Skin disease classification using a SWIN Transformer

1. Importing (and/or installing) the necessary libraries

We also define a function to print the elapsed time for resource management purposes.

In [1]:
```
# Debugging code removed for clarity
```

In [2]:
```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader, random_split
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import pandas as pd
from sklearn.metrics import f1_score, accuracy_score
import os
from torchvision.datasets import VisionDataset
from torchvision.datasets.folder import IMG_EXTENSIONS, default_loader
from torchvision.datasets.utils import list_files
import torch.optim.lr_scheduler as lr_scheduler
from transformers import AutoImageProcessor, AutoModelForImageClassificat
import random
from holisticai.bias.metrics import disparate_impact, precision_matrix
from fairlearn.metrics import equalized_odds_ratio
from IPython.display import Image, display
import PIL
from tqdm.notebook import tqdm
```

2. Building the dataset class

In [3]:
```
# Debugging code removed for clarity
```

```
Device: cpu
```

In [4]:
```
# Debugging code removed for clarity
```

In [5]:
```
###################### SET THE SEED FOR REPRODUCIBILITY ################
seed = 42
```

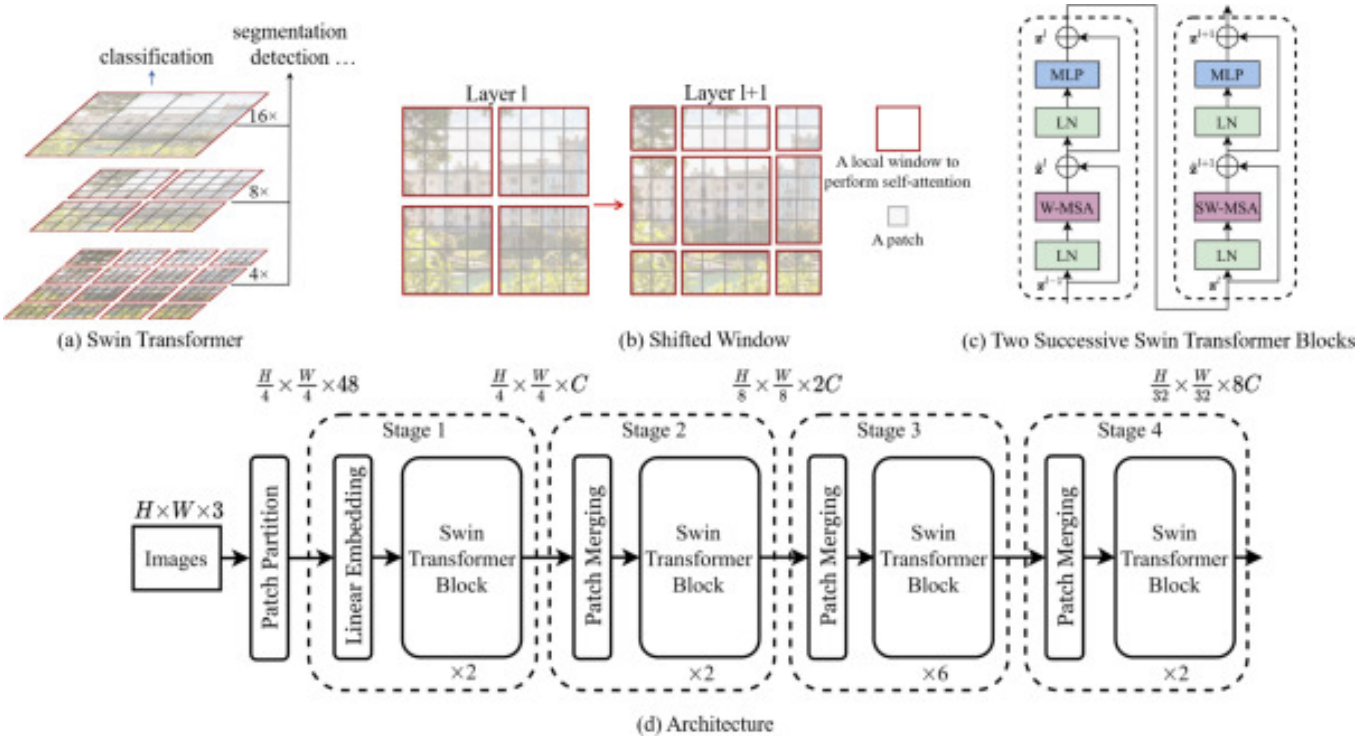
```
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)

############################### INSTANTIATING THE DATASET AND SPLITTING #
dataset = CustomImageFolderColor(folder_path)

# Split the dataset into train and test sets
train_size = int(0.6 * len(dataset))  # 60% train, 20% validation, 20% te
val_size = int(0.2 * len(dataset))
test_size = len(dataset) - train_size - val_size
train_dataset, val_dataset, test_dataset = random_split(dataset, [train_s
```

```
0%|          | 0/9 [00:00<?, ?it/s]
```

3. Building the model

The chosen architecture is a **Swin Transformer**, firstly introduced by Liu et al (https://arxiv.org/abs/2103.14030). An image of the architecture is shown below. The Swin Transformer is a type of Vision Transformer. It builds hierarchical feature maps by merging image patches in deeper layers and has linear computation complexity to input image size due to computation of self-attention only within each local window. It can thus serve as a general-purpose backbone for both image classification and dense recognition tasks. In contrast, previous vision Transformers produce feature maps of a single low resolution and have quadratic computation complexity to input image size due to computation of self-attention globally.

In [15]:
```
display(Image(url='https://ars.els-cdn.com/content/image/1-s2.0-S09521976
```



In [17]:
```
# Debugging code removed for clarity
```

```
Detected 0 devices

SwinForImageClassification(
  (swin): SwinModel(
    (embeddings): SwinEmbeddings(
      (patch_embeddings): SwinPatchEmbeddings(
        (projection): Conv2d(3, 96, kernel_size=(4, 4), stride=(4, 4))
      )
      (norm): LayerNorm((96,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): SwinEncoder(
      (layers): ModuleList(
        (0): SwinStage(
          (blocks): ModuleList(
            (0-1): 2 x SwinLayer(
              (layernorm_before): LayerNorm((96,), eps=1e-05, elementwise_affine=T
rue)
              (attention): SwinAttention(
                (self): SwinSelfAttention(
                  (query): Linear(in_features=96, out_features=96, bias=True)
                  (key): Linear(in_features=96, out_features=96, bias=True)
                  (value): Linear(in_features=96, out_features=96, bias=True)
                  (dropout): Dropout(p=0.0, inplace=False)
                )
                (output): SwinSelfOutput(
                  (dense): Linear(in_features=96, out_features=96, bias=True)
                  (dropout): Dropout(p=0.0, inplace=False)
                )
              )
              (drop_path): SwinDropPath(p=0.1)
              (layernorm_after): LayerNorm((96,), eps=1e-05, elementwise_affine=Tr
ue)
              (intermediate): SwinIntermediate(
                (dense): Linear(in_features=96, out_features=384, bias=True)
                (intermediate_act_fn): GELUActivation()
              )
              (output): SwinOutput(
                (dense): Linear(in_features=384, out_features=96, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
            )
          )
          (downsample): SwinPatchMerging(
            (reduction): Linear(in_features=384, out_features=192, bias=False)
            (norm): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
          )
        )
        (1): SwinStage(
          (blocks): ModuleList(
            (0-1): 2 x SwinLayer(
              (layernorm_before): LayerNorm((192,), eps=1e-05, elementwise_affine=
True)
              (attention): SwinAttention(
                (self): SwinSelfAttention(
                  (query): Linear(in_features=192, out_features=192, bias=True)
                  (key): Linear(in_features=192, out_features=192, bias=True)
```

```
                (value): Linear(in_features=192, out_features=192, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
              (output): SwinSelfOutput(
                (dense): Linear(in_features=192, out_features=192, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
            )
            (drop_path): SwinDropPath(p=0.1)
            (layernorm_after): LayerNorm((192,), eps=1e-05, elementwise_affine=T
rue)
            (intermediate): SwinIntermediate(
              (dense): Linear(in_features=192, out_features=768, bias=True)
              (intermediate_act_fn): GELUActivation()
            )
            (output): SwinOutput(
              (dense): Linear(in_features=768, out_features=192, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
        )
        (downsample): SwinPatchMerging(
          (reduction): Linear(in_features=768, out_features=384, bias=False)
          (norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
      )
      (2): SwinStage(
        (blocks): ModuleList(
          (0-5): 6 x SwinLayer(
            (layernorm_before): LayerNorm((384,), eps=1e-05, elementwise_affine=
True)
            (attention): SwinAttention(
              (self): SwinSelfAttention(
                (query): Linear(in_features=384, out_features=384, bias=True)
                (key): Linear(in_features=384, out_features=384, bias=True)
                (value): Linear(in_features=384, out_features=384, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
              (output): SwinSelfOutput(
                (dense): Linear(in_features=384, out_features=384, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
            )
            (drop_path): SwinDropPath(p=0.1)
            (layernorm_after): LayerNorm((384,), eps=1e-05, elementwise_affine=T
rue)
            (intermediate): SwinIntermediate(
              (dense): Linear(in_features=384, out_features=1536, bias=True)
              (intermediate_act_fn): GELUActivation()
            )
            (output): SwinOutput(
              (dense): Linear(in_features=1536, out_features=384, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
        )
        (downsample): SwinPatchMerging(
```

```
            (reduction): Linear(in_features=1536, out_features=768, bias=False)
            (norm): LayerNorm((1536,), eps=1e-05, elementwise_affine=True)
          )
        )
        (3): SwinStage(
          (blocks): ModuleList(
            (0-1): 2 x SwinLayer(
              (layernorm_before): LayerNorm((768,), eps=1e-05, elementwise_affine=
True)
              (attention): SwinAttention(
                (self): SwinSelfAttention(
                  (query): Linear(in_features=768, out_features=768, bias=True)
                  (key): Linear(in_features=768, out_features=768, bias=True)
                  (value): Linear(in_features=768, out_features=768, bias=True)
                  (dropout): Dropout(p=0.0, inplace=False)
                )
                (output): SwinSelfOutput(
                  (dense): Linear(in_features=768, out_features=768, bias=True)
                  (dropout): Dropout(p=0.0, inplace=False)
                )
              )
              (drop_path): SwinDropPath(p=0.1)
              (layernorm_after): LayerNorm((768,), eps=1e-05, elementwise_affine=T
rue)
              (intermediate): SwinIntermediate(
                (dense): Linear(in_features=768, out_features=3072, bias=True)
                (intermediate_act_fn): GELUActivation()
              )
              (output): SwinOutput(
                (dense): Linear(in_features=3072, out_features=768, bias=True)
                (dropout): Dropout(p=0.0, inplace=False)
              )
            )
          )
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (pooler): AdaptiveAvgPool1d(output_size=1)
  )
  (classifier): Linear(in_features=768, out_features=9, bias=True)
)
```

We decide to train the last two stages of the Transformer, for a total of about 26 million parameters.

In [18]:
```
# Debugging code removed for clarity
```

```
Trainable Layer: swin.encoder.layers.2.blocks.0.layernorm_before.weight | Size: to
rch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.layernorm_before.bias | Size: torc
h.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.relative_position_b
ias_table | Size: torch.Size([169, 12])
```

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.query.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.query.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.key.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.key.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.value.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.self.value.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.output.dense.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.0.attention.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.layernorm_after.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.layernorm_after.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.0.intermediate.dense.weight | Size: torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.0.intermediate.dense.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.0.output.dense.weight | Size: torch.Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.0.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.layernorm_before.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.layernorm_before.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.relative_position_bias_table | Size: torch.Size([169, 12])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.query.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.query.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.key.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.key.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.value.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.self.value.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.output.dense.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.1.attention.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.layernorm_after.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.layernorm_after.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.1.intermediate.dense.weight | Size: torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.1.intermediate.dense.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.1.output.dense.weight | Size: torch.Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.1.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.layernorm_before.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.layernorm_before.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.relative_position_bias_table | Size: torch.Size([169, 12])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.query.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.query.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.key.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.key.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.value.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.self.value.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.output.dense.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.2.attention.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.layernorm_after.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.layernorm_after.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.2.intermediate.dense.weight | Size: torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.2.intermediate.dense.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.2.output.dense.weight | Size: torch.Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.2.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.layernorm_before.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.layernorm_before.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.relative_position_bias_table | Size: torch.Size([169, 12])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.query.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.query.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.key.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.key.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.value.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.self.value.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.output.dense.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.3.attention.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.layernorm_after.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.layernorm_after.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.3.intermediate.dense.weight | Size: torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.3.intermediate.dense.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.3.output.dense.weight | Size: torch.Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.3.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.layernorm_before.weight | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.layernorm_before.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.relative_position_bias_table | Size: torch.Size([169, 12])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.query.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.query.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.key.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.key.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.value.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.self.value.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.output.dense.weight | Size: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.4.attention.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.layernorm_after.weight | Size: tor
ch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.layernorm_after.bias | Size: torc
h.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.4.intermediate.dense.weight | Size:
torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.4.intermediate.dense.bias | Size: to
rch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.4.output.dense.weight | Size: torch.
Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.4.output.dense.bias | Size: torch.Si
ze([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.layernorm_before.weight | Size: to
rch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.layernorm_before.bias | Size: torc
h.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.relative_position_b
ias_table | Size: torch.Size([169, 12])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.query.weight | Siz
e: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.query.bias | Size:
torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.key.weight | Size:
torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.key.bias | Size: to
rch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.value.weight | Siz
e: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.self.value.bias | Size:
torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.output.dense.weight | Si
ze: torch.Size([384, 384])

Trainable Layer: swin.encoder.layers.2.blocks.5.attention.output.dense.bias | Siz
e: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.layernorm_after.weight | Size: tor
ch.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.layernorm_after.bias | Size: torc
h.Size([384])

Trainable Layer: swin.encoder.layers.2.blocks.5.intermediate.dense.weight | Size: torch.Size([1536, 384])

Trainable Layer: swin.encoder.layers.2.blocks.5.intermediate.dense.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.blocks.5.output.dense.weight | Size: torch.Size([384, 1536])

Trainable Layer: swin.encoder.layers.2.blocks.5.output.dense.bias | Size: torch.Size([384])

Trainable Layer: swin.encoder.layers.2.downsample.reduction.weight | Size: torch.Size([768, 1536])

Trainable Layer: swin.encoder.layers.2.downsample.norm.weight | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.2.downsample.norm.bias | Size: torch.Size([1536])

Trainable Layer: swin.encoder.layers.3.blocks.0.layernorm_before.weight | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.layernorm_before.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.relative_position_bias_table | Size: torch.Size([169, 24])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.query.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.query.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.key.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.key.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.value.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.self.value.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.output.dense.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.0.attention.output.dense.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.layernorm_after.weight | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.layernorm_after.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.0.intermediate.dense.weight | Size: torch.Size([3072, 768])

Trainable Layer: swin.encoder.layers.3.blocks.0.intermediate.dense.bias | Size: torch.Size([3072])

Trainable Layer: swin.encoder.layers.3.blocks.0.output.dense.weight | Size: torch.Size([768, 3072])

Trainable Layer: swin.encoder.layers.3.blocks.0.output.dense.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.layernorm_before.weight | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.layernorm_before.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.relative_position_bias_table | Size: torch.Size([169, 24])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.query.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.query.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.key.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.key.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.value.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.self.value.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.output.dense.weight | Size: torch.Size([768, 768])

Trainable Layer: swin.encoder.layers.3.blocks.1.attention.output.dense.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.layernorm_after.weight | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.layernorm_after.bias | Size: torch.Size([768])

Trainable Layer: swin.encoder.layers.3.blocks.1.intermediate.dense.weight | Size: torch.Size([3072, 768])

```
Trainable Layer: swin.encoder.layers.3.blocks.1.intermediate.dense.bias | Size: to
rch.Size([3072])

Trainable Layer: swin.encoder.layers.3.blocks.1.output.dense.weight | Size: torch.
Size([768, 3072])

Trainable Layer: swin.encoder.layers.3.blocks.1.output.dense.bias | Size: torch.Si
ze([768])

Trainable Layer: swin.layernorm.weight | Size: torch.Size([768])

Trainable Layer: swin.layernorm.bias | Size: torch.Size([768])

Trainable Layer: classifier.weight | Size: torch.Size([9, 768])

Trainable Layer: classifier.bias | Size: torch.Size([9])
```

In [19]:
```
# Debugging code removed for clarity
```

```
Trainable parameters: 26033985
```

4. Utilities

Dataloaders

First we define the batch size, the epoch and the learning rate hyperparameters:

In [125…
```
# Debugging code removed for clarity
```

```
LEARNING_RATE: 0.01, BATCH_SIZE: 512, N_EPOCHS: 16
```

In [27]:
```
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=T
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=Fal
```

Loss, Optimizer, Scheduler and EarlyStopper

In [23]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(filter(lambda p: p.requires_grad, model.parameters(
scheduler = lr_scheduler.MultiStepLR(optimizer, milestones=[9, 17], gamma
```

4. Training the model

In [11]:
```
# Debugging code removed for clarity
```

```
[Epoch 1, Batch     1 / 387] Training Loss: 2.2712771892547607
Accuracy on train set: 0.6737851119335072
```

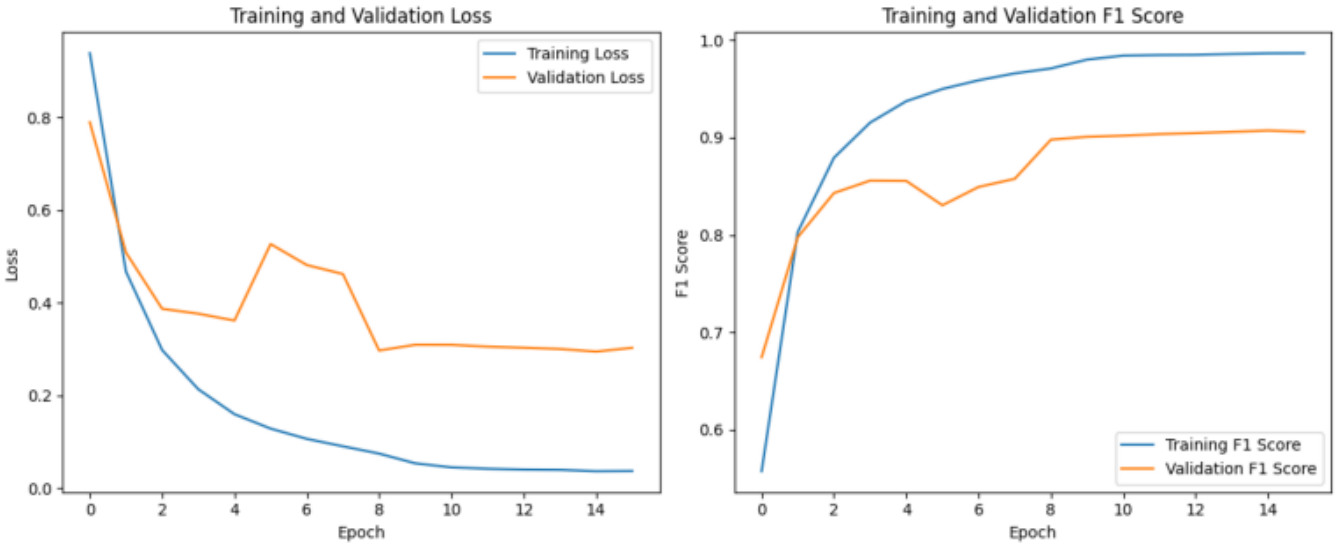
```
F1 score on train set: 0.5576003047411322
Accuracy on validation set: 0.719165200509616
F1 score on validation set: 0.6744923962320357
[Epoch 1, Validation Loss: 0.790]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 0 hours, 58 minutes, 36 seconds
[Epoch 2, Batch     1 / 387] Training Loss: 0.682073712348938
Accuracy on train set: 0.8358510788893607
F1 score on train set: 0.8030003023878756
Accuracy on validation set: 0.8243493296123279
F1 score on validation set: 0.7976468150244523
[Epoch 2, Validation Loss: 0.509]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 1 hours, 57 minutes, 20 seconds
[Epoch 3, Batch     1 / 387] Training Loss: 0.341732919216156
Accuracy on train set: 0.896064632247366
F1 score on train set: 0.8791760608344547
Accuracy on validation set: 0.8683643754170964
F1 score on validation set: 0.8429461742969863
[Epoch 3, Validation Loss: 0.387]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 2 hours, 56 minutes, 46 seconds
[Epoch 4, Batch     1 / 387] Training Loss: 0.18321247398853302
Accuracy on train set: 0.9252613804121418
F1 score on train set: 0.9151647804633384
Accuracy on validation set: 0.8744160650367044
F1 score on validation set: 0.8556909161767179
[Epoch 4, Validation Loss: 0.377]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 3 hours, 56 minutes, 23 seconds
[Epoch 5, Batch     1 / 387] Training Loss: 0.14888884127140045
Accuracy on train set: 0.9440281906610851
F1 score on train set: 0.9371706059865469
Accuracy on validation set: 0.8807255960686768
F1 score on validation set: 0.8553979240161621
[Epoch 5, Validation Loss: 0.362]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 4 hours, 55 minutes, 17 seconds
[Epoch 6, Batch     1 / 387] Training Loss: 0.13273616135120392
Accuracy on train set: 0.95525693138385
F1 score on train set: 0.9497076292974894
Accuracy on validation set: 0.8459321725414063
F1 score on validation set: 0.8303987144742642
[Epoch 6, Validation Loss: 0.527]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 5 hours, 53 minutes, 25 seconds
[Epoch 7, Batch     1 / 387] Training Loss: 0.1280478537082672
Accuracy on train set: 0.9628506137636758
F1 score on train set: 0.9586233786555154
Accuracy on validation set: 0.8605532973366499
F1 score on validation set: 0.8492141899765208
[Epoch 7, Validation Loss: 0.482]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 6 hours, 50 minutes, 9 seconds
[Epoch 8, Batch     1 / 387] Training Loss: 0.09685322642326355
Accuracy on train set: 0.9691348662258084
F1 score on train set: 0.9657024333434006
```

```
Accuracy on validation set: 0.8700934295941273
F1 score on validation set: 0.8575349364914756
[Epoch 8, Validation Loss: 0.462]
UPDATING LEARNING RATE TO:  0.01
Elapsed time: 7 hours, 47 minutes, 25 seconds
[Epoch 9, Batch     1 / 387] Training Loss: 0.0526040755212307
Accuracy on train set: 0.9739984630629538
F1 score on train set: 0.9707371473046711
Accuracy on validation set: 0.9113328884305042
F1 score on validation set: 0.8977098778950536
[Epoch 9, Validation Loss: 0.297]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 8 hours, 43 minutes, 57 seconds
[Epoch 10, Batch     1 / 387] Training Loss: 0.07101761549711227
Accuracy on train set: 0.9819409897065664
F1 score on train set: 0.9798617505284886
Accuracy on validation set: 0.9102863556391434
F1 score on validation set: 0.9006601362601963
[Epoch 10, Validation Loss: 0.310]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 9 hours, 43 minutes, 8 seconds
[Epoch 11, Batch     1 / 387] Training Loss: 0.055171433836221695
Accuracy on train set: 0.9852878723533337
F1 score on train set: 0.9839289915613529
Accuracy on validation set: 0.9110598798762362
F1 score on validation set: 0.9017908818741633
[Epoch 11, Validation Loss: 0.309]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 10 hours, 41 minutes, 4 seconds
[Epoch 12, Batch     1 / 387] Training Loss: 0.04062988981604576
Accuracy on train set: 0.9861675665837529
F1 score on train set: 0.9845301449374129
Accuracy on validation set: 0.9126220954923254
F1 score on validation set: 0.9034122569843714
[Epoch 12, Validation Loss: 0.305]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 11 hours, 38 minutes, 9 seconds
[Epoch 13, Batch     1 / 387] Training Loss: 0.03671315312385559
Accuracy on train set: 0.9867186394062569
F1 score on train set: 0.9846953816029687
Accuracy on validation set: 0.9135321240065523
F1 score on validation set: 0.9044388276788475
[Epoch 13, Validation Loss: 0.303]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 12 hours, 36 minutes, 37 seconds
[Epoch 14, Batch     1 / 387] Training Loss: 0.05769019201397896
Accuracy on train set: 0.9870169265303647
F1 score on train set: 0.9856151102837631
Accuracy on validation set: 0.9145634896560092
F1 score on validation set: 0.9056941852249099
[Epoch 14, Validation Loss: 0.300]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 13 hours, 34 minutes, 27 seconds
[Epoch 15, Batch     1 / 387] Training Loss: 0.03003191202878952
Accuracy on train set: 0.9877095593439705
F1 score on train set: 0.9862823786401523
Accuracy on validation set: 0.9160650367044834
```

```
F1 score on validation set: 0.9070744264850175
[Epoch 15, Validation Loss: 0.295]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 14 hours, 32 minutes, 20 seconds
[Epoch 16, Batch     1 / 387] Training Loss: 0.0344519279897213
Accuracy on train set: 0.9877045036300026
F1 score on train set: 0.9864185481686728
Accuracy on validation set: 0.9141691439665109
F1 score on validation set: 0.9057189878458429
[Epoch 16, Validation Loss: 0.303]
UPDATING LEARNING RATE TO:  0.0001
Elapsed time: 15 hours, 30 minutes, 6 seconds
Finished Training
```

Results

In [12]:
```python
image_path = "leonardo/colab/SWIN/plots/epcs_16__lr_0.01__bs_512_def2.png

image = PIL.Image.open(image_path)
plt.figure(figsize=(10, 10))
plt.imshow(image)
plt.axis('off')
plt.show()
```



5. Evaluating the model

This step involves the evaluation of the network's performance according to 'traditional' metrics (accuracy, F1 scores) and fairness metrics, which are specifically designed to assess the presence of bias.

First, we retrieve the predictions of the model on the test set:

In [ ]:
```python
# Debugging code removed for clarity
```

In [101…
```python
import pandas as pd
```

```
df = pd.read_csv("final_predictions1e-4_swin_def.csv")
labels_fairness = df['labels_fairness']
pred_fairness = df ['pred_fairness']
skin_labels_fairness = df['skin_labels_fairness']
```

Overall accuracy and F1 score

We compute the overall accuracy and F1 score:

In [127…
```
# Debugging code removed for clarity
```

```
Overall accuracy: 91.3%
Overall F1 score: 0.90
```

We evaluate the accuracy of the network for every skin tone.

In [21]:
```
# Debugging code removed for clarity
```

```
Accuracy for skin tone light: 90.3%
Accuracy for skin tone very light: 91.5%
Accuracy for skin tone tan: 91.9%
Accuracy for skin tone intermediate: 91.7%
Accuracy for skin tone brown: 91.2%
Accuracy for skin tone dark: 90.8%
```

We also evaluate the F1 score of the network for every skin tone.

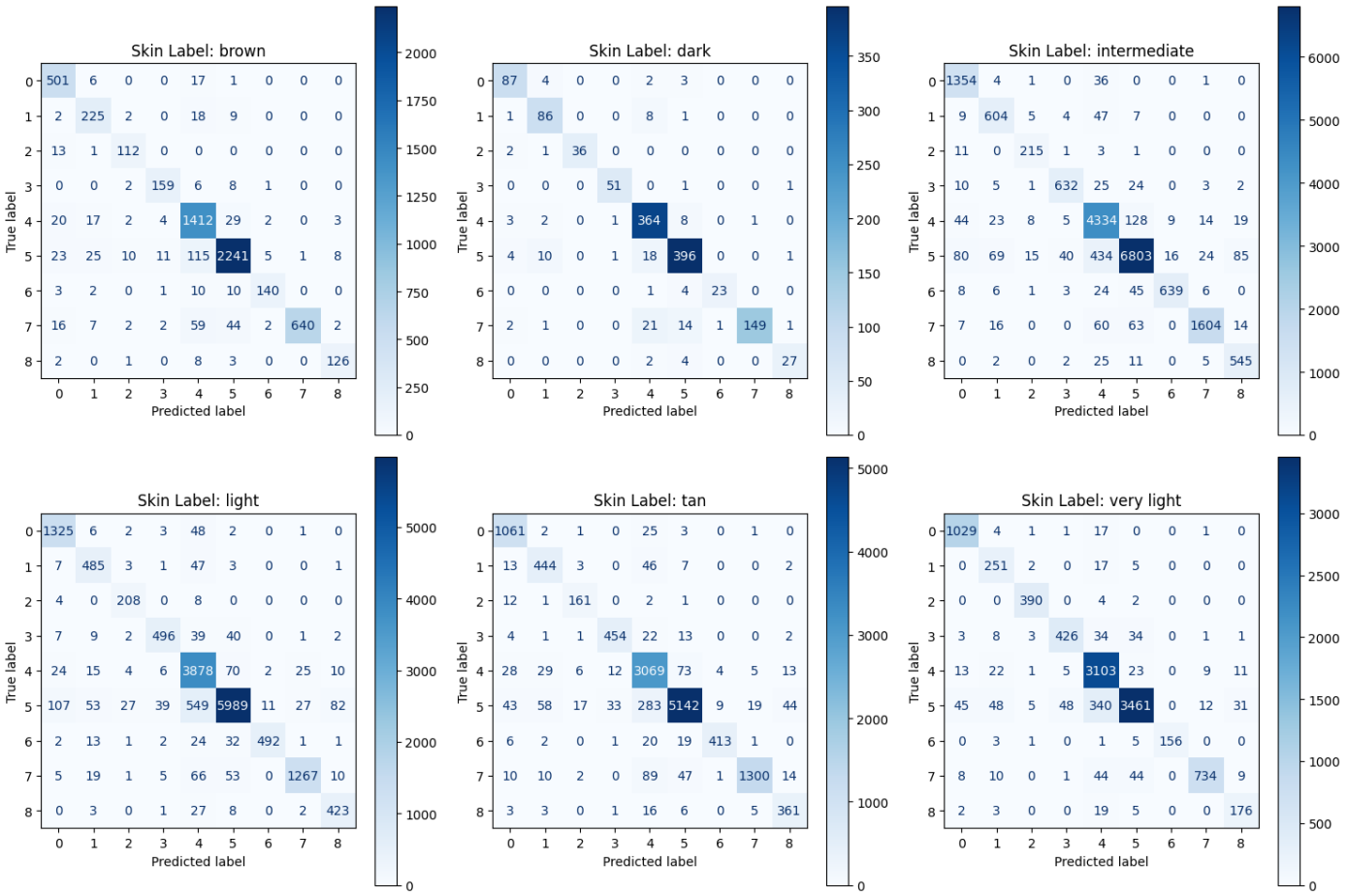
In [128…
```
# Debugging code removed for clarity
```

```
F1 score for skin tone brown: 0.91
F1 score for skin tone tan: 0.92
F1 score for skin tone light: 0.90
F1 score for skin tone very light: 0.92
F1 score for skin tone dark: 0.91
F1 score for skin tone intermediate: 0.92
```

Additionally, we show the overall confusion matrices for each skin tone.

In [129…
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Convert lists to numpy arrays for easier processing
labels_fairness = np.array(labels_fairness)
pred_fairness = np.array(pred_fairness)
skin_labels_fairness = np.array(skin_labels_fairness)

# Get unique skin labels
unique_skin_labels = np.unique(skin_labels_fairness)

# Create a figure with 2 rows and 3 columns
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.ravel()  # Flatten the axes array for easier iteration
```

```python
# Plot confusion matrices for each skin label
for i, skin_label in enumerate(unique_skin_labels):
    # Filter labels and predictions for the current skin label
    indices = np.where(skin_labels_fairness == skin_label)
    filtered_labels = labels_fairness[indices]
    filtered_predictions = pred_fairness[indices]

    # Compute confusion matrix
    cm = confusion_matrix(filtered_labels, filtered_predictions)

    # Create a ConfusionMatrixDisplay
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)

    # Plot the confusion matrix on the corresponding subplot
    disp.plot(ax=axes[i], cmap=plt.cm.Blues)
    axes[i].set_title(f"Skin Label: {skin_label}")

# Adjust layout
plt.tight_layout()
plt.show()
```



From now on, we will consider the following aggregation for the skin tones:

- "brown", "dark" ---> Minority
- all the other skin tones ---> Majority

We compute the overall Accuracy and F1 score values for the Minority and the Majority groups:

In [132…
```
# Debugging code removed for clarity
```

```
Overall Minority F1: 0.90,      Overall Majority F1: 0.90
Overall Minority Accuracy: 91.1%,      Overall Majority Accuracy: 91.3%
```

Second, we compute the Accuracy and the F1 score values for each skin disease, both for the Majority and Minority group.

In [117…
```
# Debugging code removed for clarity
```

```
Disease 0: Minority Accuracy: 94.69%, Majority Accuracy: 96.75%
Disease 1: Minority Accuracy: 88.35%, Majority Accuracy: 88.62%
Disease 2: Minority Accuracy: 89.70%, Majority Accuracy: 95.12%
Disease 3: Minority Accuracy: 91.70%, Majority Accuracy: 87.11%
Disease 4: Minority Accuracy: 95.07%, Majority Accuracy: 95.61%
Disease 5: Minority Accuracy: 91.91%, Majority Accuracy: 88.82%
Disease 6: Minority Accuracy: 84.02%, Majority Accuracy: 88.17%
Disease 7: Minority Accuracy: 81.93%, Majority Accuracy: 88.97%
Disease 8: Minority Accuracy: 88.44%, Majority Accuracy: 90.99%
```

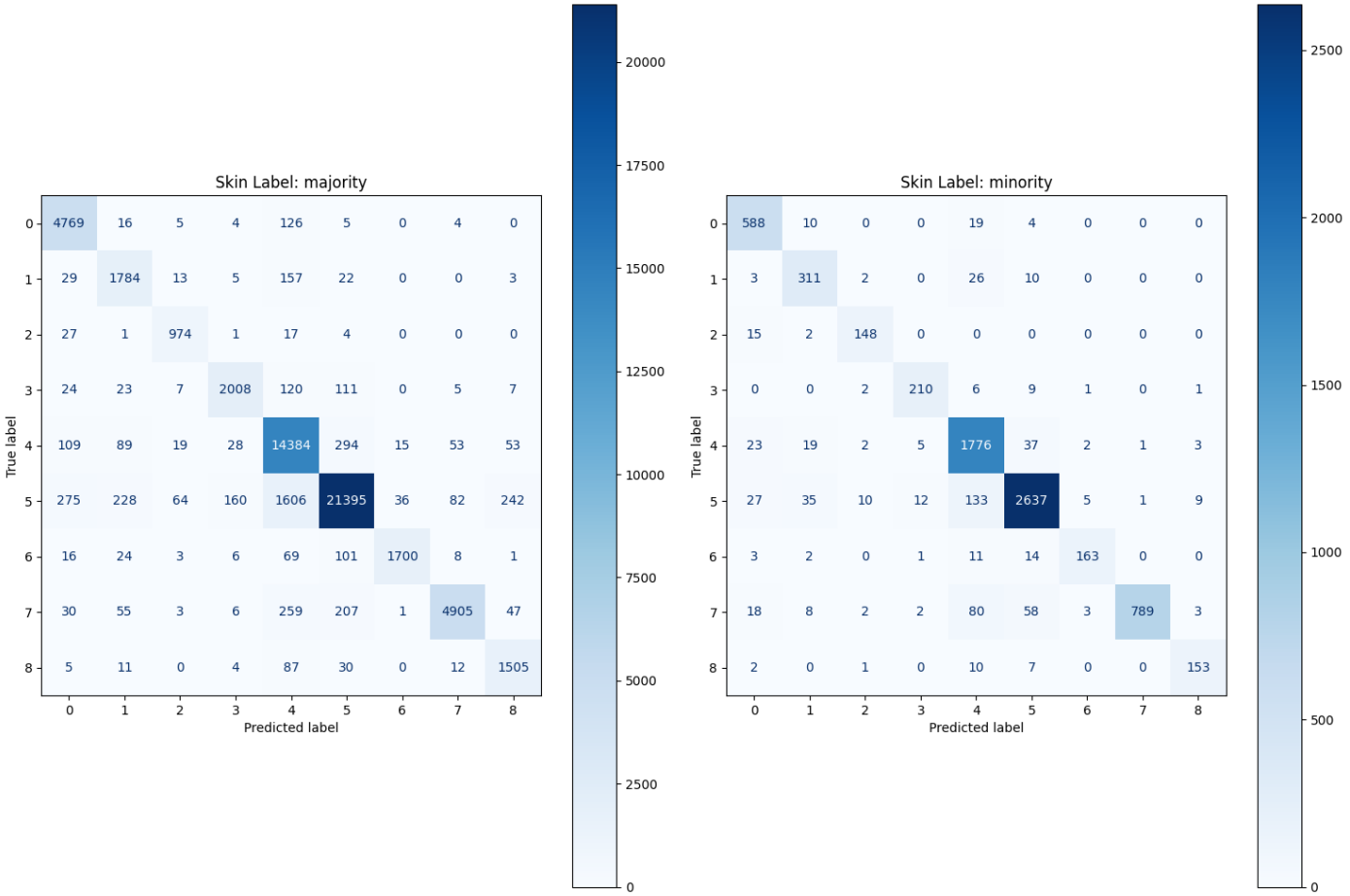
In [119…
```
# Debugging code removed for clarity
```

```
Disease 0: Minority F1: 0.90, Majority F1: 0.93
Disease 1: Minority F1: 0.84, Majority F1: 0.84
Disease 2: Minority F1: 0.89, Majority F1: 0.92
Disease 3: Minority F1: 0.92, Majority F1: 0.89
Disease 4: Minority F1: 0.90, Majority F1: 0.90
Disease 5: Minority F1: 0.93, Majority F1: 0.93
Disease 6: Minority F1: 0.89, Majority F1: 0.92
Disease 7: Minority F1: 0.90, Majority F1: 0.93
Disease 8: Minority F1: 0.89, Majority F1: 0.86
```

We also provide the confusion matrices for the Majority and Minority group.

In [34]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Convert lists to numpy arrays for easier processing
labels_fairness = np.array(labels_fairness)
pred_fairness = np.array(pred_fairness)
groups = np.array(['minority' if skin_labels_fairness[i] == 'dark' or ski

# Get unique skin labels
unique_skin_labels = np.unique(groups)

# Create a figure with 2 rows and 3 columns
fig, axes = plt.subplots(1, 2, figsize=(15, 10))
axes = axes.ravel()  # Flatten the axes array for easier iteration

# Plot confusion matrices for each skin label
for i, skin_label in enumerate(unique_skin_labels):
    # Filter labels and predictions for the current skin label
    indices = np.where(groups == skin_label)
    filtered_labels = labels_fairness[indices]
    filtered_predictions = pred_fairness[indices]
```

```
    # Compute confusion matrix
    cm = confusion_matrix(filtered_labels, filtered_predictions)

    # Create a ConfusionMatrixDisplay
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)

    # Plot the confusion matrix on the corresponding subplot
    disp.plot(ax=axes[i], cmap=plt.cm.Blues)
    axes[i].set_title(f"Skin Label: {skin_label}")

# Adjust layout
plt.tight_layout()
plt.show()
```



Disparate Impact

In the binary case, **disparate impact** is defined as

$$\frac{Pr(\hat{Y} = 1 | X \in minority\_group)}{Pr(\hat{Y} = 1 | X \in majority\_group)}$$

i.e. the ratio between the selection rate for the minority and majority group. In our case, we have a multiclass task at hand. We handle it by computing the Disparate Impact for each disease, considering $\hat{Y} = 1$ if the single disease is detected, $\hat{Y} = 0$ otherwise. Plus, we aggregate the skin tones considering the *brown* and *dark* skin tones as the **minority group** and the remaining four skin tones as the **majority group**. The disparate impact is computed using the implementation provided by the

library *holisticai*.

In [121…
```
# Debugging code removed for clarity
```

```
Disparate impact for disease esantema-iatrogeno-farmaco-indotta: 1.01

Disparate impact for disease esantema-maculo-papuloso: 1.36

Disparate impact for disease esantema-morbilliforme: 1.21

Disparate impact for disease esantema-polimorfo-like: 0.81

Disparate impact for disease esantema-virale: 0.96

Disparate impact for disease orticaria: 0.99

Disparate impact for disease pediculosi: 0.78

Disparate impact for disease scabbia: 1.23

Disparate impact for disease varicella: 0.72
```

Equalized Odds Ratio

A classifier satisfies Equalized Odds under a distribution over (X, A, Y) (where A indicates the sensitive feature) if its prediction $\hat{Y}$ is conditionally independent of the sensitive feature A given the label Y . This is equivalent to $E(\hat{Y}|A = a, Y = y) = E(\hat{Y}|Y = y)\ \forall\ a, y$. Equalized odds requires that the true positive rate, $Pr(\hat{Y} = 1|Y = 1)$, and the false positive rate, $Pr(\hat{Y} = 1|Y = 0)$, are equal across groups. In our case, Equalized Odds Ratio (EOR) was computed for every disease using the implementation of the library **fairlearn**, in which EOR is defined as 'the smaller of two metrics: *true positive rate ratio* and *false positive rate ratio*. The former is the ratio between the smallest and largest of $Pr(\hat{Y} = 1|A = a, Y = 1)$, across all values of the sensitive feature(s). The latter is defined similarly, but for $Pr(\hat{Y} = 1|A = a, Y = 0)$. The equalized odds ratio of 1 means that all groups have the same true positive, true negative, false positive, and false negative rates' [https://fairlearn.org/main/api_reference/generated/fairlearn.metrics.equalized_odds_rati
Even in this case, the skin tones were aggregated into the same two groups used to compute the Disparate Impact Ratio.

In [134…
```
# Debugging code removed for clarity
```

```
Equalized Odds Ratio for esantema-iatrogeno-farmaco-indotta: 0.72

Equalized Odds Ratio for esantema-maculo-papuloso: 0.74

Equalized Odds Ratio for esantema-morbilliforme: 0.76

Equalized Odds Ratio for esantema-polimorfo-like: 0.73

Equalized Odds Ratio for esantema-virale: 0.91

Equalized Odds Ratio for orticaria: 0.74

Equalized Odds Ratio for pediculosi: 0.61

Equalized Odds Ratio for scabbia: 0.10

Equalized Odds Ratio for varicella: 0.35
```

Predictive Rate Ratio

The **Predictive Rate Parity** is achieved if the **Positive Predictive Value (PPV)**, also named as **precision**, is the same across all groups. The formula for the precision is

$$PPV = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Positives\ (FP)}$$

The **Predictive Rate Ratio** compares the precision between the two groups. It is calculated by taking the ratio of the PPV of one group to the PPV of another group:

$$Predictive\ Rate\ Ratio = \frac{PPV_{minority\_group}}{PPV_{majority\_group}}$$

In our implementation, we compute this value for each disease, with the usual division of the skin tones into the minority group and majority group.

We can retrieve the Predictive Rate Ratio value by considering the precision matrix relative to each disease for the minority and majority group. The function for the precision matrix is taken from the library **holisticai**. Observing the matrix, we simply compute the ratio between the precision of the minority group and of the majority group.

In [123…

```python
groups = ['minority' if skin_labels_fairness[i] == 'dark' or skin_labels_
prec_matrix = precision_matrix(groups, pred_fairness, labels_fairness)
prec_matrix.head()
```

	0	1	2	3	4	5	6	7	
majority	0.967539	0.886239	0.951172	0.871150	0.956129	0.888202	0.881743	0.889715	0
minority	0.946860	0.883523	0.896970	0.917031	0.950749	0.919136	0.840206	0.819315	0.

In [135…
```
# Debugging code removed for clarity
```

```
Predictive Rate Ratio for disease esantema-iatrogeno-farmaco-indotta: 0.98

Predictive Rate Ratio for disease esantema-maculo-papuloso: 1.00

Predictive Rate Ratio for disease esantema-morbilliforme: 0.94

Predictive Rate Ratio for disease esantema-polimorfo-like: 1.05

Predictive Rate Ratio for disease esantema-virale: 0.99

Predictive Rate Ratio for disease orticaria: 1.03

Predictive Rate Ratio for disease pediculosi: 0.95

Predictive Rate Ratio for disease scabbia: 0.92

Predictive Rate Ratio for disease varicella: 0.97
```