PROJECT TITLE: FACIAL EXPRESSIONS RECOGNITION

TEAM: 7

CHIBUIKE OKOROAMA - C0892150

JUMOKE KADIJAT YEKEEN - C0900481

MODUPEOLA OYATOKUN - C0895705

Lecturer: Ishant Gupta

Table to content

Project Objective

The objective is to predict facial expression recognition of some image dataset.

Dataset Selection

About Dataset

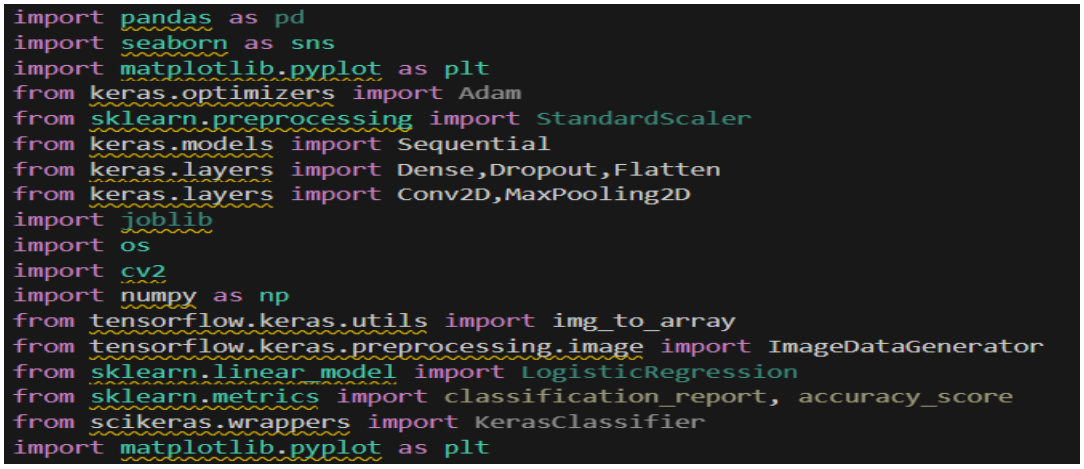
The dataset used for this project was about 63MB obtained from Kaggle and consists of 48x48 pixel grayscale images of faces. The faces have been automatically registered so that the face is centred and occupies about the same amount of space in each image.

The task is to categorize each face based on the emotion shown in the facial expression into one of seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). The training set consists of 28,709 examples and the public test set consists of 3,589 examples.

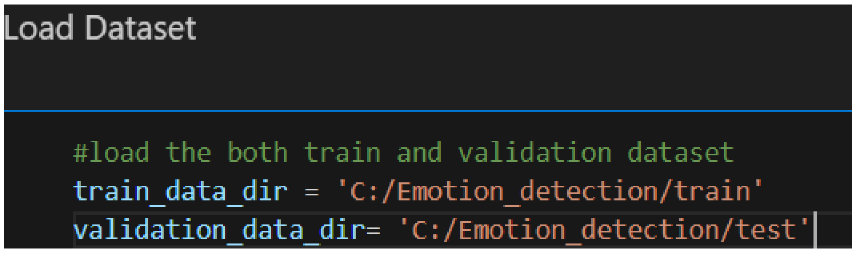
The first step is to create a new directory for the facial expression recognition in VS Code.

Created and activated a virtual environment

Import Libraries



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense,Dropout,Flatten
from keras.layers import Conv2D,MaxPooling2D
import joblib
import os
import cv2
import numpy as np
from tensorflow.keras.utils import img_to_array
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score
from scikeras.wrappers import KerasClassifier
import matplotlib.pyplot as plt
```



Load Dataset

```
#load the both train and validation dataset
train_data_dir = 'C:/Emotion_detection/train'
validation_data_dir= 'C:/Emotion_detection/test'
```

Data Pre-Processing

The data preprocessing stage involves preparing the image data for training and validation in a deep learning model for emotion detection. The preprocessing pipeline is carefully designed to enhance the model's performance by augmenting the training data and ensuring consistency in the validation data. The main steps include data augmentation, rescaling, and batch generation.

1. Directory Setup

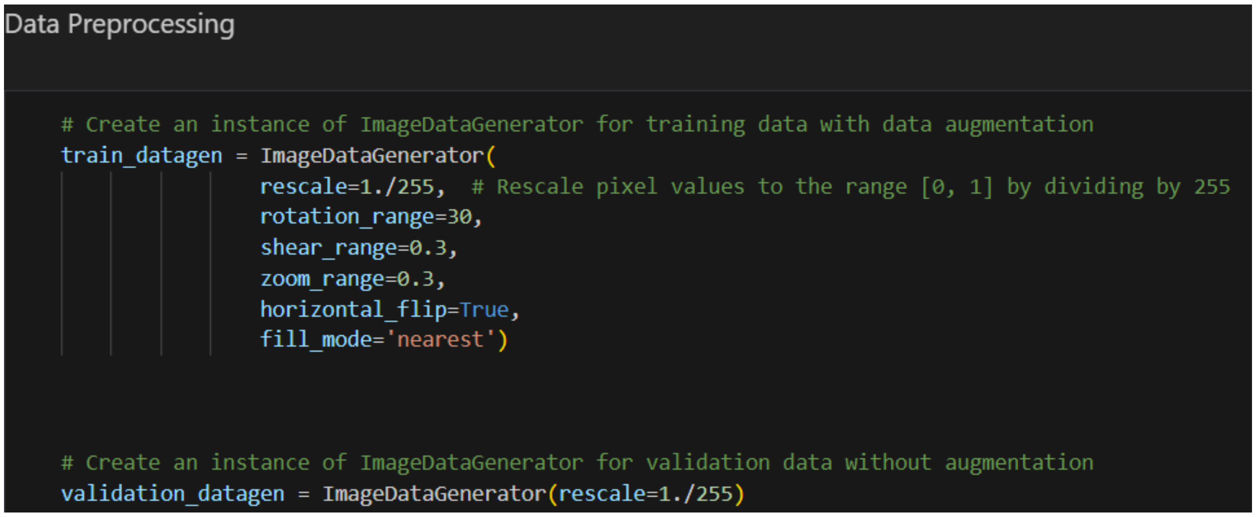
The image data is organized into two directories:

Training Data Directory: This directory contains images used to train the model.

Validation Data Directory: This directory contains images used to validate the model's performance.

2. Data Augmentation for Training Data

To improve the generalization of the model, various data augmentation techniques are applied to the training data:



Data Preprocessing

```python
# Create an instance of ImageDataGenerator for training data with data augmentation
train_datagen = ImageDataGenerator(
                rescale=1./255,  # Rescale pixel values to the range [0, 1] by dividing by 255
                rotation_range=30,
                shear_range=0.3,
                zoom_range=0.3,
                horizontal_flip=True,
                fill_mode='nearest')



# Create an instance of ImageDataGenerator for validation data without augmentation
validation_datagen = ImageDataGenerator(rescale=1./255)
```

Rescaling: All pixel values are rescaled to the range [0, 1] by dividing by 255. This normalization step ensures that the model trains faster and with greater accuracy.

Rotation: Images are randomly rotated by up to 30 degrees, helping the model become invariant to slight rotations in the input images.

Shear Transformation: A shear range of 0.3 is applied, which skews the image along one axis, augmenting the dataset with different perspectives.

Zoom: A random zoom of up to 30% is applied, allowing the model to learn from images at different scales.

Horizontal Flip: Images are randomly flipped horizontally, which helps the model recognize features that are mirrored.

Fill Mode: The nearest pixel value is used to fill in new pixels that are created as a result of these transformations.

These augmentations generate a more diverse set of images during training, which can help the model learn more robust features and reduce overfitting.

3. Validation Data Preprocessing

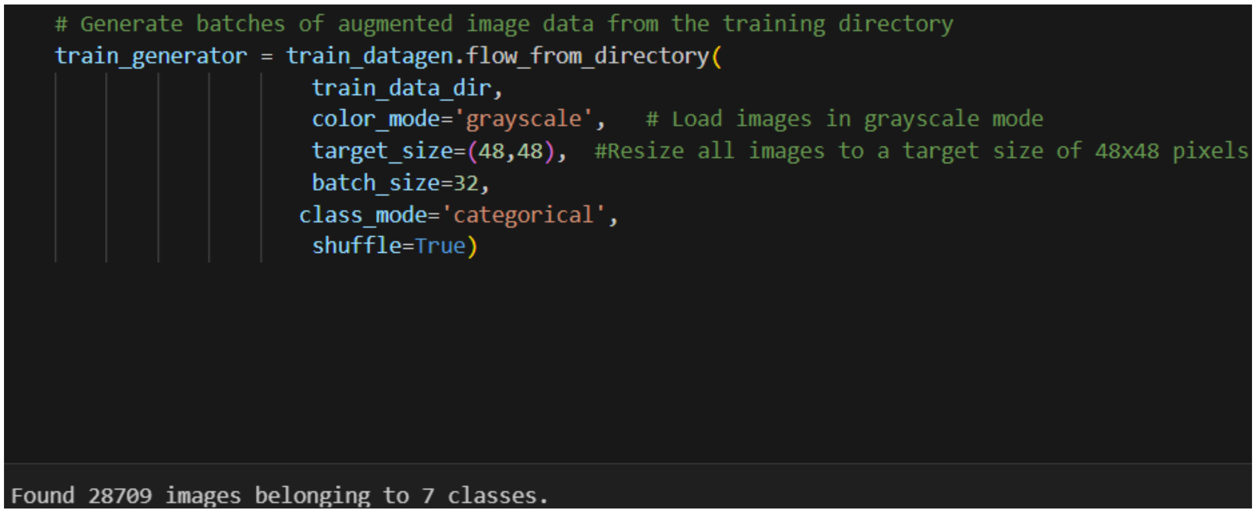
The validation data undergoes a simpler preprocessing pipeline:

Rescaling: Similar to the training data, the validation images are rescaled to the [0, 1] range.

No Augmentation: Unlike the training data, the validation data is not augmented. This ensures that the validation performance is measured on data that closely resembles the real-world scenario, without artificial transformations.

4. Image Data Generation

The processed images are then loaded in batches for both training and validation:



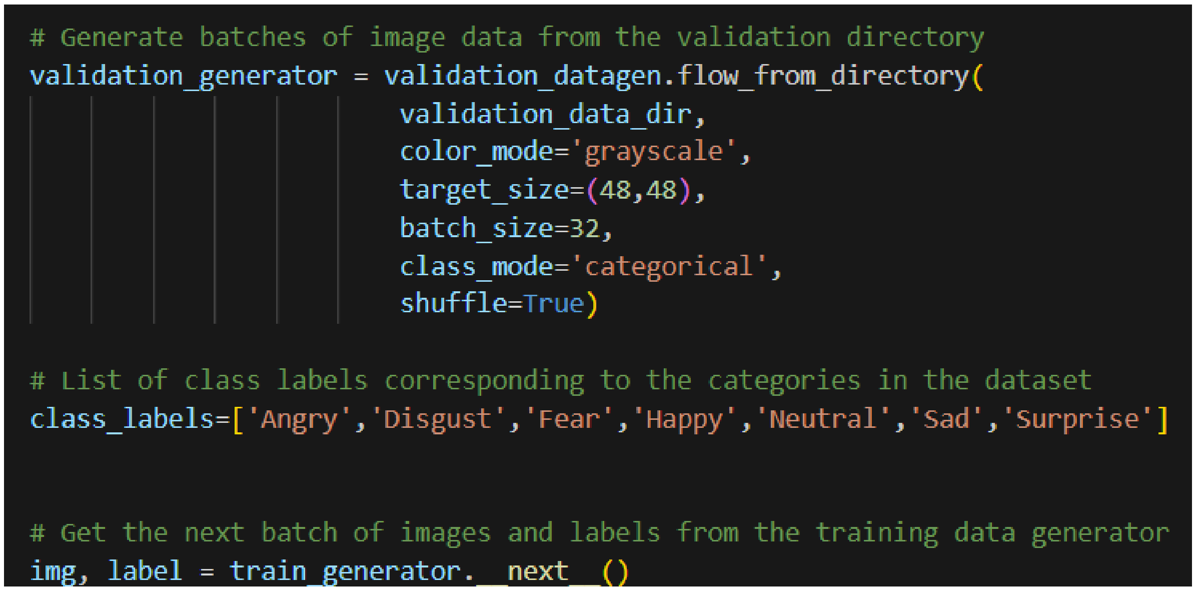
```
# Generate batches of augmented image data from the training directory
train_generator = train_datagen.flow_from_directory(
                    train_data_dir,
                    color_mode='grayscale',   # Load images in grayscale mode
                    target_size=(48,48),  #Resize all images to a target size of 48x48 pixels
                    batch_size=32,
                   class_mode='categorical',
                    shuffle=True)
```

```
Found 28709 images belonging to 7 classes.
```

Grayscale Conversion: All images are loaded in grayscale mode to reduce the computational complexity and focus the model on key features that are independent of color.

Target Size: Each image is resized to 48x48 pixels, a common size for facial emotion recognition tasks, which balances computational efficiency with the need to retain sufficient detail.

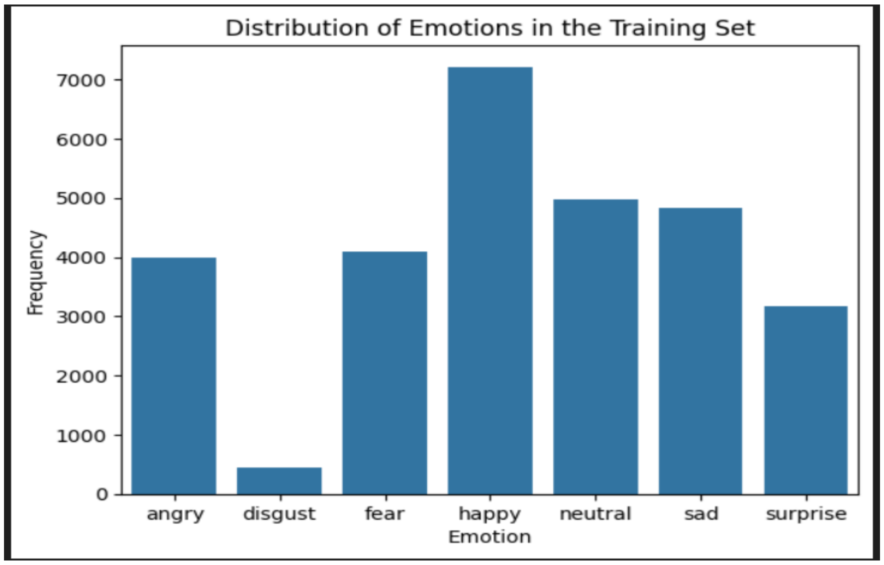
Batch Size: The images are grouped into batches of 64, which is a commonly used size that balances memory usage and computational efficiency.


```
# Generate batches of image data from the validation directory
validation_generator = validation_datagen.flow_from_directory(
                        validation_data_dir,
                        color_mode='grayscale',
                        target_size=(48,48),
                        batch_size=32,
                        class_mode='categorical',
                        shuffle=True)

# List of class labels corresponding to the categories in the dataset
class_labels=['Angry','Disgust','Fear','Happy','Neutral','Sad','Surprise']


# Get the next batch of images and labels from the training data generator
img, label = train_generator.__next__()
```

Class Mode: The class labels are returned in a categorical format, suitable for multi-class classification tasks.

Shuffling: The training images are shuffled before each epoch to ensure that the model does not learn the order of the images, which helps in better generalization.

Data Exploration



This image shows a bar chart titled "Distribution of Emotions in the Training Set". It visualizes the frequency of different emotions in a dataset, likely used for training an emotion recognition model or for sentiment analysis.

The chart has seven bars, each representing a distinct emotion: Angry,Disgust,Fear,Happy,Neutral,Sad,Surprise

The y-axis represents the frequency, ranging from 0 to about 7000. The x-axis lists the emotion categories.

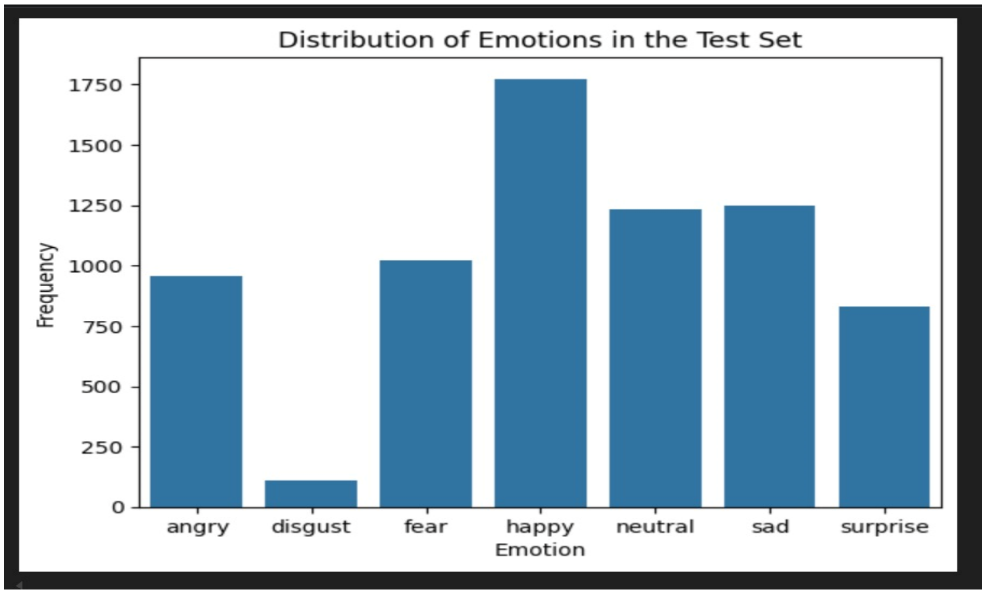
Key observations:

"Happy" is the most frequent emotion, with over 7000 instances.

"Neutral" and "Sad" are the next most common, each with around 5000 instances.

"Angry", "Fear", and "Surprise" have moderate frequencies, between 3000-4000 instances each.

"Disgust" is the least represented emotion, with fewer than 1000 instances.



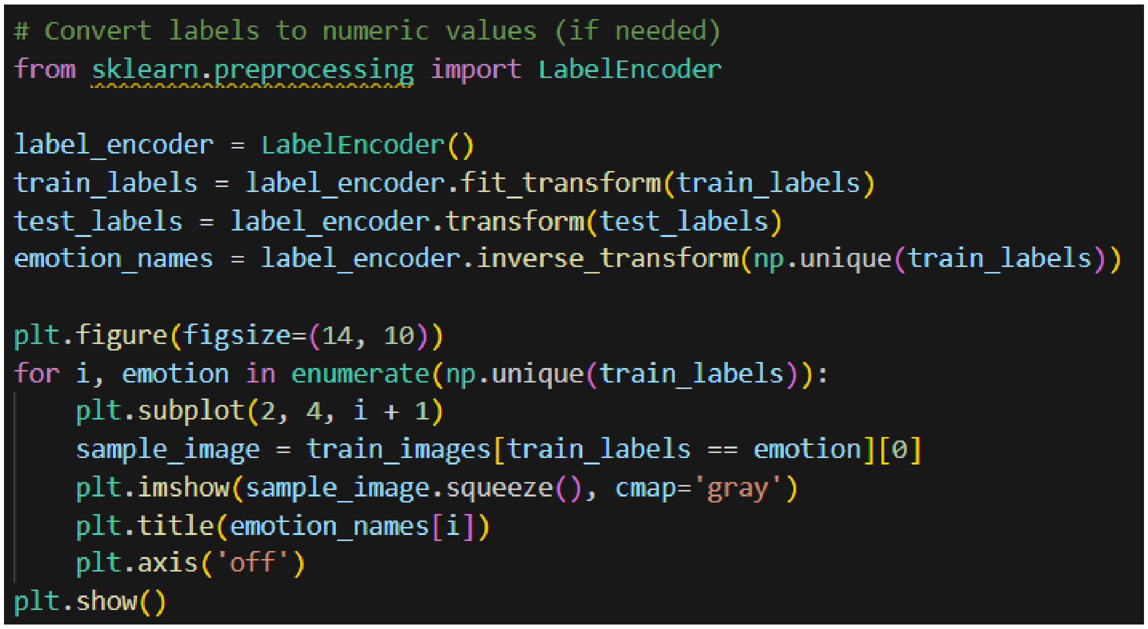
This image shows a bar chart titled "Distribution of Emotions in the Test Set". It illustrates the frequency of different emotions in a dataset used for testing an emotion recognition model.

Key points about the visualization:

1. Emotions represented: The chart includes seven emotion categories: angry, disgust, fear, happy, neutral, sad, and surprise.
2. Frequency distribution:
    - "Happy" is the most frequent emotion, with about 1750 instances.
    - "Neutral" and "Sad" are the next most common, each with around 1250 instances.
    - "Fear" and "Angry" have moderate frequencies, with about 1000 and 950 instances respectively.
    - "Surprise" has around 800 instances.
    - "Disgust" is the least represented, with only about 100 instances.

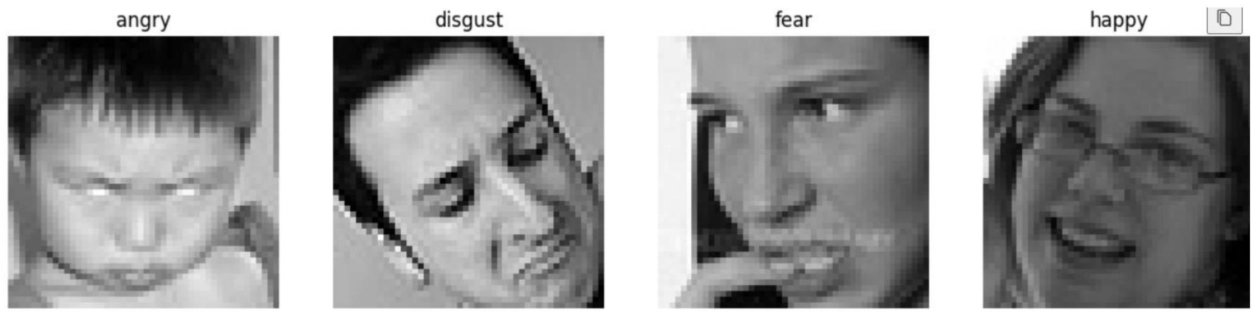
Feature Engineering

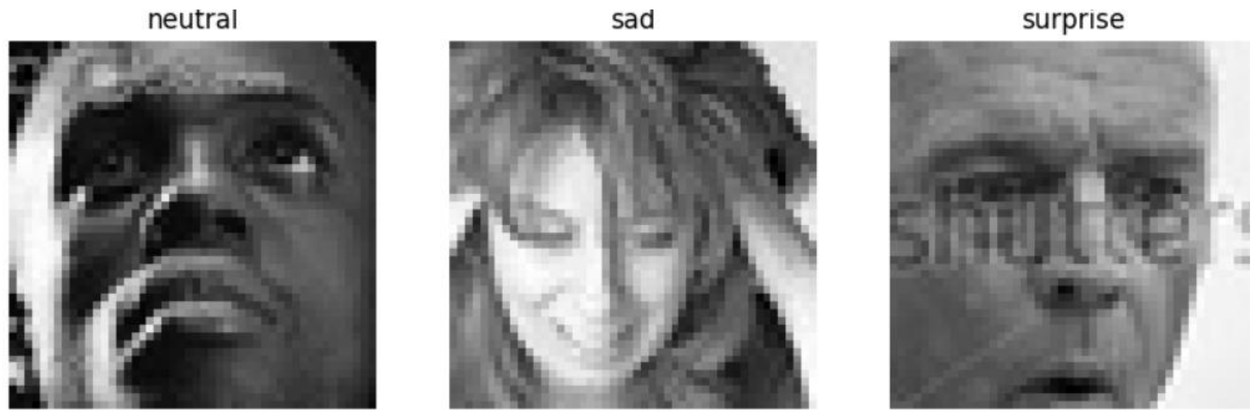
This converts categorical emotion labels to numeric values for machine learning purposes. This retrieves the original emotion names corresponding to the encoded labels.



```python
# Convert labels to numeric values (if needed)
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
test_labels = label_encoder.transform(test_labels)
emotion_names = label_encoder.inverse_transform(np.unique(train_labels))

plt.figure(figsize=(14, 10))
for i, emotion in enumerate(np.unique(train_labels)):
    plt.subplot(2, 4, i + 1)
    sample_image = train_images[train_labels == emotion][0]
    plt.imshow(sample_image.squeeze(), cmap='gray')
    plt.title(emotion_names[i])
    plt.axis('off')
plt.show()
```

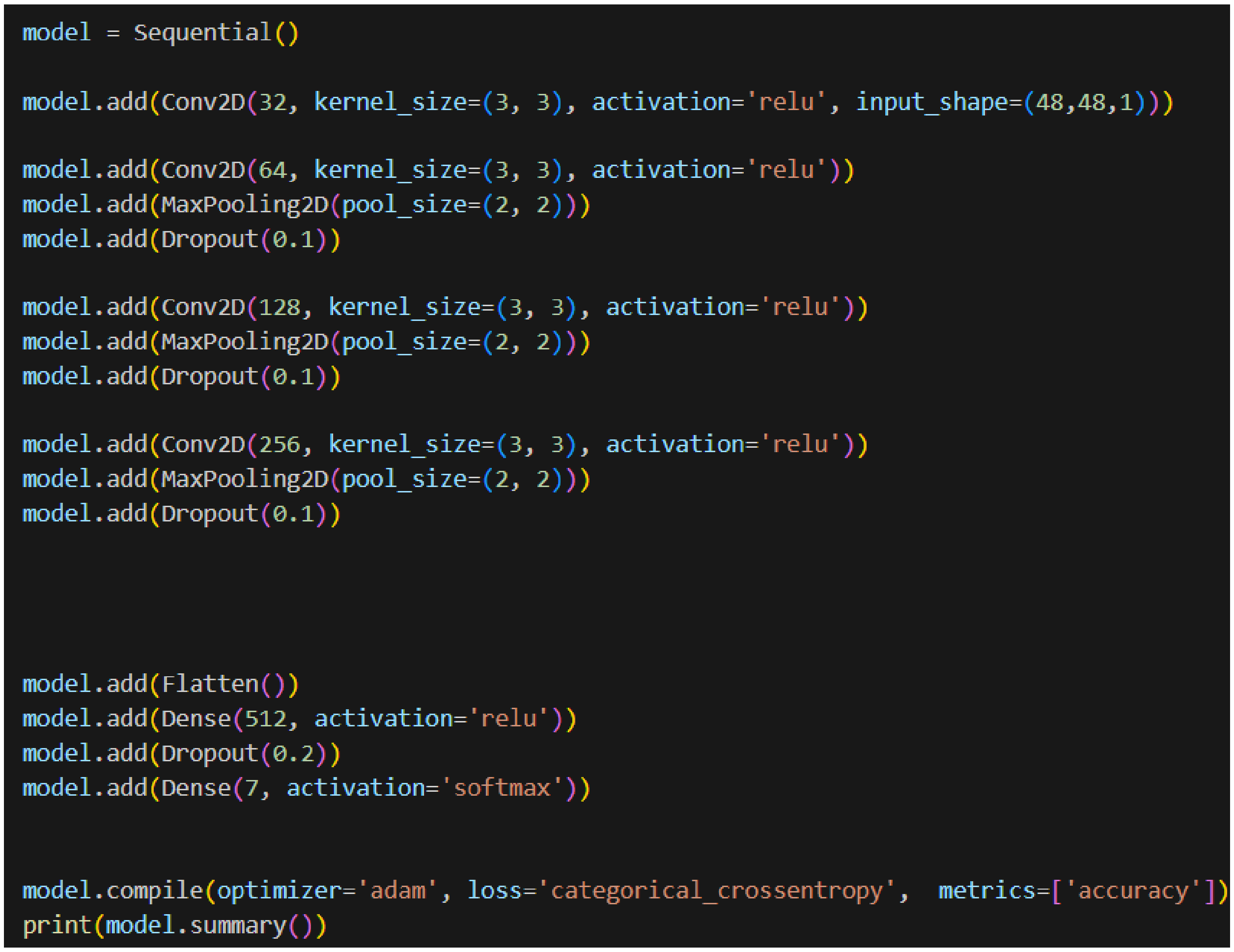
This code will create a visual grid of sample images, one for each emotion category in the dataset, allowing for a quick visual inspection of the types of images used in the emotion recognition task.


angry
disgust
fear
happy


neutral
sad
surprise

Model Building

Convolutional Neural Network (CNN)



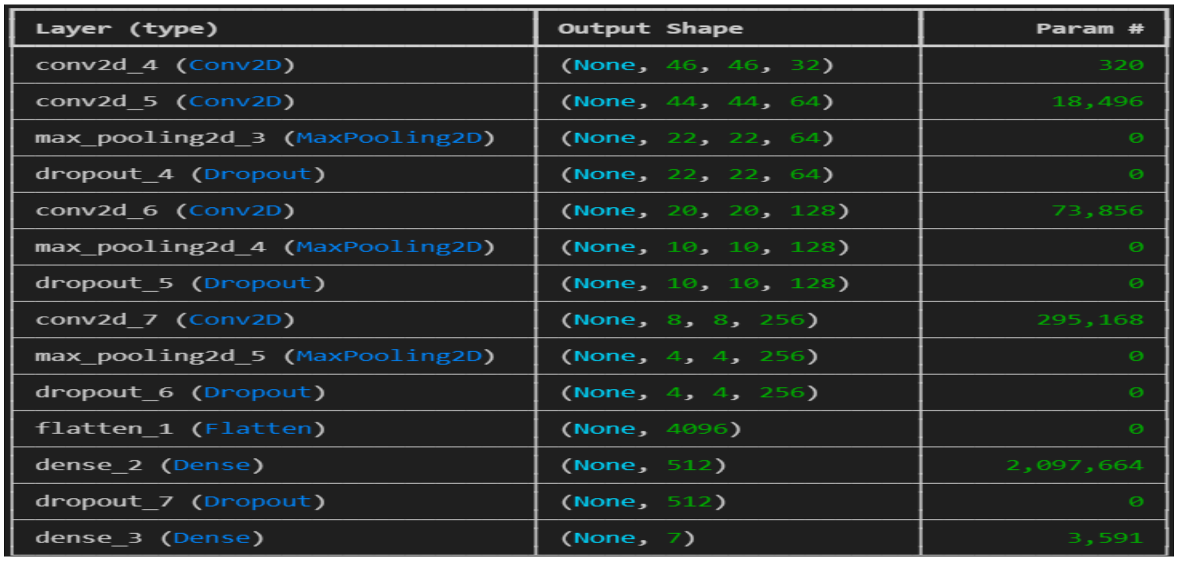
```
model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(48,48,1)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.1))

model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.1))

model.add(Conv2D(256, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.1))




model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(7, activation='softmax'))


model.compile(optimizer='adam', loss='categorical_crossentropy',  metrics=['accuracy'])
print(model.summary())
```

This code defines a Convolutional Neural Network (CNN) model using Keras for image classification, likely for the emotion recognition task we've been discussing. Let's break it down:

1. Model Structure:
   - It's a Sequential model with multiple layers.
   - Input shape is 48x48x1, suggesting grayscale images of size 48x48 pixels.
2. Convolutional Layers:
   - Four Conv2D layers with increasing filter sizes (32, 64, 128, 256).S
   - All use 3x3 kernel size and ReLU activation.

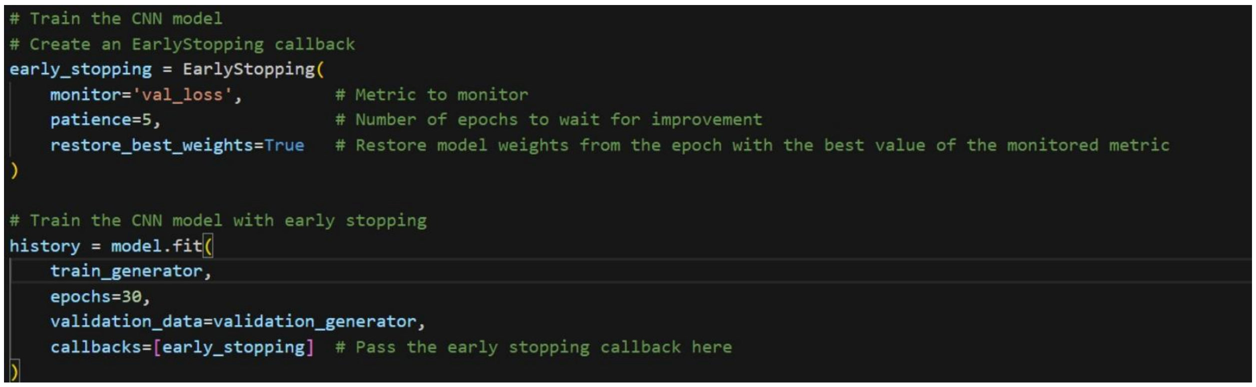
3. Pooling and Regularization:
   - MaxPooling2D layers with 2x2 pool size after each conv layer.
   - Dropout layers (0.1 rate) for regularization.
4. Flatten and Dense Layers:
   - Flatten layer to transition from convolutional to dense layers.
   - One dense layer with 512 units and ReLU activation.
   - Final dense layer with 7 units (likely corresponding to 7 emotions) and softmax activation for classification.
5. Compilation:

- Uses Adam optimizer.
- Categorical crossentropy loss function (suitable for multi-class classification).
- Accuracy as the evaluation metric.



```
Layer (type)                          Output Shape                    Param #
conv2d_4 (Conv2D)                     (None, 46, 46, 32)                  320
conv2d_5 (Conv2D)                     (None, 44, 44, 64)               18,496
max_pooling2d_3 (MaxPooling2D)        (None, 22, 22, 64)                    0
dropout_4 (Dropout)                   (None, 22, 22, 64)                    0
conv2d_6 (Conv2D)                     (None, 20, 20, 128)              73,856
max_pooling2d_4 (MaxPooling2D)        (None, 10, 10, 128)                   0
dropout_5 (Dropout)                   (None, 10, 10, 128)                   0
conv2d_7 (Conv2D)                     (None, 8, 8, 256)               295,168
max_pooling2d_5 (MaxPooling2D)        (None, 4, 4, 256)                     0
dropout_6 (Dropout)                   (None, 4, 4, 256)                     0
flatten_1 (Flatten)                   (None, 4096)                          0
dense_2 (Dense)                       (None, 512)                   2,097,664
dropout_7 (Dropout)                   (None, 512)                           0
dense_3 (Dense)                       (None, 7)                         3,591
```

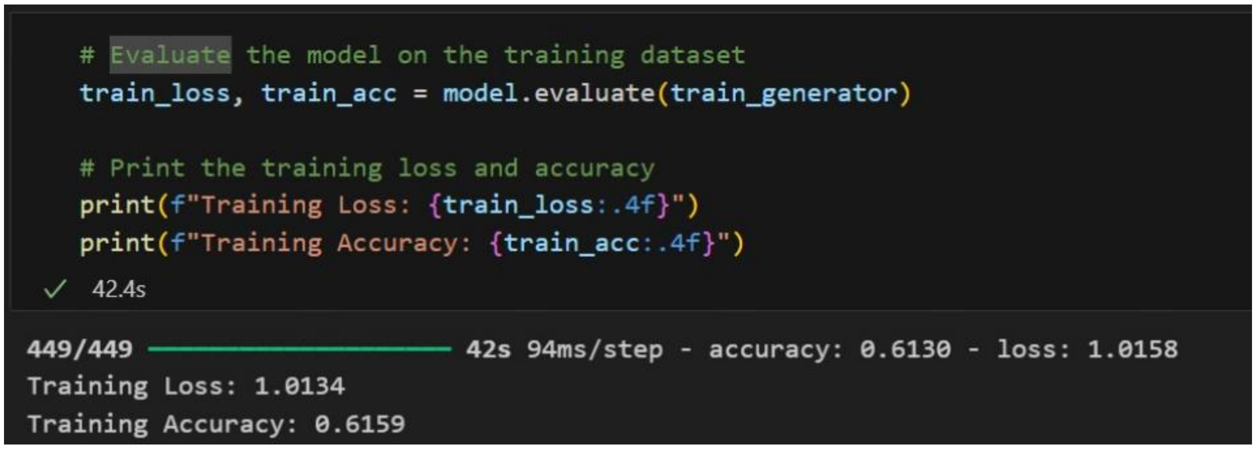
Train the Data



```
# Train the CNN model
# Create an EarlyStopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',          # Metric to monitor
    patience=5,                  # Number of epochs to wait for improvement
    restore_best_weights=True    # Restore model weights from the epoch with the best value of the monitored metric
)

# Train the CNN model with early stopping
history = model.fit(
    train_generator,
    epochs=30,
    validation_data=validation_generator,
    callbacks=[early_stopping]  # Pass the early stopping callback here
)
```

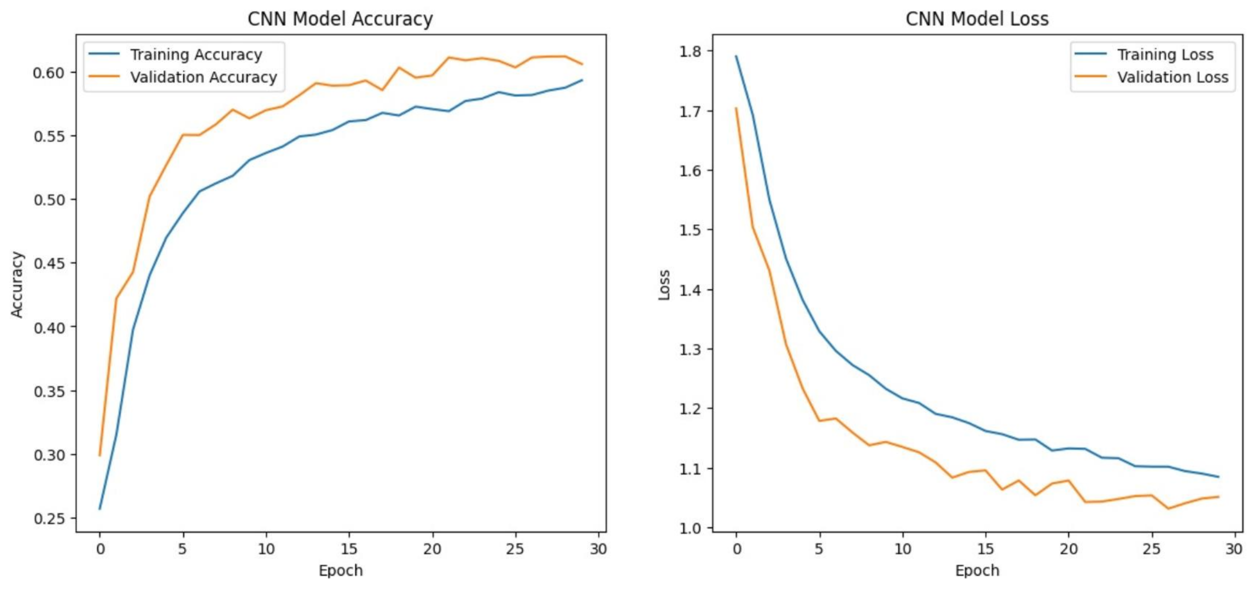
Trains a CNN model with early stopping to prevent overfitting.

1. EarlyStopping callback: The EarlyStopping callback monitors the validation loss (val_loss) during training. If the loss does not improve after 5 epochs (patience=5), training will stop early. It also restores the model to the state where validation loss was the best (restore_best_weights=True).
2. Model training: The CNN model is trained for up to 30 epochs using the training data, while evaluating on the validation data. The early stopping callback is passed to halt training early if necessary, based on the validation loss.



```
# Evaluate the model on the training dataset
train_loss, train_acc = model.evaluate(train_generator)

# Print the training loss and accuracy
print(f"Training Loss: {train_loss:.4f}")
print(f"Training Accuracy: {train_acc:.4f}")
✓ 42.4s

449/449 ━━━━━━━━━━━━━━━━━━━━ 42s 94ms/step - accuracy: 0.6130 - loss: 1.0158
Training Loss: 1.0134
Training Accuracy: 0.6159
```

The Training accuracy was 61.59%



This image presents two graphs illustrating the performance of a Convolutional Neural Network (CNN) model during training:

Left Graph (Accuracy):

- **Title:** CNN Model Accuracy
- **X-axis:** Epoch (Represents the number of times the entire dataset is passed through the model)
- **Y-axis:** Accuracy (Measures how well the model correctly classifies data)
- **Lines:**
  - **Training Accuracy:** Shows the model's accuracy on the training data. It generally increases over epochs as the model learns.
  - **Validation Accuracy:** Shows the model's accuracy on a separate validation dataset. It helps assess how well the model generalizes to unseen data.

Right Graph (Loss):

- **Title:** CNN Model Loss
- **X-axis:** Epoch (Same as the left graph)
- **Y-axis:** Loss (Represents the error between the model's predicted values and the actual values)

- **Lines:**
    - **Training Loss:** Shows the model's loss on the training data. It typically decreases over epochs as the model improves.
    - **Validation Loss:** Shows the model's loss on the validation dataset. It's used to monitor for overfitting (when the model performs well on training data but poorly on new data).

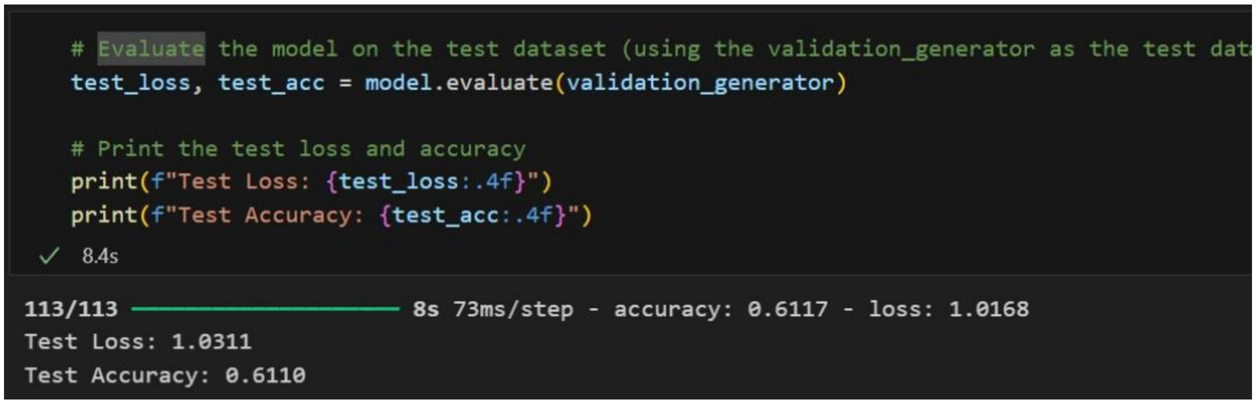
Key Observations:

- **Accuracy:** The training accuracy steadily increases, while the validation accuracy also improves but with some fluctuations.
- **Loss:** The training loss decreases, indicating the model is learning. The validation loss also decreases initially but then starts to increase slightly, suggesting potential overfitting.

Interpretation:

The CNN model is learning from the training data and improving its accuracy over time. However, there are signs of overfitting as the validation loss starts to increase. This means the model might be memorizing the training data too well and not generalizing well to new data.

Model Evaluation



```
# Evaluate the model on the test dataset (using the validation_generator as the test dat
test_loss, test_acc = model.evaluate(validation_generator)

# Print the test loss and accuracy
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
```

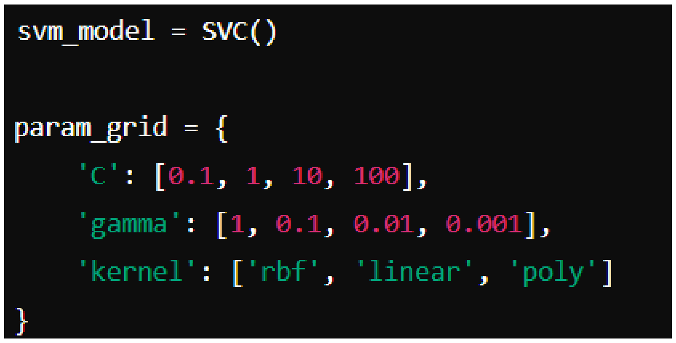
✓ 8.4s

```
113/113 ━━━━━━━━━━━━━━━━━━━━ 8s 73ms/step - accuracy: 0.6117 - loss: 1.0168
Test Loss: 1.0311
Test Accuracy: 0.6110
```

The Test Accuracy is 61.10%

SVM (Support Vector Machine)

using an SVM (Support Vector Machine) for classification on a dataset of images, presumably related to emotions based on the class labels (e.g., "Angry", "Happy", etc.)

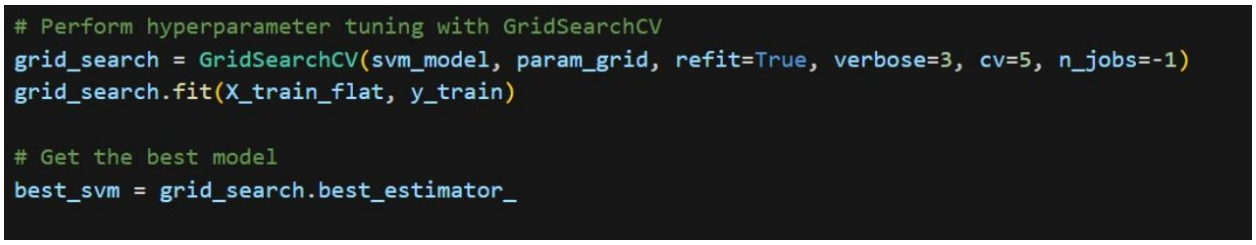


```
svm_model = SVC()

param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [1, 0.1, 0.01, 0.001],
    'kernel': ['rbf', 'linear', 'poly']
}
```

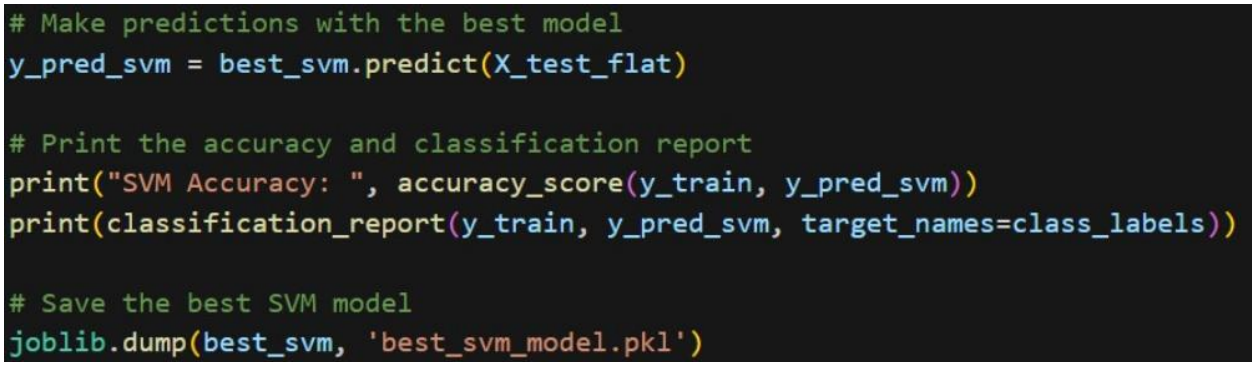
The param_grid dictionary defines the hyperparameters for the SVM model that will be tuned.

- 'C' controls the regularization strength.
- 'gamma' controls the kernel coefficient for the ‘rbf’ kernel.
- 'kernel' specifies the type of kernel to use: 'rbf' (Radial Basis Function), 'linear', or 'poly' (polynomial).

Hyperparameter Tuning with GridSearchCV



```
# Perform hyperparameter tuning with GridSearchCV
grid_search = GridSearchCV(svm_model, param_grid, refit=True, verbose=3, cv=5, n_jobs=-1)
grid_search.fit(X_train_flat, y_train)

# Get the best model
best_svm = grid_search.best_estimator_
```

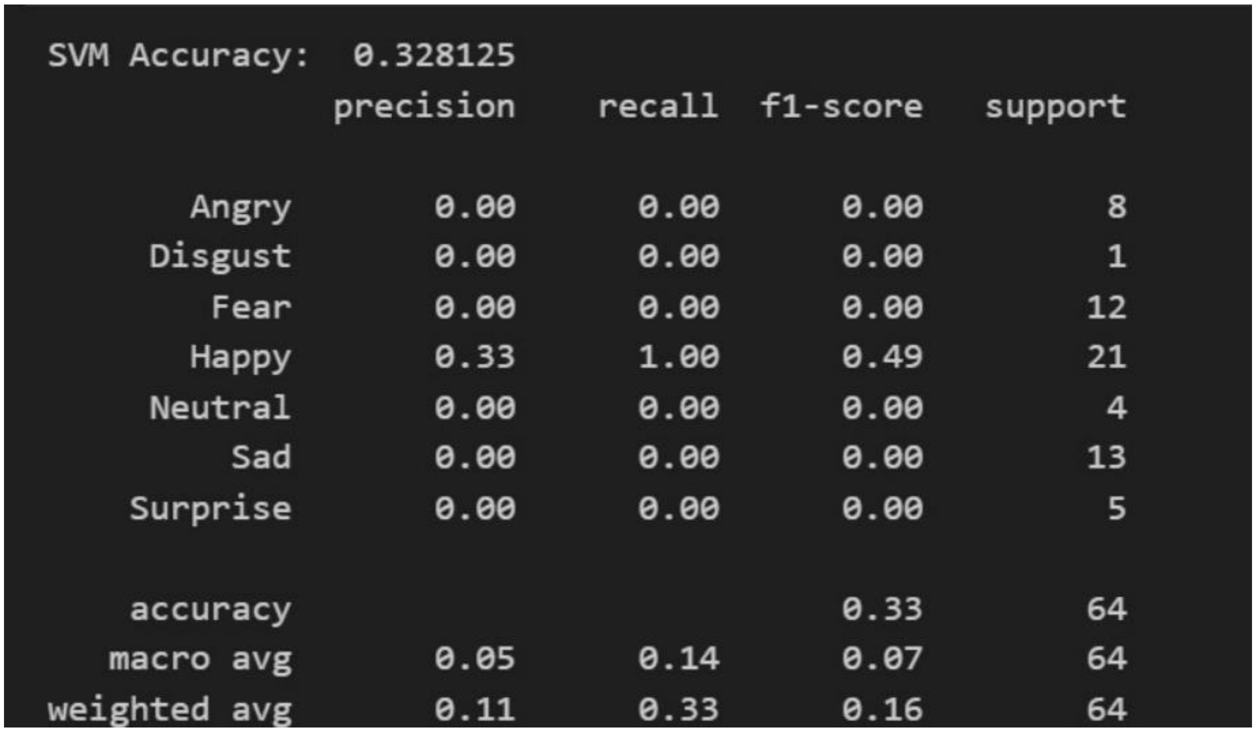


```
# Make predictions with the best model
y_pred_svm = best_svm.predict(X_test_flat)

# Print the accuracy and classification report
print("SVM Accuracy: ", accuracy_score(y_train, y_pred_svm))
print(classification_report(y_train, y_pred_svm, target_names=class_labels))

# Save the best SVM model
joblib.dump(best_svm, 'best_svm_model.pkl')
```

best_svm: This likely refers to the SVM model that achieved the best performance during training or hyperparameter tuning.

X_test_flat: This is the feature matrix of the test data, flattened into a one-dimensional array.

y_pred_svm: This variable will store the predicted class labels for the test data using the best_svm model.



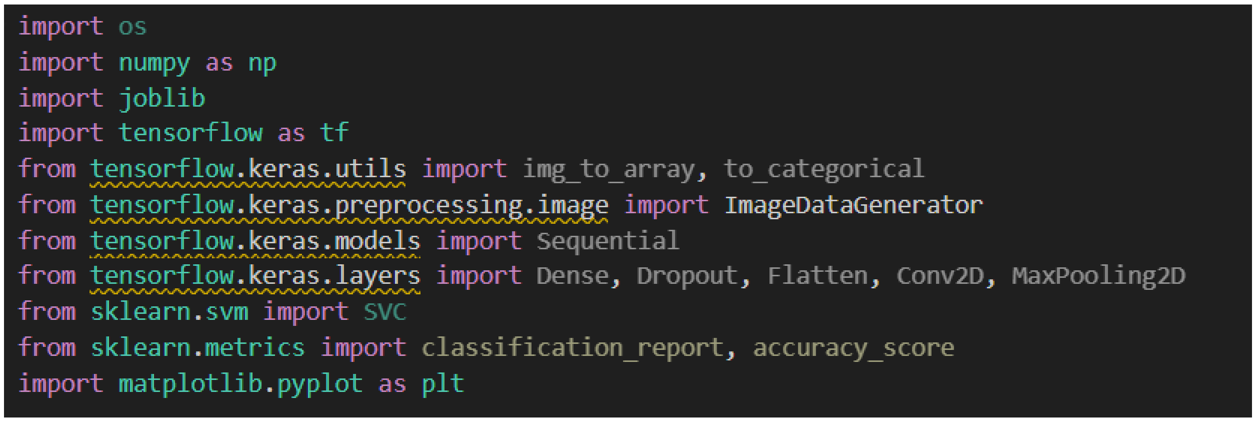
```
SVM Accuracy:  0.328125
              precision    recall  f1-score   support

       Angry       0.00      0.00      0.00         8
     Disgust       0.00      0.00      0.00         1
        Fear       0.00      0.00      0.00        12
       Happy       0.33      1.00      0.49        21
     Neutral       0.00      0.00      0.00         4
         Sad       0.00      0.00      0.00        13
    Surprise       0.00      0.00      0.00         5

    accuracy                           0.33        64
   macro avg       0.05      0.14      0.07        64
weighted avg       0.11      0.33      0.16        64
```

The accuracy for SVM is 32.81%

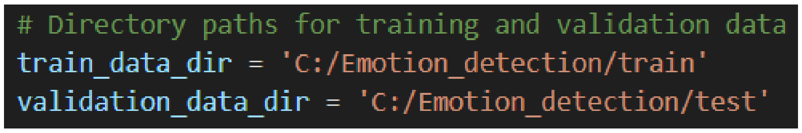
Predict

This is where our model is tested on test

Import Libaries



```python
import os
import numpy as np
import joblib
import tensorflow as tf
from tensorflow.keras.utils import img_to_array, to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt
```

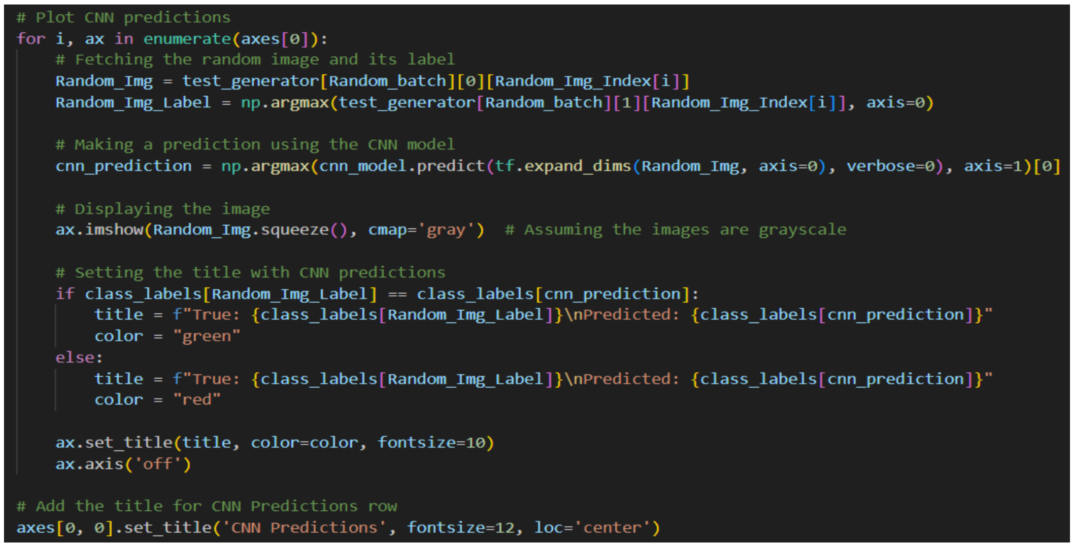


```python
# Directory paths for training and validation data
train_data_dir = 'C:/Emotion_detection/train'
validation_data_dir = 'C:/Emotion_detection/test'
```

The variables train_data_dir and validation_data_dir are storing the file paths to directories that contain the training and validation datasets for an emotion detection model.

- train_data_dir: This variable holds the path 'C:/Emotion_detection/train', which indicates that the training data (used to train the model) is stored in the train folder located inside the C:/Emotion_detection/ directory.
- validation_data_dir: This variable holds the path 'C:/Emotion_detection/test', which indicates that the validation data (used to evaluate the model's performance during training) is stored in the test folder located inside the C:/Emotion_detection/ directory.

Making predictions using a trained CNN model



```python
# Plot CNN predictions
for i, ax in enumerate(axes[0]):
    # Fetching the random image and its label
    Random_Img = test_generator[Random_batch][0][Random_Img_Index[i]]
    Random_Img_Label = np.argmax(test_generator[Random_batch][1][Random_Img_Index[i]], axis=0)

    # Making a prediction using the CNN model
    cnn_prediction = np.argmax(cnn_model.predict(tf.expand_dims(Random_Img, axis=0), verbose=0), axis=1)[0]

    # Displaying the image
    ax.imshow(Random_Img.squeeze(), cmap='gray')  # Assuming the images are grayscale

    # Setting the title with CNN predictions
    if class_labels[Random_Img_Label] == class_labels[cnn_prediction]:
        title = f"True: {class_labels[Random_Img_Label]}\nPredicted: {class_labels[cnn_prediction]}"
        color = "green"
    else:
        title = f"True: {class_labels[Random_Img_Label]}\nPredicted: {class_labels[cnn_prediction]}"
        color = "red"

    ax.set_title(title, color=color, fontsize=10)
    ax.axis('off')

# Add the title for CNN Predictions row
axes[0, 0].set_title('CNN Predictions', fontsize=12, loc='center')
```

Loop through images: The code iterates through a batch of random images and selects one image at a time for display.

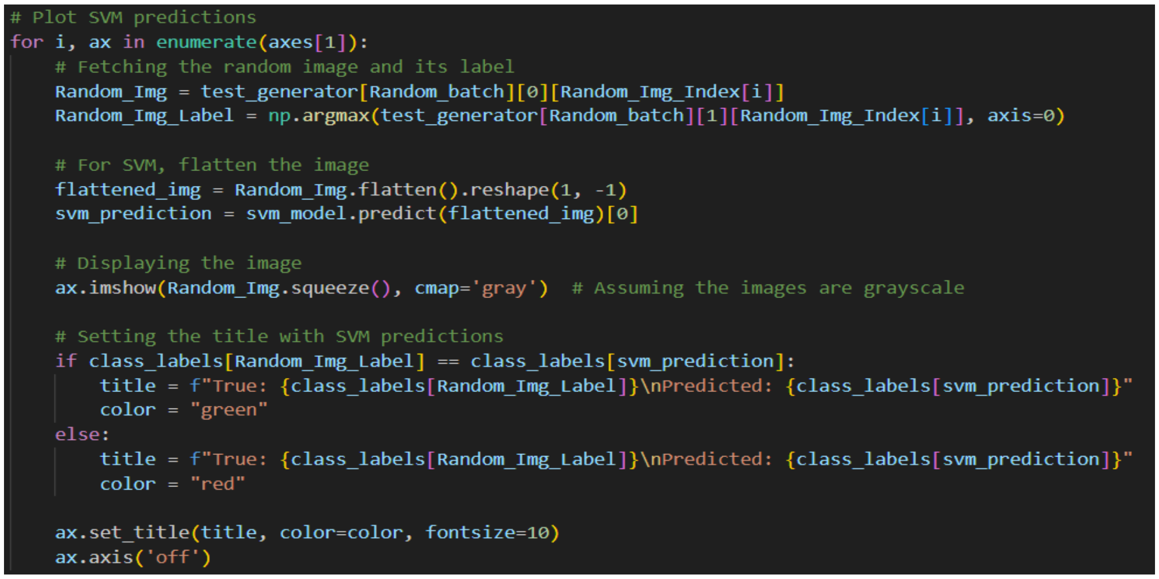
Fetch random image and label: The code retrieves an image and its corresponding true label from the test data.

Make a prediction: The CNN model predicts the class of the selected image.

Plot the image: The selected image is displayed, and its true label and predicted label are compared.

Set title and color: If the prediction is correct, the title is displayed in green; otherwise, it is displayed in red.

Making predictions using a trained SVM model



```python
# Plot SVM predictions
for i, ax in enumerate(axes[1]):
    # Fetching the random image and its label
    Random_Img = test_generator[Random_batch][0][Random_Img_Index[i]]
    Random_Img_Label = np.argmax(test_generator[Random_batch][1][Random_Img_Index[i]], axis=0)

    # For SVM, flatten the image
    flattened_img = Random_Img.flatten().reshape(1, -1)
    svm_prediction = svm_model.predict(flattened_img)[0]

    # Displaying the image
    ax.imshow(Random_Img.squeeze(), cmap='gray')  # Assuming the images are grayscale

    # Setting the title with SVM predictions
    if class_labels[Random_Img_Label] == class_labels[svm_prediction]:
        title = f"True: {class_labels[Random_Img_Label]}\nPredicted: {class_labels[svm_prediction]}"
        color = "green"
    else:
        title = f"True: {class_labels[Random_Img_Label]}\nPredicted: {class_labels[svm_prediction]}"
        color = "red"

    ax.set_title(title, color=color, fontsize=10)
    ax.axis('off')
```

predictions made by an SVM model on random test images and compares them with the true labels.

Loop through images: The code iterates over a batch of randomly selected test images.

Fetch random image and label: Each iteration retrieves a random image and its corresponding true label.

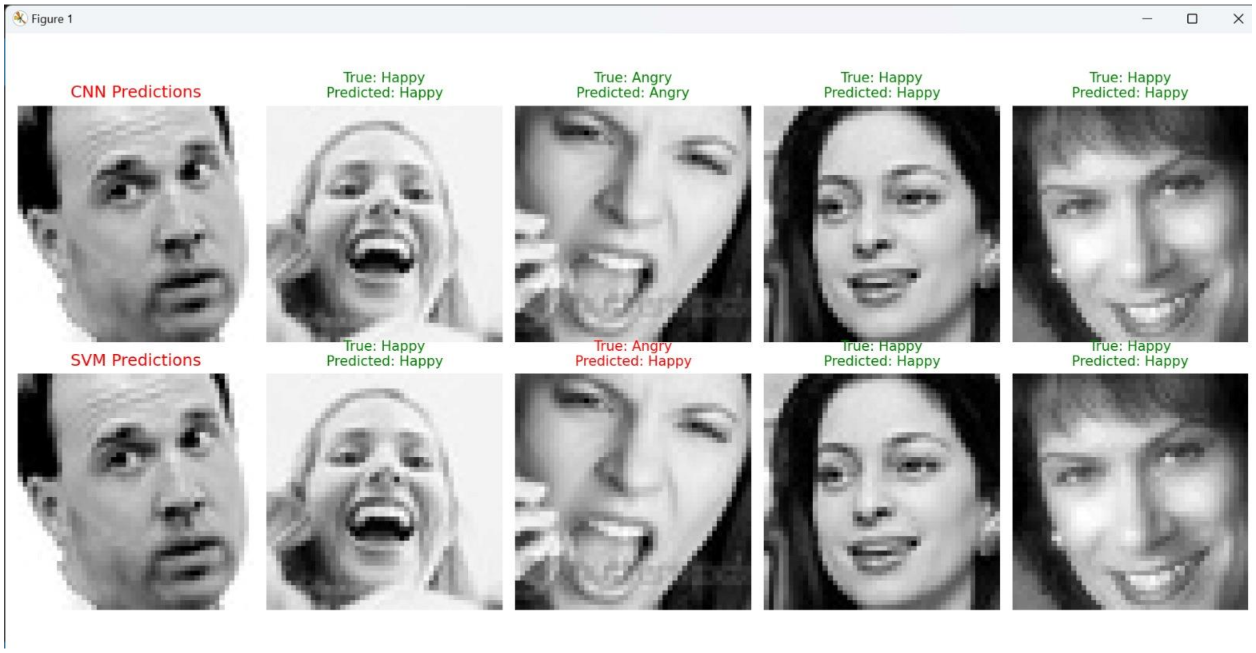
Flatten image for SVM: The image is flattened (converted from a 2D array to 1D) to match the input format expected by the SVM model.

Make a prediction: The SVM model predicts the class of the flattened image.

Plot the image: The image is displayed along with its true and predicted labels.

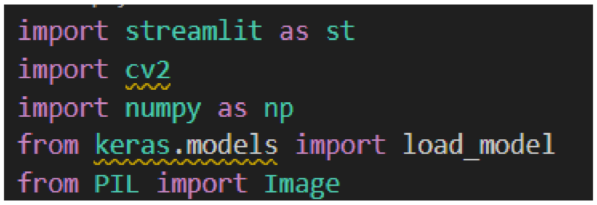
Set title and color: If the prediction is correct, the title is displayed in green; otherwise, it's shown in red.

Display the prediction for CNN and SVM

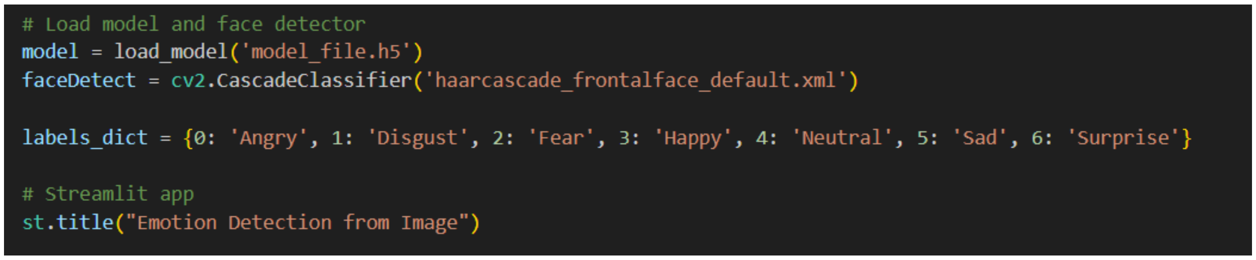


Streamlit App.

Import liabaries:



```python
import streamlit as st
import cv2
import numpy as np
from keras.models import load_model
from PIL import Image
```



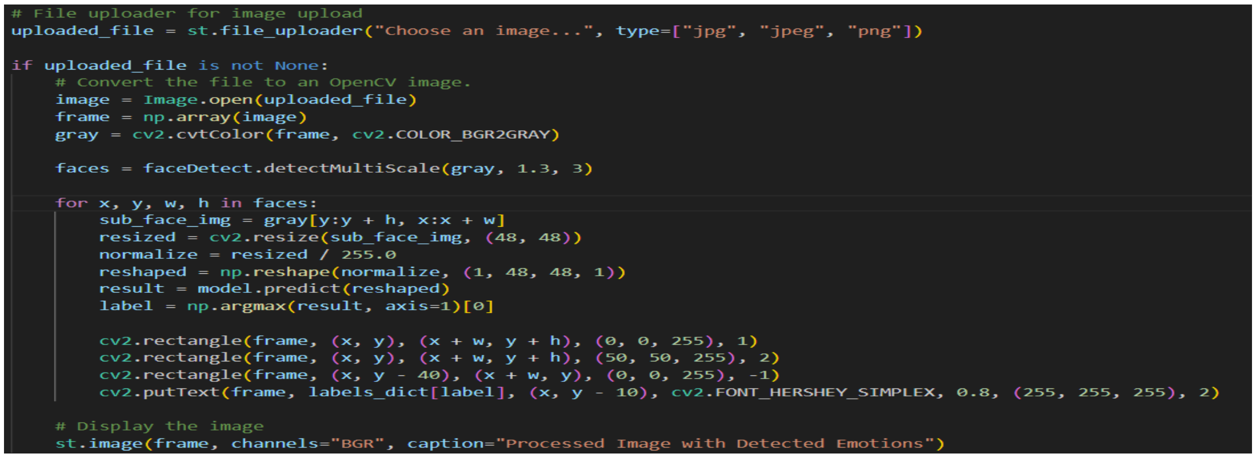
```python
# Load model and face detector
model = load_model('model_file.h5')
faceDetect = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

labels_dict = {0: 'Angry', 1: 'Disgust', 2: 'Fear', 3: 'Happy', 4: 'Neutral', 5: 'Sad', 6: 'Surprise'}

# Streamlit app
st.title("Emotion Detection from Image")
```

A pre-trained model (model_file.h5) is loaded using load_model. This model is expected to predict the emotion in faces.

haarcascade_frontalface_default.xml is a Haar cascade classifier used for detecting faces in images. It is loaded with OpenCV's CascadeClassifier.

This dictionary maps the predicted numerical labels (from 0 to 6) to corresponding emotion categories (e.g., Angry, Happy, etc.)

creates the Streamlit interface for the web application.



```python
# File uploader for image upload
uploaded_file = st.file_uploader("Choose an image...", type=["jpg", "jpeg", "png"])

if uploaded_file is not None:
    # Convert the file to an OpenCV image.
    image = Image.open(uploaded_file)
    frame = np.array(image)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = faceDetect.detectMultiScale(gray, 1.3, 3)

    for x, y, w, h in faces:
        sub_face_img = gray[y:y + h, x:x + w]
        resized = cv2.resize(sub_face_img, (48, 48))
        normalize = resized / 255.0
        reshaped = np.reshape(normalize, (1, 48, 48, 1))
        result = model.predict(reshaped)
        label = np.argmax(result, axis=1)[0]

        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 0, 255), 1)
        cv2.rectangle(frame, (x, y), (x + w, y + h), (50, 50, 255), 2)
        cv2.rectangle(frame, (x, y - 40), (x + w, y), (0, 0, 255), -1)
        cv2.putText(frame, labels_dict[label], (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2)

    # Display the image
    st.image(frame, channels="BGR", caption="Processed Image with Detected Emotions")
```

Streamlit's file_uploader is used to allow users to upload images in JPG, JPEG, or PNG formats.

The uploaded image is opened using PIL.Image.open(), converted to a NumPy array (np.array(image)) to make it compatible with OpenCV, and converted to grayscale (cv2.cvtColor). Grayscale images are preferred for face detection and many deep learning models for facial expressions.

Face Detection: The detectMultiScale function from OpenCV detects faces in the grayscale image. The parameters:

- 1.3: The scaling factor that specifies how much the image size is reduced at each image scale.
- 3: Specifies how many neighbors each candidate rectangle should have to retain it (higher values give fewer detections but with higher quality).

Emotion Prediction:

The face is cropped (sub_face_img) from the grayscale image, resized to a 48x48 image (as expected by the model).

The pixel values are normalized by dividing by 255 (to scale them between 0 and 1).

The image is reshaped to (1, 48, 48, 1) to fit the model's input requirements.

model.predict(reshaped) returns the model's prediction, and np.argmax(result, axis=1)[0] extracts the predicted label (the index of the class with the highest probability).

Drawing Rectangles and Labeling the Detected Faces:

Rectangles are drawn around the detected faces in the image. Three rectangles are drawn:

- The first one outlines the face.
- The second one emphasizes the border.
- The third one serves as the background for the emotion label.

cv2.putText places the emotion label (from labels_dict) on top of the rectangle.

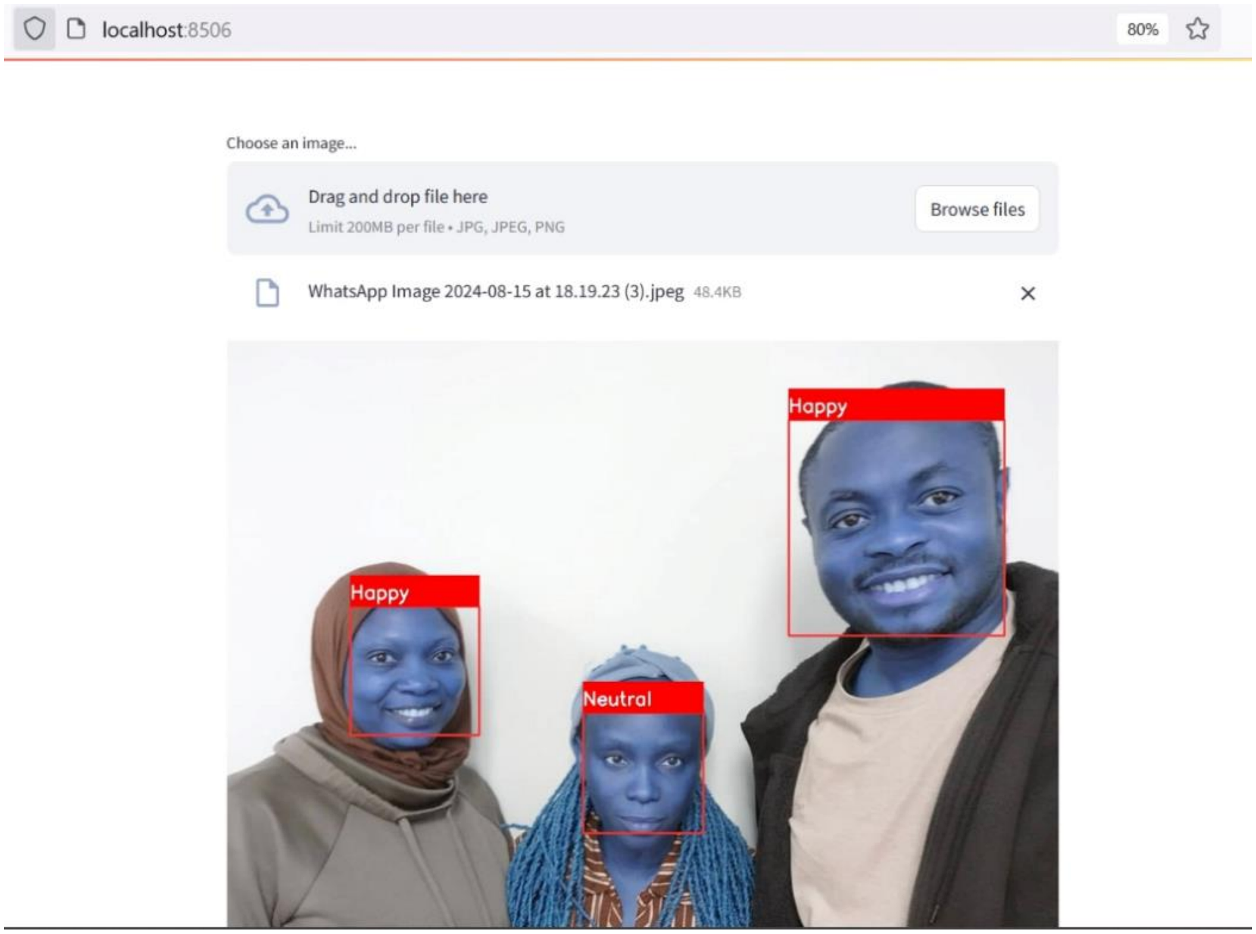
Displaying the Processed Image

The processed image with rectangles and labels is displayed using st.image. The channels="BGR" specifies the color format used by OpenCV, and a caption is provided.

This Streamlit app provides a simple and interactive way for users to upload images, detect faces, and classify the emotions present in those faces using a pre-trained deep learning model.

Deployment

We were about to predict with our picture facial expressions



Choose an image...

Drag and drop file here

Limit 200MB per file • JPG, JPEG, PNG

WhatsApp Image 2024-08-15 at 18.19.23 (3).jpeg 48.4KB

Conclusion

The Convolutional Neural Network (CNN) demonstrated improved learning over time with a a test accuracy of 61.10%. The model showed reasonable performance in classifying emotions from images. In contrast, the Support Vector Machine (SVM) model, though employing hyperparameter tuning, underperformed significantly, achieving only 32.81% accuracy. This indicates that the CNN is better suited for this image classification task, likely due to its ability to capture spatial hierarchies and patterns in image data, while the SVM struggled with the complexity of the visual features

Industries where Facial Expressions Recognition can be used

1. **Healthcare**: Emotion recognition can assist in mental health diagnosis and monitoring, offering insights into patients' emotional states, especially for those with anxiety, depression, or other mood disorders. It can also be useful in elder care or autism spectrum disorder interventions.
2. **Education**: Emotion recognition can be integrated into e-learning platforms to assess student engagement and emotional responses to learning materials, enabling personalized interventions for better learning outcomes.
3. **Customer Service**: Emotion recognition can enhance virtual assistants, chatbots, and call centers by identifying customer emotions during interactions, enabling more empathetic responses and improving customer satisfaction.

Reference

FER-2013 (kaggle.com) --dataset