UNIVERSIDADE ESTADUAL DE LONDRINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GABRYEL CAMILLO LEITE
MURILO ALDIGUERI MARINO

IDENTIFICAÇÂO DE PERSONAGENS DO ANIME JOJO POR MEIO DA CLASSIFICAÇÃO DE IMAGENS

Londrina
2024

GABRYEL CAMILLO LEITE
MURILO ALDIGUERI MARINO

IDENTIFICAÇÂO DE PERSONAGENS DO ANIME JOJO POR MEIO DA CLASSIFICAÇÃO DE IMAGENS

Trabalho submetido à matéria de Inteligência Artificial do curso de Bacharelado em Ciência da Computação, como requisito parcial de nota.

Área de pesquisa: Aprendizagem de Máquina

Professor: Bruno Squizato Faiçal

Londrina
2024

Conteúdo

Londrina
2024

1 Considerações Iniciais

1.1 Objetivos e Mudanças

Montar nosso próprio dataset, coletando um conjunto de dados de imagens dos principais personagens do anime JoJo's Bizarre Adventure, o que acabamos limitando apenas para o arco Stardust Crusaders. Mudamos e aumentamos o dataset com o decorrer do projeto, sendo um dos pontos cruciais para a melhora na acurácia. Utilizamos webscrapping para criação do dataset, usando o site MyAnimesList como referência.

Após a criação do dataset, poderiamos partir para a criação e treinamento do modelo de classificação de imagens para identificar corretamente os personagens.

Para a avaliação do modelo utizamos as métricas recomendadas para modelos de classificaçao de imagens, sendo: acurácia, precisão, recall e a média harmônica dessas duas (F1-score).

2 Pré-Processamento de Dados

2.1 Importação de Bibliotecas

Bibliotecas necessárias para a construção e treinamento do modelo.

```
import os
import numpy as np
from PIL import Image
import torch
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
```

2.2 Inicialização

Antes de iniciar o processo de treinamento do modelo, preparamos o ambiente de trabalho. No código a seguir, importamos as bibliotecas necessárias e definimos algumas constantes e diretórios para armazenar as imagens que serão utilizadas. A variável IMAGE-SIZE é definida como 224, que é a dimensão padrão para as imagens de entrada no modelo VGG16.

```
from torch.utils.data import DataLoader, random_split

# Variável para determinar o tamanho das imagens geradas no pré-
    processamento
IMAGE_SIZE = 224

# Definindo a estrutura de pastas
CHAR_FOLDER = Path("Characters")
IMAGE_FOLDER = CHAR_FOLDER / "Image"
AUGMENTED_TRAIN_FOLDER = CHAR_FOLDER / "Augmented_Train"
```

```

# Criar diretório para imagens aumentadas, se não existir
AUGMENTED_TRAIN_FOLDER.mkdir(parents=True, exist_ok=True)

logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(message)s",
    level=logging.INFO,
    handlers=[
        logging.FileHandler("info.log", "w", "utf-8"),
        logging.StreamHandler(sys.stdout)
    ],
)
```

2.3 Data Augmentation

A função save-augmented-images percorre todas as imagens do dataset original, salvando cada uma e, em seguida, aplica uma série de transformações para gerar versões alternativas de cada imagem.

As transformações incluem redimensionamento, flips horizontais e verticais, e rotação aleatória. Após as transformações, as imagens são salvas nos diretórios específicos de cada classe, aumentando consideravelmente o dataset e contribuindo para o treinamento do modelo.

```
def save_augmented_images(dataset, transform, save_folder):
    for idx in range(len(dataset)):
        img, label = dataset[idx]
        class_folder = save_folder / dataset.classes[label]
        class_folder.mkdir(parents=True, exist_ok=True)

        # Salvar a imagem original
        save_path = class_folder / f"{idx}_original.png"
        img.save(save_path)

        # Aplica as transformações e salva as imagens aumentadas
        for i in range(5):  # Aplica e salva augmentações por
    imagem
            augmented_img = transform(img)

            if isinstance(augmented_img, torch.Tensor):
                augmented_img_pil = transforms.ToPILImage()(
    augmented_img)
            else:
                augmented_img_pil = augmented_img

            # Salva a imagem aumentada
            save_path = class_folder / f"{idx}_{i}.png"
            augmented_img_pil.save(save_path)

# Transformações para pré-processamento das imagens
data_transforms = {
    'augment': transforms.Compose([
        transforms.Resize(IMAGE_SIZE),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
```

```
        transforms.RandomRotation(degrees=30),
        transforms.ToTensor(),
    ]),
    'final': transforms.Compose([
        transforms.Resize(IMAGE_SIZE + 32),
        transforms.CenterCrop(IMAGE_SIZE),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
    ]),
}

# Carregar o dataset original
original_dataset = datasets.ImageFolder(IMAGE_FOLDER)

# Salvar as imagens aumentadas
save_augmented_images(original_dataset, data_transforms['augment'], AUGMENTED_TRAIN_FOLDER)
```

2.4 Dataset

Dividimos o dataset em conjuntos de treino e teste, com 80% dos dados sendo utilizados para treino e 20% para teste.

```
# Carregar o dataset aumentado
augmented_dataset = datasets.ImageFolder(AUGMENTED_TRAIN_FOLDER, transform=data_transforms['final'])

# Separar o dataset em treino (80%) e teste (20%)
train_size = int(0.8 * len(augmented_dataset))
test_size = len(augmented_dataset) - train_size
train_dataset, test_dataset = random_split(augmented_dataset, [train_size, test_size])

# Criando os data loaders
dataloaders = {
    'train': DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4),
    'test': DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=4)
}

# Obtendo o número de classes
class_names = [name.replace('-', ' ') for name in augmented_dataset.classes]
num_classes = len(class_names)

print(f"Classes: {class_names}")
```

Classes: ['Dio Brando', 'Iggy', 'Jean Pierre Polnareff', 'Joseph Joestar', 'Jotarou Kuujou', 'Muhammad Avdol', 'Noriaki Kakyouin']

2.5 Exibição de Imagens

A visualização é feita utilizando a biblioteca matplotlib, ajustando o tamanho da figura e removendo os eixos para uma apresentação mais clara.

Desta forma podemos inspecionar visualmente os dados e verificar se as transformações de augmentação e normalização foram aplicadas corretamente, assegurando que as imagens estejam prontas para serem usadas no treinamento do modelo.

```
# Função para mostrar imagens
def imshow(inp):
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    plt.axis('off')

# Obtendo um batch de imagens de treino
inputs, classes = next(iter(dataloaders['train']))

# Fazendo um grid de imagens
out = utils.make_grid(inputs)

# Aumentando o tamanho da figura
plt.figure(figsize=(12, 12))

# Mostrando as imagens
imshow(out)
plt.show()
```

3 Definição do Modelo

3.1 Importação do Modelo VGG16 Pré-treinado

O modelo VGG16 é carregado utilizando a função models.vgg16 com pesos pré-treinados do ImageNet. Em seguida, os parâmetros do modelo são congelados, ou seja, definidos como não atualizáveis (requires-grad = False), para que apenas os parâmetros da última camada totalmente conectada sejam treinados durante o processo de fine-tuning.

Para adaptar o modelo às novas classes específicas do problema, a última camada do classificador é substituída por uma nova camada totalmente conectada com o número de neurônios igual ao número de classes do novo conjunto de dados.

```
# Carregando o modelo VGG16 pré-treinado
model = models.vgg16(weights=VGG16_Weights.IMAGENET1K_V1)

# Congelando os parâmetros do modelo
```

```
for param in model.parameters():
    param.requires_grad = False

# Número de características de entrada para a última camada
num_ftrs = model.classifier[6].in_features
model.classifier[6] = nn.Linear(num_ftrs, num_classes)

# Exibindo o modelo para verificar as mudanças
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
     ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=7, bias=True)
  )
)
```

4 Treinamento do Modelo

No código a seguir, definimos o dispositivo (GPU ou CPU) a ser utilizado, movemos o modelo para o dispositivo adequado e configuramos a função de perda e o otimizador.

Utilizamos a função train-model para gerenciar o processo de treinamento e validação do modelo. A cada época, o modelo alterna entre as fases de treino e validação. Durante a fase de treino, o modelo é ajustado (treinado) com os dados de entrada, e o otimizador ajusta os pesos do modelo para minimizar a função de perda. Na fase de validação, o modelo é avaliado com um conjunto de dados diferente, e as métricas de desempenho são registradas sem realizar ajustes nos pesos.

A função train-model registra o histórico de perda e precisão para ambas as fases, e salva o melhor modelo baseado na precisão de validação. Gráficos são atualizados dinamicamente para visualizar o progresso do treinamento. Ao final, o melhor modelo é carregado e salvo.

```
# Definindo o dispositivo (GPU ou CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "
    cpu")
model = model.to(device)
logging.info(f"Usando {'cuda' if torch.cuda.is_available() else '
    cpu'}")

# Definindo a função de perda e o otimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier[6].parameters(), lr=0.001)

# Função para treinar o modelo
def train_model(model, dataloaders, criterion, optimizer,
    num_epochs, model_path):
    best_model_wts = model.state_dict()
    best_acc = 0.0

```

```
    train_loss_history = []
    train_acc_history = []
    val_loss_history = []
    val_acc_history = []

    for epoch in range(num_epochs):
        logging.info(f'Epoch {epoch}/{num_epochs - 1}')
        logging.info('-' * 10)

        # Cada época tem uma fase de treino e uma de validação
        for phase in ['train', 'test']:
            if phase == 'train':
                model.train()  # Definir o modelo para treinamento
            else:
                model.eval()   # Definir o modelo para validação

            running_loss = 0.0
            running_corrects = 0

            # Iterar sobre os dados
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()

                # Somente faça cálculo de gradiente na fase de
    treinamento
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backpropagation e otimização na fase de
    treinamento
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].
    dataset)
            epoch_acc = running_corrects.double() / len(dataloaders
    [phase].dataset)

            logging.info(f'{phase} Loss: {epoch_loss:.4f} Acc: {
    epoch_acc:.4f}')

            # Armazenar a perda e precisão para plotagem
            if phase == 'train':
                train_loss_history.append(epoch_loss)
                train_acc_history.append(epoch_acc.item())
            else:
                val_loss_history.append(epoch_loss)
                val_acc_history.append(epoch_acc.item())
```

```
67
68              # Salvar o melhor modelo
69              if phase == 'test' and epoch_acc > best_acc:
70                  best_acc = epoch_acc
71                  best_model_wts = model.state_dict()
72
73          # Atualizar gráficos dinamicamente
74          clear_output(wait=True)
75          plt.figure(figsize=(12, 4))
76          plt.subplot(1, 2, 1)
77          plt.plot(train_loss_history, label='Train Loss')
78          plt.plot(val_loss_history, label='Val Loss')
79          plt.xlabel('Epoch')
80          plt.ylabel('Loss')
81          plt.legend()
82          plt.subplot(1, 2, 2)
83          plt.plot(train_acc_history, label='Train Acc')
84          plt.plot(val_acc_history, label='Val Acc')
85          plt.xlabel('Epoch')
86          plt.ylabel('Accuracy')
87          plt.legend()
88          display(plt.gcf())
89
90      logging.info(f'Best val Acc: {best_acc:.4f}')
91
92      # Carregar os melhores pesos do modelo
93      model.load_state_dict(best_model_wts)
94
95      # Salvar o modelo treinado
96      torch.save(model.state_dict(), model_path)
97      logging.info("Modelo salvo!")
98
99      return model, train_loss_history, val_loss_history,
        train_acc_history, val_acc_history
100
101 # Carregar o modelo salvo, se existir
102 model_path = 'model.pth'
103 if Path(model_path).exists():
104     model.load_state_dict(torch.load(model_path))
105     logging.info("Carregado modelo salvo!")
106
107 # Treinar o modelo
108 model, train_loss_history, val_loss_history, train_acc_history,
        val_acc_history  = train_model(model, dataloaders, criterion,
        optimizer, 25, model_path)
```

5 Avaliação do Modelo

5.1 Métricas

A função evaluate-model coloca o modelo em modo de avaliação (eval) e desativa a autograd para evitar cálculos desnecessários de gradientes. Em seguida, percorre os dados de teste, coleta as previsões e os rótulos verdadeiros, e calcula as métricas de desempenho: acurácia, precisão, recall e F1-score.

Essas métricas fornecem uma visão abrangente do desempenho do modelo, considerando não apenas a proporção de previsões corretas (acurácia), mas também a qualidade das previsões positivas (precisão), a capacidade de identificar todas as instâncias relevantes (recall) e uma média harmônica dessas duas (F1-score).

```
from sklearn.metrics import accuracy_score, f1_score,
    precision_score, recall_score


def evaluate_model(model, dataloaders):
    model.eval()
    all_labels = []
    all_preds = []

    with torch.no_grad():
        for inputs, labels in dataloaders['test']:
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())

    accuracy = accuracy_score(all_labels, all_preds)
    precision = precision_score(all_labels, all_preds, average='
    weighted')
    recall = recall_score(all_labels, all_preds, average='weighted'
    )
    f1 = f1_score(all_labels, all_preds, average='weighted')

    print(f'Acurácia: {accuracy*100:.2f}%')
    print(f'Precisão: {precision*100:.2f}%')
    print(f'Recall: {recall*100:.2f}%')
    print(f'F1-Score: {f1*100:.2f}%')

    return accuracy, precision, recall, f1

# Avaliar o modelo
accuracy, precision, recall, f1 = evaluate_model(model, dataloaders
    )
```

Com um dos treinamentos que fizemos obtemos resultados muito semelhantes:
Acurácia: 93.43 %
Precisão: 93.53 %
Recall: 93.43 %
F1-Score: 93.41 %

5.2 Apresentação dos Gráficos

5.2.1 Métricas Gerais

A função plot-metrics cria dois gráficos. O primeiro mostrando a perda e a acurácia tanto para o treinamento quanto para a validação ao longo das épocas, ajudando a entender como o modelo está se ajustando durante o treinamento e se está generalizando bem para novos dados. O segundo gráfico exibe as métricas de desempenho finais do modelo, incluindo acurácia, precisão, recall e F1-score.

```
def plot_metrics(train_loss_history, val_loss_history,
    train_acc_history, val_acc_history, accuracy, precision, recall
    , f1):
    epochs = range(len(train_loss_history))

    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.plot(epochs, train_loss_history, label='Train Loss')
    plt.plot(epochs, val_loss_history, label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(2, 2, 2)
    plt.plot(epochs, train_acc_history, label='Train Acc')
    plt.plot(epochs, val_acc_history, label='Val Acc')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.figure(figsize=(8, 8))
    plt.bar(['Accuracy', 'Precision', 'Recall', 'F1-Score'], [
    accuracy, precision, recall, f1])
    plt.ylim(0, 1)
    plt.ylabel('Score')

    plt.tight_layout()
    plt.show()

plot_metrics(train_loss_history, val_loss_history,
    train_acc_history, val_acc_history, accuracy, precision, recall
    , f1)
```

Analisando o gráfico de acurácia podemos notar o crescimento exponencial nas primeiras épocas e a estabelização com o decorrer delas, ficando basicamente constante já a partir da décima época, obtendo declínios mínimos, alcaçando e permanecendo em 0.935.

Paralelamente, a função de perda descresce exponencialmente e também se estabeliza com o decorrer das épocas, permanecendo em 0.2.

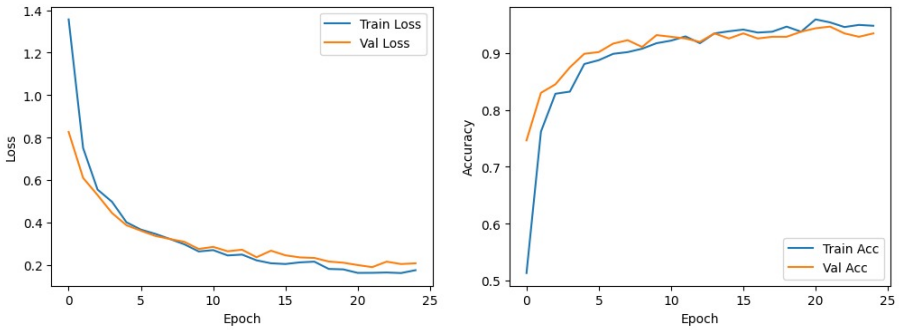

Figura 1: Gráficos de treinamento e teste em relação as épocas.

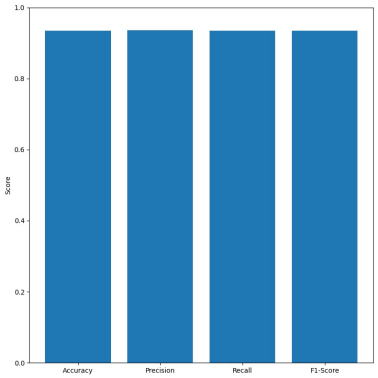

Figura 2: Métricas de avaliação (Acurácia, Precisão, Recall e F1-Score).

5.2.2 Métricas por Classe

A função evaluate-model-per-class avalia o modelo para cada classe individualmente, coletando todas as previsões e rótulos verdadeiros. Em seguida, calcula a precisão, recall e F1-score para cada classe.

Além disso, a função cria uma matriz de confusão que ilustra visualmente as previsões versus os rótulos verdadeiros para todas as classes.

```
def evaluate_model_per_class(model, dataloaders, class_names):
    model.eval()

    y_true = []
    y_pred = []

    with torch.no_grad():
        for inputs, labels in dataloaders['test']:
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

```

```
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())

    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    # Calcula as métricas por classe
    precision = precision_score(y_true, y_pred, average=None)
    recall = recall_score(y_true, y_pred, average=None)
    f1 = f1_score(y_true, y_pred, average=None)

    # Cria a matriz de confusão
    conf_matrix = confusion_matrix(y_true, y_pred)

    # Plota a matriz de confusão
    plt.figure(figsize=(10, 8))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
    xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

    # Plota as métricas por classe
    fig, axes = plt.subplots(1, 3, figsize=(18, 5))

    axes[0].bar(class_names, precision, color='blue')
    axes[0].set_title('Precision by Class')
    axes[0].set_xticklabels(class_names, rotation=90)

    axes[1].bar(class_names, recall, color='green')
    axes[1].set_title('Recall by Class')
    axes[1].set_xticklabels(class_names, rotation=90)

    axes[2].bar(class_names, f1, color='red')
    axes[2].set_title('F1 Score by Class')
    axes[2].set_xticklabels(class_names, rotation=90)

    plt.tight_layout()
    plt.show()


evaluate_model_per_class(model, dataloaders, class_names)
```

Analisando a matriz de confusão podemos perceber os personagem que são melhores reconhecidos e quantas vezes cada personagem foi confudido com outro.

O Iggy por exemplo foi nitidamente o melhor classificado, isso se deve ao fato dele ser o único animal, enquanto os demais persongaens são humanos.

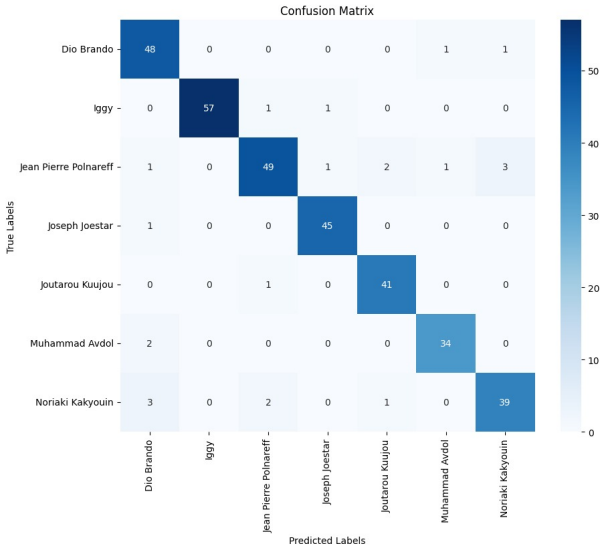

Figura 3: Matriz de Confusão.

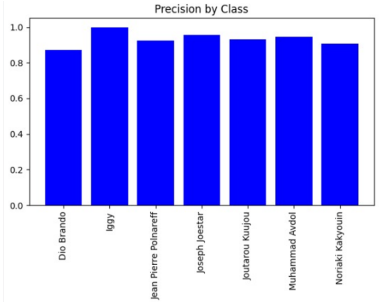

Figura 4: Gráfico de precisão dos personagens.

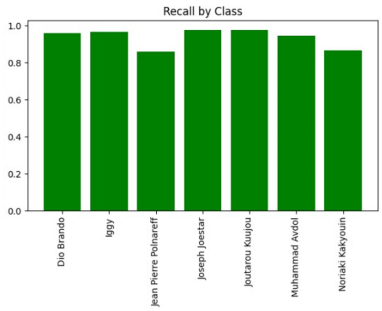

Figura 5: Gráfico recall dos personagens.

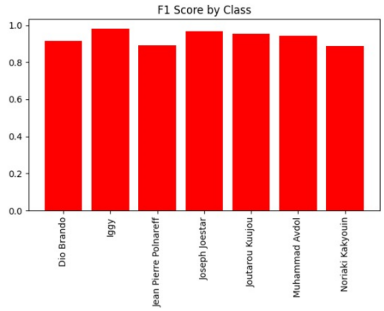

Figura 6: Gráfico F1-Score dos personagens.

6 Conclusão e Observações

Ficamos satisfeitos com os resultados obtidos, alcançando uma boa acurácia de 93.43 porcento, mostrando que nosso objetivo inicial de conseguir classificar e identificar corretamente os personagens foi realizado.

Em relação ao desenvolvimento tiveram pontos que foram de suma importância para obter os resultados finais. A matriz de confusão foi um dos que mais nos baseamos, visto que a partir da sua analise podemos notar quais personagens estão sendo melhor identificados e com quais está mais sendo confudido, desta forma alteramos o dataset conseguindo uma melhora significativa no modelo.

Na imagem abaixo está a matriz de confusão antes da melhora no dataset. Neste teste obteve uma acurácia de 77.81 porcento.

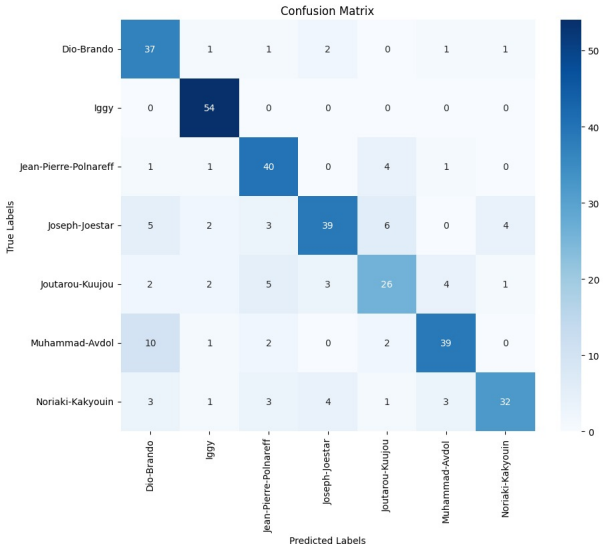

Figura 6: Matriz de Confusão (2).

Se compararmos com a matriz de confusão apresentada anteriormente vemos a significativa mudança. Onde neste por exemplo, o personagem Joutarou foi o pior classificado, sendo reconhecido corretamente apenas 26 vezes no testes, já depois das mudanças foi para 41 vezes.