==============================] - 27s 266ms/step - loss: 1.1168 - accuracy: 0.6001 - val_loss: 1.1189 - va
l_accuracy: 0.6098
Epoch 7/20
100/100 [==============================] - 27s 272ms/step - loss: 1.0534 - accuracy: 0.6241 - val_loss: 1.0436 - va
l_accuracy: 0.6297
Epoch 8/20
100/100 [==============================] - 28s 275ms/step - loss: 0.9964 - accuracy: 0.6498 - val_loss: 1.0589 - va
l_accuracy: 0.6286
Epoch 9/20
100/100 [==============================] - 28s 279ms/step - loss: 0.9573 - accuracy: 0.6611 - val_loss: 0.9307 - va
l_accuracy: 0.6671
```

```
Epoch 10/20
100/100 [==============================] - 27s 269ms/step - loss: 0.9043 - accuracy: 0.6807 - val_loss: 0.9176 - va
l_accuracy: 0.6768
Epoch 11/20
100/100 [==============================] - 26s 261ms/step - loss: 0.8829 - accuracy: 0.6876 - val_loss: 0.9549 - va
l_accuracy: 0.6695
Epoch 12/20
100/100 [==============================] - 26s 260ms/step - loss: 0.8465 - accuracy: 0.7015 - val_loss: 0.8903 - va
l_accuracy: 0.6906
Epoch 13/20
100/100 [==============================] - 26s 260ms/step - loss: 0.8166 - accuracy: 0.7125 - val_loss: 0.8582 - va
l_accuracy: 0.6996
Epoch 14/20
100/100 [==============================] - 28s 279ms/step - loss: 0.7809 - accuracy: 0.7257 - val_loss: 0.8378 - va
l_accuracy: 0.7106
Epoch 15/20
100/100 [==============================] - 32s 320ms/step - loss: 0.7618 - accuracy: 0.7308 - val_loss: 0.8231 - va
l_accuracy: 0.7132
Epoch 16/20
100/100 [==============================] - 29s 293ms/step - loss: 0.7299 - accuracy: 0.7431 - val_loss: 0.8184 - va
l_accuracy: 0.7104
Epoch 17/20
100/100 [==============================] - 32s 321ms/step - loss: 0.7048 - accuracy: 0.7519 - val_loss: 0.8048 - va
l_accuracy: 0.7183
Epoch 18/20
100/100 [==============================] - 29s 292ms/step - loss: 0.6915 - accuracy: 0.7564 - val_loss: 0.8009 - va
l_accuracy: 0.7212
Epoch 19/20
100/100 [==============================] - 27s 275ms/step - loss: 0.6611 - accuracy: 0.7656 - val_loss: 0.7544 - va
l_accuracy: 0.7364
Epoch 20/20
100/100 [==============================] - 28s 278ms/step - loss: 0.6413 - accuracy: 0.7718 - val_loss: 0.7496 - va
l_accuracy: 0.7428
Accuracy after 20 epochs: 74.28%
```

The results from step 4, which extend the training of the CNN from step 3 to 20 epochs, show a notable improvement in performance:

- Increased Accuracy: The model's accuracy has significantly increased to 74.28% after 20 epochs, compared to 57.86% after 5 epochs in step 3. This suggests that the additional layers added in step 3 needed more epochs to learn effectively from the data.
- Progress Over Time: The accuracy improves steadily over the 20 epochs, indicating that the model continues to learn and improve its predictions with more training. The plateauing of accuracy seen in the later epochs suggests that the model might be reaching the limits of what it can learn with its current configuration.
- Validation Loss: The validation loss decreases over time, which is a good sign that the model is generalizing well to unseen data. However, there are epochs where the validation loss slightly increases, which could be a sign of beginning to overfit, but it doesn't appear to be a major issue since the overall trend is downward.
- Comparing to Previous Results: The extended training has outperformed the previous 5-epoch trainings (61.56% in step 2 and 57.86% in step 3). This demonstrates the importance of sufficient training time, especially for more complex models.

From these results, the model benefits from additional training time. While more complex models can initially perform worse than simpler ones due to the requirement of more training to effectively tune the greater number of parameters, given enough epochs, they often outperform simpler models. The results of step 4 confirm this, as the model has now surpassed the performance of the model in step 2.

Step 5

```
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier

# Flatten the image data
x_train_flat = x_train.reshape(x_train.shape[0], -1)
x_test_flat = x_test.reshape(x_test.shape[0], -1)

# Reshape the labels for Naïve Bayes
y_train_flat = y_train.argmax(axis=1)
y_test_flat = y_test.argmax(axis=1)

# Train a Naïve Bayes model
naive_bayes_model = MultinomialNB()
naive_bayes_model.fit(x_train_flat, y_train_flat)
```

```
# Train a Random Forest model with 100 trees and max depth of 10
random_forest_model = RandomForestClassifier(n_estimators=100, max_depth=10,
random_state=42)
random_forest_model.fit(x_train_flat, y_train_flat)

# Predictions using Naïve Bayes
y_pred_nb = naive_bayes_model.predict(x_test_flat)
accuracy_nb = accuracy_score(y_test_flat, y_pred_nb)

# Predictions using Random Forest
y_pred_rf = random_forest_model.predict(x_test_flat)
accuracy_rf = accuracy_score(y_test_flat, y_pred_rf)
accuracy_nb, accuracy_rf
```

Part 2 - Step 5

In [17]:
```
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier

# Flatten the image data
x_train_flat = x_train.reshape(x_train.shape[0], -1)
x_test_flat = x_test.reshape(x_test.shape[0], -1)

# Reshape the labels for Naïve Bayes
y_train_flat = y_train.argmax(axis=1)
y_test_flat = y_test.argmax(axis=1)

# Train a Naïve Bayes model
naive_bayes_model = MultinomialNB()
naive_bayes_model.fit(x_train_flat, y_train_flat)

# Train a Random Forest model with 100 trees and max depth of 10
random_forest_model = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
random_forest_model.fit(x_train_flat, y_train_flat)

# Predictions using Naïve Bayes
y_pred_nb = naive_bayes_model.predict(x_test_flat)
accuracy_nb = accuracy_score(y_test_flat, y_pred_nb)

# Predictions using Random Forest
y_pred_rf = random_forest_model.predict(x_test_flat)
accuracy_rf = accuracy_score(y_test_flat, y_pred_rf)

accuracy_nb, accuracy_rf
```

Out[17]: (0.2933, 0.4331)

The accuracy rates for the different models trained on the CIFAR-10 dataset are as follows:

- Step 2 (CNN - 5 Epochs): The CNN achieved an accuracy of 61.56%.
- Step 3 (Enhanced CNN - 5 Epochs): The more complex CNN with additional layers achieved an accuracy of 57.86%.
- Step 4 (Enhanced CNN - 20 Epochs): Extending the training to 20 epochs improved the accuracy to 74.28%.
- Step 5 (Naïve Bayes and Random Forest): The Naïve Bayes model yielded an accuracy of 29.33%, and the Random Forest model achieved 43.31%.

Comparing these results:

- CNN Models (Steps 2, 3, 4): CNNs performed the best, with the extended training CNN (Step 4) achieving the highest accuracy among all models. This is expected as CNNs are designed to work with image data by capturing spatial hierarchies and feature correlations.

- Traditional Machine Learning Models (Step 5): Both Naïve Bayes and Random Forest models performed significantly worse than CNNs. This is due to their inability to capture the complex patterns and spatial relationships in image data effectively.
- Best Performing Model: The model from Step 4, the enhanced CNN trained for 20 epochs, performed the best with an accuracy rate of 74.28%. The improvement in performance from steps 2 to 4 underscores the effectiveness of deeper architectures and longer training times for complex datasets like CIFAR-10.
- However, it's also important to note that while the CNN models outperform traditional machine learning models, they also tend to be more computationally intensive and may require more data and training time to achieve these results.