Name: **Harsh Pramod Padyal**

Roll No : **40**

Div: **D20B**

Experiment 03

AIM: To build a Cognitive based application to acquire knowledge through images for a Customer service application/ Insurance/ Healthcare Application/ Smarter Cities/Government etc.

Theory

A **cognitive-based application** is an intelligent system that can learn from data and provide meaningful insights, similar to how humans learn and make decisions. In this experiment, we focus on building such an application that can acquire **knowledge through images** and use it for different real-world domains like **Customer Service, Insurance, Healthcare, Smarter Cities, and Government services**.

Cognitive Learning with Images

Images are a powerful source of information. With the help of **computer vision** and **deep learning**, machines can recognize patterns, objects, and categories from images. This helps in tasks such as:

- Identifying documents or ID cards in customer service.
- Detecting damages in insurance claims.
- Recognizing medical scans in healthcare.
- Monitoring traffic or waste management in smarter cities.
- Analyzing satellite images for government projects.

Steps Involved in Building the Application

1. **Data Acquisition** – Images are collected from reliable sources (e.g., Kaggle dataset). These images act as knowledge input for the system.
2. **Data Preprocessing** – Images are cleaned, resized, and balanced to ensure the model learns fairly from all categories.
3. **Label Encoding** – Each category (e.g., type of food, document, or object) is converted into machine-understandable numerical form.
4. **Model Selection** – A **Vision Transformer (ViT)** or similar pretrained model is used. These models are powerful because they are trained on large image datasets and can be fine-tuned for specific tasks.
5. **Training** – The dataset is split into training and testing sets. The model learns patterns from the training set.
6. **Evaluation** – The model is tested using unseen images to measure its accuracy and reliability. Metrics like **accuracy and confusion matrix** are used.
7. **Prediction/Inference** – Once trained, the system can analyze a new image and provide a prediction, thereby "acquiring knowledge" automatically.

Why Vision Transformer (ViT)?

Traditional models like CNNs (Convolutional Neural Networks) are good for image recognition, but **Transformers** (originally made for NLP) are now widely used for vision tasks because they:

- Understand global relationships in images.

- Work well with large and complex datasets.
- Provide high accuracy with pretrained weights.

Applications

- **Customer Service:** Auto-verification of documents, image-based query solving.
- **Insurance:** Detecting car or property damage from images.
- **Healthcare:** Disease detection from medical scans.
- **Smarter Cities:** Smart surveillance, traffic monitoring.
- **Government:** Land use analysis, digital governance solutions.

Importing Required Libraries

```
!pip install -U -q evaluate transformers datasets>=2.14.5 accelerate>=0.27 2>/dev/null
```

Upload Kaggle API Key

```
from google.colab import files
files.upload()
```

Choose Files kaggle (3).json

- **kaggle (3).json**(application/json) - 67 bytes, last modified: 8/30/2025 - 100% done

```
Saving kaggle (3).json to kaggle (3).json
{'kaggle (3).json': b'{"username":"robstark143","key":"cfc32d778a754be45dee9d7207d5eca4"}'}
```

Setup Kaggle Credentials

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

Download Dataset from Kaggle

```
!kaggle datasets download -d iamsouravbanerjee/indian-food-images-dataset
```

```
Dataset URL: https://www.kaggle.com/datasets/iamsouravbanerjee/indian-food-images-dataset
License(s): other
Downloading indian-food-images-dataset.zip to /content
 90% 318M/355M [00:00<00:00, 516MB/s]
100% 355M/355M [00:00<00:00, 528MB/s]
```

Unzip Dataset

```
!unzip indian-food-images-dataset.zip -d indian-food-images-dataset
```

Show hidden output

Import Required Libraries

```
# Importing necessary libraries and modules
import warnings  # Import the 'warnings' module for handling warnings
warnings.filterwarnings("ignore")  # Ignore warnings during execution
```

```
import gc  # Import the 'gc' module for garbage collection
import numpy as np  # Import NumPy for numerical operations
import pandas as pd  # Import Pandas for data manipulation
import itertools  # Import 'itertools' for iterators and looping
from collections import Counter  # Import 'Counter' for counting elements
import matplotlib.pyplot as plt  # Import Matplotlib for data visualization
from sklearn.metrics import (  # Import various metrics from scikit-learn
    accuracy_score,  # For calculating accuracy
    roc_auc_score,  # For ROC AUC score
    confusion_matrix,  # For confusion matrix
    classification_report,  # For classification report
    f1_score  # For F1 score
)
# Import custom modules and classes
from imblearn.over_sampling import RandomOverSampler # import RandomOverSampler
import accelerate # Import the 'accelerate' module
import evaluate  # Import the 'evaluate' module
from datasets import Dataset, Image, ClassLabel  # Import custom 'Dataset', 'ClassLabel', and 'Image' classes
from transformers import (  # Import various modules from the Transformers library
    TrainingArguments,  # For training arguments
    Trainer,  # For model training
    ViTImageProcessor,  # For processing image data with ViT models
    ViTForImageClassification,  # ViT model for image classification
    DefaultDataCollator  # For collating data in the default way
)
import torch  # Import PyTorch for deep learning
from torch.utils.data import DataLoader  # For creating data loaders
from torchvision.transforms import (  # Import image transformation functions
    CenterCrop,  # Center crop an image
    Compose,  # Compose multiple image transformations
    Normalize,  # Normalize image pixel values
    RandomRotation,  # Apply random rotation to images
    RandomResizedCrop,  # Crop and resize images randomly
    RandomHorizontalFlip,  # Apply random horizontal flip
    RandomAdjustSharpness,  # Adjust sharpness randomly
    Resize,  # Resize images
    ToTensor  # Convert images to PyTorch tensors
)
```

Image Preprocessing Setup

```
# Import the necessary module from the Python Imaging Library (PIL).
from PIL import ImageFile

# Enable the option to load truncated images.
# This setting allows the PIL library to attempt loading images even if they are corrupted or incomplete.
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

Load Image Files and Labels

```
# use https://huggingface.co/docs/datasets/image_load for reference

image_dict = {}
# Define the list of file names
from pathlib import Path
from tqdm import tqdm
file_names = []
labels = []
for file in sorted((Path('/content/indian-food-images-dataset/Indian Food Images/Indian Food Images').glob('*/*.jpg'
    file_names.append(str(file))
    label = str(file).split('/')[-2]
```

```
    labels.append(label)
print(len(file_names), len(labels))

# Create a pandas dataframe from the collected file names and labels
df = pd.DataFrame.from_dict({"image": file_names, "label": labels})
print(df.shape)
```

```
4000 4000
(4000, 2)
```

Preview Dataset

```
df.head()
```

	image	label
0	/content/indian-food-images-dataset/Indian Foo...	adhirasam
1	/content/indian-food-images-dataset/Indian Foo...	adhirasam
2	/content/indian-food-images-dataset/Indian Foo...	adhirasam
3	/content/indian-food-images-dataset/Indian Foo...	adhirasam
4	/content/indian-food-images-dataset/Indian Foo...	adhirasam

Next steps: Generate code with df | View recommended plots | New interactive sheet

Check Unique Labels

```
df['label'].unique()
```

```
array(['adhirasam', 'aloo_gobi', 'aloo_matar', 'aloo_methi',
       'aloo_shimla_mirch', 'aloo_tikki', 'anarsa', 'ariselu',
       'bandar_laddu', 'basundi', 'bhatura', 'bhindi_masala', 'biryani',
       'boondi', 'butter_chicken', 'chak_hao_kheer', 'cham_cham',
       'chana_masala', 'chapati', 'chhena_kheeri', 'chicken_razala',
       'chicken_tikka', 'chicken_tikka_masala', 'chikki',
       'daal_baati_churma', 'daal_puri', 'dal_makhani', 'dal_tadka',
       'dharwad_pedha', 'doodhpak', 'double_ka_meetha', 'dum_aloo',
       'gajar_ka_halwa', 'gavvalu', 'ghevar', 'gulab_jamun', 'imarti',
       'jalebi', 'kachori', 'kadai_paneer', 'kadhi_pakoda', 'kajjikaya',
       'kakinada_khaja', 'kalakand', 'karela_bharta', 'kofta',
       'kuzhi_paniyaram', 'lassi', 'ledikeni', 'litti_chokha', 'lyangcha',
       'maach_jhol', 'makki_di_roti_sarson_da_saag', 'malapua',
       'misi_roti', 'misti_doi', 'modak', 'mysore_pak', 'naan',
       'navrattan_korma', 'palak_paneer', 'paneer_butter_masala',
       'phirni', 'pithe', 'poha', 'poornalu', 'pootharekulu',
       'qubani_ka_meetha', 'rabri', 'ras_malai', 'rasgulla', 'sandesh',
       'shankarpali', 'sheer_korma', 'sheera', 'shrikhand', 'sohan_halwa',
       'sohan_papdi', 'sutar_feni', 'unni_appam'], dtype=object)
```

Balance Dataset (Random Oversampling)

```
# random oversampling of minority class
# 'y' contains the target variable (label) we want to predict
y = df[['label']]

# Drop the 'label' column from the DataFrame 'df' to separate features from the target variable
df = df.drop(['label'], axis=1)

# Create a RandomOverSampler object with a specified random seed (random_state=83)
```

```
ros = RandomOverSampler(random_state=83)

# Use the RandomOverSampler to resample the dataset by oversampling the minority class
# 'df' contains the feature data, and 'y_resampled' will contain the resampled target variable
df, y_resampled = ros.fit_resample(df, y)

# Delete the original 'y' variable to save memory as it's no longer needed
del y

# Add the resampled target variable 'y_resampled' as a new 'label' column in the DataFrame 'df'
df['label'] = y_resampled

# Delete the 'y_resampled' variable to save memory as it's no longer needed
del y_resampled

# Perform garbage collection to free up memory used by discarded variables
gc.collect()

print(df.shape)
```

```
(4000, 2)
```

Convert Pandas DataFrame to HuggingFace Dataset

```
# Create a dataset from a Pandas DataFrame.
dataset = Dataset.from_pandas(df).cast_column("image", Image())
```

Display Sample Image

```
# Display the first image in the dataset
dataset[0]["image"]
```

Preview Label List

```
# Extracting a subset of elements from the 'labels' list using slicing.
# The slicing syntax [:5] selects elements from the beginning up to (but not including) the 5th element.
# This will give us the first 5 elements of the 'labels' list.
# The result will be a new list containing these elements.
labels_subset = labels[:5]

# Printing the subset of labels to inspect the content.
print(labels_subset)
```

```
['adhirasam', 'adhirasam', 'adhirasam', 'adhirasam', 'adhirasam']
```

Prepare Label Mappings (Label → ID & ID → Label)

```

```

```
# Create a list of unique labels by converting 'labels' to a set and then back to a list
labels_list = sorted(list(set(labels)))

# Initialize empty dictionaries to map labels to IDs and vice versa
label2id, id2label = dict(), dict()

# Iterate over the unique labels and assign each label an ID, and vice versa
for i, label in enumerate(labels_list):
    label2id[label] = i  # Map the label to its corresponding ID
    id2label[i] = label  # Map the ID to its corresponding label

# Print the resulting dictionaries for reference
print("Mapping of IDs to Labels:", id2label, '\n')
print("Mapping of Labels to IDs:", label2id)
```

```
Mapping of IDs to Labels: {0: 'adhirasam', 1: 'aloo_gobi', 2: 'aloo_matar', 3: 'aloo_methi', 4: 'aloo_shimla_mir

Mapping of Labels to IDs: {'adhirasam': 0, 'aloo_gobi': 1, 'aloo_matar': 2, 'aloo_methi': 3, 'aloo_shimla_mirch'
```

Create Class Labels and Mapping Functions

```
# Creating classlabels to match labels to IDs
ClassLabels = ClassLabel(num_classes=len(labels_list), names=labels_list)

# Mapping labels to IDs
def map_label2id(example):
    example['label'] = ClassLabels.str2int(example['label'])
    return example

dataset = dataset.map(map_label2id, batched=True)

# Casting label column to ClassLabel Object
dataset = dataset.cast_column('label', ClassLabels)

# Splitting the dataset into training and testing sets using an 60-40 split ratio.
dataset = dataset.train_test_split(test_size=0.4, shuffle=True, stratify_by_column="label")

# Extracting the training data from the split dataset.
train_data = dataset['train']

# Extracting the testing data from the split dataset.
test_data = dataset['test']
```

```
Map: 100%                                                    4000/4000 [00:00<00:00, 88754.25 examples/s]

Casting the dataset: 100%                                    4000/4000 [00:00<00:00, 161320.94 examples/s]
```

Load Pretrained Vision Transformer (ViT) Model & Processor

```
model_str = "dima806/indian_food_image_detection" #'google/vit-base-patch16-224-in21k'

# Create a processor for ViT model input from the pre-trained model
processor = ViTImageProcessor.from_pretrained(model_str)

# Retrieve the image mean and standard deviation used for normalization
image_mean, image_std = processor.image_mean, processor.image_std

# Get the size (height) of the ViT model's input images
size = processor.size["height"]
print("Size: ", size)
```

```
# Define a normalization transformation for the input images
normalize = Normalize(mean=image_mean, std=image_std)

# Define a set of transformations for training data
_train_transforms = Compose(
    [
        Resize((size, size)),             # Resize images to the ViT model's input size
        RandomRotation(90),               # Apply random rotation
        RandomAdjustSharpness(2),         # Adjust sharpness randomly
        RandomHorizontalFlip(0.5),        # Random horizontal flip
        ToTensor(),                       # Convert images to tensors
        normalize                         # Normalize images using mean and std
    ]
)

# Define a set of transformations for validation data
_val_transforms = Compose(
    [
        Resize((size, size)),             # Resize images to the ViT model's input size
        ToTensor(),                       # Convert images to tensors
        normalize                         # Normalize images using mean and std
    ]
)

# Define a function to apply training transformations to a batch of examples
def train_transforms(examples):
    examples['pixel_values'] = [_train_transforms(image.convert("RGB")) for image in examples['image']]
    return examples

# Define a function to apply validation transformations to a batch of examples
def val_transforms(examples):
    examples['pixel_values'] = [_val_transforms(image.convert("RGB")) for image in examples['image']]
    return examples
```

```
Fetching 1 files: 100%                                                  1/1 [00:00<00:00,  5.86it/s]
preprocessor_config.json: 100%                                          578/578 [00:00<00:00, 60.5kB/s]
Size:  224
```

Apply Train and Validation Transforms

```
# Set the transforms for the training data
train_data.set_transform(train_transforms)

# Set the transforms for the test/validation data
test_data.set_transform(val_transforms)
```

Define Data Collator for Training

```
# Set the transforms for the training data
train_data.set_transform(train_transforms)

# Set the transforms for the test/validation data
test_data.set_transform(val_transforms)
# Define a collate function that prepares batched data for model training.
def collate_fn(examples):
    # Stack the pixel values from individual examples into a single tensor.
    pixel_values = torch.stack([example["pixel_values"] for example in examples])

    # Convert the label strings in examples to corresponding numeric IDs using label2id dictionary.
```

```
    labels = torch.tensor([example['label'] for example in examples])

    # Return a dictionary containing the batched pixel values and labels.
    return {"pixel_values": pixel_values, "labels": labels}
```

Initialize Vision Transformer Model for Classification

```
# Create a ViTForImageClassification model from a pretrained checkpoint with a specified number of output labels.
model = ViTForImageClassification.from_pretrained(model_str, num_labels=len(labels_list))

# Configure the mapping of class labels to their corresponding indices for later reference.
model.config.id2label = id2label
model.config.label2id = label2id

# Calculate and print the number of trainable parameters in millions for the model.
print(model.num_parameters(only_trainable=True) / 1e6)
```

```
config.json:          4.33k/?  [00:00<00:00,  290kB/s]
model.safetensors:  100%                                                343M/343M  [00:01<00:00,  274MB/s]
85.860176
```

Define Evaluation Metric Function

```
# Load the accuracy metric from a module named 'evaluate'
accuracy = evaluate.load("accuracy")

# Define a function 'compute_metrics' to calculate evaluation metrics
def compute_metrics(eval_pred):
    # Extract model predictions from the evaluation prediction object
    predictions = eval_pred.predictions

    # Extract true labels from the evaluation prediction object
    label_ids = eval_pred.label_ids

    # Calculate accuracy using the loaded accuracy metric
    # Convert model predictions to class labels by selecting the class with the highest probability (argmax)
    predicted_labels = predictions.argmax(axis=1)

    # Calculate accuracy score by comparing predicted labels to true labels
    acc_score = accuracy.compute(predictions=predicted_labels, references=label_ids)['accuracy']

    # Return the computed accuracy as a dictionary with the key "accuracy"
    return {
        "accuracy": acc_score
    }
```

```
Downloading builder script:          4.20k/?  [00:00<00:00,  402kB/s]
```

Re-install/Verify Transformers & Dependencies

```
!pip -q install -U transformers accelerate datasets evaluate
import transformers, accelerate, datasets, evaluate
print("Transformers:", transformers.__version__)
from transformers import TrainingArguments
```

```
Transformers: 4.56.0
```

Setup Training Arguments

```
# Define the name of the evaluation metric to be used during training and evaluation.
metric_name = "accuracy"

# Define the name of the model, which will be used to create a directory for saving model checkpoints and outputs.
model_name = "indian_food_image_detection"

# Define the number of training epochs for the model.
num_train_epochs = 20

# Create an instance of TrainingArguments to configure training settings.
args = TrainingArguments(
    # Specify the directory where model checkpoints and outputs will be saved.
    output_dir=model_name,

    # Specify the directory where training logs will be stored.
    logging_dir='./logs',

    # Define the evaluation strategy, which is performed at the end of each epoch.
    eval_strategy="epoch",

    # Set the learning rate for the optimizer.
    learning_rate=1e-6,

    # Define the batch size for training on each device.
    per_device_train_batch_size=64,

    # Define the batch size for evaluation on each device.
    per_device_eval_batch_size=32,

    # Specify the total number of training epochs.
    num_train_epochs=num_train_epochs,

    # Apply weight decay to prevent overfitting.
    weight_decay=0.02,

    # Set the number of warm-up steps for the learning rate scheduler.
    warmup_steps=50,

    # Disable the removal of unused columns from the dataset.
    remove_unused_columns=False,

    # Define the strategy for saving model checkpoints (per epoch in this case).
    save_strategy='epoch',

    # Load the best model at the end of training.
    load_best_model_at_end=True,

    # Limit the total number of saved checkpoints to save space.
    save_total_limit=1,

    # Specify that training progress should not be reported.
    report_to="none"
)
```

Initialize Trainer

```
trainer = Trainer(
    model,
    args,
```

```
    train_dataset=train_data,
    eval_dataset=test_data,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    tokenizer=processor,
)
```

Evaluate Model Before Training

```
trainer.evaluate()
```


```
{'eval_loss': 3.1410443782806396,
 'eval_model_preparation_time': 0.0042,
 'eval_accuracy': 0.76625,
 'eval_runtime': 40.2098,
 'eval_samples_per_second': 39.791,
 'eval_steps_per_second': 1.243}
```

Train the Model

```
# Start training the model using the trainer object.
trainer.train()
```

[760/760 45:32, Epoch 20/20]

Epoch	Training Loss	Validation Loss	Model Preparation Time	Accuracy
1	No log	3.139473	0.004200	0.766250
2	No log	3.135326	0.004200	0.766250
3	No log	3.131135	0.004200	0.766875
4	No log	3.126826	0.004200	0.768125
5	No log	3.123134	0.004200	0.766875
6	No log	3.119579	0.004200	0.766875
7	No log	3.116225	0.004200	0.768125
8	No log	3.113201	0.004200	0.767500
9	No log	3.110483	0.004200	0.767500
10	No log	3.108049	0.004200	0.766875
11	No log	3.105657	0.004200	0.767500
12	No log	3.103495	0.004200	0.767500
13	No log	3.101722	0.004200	0.767500
14	3.077200	3.100129	0.004200	0.766875
15	3.077200	3.098818	0.004200	0.766875
16	3.077200	3.097742	0.004200	0.768125
17	3.077200	3.096916	0.004200	0.767500
18	3.077200	3.096251	0.004200	0.766875
19	3.077200	3.095862	0.004200	0.767500
20	3.077200	3.095746	0.004200	0.767500

[50/50 03:00]

```
TrainOutput(global_step=760, training_loss=3.066607746325041, metrics={'train_runtime': 2740.2134,
'train_samples_per_second': 17.517, 'train_steps_per_second': 0.277, 'total_flos': 3.722215845003264e+18,
'train_loss': 3.066607746325041, 'epoch': 20.0})
```

Evaluate Model After Training

```
trainer.evaluate()
```

[50/50 00:26]

```
{'eval_loss': 3.0957460403442383,
 'eval_model_preparation_time': 0.0042,
 'eval_accuracy': 0.7675,
 'eval_runtime': 27.2294,
 'eval_samples_per_second': 58.76,
 'eval_steps_per_second': 1.836,
 'epoch': 20.0}
```

Generate Predictions on Test Data

```
# Use the trained 'trainer' to make predictions on the 'test_data'.
outputs = trainer.predict(test_data)

# Print the metrics obtained from the prediction outputs.
print(outputs.metrics)
```

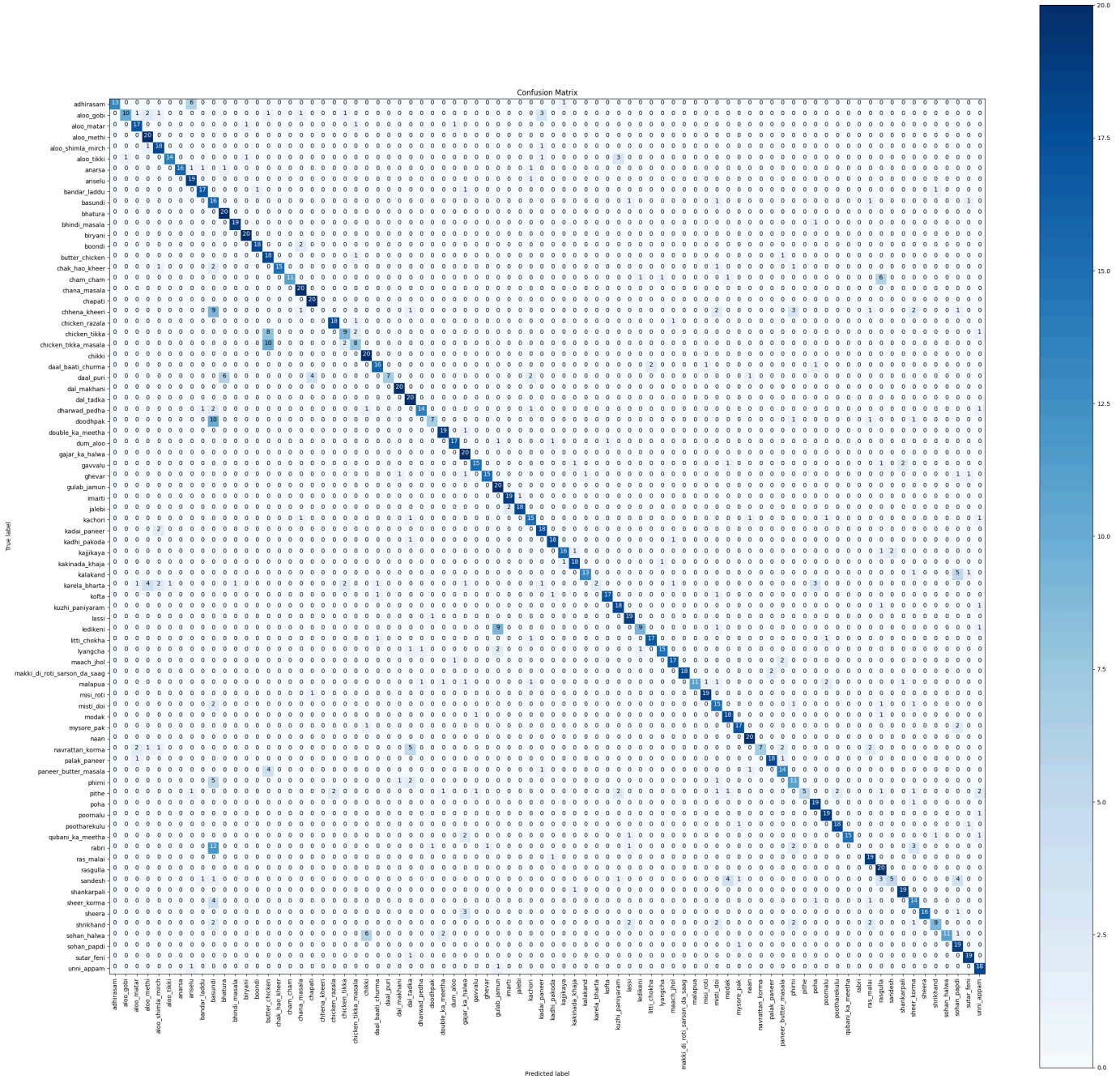
```
{'test_loss': 3.0957460403442383, 'test_model_preparation_time': 0.0042, 'test_accuracy': 0.7675, 'test_runtime'
```

Plot Confusion Matrix

```
# Extract the true labels from the model outputs
y_true = outputs.label_ids

# Predict the labels by selecting the class with the highest probability
y_pred = outputs.predictions.argmax(1)

# Define a function to plot a confusion matrix
def plot_confusion_matrix(cm, classes, title='Confusion Matrix', cmap=plt.cm.Blues, figsize=(10, 8)):
    """
    This function plots a confusion matrix.

    Parameters:
        cm (array-like): Confusion matrix as returned by sklearn.metrics.confusion_matrix.
        classes (list): List of class names, e.g., ['Class 0', 'Class 1'].
        title (str): Title for the plot.
        cmap (matplotlib colormap): Colormap for the plot.
    """
    # Create a figure with a specified size
    plt.figure(figsize=figsize)

    # Display the confusion matrix as an image with a colormap
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    # Define tick marks and labels for the classes on the axes
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=90)
    plt.yticks(tick_marks, classes)

    fmt = '.0f'
    # Add text annotations to the plot indicating the values in the cells
    thresh = cm.max() / 2.0
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white" if cm[i, j] > thresh else

    # Label the axes
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

    # Ensure the plot layout is tight
    plt.tight_layout()
    # Display the plot
    plt.show()

# Calculate accuracy and F1 score
accuracy = accuracy_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred, average='macro')

# Display accuracy and F1 score
print(f"Accuracy: {accuracy:.4f}")
print(f"F1 Score: {f1:.4f}")

# Get the confusion matrix if there are a small number of labels
if len(labels_list) <= 250:
    # Compute the confusion matrix
    cm = confusion_matrix(y_true, y_pred)
```

```
    # Plot the confusion matrix using the defined function
    plot_confusion_matrix(cm, labels_list, figsize=(28, 26))

# Finally, display classification report
print()
print("Classification report:")
print()
print(classification_report(y_true, y_pred, target_names=labels_list, digits=4))
```

Accuracy: 0.7675
F1 Score: 0.7517



Classification report:

	precision	recall	f1-score	support
adhirasam	1.0000	0.6500	0.7879	20
aloo_gobi	0.9091	0.5000	0.6452	20
aloo_matar	0.7727	0.8500	0.8095	20
aloo_methi	0.7143	1.0000	0.8333	20
aloo_shimla_mirch	0.7200	0.9000	0.8000	20
aloo_tikki	0.9333	0.7000	0.8000	20
anarsa	1.0000	0.8000	0.8889	20
ariselu	0.6786	0.9500	0.7917	20
bandar_laddu	0.8500	0.8500	0.8500	20
basundi	0.2462	0.8000	0.3765	20
bhatura	0.7407	1.0000	0.8511	20
bhindi_masala	0.9500	0.9500	0.9500	20
biryani	0.9091	1.0000	0.9524	20
boondi	0.9474	0.9000	0.9231	20
butter_chicken	0.4390	0.9000	0.5902	20

chak_hao_kheer	1.0000	0.7500	0.8571	20
cham_cham	1.0000	0.5500	0.7097	20
chana_masala	0.8000	1.0000	0.8889	20
chapati	0.8000	1.0000	0.8889	20
chhena_kheeri	0.0000	0.0000	0.0000	20
chicken_razala	0.9000	0.9000	0.9000	20
chicken_tikka	0.6429	0.4500	0.5294	20
chicken_tikka_masala	0.6154	0.4000	0.4848	20
chikki	0.7143	1.0000	0.8333	20
daal_baati_churma	0.8421	0.8000	0.8205	20
daal_puri	1.0000	0.3500	0.5185	20
dal_makhani	0.9091	1.0000	0.9524	20
dal_tadka	0.6250	1.0000	0.7692	20
dharwad_pedha	0.8750	0.7000	0.7778	20
doodhpak	0.7778	0.3500	0.4828	20
double_ka_meetha	0.8261	0.9500	0.8837	20
dum_aloo	0.8947	0.8500	0.8718	20
gajar_ka_halwa	0.6667	1.0000	0.8000	20
gavvalu	0.8824	0.7500	0.8108	20
ghevar	0.9375	0.7500	0.8333	20
gulab_jamun	0.6061	1.0000	0.7547	20
imarti	0.9048	0.9500	0.9268	20
jalebi	0.9474	0.9000	0.9231	20
kachori	0.6818	0.7500	0.7143	20
kadai_paneer	0.7200	0.9000	0.8000	20
kadhi_pakoda	0.8571	0.9000	0.8780	20
kajjikaya	0.8889	0.8000	0.8421	20
kakinada_khaja	0.8571	0.9000	0.8780	20
kalakand	0.9286	0.6500	0.7647	20
karela_bharta	1.0000	0.1000	0.1818	20
kofta	0.9444	0.8500	0.8947	20
kuzhi_paniyaram	0.7500	0.9000	0.8182	20
lassi	0.7917	0.9500	0.8636	20
ledikeni	0.8182	0.4500	0.5806	20
litti_chokha	0.8947	0.8500	0.8718	20
lyangcha	0.8824	0.7500	0.8108	20
maach_jhol	0.8500	0.8500	0.8500	20
makki_di_roti_sarson_da_saag	1.0000	0.9000	0.9474	20
malapua	1.0000	0.5500	0.7097	20
misi_roti	0.9048	0.9500	0.9268	20
misti_doi	0.5769	0.7500	0.6522	20
modak	0.7200	0.9000	0.8000	20
mysore_pak	0.8500	0.8500	0.8500	20
naan	0.8696	1.0000	0.9302	20
navrattan_korma	1.0000	0.3500	0.5185	20
palak_paneer	0.9000	0.9000	0.9000	20
paneer_butter_masala	0.7000	0.7000	0.7000	20
phirni	0.5238	0.5500	0.5366	20
pithe	1.0000	0.2500	0.4000	20
poha	0.7600	0.9500	0.8444	20
poornalu	0.8261	0.9500	0.8837	20
pootharekulu	0.9000	0.9000	0.9000	20
qubani_ka_meetha	1.0000	0.7500	0.8571	20
rabri	0.0000	0.0000	0.0000	20
ras_malai	0.7037	0.9500	0.8085	20
rasgulla	0.5714	1.0000	0.7273	20
sandesh	0.7143	0.2500	0.3704	20
shankarpali	0.8636	0.9500	0.9048	20
sheer_korma	0.5600	0.7000	0.6222	20
sheera	1.0000	0.8000	0.8889	20
shrikhand	0.8182	0.4500	0.5806	20
sohan_halwa	1.0000	0.5500	0.7097	20
sohan_papdi	0.5588	0.9500	0.7037	20
sutar_feni	0.8261	0.9500	0.8837	20
unni_appam	0.6667	0.9000	0.7660	20
accuracy			0.7675	1600
macro avg	0.7957	0.7675	0.7517	1600
weighted avg	0.7957	0.7675	0.7517	1600

Save the Trained Model

```
trainer.save_model()
```

Setup Image Classification Pipeline

```
# Import the 'pipeline' function from the 'transformers' library.
from transformers import pipeline

# Create a pipeline for image classification tasks.
# You need to specify the 'model_name' and the 'device' to use for inference.
# - 'model_name': The name of the pre-trained model to be used for image classification.
# - 'device': Specifies the device to use for running the model (0 for GPU, -1 for CPU).
pipe = pipeline('image-classification', model=model_name, device=0)
```

```
Using a slow image processor as `use_fast` is unset and a slow processor was saved with this model. `use_fast=Tr
Device set to use cuda:0
```

Load and Display Test Image

```
# Accessing an image from the 'test_data' dataset using index 1.
image = test_data[2]["image"]

# Displaying the 'image' variable.
image
```

Perform Inference on Test Image

```
# Apply the 'pipe' function to process the 'image' variable.
pipe(image)
```

```
[{'label': 'sheera', 'score': 0.03791926056146622},
 {'label': 'bhatura', 'score': 0.02033482864499092},
 {'label': 'cham_cham', 'score': 0.0195829588919878},
 {'label': 'boondi', 'score': 0.019102953374385834},
 {'label': 'rasgulla', 'score': 0.019010771065950394}]
```

Compare Prediction with Ground Truth

```
# This line of code accesses the "label" attribute of a specific element in the test_data list.
# It's used to retrieve the actual label associated with a test data point.
id2label[test_data[2]["label"]]
```

```
'sheera'
```

Conclusion

This experiment shows how cognitive applications can **learn from images and assist in decision-making**. By using advanced models like Vision Transformers, we can make systems that are more intelligent, adaptive, and useful across many industries. The integration of image recognition with cognitive learning is a major step toward building **real-world AI solutions** that reduce human effort and improve accuracy.

```
Start coding or generate with AI.
```