This file contains explanation of what we did in each step and results of train set.

```
[ ]  # Import required packages
     import numpy as np
     import cv2
     import matplotlib.pyplot as plt
     from sklearn.metrics import classification_report
     from sklearn.linear_model import LogisticRegression
```

We import the required packages:

- numpy for numerical operations and handling arrays.
- cv2 (OpenCV) for image processing tasks.
- matplotlib.pyplot for plotting images and graphs for visualization.
- sklearn.metrics for evaluating the model with classification_report.
- sklearn.linear_model to use the LogisticRegression model for classification tasks.

1. Load the datasets

For the project, we provide a training set with 50000 images in the directory `../data/images/` with:

- noisy labels for all images provided in `../data/noisy_label.csv`;
- clean labels for the first 10000 images provided in `../data/clean_labels.csv`.

The first step is load the datasets. This is given in the stater code.

```
[5]  !unzip -q data.zip
```

- **Unzip the Data**: this is used to unzip the dataset archive named data.zip. This step is crucial for accessing the images and label files contained within the zip file.

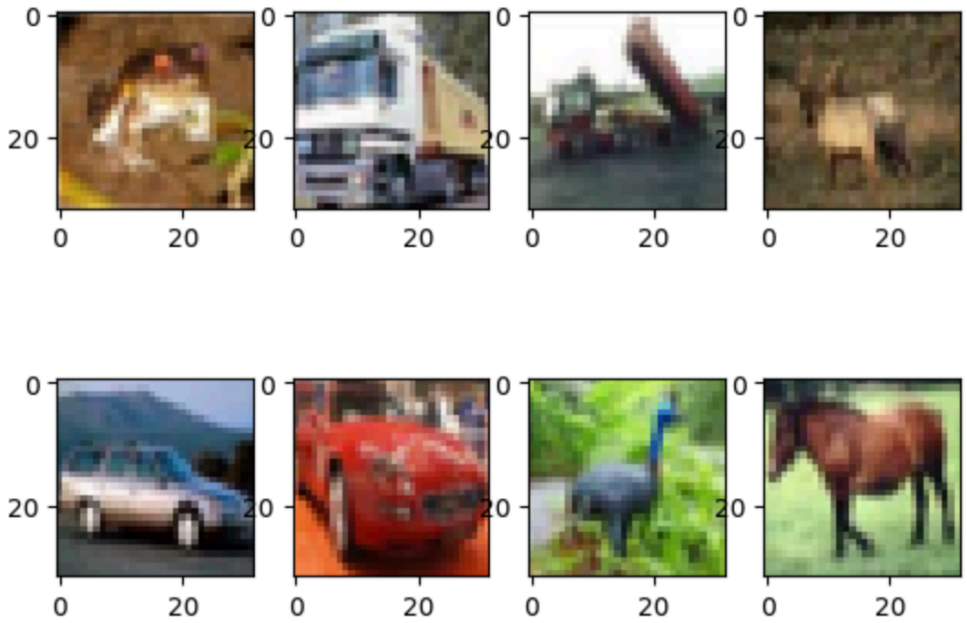
```
[6]  # [DO NOT MODIFY THIS CELL]

     # load the images
     n_img = 50000
     n_noisy = 40000
     n_clean_noisy = n_img - n_noisy
     imgs = np.empty((n_img,32,32,3))
     for i in range(n_img):
         img_fn = f'../content/data/images/{i+1:05d}.png'
         imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
     # load the labels
     clean_labels = np.genfromtxt('../content/data/clean_labels.csv', delimiter=',', dtype="int8")
     noisy_labels = np.genfromtxt('../content/data/noisy_labels.csv', delimiter=',', dtype="int8")
```

This step is crucial for preparing data for any machine learning tasks. By loading both the images and their labels into memory, we are setting the stage for further data exploration, preprocessing, and eventually feeding the data into a machine learning model for training. The distinction between clean and noisy labels suggests that part of this project involve correcting label noise, which can be an important aspect of training robust models.

For illustration, we present a small subset (of size 8) of the images with their clean and noisy labels in `clean_noisy_trainset`. You are encouraged to explore more characteristics of the label noises on the whole dataset.

```
# [DO NOT MODIFY THIS CELL]

fig = plt.figure()

ax1 = fig.add_subplot(2,4,1)
ax1.imshow(imgs[0]/255)
ax2 = fig.add_subplot(2,4,2)
ax2.imshow(imgs[1]/255)
ax3 = fig.add_subplot(2,4,3)
ax3.imshow(imgs[2]/255)
ax4 = fig.add_subplot(2,4,4)
ax4.imshow(imgs[3]/255)
ax1 = fig.add_subplot(2,4,5)
ax1.imshow(imgs[4]/255)
ax2 = fig.add_subplot(2,4,6)
ax2.imshow(imgs[5]/255)
ax3 = fig.add_subplot(2,4,7)
ax3.imshow(imgs[6]/255)
ax4 = fig.add_subplot(2,4,8)
ax4.imshow(imgs[7]/255)

# The class-label correspondence
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# print clean labels
print('Clean labels:')
print(' '.join('%5s' % classes[clean_labels[j]] for j in range(8)))
# print noisy labels
print('Noisy labels:')
print(' '.join('%5s' % classes[noisy_labels[j]] for j in range(8)))
```

```
Clean labels:
 frog truck truck  deer   car   car  bird horse
Noisy labels:
  cat   dog truck  frog   dog  ship  bird  deer
```



In this step, we visualize the first 8 images from our dataset and print their corresponding clean and noisy labels.

2. The predictive model

We consider a baseline model directly on the noisy dataset without any label corrections. RGB histogram features are extracted to fit a logistic regression model.

2.1. Baseline Model

```
[8]  # [DO NOT MODIFY THIS CELL]
     # RGB histogram dataset construction
     no_bins = 6
     bins = np.linspace(0,255,no_bins) # the range of the rgb histogram
     target_vec = np.empty(n_img)
     feature_mtx = np.empty((n_img,3*(len(bins)-1)))
     i = 0
     for i in range(n_img):
         # The target vector consists of noisy labels
         target_vec[i] = noisy_labels[i]

         # Use the numbers of pixels in each bin for all three channels as the features
         feature1 = np.histogram(imgs[i][:,:,0],bins=bins)[0]
         feature2 = np.histogram(imgs[i][:,:,1],bins=bins)[0]
         feature3 = np.histogram(imgs[i][:,:,2],bins=bins)[0]

         # Concatenate three features
         feature_mtx[i,] = np.concatenate((feature1, feature2, feature3), axis=None)
         i += 1
```

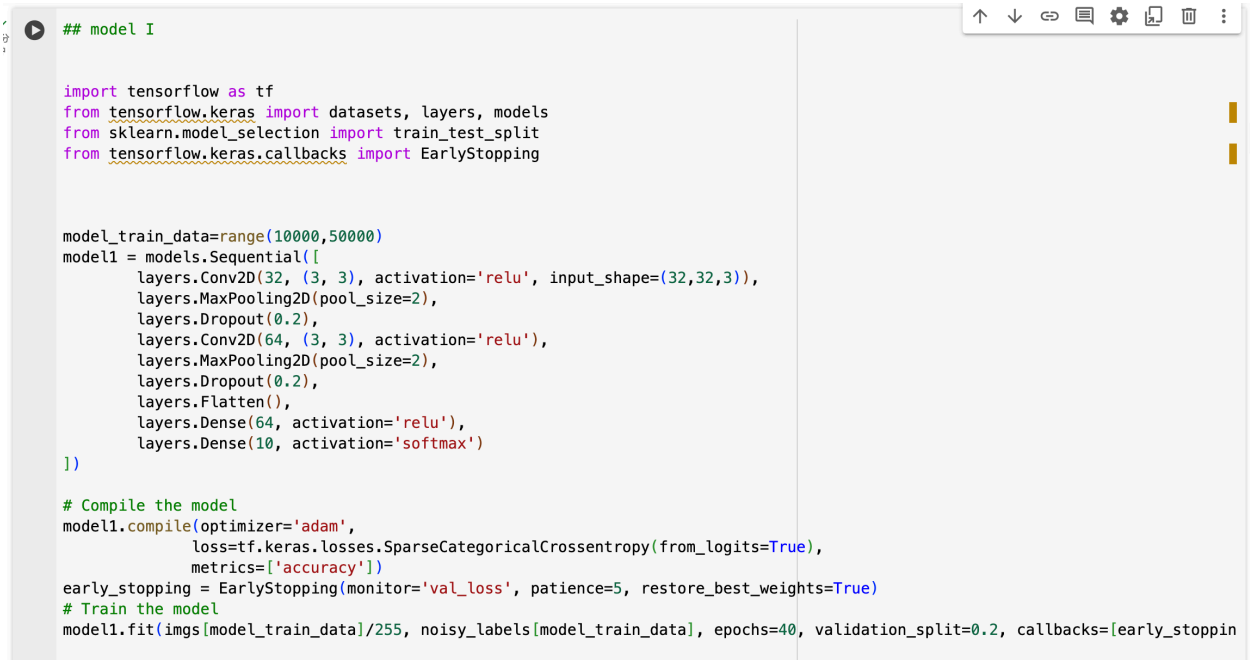
The second step is predicting models.

In this step, we constructed a dataset for a baseline model using RGB histogram features from the images and their noisy labels. This process prepares the data (feature vectors and target labels) for fitting a logistic regression model by transforming the image data into a form that represents the color distribution within each image.

```
[9]  print(imgs[0].shape)

     (32, 32, 3)
```

```
[10] # [DO NOT MODIFY THIS CELL]
     # Train a logistic regression model
     clf = LogisticRegression(random_state=0).fit(feature_mtx, target_vec)
```

This step trained a logistic regression model using the RGB histogram features (feature_mtx) and the noisy labels (target_vec) as input. This trained model can now be used to make predictions on unseen data or evaluate its performance on a test set. The use of noisy labels for training represents a baseline approach, where no preprocessing or correction of labels is done before model training.



```
## model I


import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping



model_train_data=range(10000,50000)
model1 = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,32,3)),
        layers.MaxPooling2D(pool_size=2),
        layers.Dropout(0.2),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D(pool_size=2),
        layers.Dropout(0.2),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')
])

# Compile the model
model1.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
# Train the model
model1.fit(imgs[model_train_data]/255, noisy_labels[model_train_data], epochs=40, validation_split=0.2, callbacks=[early_stoppin
```

In this step, we defined, compiled, and trained a **convolutional neural network (CNN)** model named **model1** using **TensorFlow** and **Keras**, specifically for image classification tasks. Here is the breakdown of our process:

- There is a **sequential model model1** was created, indicating a linear stack of layers. The model consists of two convolutional layers (each followed by max-pooling and dropout layers for feature extraction and regularization to prevent overfitting), a flattening layer to transform 2D feature maps to a 1D feature vector, and two dense layers for classification. The final layer uses 'softmax' activation to output probabilities for each of the 10 classes.
- input_shape=(32,32,3) specifies that each input image has dimensions of 32x32 pixels with 3 color channels (RGB).

- For Model Compilation:
  - The model was compiled with the '**Adam**' optimizer and '**sparse categorical crossentropy**' loss function, suitable for multi-class classification tasks where each label is an integer (not one-hot encoded).
  - The '**metric**' used to evaluate model performance during training and validation is accuracy.

  - An '**EarlyStopping**' callback was defined to monitor the validation loss (val_loss) and stop training if it doesn't improve after five epochs (patience=5). This helps prevent overfitting and saves training time by stopping early if the model isn't improving.
  - **restore_best_weights=True** ensures that the model's weights are rolled back to those of the epoch with the best value of the monitored metric (in this case, val_loss).
- For Model Training:
  - The model was trained using images indexed by model_train_data (indices 10000 to 49999 in imgs, effectively using the latter 40,000 images) normalized by dividing by 255 (to scale pixel values to the range [0,1]), with corresponding noisy_labels.
  - A validation split of 20% (validation_split=0.2) was used to evaluate the model on a portion of the training data not seen during the epoch's training phase.
  - Training was set to run for a maximum of 40 epochs, but could stop early if the early stopping criterion was met.

This step is a critical part of the machine learning workflow, where a CNN model is prepared and trained for classifying images based on their content, using noisy labels as the target variable for training. The model's performance is monitored on a validation set, with early stopping to mitigate overfitting.

```python
import numpy as np
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, concatenate, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import regularizers

image_input_shape = (32, 32, 3)  # Define your image dimensions
prior_label_input_shape = (1,)  # Assuming prior label is a single integer

# Define input layers
image_input = Input(shape=image_input_shape, name='image_input')
prior_label_input = Input(shape=prior_label_input_shape, name='prior_label_input')

# CNN layers for image processing
conv1 = Conv2D(32, kernel_size=(3, 3), activation='relu')(image_input)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(64, kernel_size=(3, 3), activation='relu')(pool1)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
flatten = Flatten()(pool2)

# Concatenate image features and prior label
concatenated_features = concatenate([flatten, prior_label_input])

# Fully connected layers for classification
dense1 = Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.01))(concatenated_features)
dropout1 = Dropout(0.5)(dense1)  # Adding dropout regularization
output = Dense(10, activation='softmax')(dropout1)  # Assuming num_classes is the number of output classes
```

```python
# Define the model
label_correction_model = Model(inputs=[image_input, prior_label_input], outputs=output)

# Compile the model
label_correction_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
# Train the model
label_correction_model.fit([imgs[:10000]/255, noisy_labels[:10000]], clean_labels, epochs=50, batch_size=32, validation_split=0.
corrected_labels = label_correction_model.predict([imgs, noisy_labels])
corrected_labels = np.argmax(corrected_labels, axis=1)
```

This step defines, compiles, and trains a multi-input convolutional neural network (CNN) model for label correction, integrating both image data and prior (noisy) label information. Here's a summary of the process:

- **For Model Architecture**:
  - We define two input layers: one for the images (image_input) and another for the prior labels (prior_label_input).
  - Construct a CNN pathway for image processing, including two convolutional layers each followed by ‘max pooling’.
  - Flatten the output of the CNN pathway and concatenates it with the prior label input to combine image features with label information.
  - Add fully connected (dense) layers for classification, including regularization (L2 and dropout) to prevent overfitting.

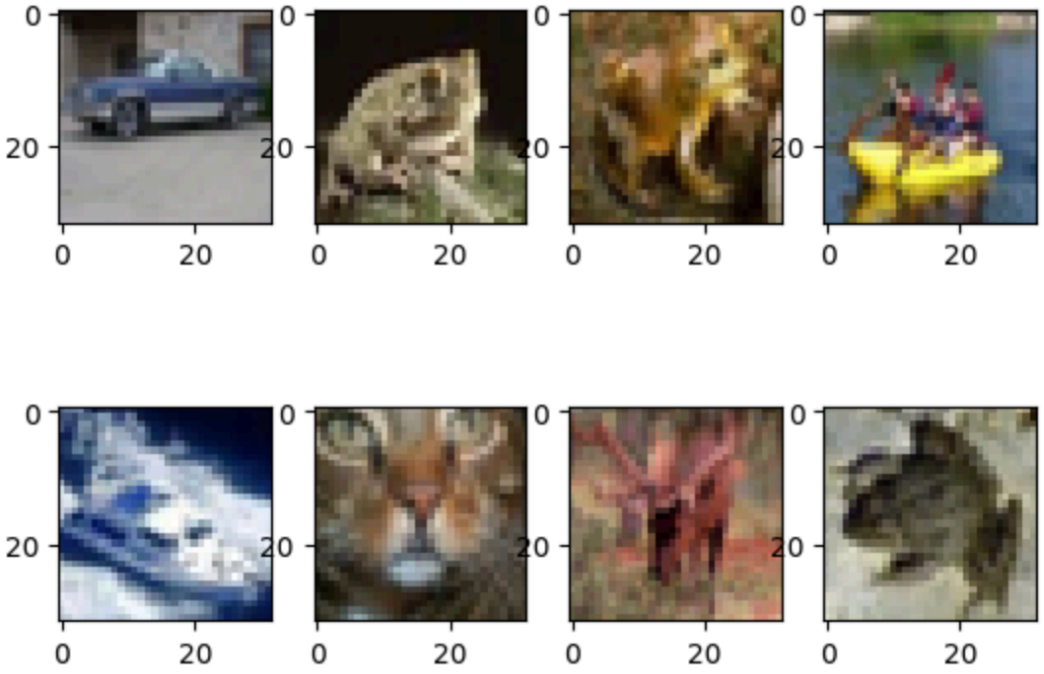
    - The output layer uses 'softmax' activation for multi-class classification (assumed to be 10 classes).
- **For Model Compilation**:
    - We compile the model with the 'Adam' optimizer, 'sparse categorical crossentropy' loss (appropriate for integer-labeled multi-class classification), and tracks accuracy as a metric.
- **For Model Training**:
    - We train the model on the first 10,000 images (normalized by dividing by 255) and their noisy labels, using the clean labels as the target for training. This subset acts as training data where the model learns to correct labels based on image features and prior label information.
    - We apply a validation split of 20% to monitor the model's performance on unseen data during training.
    - Also include an 'early stopping' callback to halt training if validation loss does not improve for three consecutive epochs, restoring weights from the best epoch to prevent overfitting.
- **For Label Prediction and Correction**:
    - We use the trained model to predict labels for all images, including those not used during training, by feeding both the images and their noisy labels as input.
    - Processes the prediction outputs (probabilities for each class) to determine the most likely class for each image (np.argmax(corrected_labels, axis=1)), resulting in corrected_labels which are the model's attempt to correct or refine the initial noisy labels based on learned image features and initial label information.

This step essentially builds and utilizes a machine learning model to refine or correct initial noisy labels based on the correlation between image content and the given labels, aiming to improve the quality of the dataset for further machine learning tasks.

```
fig = plt.figure()

ax1 = fig.add_subplot(2,4,1)
ax1.imshow(imgs[10000]/255)
ax2 = fig.add_subplot(2,4,2)
ax2.imshow(imgs[10001]/255)
ax3 = fig.add_subplot(2,4,3)
ax3.imshow(imgs[10002]/255)
ax4 = fig.add_subplot(2,4,4)
ax4.imshow(imgs[10003]/255)
ax1 = fig.add_subplot(2,4,5)
ax1.imshow(imgs[10004]/255)
ax2 = fig.add_subplot(2,4,6)
ax2.imshow(imgs[10005]/255)
ax3 = fig.add_subplot(2,4,7)
ax3.imshow(imgs[10006]/255)
ax4 = fig.add_subplot(2,4,8)
ax4.imshow(imgs[10007]/255)
```

```
# The class-label correspondence
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# print clean labels
#print('Clean labels:')
#print(' '.join('%5s' % classes[clean_labels[j]] for j in range(8)))
# print noisy labels
print('Noisy labels:')
print(' '.join('%5s' % classes[noisy_labels[j]] for j in range(10000,10008)))
print('corrected labels:')
for j in range(10000,10008):
    print(classes[corrected_labels[j]], end="  ")
```

```
Noisy labels:
  car horse   cat  frog plane   cat   cat  bird
corrected labels:
plane  dog  horse  car  car  cat  horse  cat
```



This step visualizes a set of images from the dataset and displays their corresponding noisy labels and the labels corrected by the label correction model. Here is a breakdown of process:

1. For **Image Visualization**: There are 8 images, indexed from 10,000 to 10,007 in the `imgs` array, are displayed in a 2x4 grid. Each image is normalized by dividing by 255 to scale the pixel values to the [0,1] range suitable for display with `matplotlib`.

2. For **Class–Label Correspondence**: We define a tuple `classes` that contains string representations of the 10 possible classes (e.g., 'plane', 'car', 'bird', etc.).

3. For **Noisy Labels Display**: We print the noisy labels for the displayed images, using the `classes` tuple to convert the label indices into human–readable class names. This shows the initial label assigned to each image before any correction.

4. For **Corrected Labels Display**: We print the corrected labels for the same set of images. The corrected labels were obtained by predicting with the label correction model, which aimed to improve label accuracy based on the image content and initial (noisy) labels.

This step is useful for qualitative evaluation, allowing us to compare the initial noisy labels with the corrected labels generated by the model. Such visual inspection can provide insights into the effectiveness of the label correction process in cleaning the dataset and improving the reliability of the labels for further machine learning tasks.

```
[14] model1.save("../output/model_I.h5")
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning: You are saving your model as an HDF5 file
  saving_api.save_model(
```

```
[15] ## model II


    import tensorflow as tf
    from tensorflow.keras import datasets, layers, models
    from sklearn.model_selection import train_test_split

    model_train_data=range(10000,50000)
    #clean_corrected_data=np.concatenate((clean_labels, corrected_labels[10000:]), axis=0)
    #train_labels=clean_corrected_data[model_train_data]

    model2 = models.Sequential([
            layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,32,3)),
            layers.MaxPooling2D(pool_size=2),
            layers.Dropout(0.2),
            layers.Conv2D(64, (3, 3), activation='relu'),
            layers.MaxPooling2D(pool_size=2),
            layers.Dropout(0.2),
            layers.Flatten(),
            layers.Dense(64, activation='relu'),
            layers.Dense(10, activation='softmax')
    ])

    # Compile the model
    model2.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
    # Train the model
    model2.fit(imgs[model_train_data]/255, corrected_labels[model_train_data], epochs=50, validation_split=0.2, callbacks=[early_sto
```

This step defines, compiles, and trains a **second** convolutional neural network (**CNN**) model (`**model2**`) using TensorFlow and Keras, similar to `model1` but with a crucial difference in the **training labels** used. Here's an overview:

1. For **Model Training Data**: Unlike `model1`, which used noisy labels for training, `model2` is trained on a subset of images (indices 10000 to 49999 from `imgs`) using the `corrected_labels` generated by the label correction model as the target for training. This suggests an attempt to improve model performance by training on labels that have been refined based on image content and initial label information, potentially reducing the impact of label noise.

2. For **Model Architecture**: The architecture of `model2` mirrors that of `model1`, with 2 convolutional layers (each followed by max pooling and dropout for regularization), a flattening layer, and two dense layers for classification. The final layer uses '**softmax**' activation for predicting the class probabilities of 10 possible outcomes.

3. For **Model Compilation**: The model is compiled with the '**Adam**' optimizer and '**sparse categorical crossentropy**' as the loss function. This setup is appropriate for multi-class classification tasks with labels as integers. The model's performance will be evaluated based on accuracy.

4. **Model Training**:
   - The model is trained using the corrected labels for the images, which have been normalized by dividing pixel values by 255 to scale them to the [0,1] range. This normalization is standard practice to facilitate model learning.
   - A **validation split** of 20% is used to monitor the model's performance on unseen data during training, helping to gauge its generalization ability.
   - An `**EarlyStopping**` callback is included to halt training if the validation loss does not improve for five consecutive epochs, mitigating overfitting by restoring the model to its best-performing state.

This step aims to leverage the potentially higher quality of the corrected labels to train a CNN model that might exhibit better performance on classification tasks compared to a model trained on the original, noisy labels. It represents an iterative approach to model development, where insights and improvements from previous steps are used to enhance subsequent models.

```
[20] model2.save("../output/model_II.h5")
```

For the convenience of evaluation, we write the following function `predictive_model` that does the label prediction. **For your predictive model, feel free to modify the function, but make sure the function takes an RGB image of numpy.array format with dimension $32 \times 32 \times 3$ as input, and returns one single label as output.**

```
[21] # [DO NOT MODIFY THIS CELL]
     def baseline_model(image):
         '''
         This is the baseline predictive model that takes in the image and returns a label prediction
         '''
         feature1 = np.histogram(image[:,:,0],bins=bins)[0]
         feature2 = np.histogram(image[:,:,1],bins=bins)[0]
         feature3 = np.histogram(image[:,:,2],bins=bins)[0]
         feature = np.concatenate((feature1, feature2, feature3), axis=None).reshape(1,-1)
         return clf.predict(feature)
```

This step defines a function named `baseline_model` that serves as a predictive model for classifying images. The function encapsulates the process of making predictions based on RGB histogram features of an input image using a previously trained logistic regression model (`clf`).
It demonstrates how a machine learning model trained on specific features (in this case, RGB histogram features) can be integrated into a callable function for easy use.

2.2. Model I

```
[22] # [BUILD A MORE SOPHISTICATED PREDICTIVE MODEL]

     # write your code here...

     def model_I(image):
         '''
         This function should takes in the image of dimension 32*32*3 as input and returns a label prediction
         '''
         # write your code here...
         image=np.expand_dims(image, axis=0)
         prediction=model1.predict(image/255, verbose=0)
         return np.argmax(prediction)
```

2.3. Model II

```
[23] # [ADD WEAKLY SUPERVISED LEARNING FEATURE TO MODEL I]

     # write your code here...

     def model_II(image):
         '''
         This function should takes in the image of dimension 32*32*3 as input and returns a label prediction
         '''
         # write your code here...
         image=np.expand_dims(image, axis=0)
         prediction=model2.predict(image/255, verbose=0)
         return np.argmax(prediction)
```

These steps define two functions, `model_I` and `model_II`, each designed to make label predictions for a single input image using two different trained convolutional neural network (CNN) models, `model1` and `model2`, respectively.

For the **`model_I` Function**:
We Use the first CNN model to predict the class of a given image. `model1` is being trained on a subset of images with noisy labels. Then returns the predicted class label as an integer.

The differences of '**model_II' Function** is it uses the second CNN model for predictions. `model2` is being trained on a subset of images with labels corrected by the label correction model.

This shows a practical application of CNN models in image classification tasks, providing a straightforward interface for predicting the class of an image with different models trained under varying conditions.

3. Evaluation

For assessment, we will evaluate your final model on a hidden test dataset with clean labels by the `evaluation` function defined as follows. Although you will not have the access to the test set, the function would be useful for the model developments. For example, you can split the small training set, using one portion for weakly supervised learning and the other for validation purpose.

```
[24] # [DO NOT MODIFY THIS CELL]
     def evaluation(model, test_labels, test_imgs):
         y_true = test_labels
         y_pred = []
         for image in test_imgs:
             y_pred.append(model(image))
         print(classification_report(y_true, y_pred))
```

This step defines an `**evaluation**` function that provides a way to assess the performance of a given predictive model using a test dataset with clean labels.

```
[25] # [DO NOT MODIFY THIS CELL]

     ## Performence for baseline model -- storage, memory and time

     import time
     import psutil

     # Record the start time and start memory for baseline model
     start_time = time.time()
     start_mem = psutil.Process().memory_info().rss
```

```
n_test = 10000
test_labels = np.genfromtxt('../content/data/clean_labels.csv', delimiter=',', dtype="int8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../content/data/images/{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)

baseline_predict=[]
for test_img in test_imgs:
  baseline_predict.append(int(baseline_model(test_img)[0]))
baseline_predict

end_time = time.time()
end_mem = psutil.Process().memory_info().rss

# Calculate the elapsed time and the used memory
elapsed_time = end_time - start_time
used_mem = end_mem - start_mem

# Print the elapsed time and used memory
print("Time taken:", elapsed_time, "seconds")
print("Function used {:.2f} MB of memory.".format((used_mem) / (1024 * 1024)))
print("Total memory available: {:.2f} GB".format(psutil.virtual_memory().total / (1024 * 1024 * 1024)))
print("CPU usage: {:.2f}%".format(psutil.cpu_percent()))
```

```
Time taken: 4.923543691635132 seconds
Function used 0.00 MB of memory.
Total memory available: 12.67 GB
CPU usage: 35.90%
```

```
## Performence for model I -- storage, memory and time

import time
import psutil

# Record the start time and start memory for model I
start_time = time.time()
start_mem = psutil.Process().memory_info().rss

n_test = 10000
test_labels = np.genfromtxt('../content/data/clean_labels.csv', delimiter=',', dtype="int8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../content/data/images/{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)


model_I_predict=np.argmax(model1.predict(test_imgs), axis=1)

end_time = time.time()
end_mem = psutil.Process().memory_info().rss

# Calculate the elapsed time and the used memory
elapsed_time = end_time - start_time
used_mem = end_mem - start_mem

# Print the elapsed time and used memory
print("Time taken:", elapsed_time, "seconds")
print("Function used {:.2f} MB of memory.".format((used_mem) / (1024 * 1024)))
print("Total memory available: {:.2f} GB".format(psutil.virtual_memory().total / (1024 * 1024 * 1024)))
print("CPU usage: {:.2f}%".format(psutil.cpu_percent()))
```

```
313/313 [==============================] - 4s 11ms/step
Time taken: 5.291825771331787 seconds
Function used 124.16 MB of memory.
Total memory available: 12.67 GB
CPU usage: 63.30%
```

```python
## Performence for model II -- storage, memory and time

import time
import psutil

# Record the start time and start memory for model II
start_time = time.time()
start_mem = psutil.Process().memory_info().rss

n_test = 10000
test_labels = np.genfromtxt('../content/data/clean_labels.csv', delimiter=',', dtype="int8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../content/data/images/{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)


model_II_predict=np.argmax(model2.predict(test_imgs), axis=1)

end_time = time.time()
end_mem = psutil.Process().memory_info().rss

# Calculate the elapsed time and the used memory
elapsed_time = end_time - start_time
used_mem = end_mem - start_mem

# Print the elapsed time and used memory
print("Time taken:", elapsed_time, "seconds")
print("Function used {:.2f} MB of memory.".format((used_mem) / (1024 * 1024)))
print("Total memory available: {:.2f} GB".format(psutil.virtual_memory().total / (1024 * 1024 * 1024)))
print("CPU usage: {:.2f}%".format(psutil.cpu_percent()))
```

```
313/313 [==============================] - 5s 17ms/step
Time taken: 15.93086552619934 seconds
Function used 259.60 MB of memory.
Total memory available: 12.67 GB
CPU usage: 12.30%
```

The above steps measure and report the performance, in terms of execution time and memory usage, of generating predictions for a test dataset using three different models: the baseline model, model I, and model II. This is done within a Python script that utilizes the `time` and `psutil` libraries to capture relevant metrics.

Outputs include the execution time (which affects user experience and operational costs), memory usage (indicating the model's footprint and scalability), and system resource availability (to understand the model's impact on the overall system).

This evaluation helps in understanding the trade-offs between model complexity, accuracy, and operational efficiency, which is crucial for deploying machine learning models in production environments.

```
[28] ## Performence for baseline model -- prediction accuracy

     evaluation(baseline_model, test_labels[:n_test], test_imgs)
```

	precision	recall	f1-score	support
0	0.32	0.43	0.37	1005
1	0.18	0.29	0.22	974
2	0.22	0.04	0.07	1032
3	0.19	0.12	0.14	1016
4	0.24	0.48	0.32	999
5	0.22	0.13	0.16	937
6	0.26	0.35	0.30	1030
7	0.29	0.04	0.07	1001
8	0.28	0.43	0.34	1025
9	0.19	0.11	0.14	981
accuracy			0.24	10000
macro avg	0.24	0.24	0.21	10000
weighted avg	0.24	0.24	0.21	10000

The average precision of baseline model is 0.24 .

```
[ ]  ## Performence for model I -- prediction accuracy

     evaluation(model_I, test_labels[:n_test], test_imgs)
```

	precision	recall	f1-score	support
0	0.55	0.54	0.55	1005
1	0.56	0.68	0.62	974
2	0.34	0.42	0.38	1032
3	0.38	0.28	0.33	1016
4	0.43	0.33	0.37	999
5	0.39	0.35	0.37	937
6	0.53	0.64	0.58	1030
7	0.48	0.62	0.54	1001
8	0.65	0.50	0.56	1025
9	0.57	0.50	0.53	981
accuracy			0.49	10000
macro avg	0.49	0.49	0.48	10000
weighted avg	0.49	0.49	0.48	10000

The average precision of model I is 0.49 .

```
[ ]  ## Performence for model II -- prediction accuracy

     evaluation(model_II, test_labels[:n_test], test_imgs)
```

	precision	recall	f1-score	support
0	0.27	0.92	0.41	1005
1	0.63	0.72	0.67	974
2	0.53	0.19	0.28	1032
3	0.52	0.05	0.09	1016
4	0.73	0.07	0.12	999
5	0.41	0.55	0.47	937
6	0.91	0.21	0.34	1030
7	0.49	0.70	0.58	1001
8	0.67	0.47	0.55	1025
9	0.53	0.66	0.59	981
accuracy			0.45	10000
macro avg	0.57	0.45	0.41	10000
weighted avg	0.57	0.45	0.41	10000

The average precision of model II is 0.57 .

The steps described above evaluate the performance of three different models—baseline, model I, and model II—using prediction accuracy on a dataset.

The purpose of these steps is to provide a comprehensive understanding of each model's effectiveness and efficiency. By examining both the computational resources required and the accuracy of each model, we can make informed decisions about the suitability for deployment in real-world applications.

label_prediction.csv (on the test set):

It should contain 4 columns with the following column names: "Index", "Baseline", "Model I", and "Model II".

```
# (38 min)

n_test = 10000
test_labels = np.genfromtxt('../content/test_data/test_labels.csv', delimiter=',', dtype="int8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../content/test_data/test_images/test{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)

test_images_normalized = test_imgs / 255.0
```

```
import pandas as pd
Index = []
baseline_pred = []
model_I_pred = []
model_II_pred =[]
#test to change to 100

for i in range(10000):
    img = test_imgs[i]
    Index.append(i+1)
    baseline_pred.append(baseline_model(img))
    model_I_pred.append(model_I(img))
    model_II_pred.append(model_II(img))

label_prediction_df = pd.DataFrame({
    "baseline": baseline_pred,
    "model_I": model_I_pred,
    "model_II": model_II_pred
})
label_prediction_df.to_csv("../output/label_prediction.csv", index=True, index_label="index")
```

This step is generating a label_prediction.csv file.

We normalize the pixel values of test images by dividing each pixel value by 255.0 to ensure that pixel values fall within the range of 0 to 1, desirable for neural network training. Then the loop iterates over each test image and appends both index (i+1) to the 'Index' list and predictions to respective lists of 'baseline_pred, 'model_I_pred', 'model_II_pred'. Finally a Pandas DataFrame named 'label_prediction_df' with columns 'baseline', 'model_I', 'model_II' with their corresponding prediction lists is created . The DataFrame can be seen as 'label_prediction.csv' file in the "../output/" directory.

Results of train set:

	Baseline Model	Model I	Model II
Accuracy	0.24	0.49	0.57
Time	12 s	7 min	13 min
Memory (MB)	0.00	124.16	259.60