In [1]:
```
from numpy import vstack
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confu
from tqdm import tqdm
from pathlib import Path
import pandas as pd
from PIL import Image
import numpy as np
from torch import long, tensor
from torch.utils.data.dataset import Dataset
from torchvision.transforms import Compose, Resize, ToTensor, Normalize
import torch
import torch.nn.init as init
import torch.nn as nn
from torch import Tensor

from torch.nn import (Conv2d, CrossEntropyLoss, Linear, MaxPool2d, ReLU,Sequential
from pathlib import Path
from typing import Dict, List, Union
import pandas as pd
import torch
import torch.nn.init as init
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from torch import Tensor
from torch.nn import (Conv2d, CrossEntropyLoss, Linear, MaxPool2d, ReLU,Sequential
from torch.optim import Adam
from torch.optim.optimizer import Optimizer
from torch.utils.data import DataLoader
import itertools
import matplotlib.pyplot as plt
```

In [2]:
```
dirPath = Path('C:/Users/myste/git/COMP6721_Applied_AI__Project_Team_AMW_Part2')
datasetPath  = Path(dirPath/'dataset')
noMaskPath  = datasetPath/'NoMask'
N95MaskPath = datasetPath/'N95'
clothMaskPath  = datasetPath/'Cloth'
surgicalMaskPath = datasetPath/'Surgical'
N95ValvePath = datasetPath/'N95Valve'
maskDF = pd.DataFrame()

for imagepath in tqdm(list(noMaskPath .iterdir()), desc='no'):
    maskDF = maskDF.append({
        'image': str(imagepath),
        'mask': 0
    }, ignore_index=True)

for imagepath in tqdm(list(clothMaskPath .iterdir()), desc='cloth'):
    maskDF = maskDF.append({
        'image': str(imagepath),
        'mask': 1
    }, ignore_index=True)

for imagepath in tqdm(list(N95MaskPath.iterdir()), desc='N95'):
    maskDF = maskDF.append({
        'image': str(imagepath),
        'mask': 2
    }, ignore_index=True)

for imagepath in tqdm(list(surgicalMaskPath.iterdir()), desc='surgical'):
    maskDF = maskDF.append({
        'image': str(imagepath),
        'mask': 3
    }, ignore_index=True)


for imagepath in tqdm(list(N95ValvePath.iterdir()), desc='valve'):
    maskDF = maskDF.append({
        'image': str(imagepath),
        'mask': 4
    }, ignore_index=True)


print("Total no. of images:",len(maskDF))
data_frame = datasetPath/'dataset.pickle'
print(f'DataFrame saved successfully: {data_frame}')
maskDF.to_pickle(data_frame)
```

```
 The frame.append method is deprecated and will be removed from pandas in a future
version. Use pandas.concat instead.
  maskDF = maskDF.append({
C:\Users\myste\AppData\Local\Temp\ipykernel_22452\741201046.py:36: FutureWarning:
 The frame.append method is deprecated and will be removed from pandas in a future
version. Use pandas.concat instead.
  maskDF = maskDF.append({
C:\Users\myste\AppData\Local\Temp\ipykernel_22452\741201046.py:36: FutureWarning:
 The frame.append method is deprecated and will be removed from pandas in a future
version. Use pandas.concat instead.
  maskDF = maskDF.append({
valve: 100%|█████████████████████████████████████████████████████████████████████
███ | 400/402 [00:07<00:00, 57.04it/s]C:\Users\myste\AppData\Local\Temp\ipykernel_
22452\741201046.py:36: FutureWarning: The frame.append method is deprecated and wi
ll be removed from pandas in a future version. Use pandas.concat instead.
  maskDF = maskDF.append({
C:\Users\myste\AppData\Local\Temp\ipykernel_22452\741201046.py:36: FutureWarning:
 The frame.append method is deprecated and will be removed from pandas in a future
version. Use pandas.concat instead.
  maskDF = maskDF.append({
valve: 100%|█████████████████████████████████████████████████████████████████████
███| 402/402 [00:07<00:00, 56.19it/s]
Total no. of images: 2069
DataFrame saved successfully: C:\Users\myste\git\COMP6721_Applied_AI__Project_Team
_AMW_Part2\dataset\dataset.pickle
```

In [3]:

```python
class mask_dataset(Dataset):
    def __init__(self, dataFrame):
        self.dataFrame = dataFrame
        self.transformations = Compose([
            Resize((32, 32)),
            ToTensor(),
            Normalize((0.5667, 0.5198, 0.4955),(0.3082, 0.2988, 0.3053))
        ])

    def __getitem__(self, key):
        if isinstance(key, slice):
            raise NotImplementedError('Supporting Slice')

        row = self.dataFrame.iloc[key]
        image = Image.open(row['image']).convert('RGB')
        return {
          'image': self.transformations(image),
          'mask': tensor([row['mask']], dtype=long),
          'path': row['image']
        }

    def __len__(self):
        return len(self.dataFrame.index)
```

In [4]:

```python

class face_mask_detection_CNN(nn.Module):
    def __init__(self):
        super(face_mask_detection_CNN, self).__init__()
        self.conv_layer = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(8 * 8 * 64, 1000),
            nn.ReLU(inplace=True),
            nn.Linear(1000, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.1),
            nn.Linear(512, 5)
        )
    def forward(self, x):
        x = self.conv_layer(x)
        x = x.view(x.size(0), -1)
        x = self.fc_layer(x)
        return x

face_mask_detector_cnn = face_mask_detection_CNN()
print(face_mask_detector_cnn)
```

```
face_mask_detection_CNN(
  (conv_layer): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (2): LeakyReLU(negative_slope=0.01, inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (5): LeakyReLU(negative_slope=0.01, inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (9): LeakyReLU(negative_slope=0.01, inplace=True)
    (10): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stat
s=True)
    (12): LeakyReLU(negative_slope=0.01, inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
  )
  (fc_layer): Sequential(
    (0): Dropout(p=0.1, inplace=False)
    (1): Linear(in_features=4096, out_features=1000, bias=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=1000, out_features=512, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.1, inplace=False)
    (6): Linear(in_features=512, out_features=5, bias=True)
  )
)
```

In [5]:
```
def conf_mat(cm, classes, normalize=False, title='Visualization of the confusion ma
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print('Normalized confusion matrix')
    else:
        print('Confusion matrix without normalization')

    print(cm)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j], horizontalalignment="center", color="white" if cm

    plt.tight_layout()
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')

def prepare_data(mask_df_path) -> None:
        mask_df = pd.read_pickle(mask_df_path)
        print(mask_df['mask'].value_counts())
        skf = StratifiedKFold(n_splits=10, shuffle=True)
        train_folds = []
        val_fold = []
        for train_index, validate_index in skf.split(mask_df, mask_df['mask']):
            train_folds.append(mask_dataset(mask_df.iloc[train_index]))
            val_fold.append(mask_dataset(mask_df.iloc[validate_index]))
        return [ train_folds, val_fold,CrossEntropyLoss() ]

def train_dataloader(train_df) -> DataLoader:
    return DataLoader(train_df, batch_size=32, shuffle=True, num_workers=0)

def val_dataloader(validate_df) -> DataLoader:
    return DataLoader(validate_df, batch_size=32, num_workers=0)

train_dfs, validate_dfs, cross_entropy_loss = prepare_data(data_frame)
```

```
1    435
3    420
0    410
2    402
4    402
Name: mask, dtype: int64
```

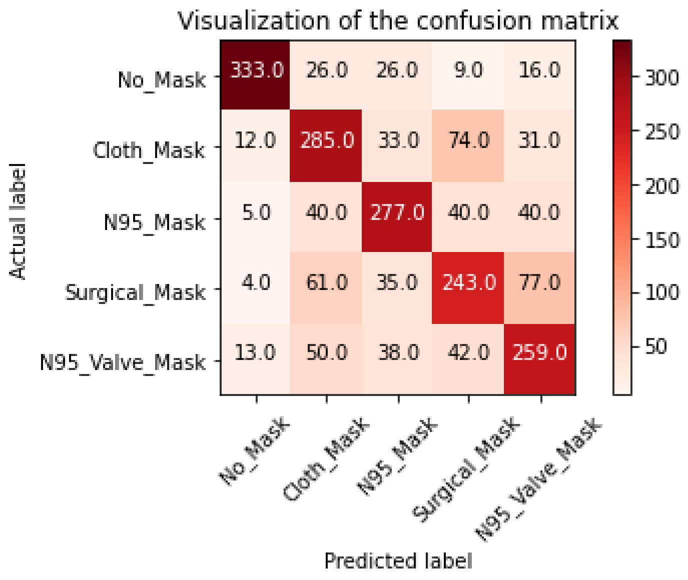
In [6]:

```
epochs = 10
lr = 0.001
retrain = False

import warnings
warnings.filterwarnings('ignore')

def training(train_fold):
    acc_list = []
    loss_list = []
    optimizer = Adam(face_mask_detector_cnn.parameters(), lr=lr)
    for epoch in range(epochs):
        total=0
        correct=0
        loss_train = 0.0
        for i, data in enumerate(train_dataloader(train_fold), 0):
            inputs, labels = data['image'], data['mask']
            labels = labels.flatten()
            outputs = face_mask_detector_cnn(inputs)
            loss = cross_entropy_loss(outputs, labels)
            loss_list.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            #training accuracy
            total += labels.size(0)
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            loss_train += loss
        print('Training Loss after epoch {} : {} Accuracy: {:.2f}%'.format(epoch, 
```

In [7]:
```
def evaluation(valid_f):
    predictions, actuals = torch.tensor([]), torch.tensor([])
    for i, data in enumerate(val_dataloader(valid_f)):
        inputs, targets = data['image'], data['mask']
        targets = targets.flatten()
        output = face_mask_detector_cnn(inputs)
        output = torch.argmax(output,axis=1)
        predictions = torch.cat((predictions, output.flatten()), dim=0)
        actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(
```

In [8]:
```
fold_results = []
fold_confusion_matrix = np.zeros((5,5))
classes = ['No_Mask', 'Cloth_Mask', 'N95_Mask', 'Surgical_Mask', 'N95_Valve_Mask']

for index in range(len(train_dfs)):
    print("Running Fold : "+ str(index+1))
    training(train_dfs[index])
    fold_result = evaluation(validate_dfs[index])
    fold_results.append(fold_result[1:-1])
    fold_confusion_matrix = np.add(fold_confusion_matrix,fold_result[0])
    if index != len(train_dfs)-1:
        face_mask_detector_cnn = face_mask_detection_CNN()
#printing the metrics (accuracy,precision,recall,f-scores )and confusion matrix
metrics_df = pd.DataFrame(fold_results, columns=['accuracy', 'precision', 'recall',
print()
print("Metrics")
print(metrics_df.mean())
print()
print("Across 10-folds")
conf_mat(fold_confusion_matrix, classes)
```

```
Running Fold : 1
Training Loss after epoch 0 : 66.5443115234375 Accuracy: 54.67%
Training Loss after epoch 1 : 46.17664337158203 Accuracy: 70.89%
Training Loss after epoch 2 : 39.4726448059082 Accuracy: 74.65%
Training Loss after epoch 3 : 32.870635986328125 Accuracy: 79.16%
Training Loss after epoch 4 : 29.61846351623535 Accuracy: 81.42%
Training Loss after epoch 5 : 25.086631774902344 Accuracy: 84.85%
Training Loss after epoch 6 : 24.429929733276367 Accuracy: 84.53%
Training Loss after epoch 7 : 22.07504653930664 Accuracy: 86.47%
Training Loss after epoch 8 : 17.488203048706055 Accuracy: 88.45%
Training Loss after epoch 9 : 13.252532958984375 Accuracy: 91.78%
Running Fold : 2
Training Loss after epoch 0 : 68.46067810058594 Accuracy: 53.65%
Training Loss after epoch 1 : 44.78142547607422 Accuracy: 71.16%
Training Loss after epoch 2 : 37.44734573364258 Accuracy: 76.75%
Training Loss after epoch 3 : 32.830039978027344 Accuracy: 80.77%
Training Loss after epoch 4 : 27.140718460083008 Accuracy: 82.33%
Training Loss after epoch 5 : 25.02231788635254 Accuracy: 85.02%
Training Loss after epoch 6 : 20.483715057373047 Accuracy: 87.59%
Training Loss after epoch 7 : 17.317785263061523 Accuracy: 88.88%
Training Loss after epoch 8 : 13.167792320251465 Accuracy: 91.78%
Training Loss after epoch 9 : 13.725604057312012 Accuracy: 92.00%
Running Fold : 3
Training Loss after epoch 0 : 71.66500091552734 Accuracy: 51.50%
Training Loss after epoch 1 : 45.31386947631836 Accuracy: 70.46%
Training Loss after epoch 2 : 34.961971282958984 Accuracy: 78.20%
Training Loss after epoch 3 : 30.57979393005371 Accuracy: 80.67%
Training Loss after epoch 4 : 27.582752227783203 Accuracy: 83.46%
Training Loss after epoch 5 : 26.1671142578125 Accuracy: 84.37%
Training Loss after epoch 6 : 20.6773738861084 Accuracy: 87.92%
Training Loss after epoch 7 : 20.407333374023438 Accuracy: 87.97%
Training Loss after epoch 8 : 11.678122520446777 Accuracy: 92.64%
Training Loss after epoch 9 : 13.500534057617188 Accuracy: 92.91%
Running Fold : 4
Training Loss after epoch 0 : 71.44699096679688 Accuracy: 49.41%
Training Loss after epoch 1 : 47.62403869628906 Accuracy: 68.96%
Training Loss after epoch 2 : 38.9819450378418 Accuracy: 76.26%
Training Loss after epoch 3 : 33.34145736694336 Accuracy: 78.63%
Training Loss after epoch 4 : 31.290252685546875 Accuracy: 81.20%
Training Loss after epoch 5 : 24.661666870117188 Accuracy: 84.48%
Training Loss after epoch 6 : 21.16436767578125 Accuracy: 86.95%
Training Loss after epoch 7 : 20.65632438659668 Accuracy: 87.86%
Training Loss after epoch 8 : 16.574350357055664 Accuracy: 89.15%
Training Loss after epoch 9 : 14.917314529418945 Accuracy: 91.89%
Running Fold : 5
Training Loss after epoch 0 : 69.76063537597656 Accuracy: 51.56%
Training Loss after epoch 1 : 46.42950439453125 Accuracy: 71.70%
Training Loss after epoch 2 : 38.811405181884766 Accuracy: 76.21%
Training Loss after epoch 3 : 33.434837341308594 Accuracy: 79.48%
Training Loss after epoch 4 : 28.527923583984375 Accuracy: 82.65%
Training Loss after epoch 5 : 27.363365173339844 Accuracy: 83.08%
Training Loss after epoch 6 : 22.185340881347656 Accuracy: 87.33%
Training Loss after epoch 7 : 20.704879760742188 Accuracy: 87.86%
Training Loss after epoch 8 : 16.27195167541504 Accuracy: 90.55%
Training Loss after epoch 9 : 12.40699291229248 Accuracy: 92.27%
Running Fold : 6
Training Loss after epoch 0 : 70.96258544921875 Accuracy: 51.29%
Training Loss after epoch 1 : 47.77632141113281 Accuracy: 69.50%
Training Loss after epoch 2 : 38.844417572021484 Accuracy: 75.19%
Training Loss after epoch 3 : 34.533355712890625 Accuracy: 78.63%
Training Loss after epoch 4 : 30.267148971557617 Accuracy: 81.31%
Training Loss after epoch 5 : 25.597944259643555 Accuracy: 84.00%
Training Loss after epoch 6 : 23.386905670166016 Accuracy: 85.34%
Training Loss after epoch 7 : 21.701906204223633 Accuracy: 87.38%
Training Loss after epoch 8 : 19.483617782592773 Accuracy: 87.38%
Training Loss after epoch 9 : 13.855963706970215 Accuracy: 91.19%
Running Fold : 7
Training Loss after epoch 0 : 77.05162811279297 Accuracy: 46.46%
Training Loss after epoch 1 : 47.88816833496094 Accuracy: 68.74%
Training Loss after epoch 2 : 38.94337463378906 Accuracy: 74.81%
Training Loss after epoch 3 : 31.712453842163086 Accuracy: 79.91%
Training Loss after epoch 4 : 26.60662078857422 Accuracy: 84.00%
Training Loss after epoch 5 : 22.646039962768555 Accuracy: 85.34%
Training Loss after epoch 6 : 21.321670532226562 Accuracy: 86.95%
Training Loss after epoch 7 : 18.926124572753906 Accuracy: 88.40%
Training Loss after epoch 8 : 15.702239036560059 Accuracy: 91.08%
Training Loss after epoch 9 : 12.532111167907715 Accuracy: 92.37%
Running Fold : 8
Training Loss after epoch 0 : 76.22260284423828 Accuracy: 48.87%
Training Loss after epoch 1 : 49.802616119384766 Accuracy: 66.17%
Training Loss after epoch 2 : 40.4404182434082 Accuracy: 74.87%
Training Loss after epoch 3 : 33.64297866821289 Accuracy: 78.84%
Training Loss after epoch 4 : 31.69550132751465 Accuracy: 78.95%
Training Loss after epoch 5 : 25.973628997802734 Accuracy: 83.62%
Training Loss after epoch 6 : 21.731613159179688 Accuracy: 86.41%
Training Loss after epoch 7 : 18.117189407348633 Accuracy: 89.26%
Training Loss after epoch 8 : 14.864117622375488 Accuracy: 90.71%
Training Loss after epoch 9 : 14.273297309875488 Accuracy: 92.16%
Running Fold : 9
Training Loss after epoch 0 : 74.76465606689453 Accuracy: 49.73%
Training Loss after epoch 1 : 48.17966079711914 Accuracy: 69.39%
Training Loss after epoch 2 : 39.74349594116211 Accuracy: 75.62%
Training Loss after epoch 3 : 33.02470397949219 Accuracy: 79.91%
Training Loss after epoch 4 : 30.035621643066406 Accuracy: 81.20%
Training Loss after epoch 5 : 28.00824737548828 Accuracy: 82.44%
Training Loss after epoch 6 : 21.065139770507812 Accuracy: 86.90%
Training Loss after epoch 7 : 19.596773147583008 Accuracy: 87.38%
Training Loss after epoch 8 : 16.2105770111084 Accuracy: 90.23%
Training Loss after epoch 9 : 13.861757278442383 Accuracy: 92.00%
Running Fold : 10
Training Loss after epoch 0 : 72.8636474609375 Accuracy: 51.53%
Training Loss after epoch 1 : 46.25029754638672 Accuracy: 70.91%
Training Loss after epoch 2 : 37.85434341430664 Accuracy: 76.70%
Training Loss after epoch 3 : 32.12782669067383 Accuracy: 80.25%
Training Loss after epoch 4 : 28.50701332092285 Accuracy: 82.77%
Training Loss after epoch 5 : 28.276500701904297 Accuracy: 83.15%
Training Loss after epoch 6 : 21.764495849609375 Accuracy: 86.63%
Training Loss after epoch 7 : 18.301864624023438 Accuracy: 88.94%
Training Loss after epoch 8 : 18.618328094482422 Accuracy: 88.89%
Training Loss after epoch 9 : 14.814053535461426 Accuracy: 91.68%

Metrics
accuracy     0.675191
precision    0.693819
recall       0.675936
f-score      0.675427
dtype: float64

Across 10-folds
Confusion matrix without normalization
[[333.  26.  26.   9.  16.]
 [ 12. 285.  33.  74.  31.]
 [  5.  40. 277.  40.  40.]
 [  4.  61.  35. 243.  77.]
 [ 13.  50.  38.  42. 259.]]
```



In [9]:
```
torch.save(face_mask_detector_cnn, dirPath/'face_mask_detection_CNN.pt')
```

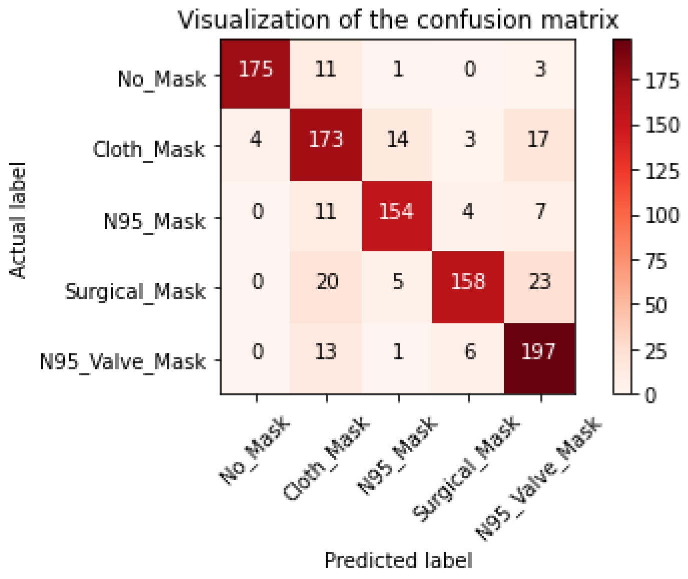
In [10]:
```
#to predict new images
from tqdm import tqdm
import matplotlib.pyplot as plt
import random

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    testDatasetPath = dirPath/'testDataset' #path to new test data. (not used in tr
    testRandomPath = testDatasetPath/'random'
    testDF = pd.DataFrame()

    for imagepath in tqdm(list(testRandomPath .iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 0
        }, ignore_index=True)

    for imagepath in tqdm(list(testRandomPath .iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 1
        }, ignore_index=True)

    for imagepath in tqdm(list(testRandomPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 2
        }, ignore_index=True)

    for imagepath in tqdm(list(testRandomPath.iterdir()), desc='surgical'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 3
        }, ignore_index=True)
    for imagepath in tqdm(list(testRandomPath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    test_df = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_df, num_samples=32, replacem
    data = iter(DataLoader(test_df, batch_size=32, num_workers=0, sampler=rand_samp
    inputs,targets = data['image'], data['mask']
    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    rand_ind = random.choice(list(range(0,32)))
    print(data['path'][rand_ind])
    img = Image.open(data['path'][rand_ind])
    plt.imshow(np.asarray(img))
    print("Predicted: ",class_mapping[output[rand_ind].tolist()])

predict()
```

```
no: 100%|██████████████████████████████████████████████████████████████████████
███| 104/104 [00:02<00:00, 43.11it/s]
cloth: 100%|███████████████████████████████████████████████████████████████████
███| 104/104 [00:02<00:00, 45.30it/s]
N95: 100%|█████████████████████████████████████████████████████████████████████
███| 104/104 [00:02<00:00, 45.47it/s]
surgical: 100%|████████████████████████████████████████████████████████████████
███| 104/104 [00:02<00:00, 44.59it/s]
valve: 100%|███████████████████████████████████████████████████████████████████
███| 104/104 [00:02<00:00, 45.11it/s]
C:\Users\myste\git\COMP6721_Applied_AI__Project_Team_AMW_Part2\testDataset\random
\test_DataSet (48).jpg
Predicted:  N95_Valve_Mask
```

In [11]:

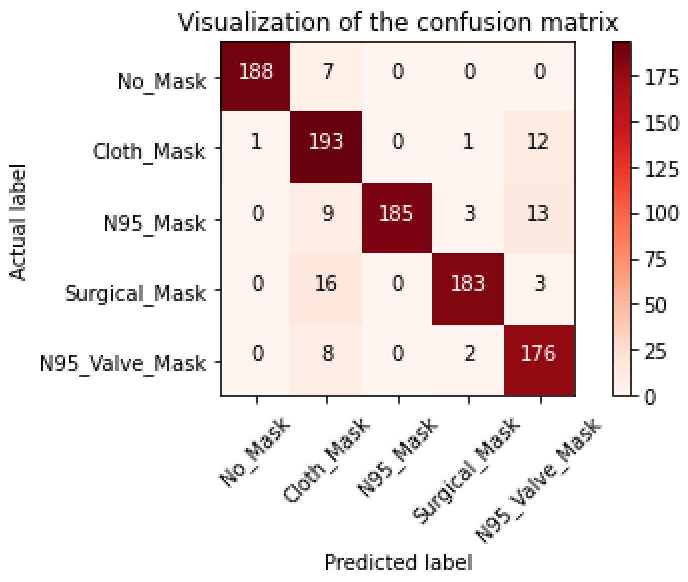
```
#GenderGroup - Male
from tqdm import tqdm
import matplotlib.pyplot as plt
import random
from torch import long, tensor
import torch

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    datasetPath = dirPath/'testDataset/GenderGroup' #path to new test data. (not us
    malePath = datasetPath/'Male'

    noMaleMaskPath  = malePath/'NoMask'
    N95MaleMaskPath = malePath/'N95'
    clothMaleMaskPath  = malePath/'Cloth'
    surgicalMaleMaskPath = malePath/'Surgical'
    N95MaleValvePath = malePath/'N95Valve'

    testDF = pd.DataFrame()

    for imgPath in tqdm(list(noMaleMaskPath.iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 0
        }, ignore_index=True)

    for imgPath in tqdm(list(clothMaleMaskPath.iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 1
        }, ignore_index=True)

    for imgPath in tqdm(list(N95MaleMaskPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 2
        }, ignore_index=True)

    for imgPath in tqdm(list(surgicalMaleMaskPath.iterdir()), desc='surgical'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 3
        }, ignore_index=True)

    for imagepath in tqdm(list(N95MaleValvePath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    predictions, actuals = torch.tensor([]), torch.tensor([])
    test_dfTest = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_dfTest, num_samples=1000, r
    data = iter(DataLoader(test_dfTest, batch_size=1000, num_workers=0, sampler=ra
    inputs,targets = data['image'], data['mask']
    targets = targets.flatten()


    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    predictions = torch.cat((predictions, output.flatten()), dim=0)
    actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(
fold_result_test = predict()
fold_confusion_matrix_test = fold_result_test[0]
fold_result_test_metrics = fold_result_test[1:-1]
conf_mat(fold_confusion_matrix_test, classes)

print("Fold results for males: ",fold_result_test_metrics)
```

```
no: 100%|██████████████████████████████████████████████████
███| 157/157 [00:03<00:00, 44.15it/s]
cloth: 100%|███████████████████████████████████████████████
███| 170/170 [00:03<00:00, 45.34it/s]
N95: 100%|█████████████████████████████████████████████████
███| 151/151 [00:03<00:00, 39.12it/s]
surgical: 100%|████████████████████████████████████████████
███| 151/151 [00:03<00:00, 45.04it/s]
valve: 100%|███████████████████████████████████████████████
███| 162/162 [00:03<00:00, 44.76it/s]
Confusion matrix without normalization
[[175  11   1   0   3]
 [  4 173  14   3  17]
 [  0  11 154   4   7]
 [  0  20   5 158  23]
 [  0  13   1   6 197]]
Fold results for males:  (0.857, 0.8675946038998082, 0.8581564474987428, 0.8602982
95517733)
```



In [12]:
```
#GenderGroup - Female
from tqdm import tqdm
import matplotlib.pyplot as plt
import random

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    datasetPath = dirPath/'testDataset/GenderGroup' #path to new test data. (not u
    femalePath = datasetPath/'Female'

    noFemaleMaskPath  = femalePath/'NoMask'
    N95FemaleMaskPath = femalePath/'N95'
    clothFemaleMaskPath  = femalePath/'Cloth'
    surgicalFemaleMaskPath = femalePath/'Surgical'
    N95FemaleValvePath = femalePath/'N95Valve'

    testDF = pd.DataFrame()

    for imgPath in tqdm(list(noFemaleMaskPath.iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 0
        }, ignore_index=True)

    for imgPath in tqdm(list(clothFemaleMaskPath.iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 1
        }, ignore_index=True)

    for imgPath in tqdm(list(N95FemaleMaskPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 2
        }, ignore_index=True)

    for imgPath in tqdm(list(surgicalFemaleMaskPath.iterdir()), desc='surgical'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 3
        }, ignore_index=True)

    for imagepath in tqdm(list(N95FemaleValvePath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imagepath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    predictions, actuals = torch.tensor([]), torch.tensor([])
    test_dfTest = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_dfTest, num_samples=1000, r
    data = iter(DataLoader(test_dfTest, batch_size=1000, num_workers=0, sampler=ra
    inputs,targets = data['image'], data['mask']
    targets = targets.flatten()


    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    predictions = torch.cat((predictions, output.flatten()), dim=0)
    actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(

fold_result_test = predict()
fold_confusion_matrix_test = fold_result_test[0]
fold_result_test_metrics = fold_result_test[1:-1]
conf_mat(fold_confusion_matrix_test, classes)

print("Fold results for females: ",fold_result_test_metrics)
```

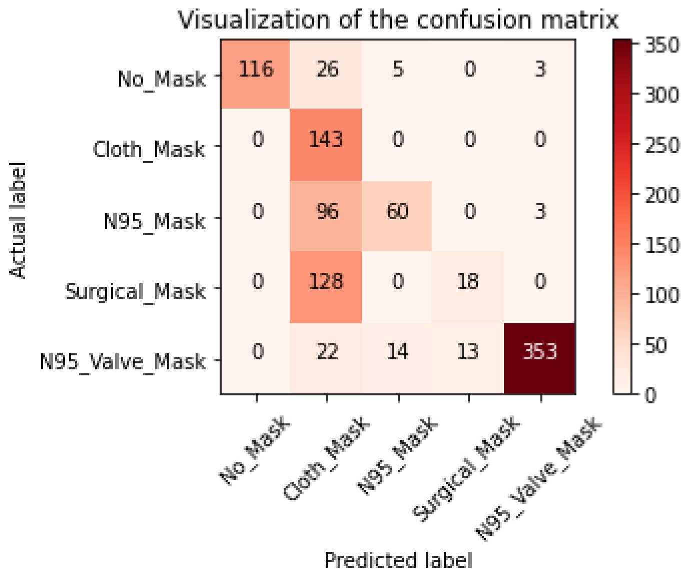
```
no: 100%|████████████████████████████████████████████████████████████
███| 161/161 [00:03<00:00, 43.16it/s]
cloth: 100%|█████████████████████████████████████████████████████████
███| 165/165 [00:03<00:00, 45.08it/s]
N95: 100%|███████████████████████████████████████████████████████████
███| 172/172 [00:03<00:00, 45.36it/s]
surgical: 100%|██████████████████████████████████████████████████████
███| 172/172 [00:03<00:00, 45.23it/s]
valve: 100%|█████████████████████████████████████████████████████████
███| 157/157 [00:03<00:00, 44.61it/s]
Confusion matrix without normalization
[[188   7   0   0   0]
 [  1 193   0   1  12]
 [  0   9 185   3  13]
 [  0  16   0 183   3]
 [  0   8   0   2 176]]
Fold results for females:  (0.925, 0.9308068482519378, 0.9259198496025182, 0.92635
47476674319)
```



In [13]:
```
#AgeGroup - GenZ
from tqdm import tqdm
import matplotlib.pyplot as plt
import random

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    datasetPath = dirPath/'testDataset/AgeGroup' #path to new test data. (not used
    genZPath = datasetPath/'GenZ'

    genZNoMaskPath = genZPath/'NoMask'
    genZClothMaskPath = genZPath/'Cloth'
    genZN95MaskPath = genZPath/'N95'
    genZSurgicalMaskPath = genZPath/'Surgical'
    genZValveMaskPath = genZPath/'N95Valve'

    testDF = pd.DataFrame()

    for imgPath in tqdm(list(genZNoMaskPath.iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 0
        }, ignore_index=True)

    for imgPath in tqdm(list(genZClothMaskPath.iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 1
        }, ignore_index=True)

    for imgPath in tqdm(list(genZN95MaskPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 2
        }, ignore_index=True)

    for imgPath in tqdm(list(genZSurgicalMaskPath.iterdir()), desc='surgical'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 3
        }, ignore_index=True)

    for imgPath in tqdm(list(genZValveMaskPath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    predictions, actuals = torch.tensor([]), torch.tensor([])
    test_dfTest = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_dfTest, num_samples=1000, re
    data = iter(DataLoader(test_dfTest, batch_size=1000, num_workers=0, sampler=ran
    inputs,targets = data['image'], data['mask']
    targets = targets.flatten()


    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    predictions = torch.cat((predictions, output.flatten()), dim=0)
    actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(a

fold_result_test = predict()
fold_confusion_matrix_test = fold_result_test[0]
fold_result_test_metrics = fold_result_test[1:-1]
conf_mat(fold_confusion_matrix_test, classes)

print("Fold results for GenZ Group: ",fold_result_test_metrics)
```

```
no: 100%|██████████████████████████████████████████████
█████| 89/89 [00:02<00:00, 40.95it/s]
cloth: 100%|███████████████████████████████████████████
█████| 92/92 [00:02<00:00, 45.20it/s]
N95: 100%|█████████████████████████████████████████████
█████| 97/97 [00:02<00:00, 45.05it/s]
surgical: 100%|████████████████████████████████████████
█████| 82/82 [00:01<00:00, 44.15it/s]
valve: 100%|███████████████████████████████████████████
███| 236/236 [00:05<00:00, 44.69it/s]
```

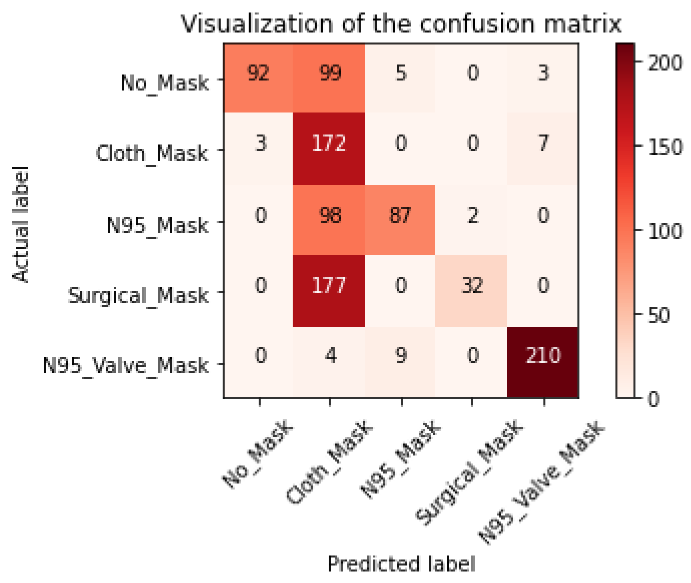
```
Confusion matrix without normalization
[[116  26   5   0   3]
 [  0 143   0   0   0]
 [  0  96  60   0   3]
 [  0 128   0  18   0]
 [  0  22  14  13 353]]
Fold results for GenZ Group:  (0.69, 0.7336008107014809, 0.6304177895737133, 0.604
0086881206033)
```



In [14]:
```python
#AgeGroup - Millennial
from tqdm import tqdm
import matplotlib.pyplot as plt
import random

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    datasetPath = dirPath/'testDataset/AgeGroup' #path to new test data. (not used
    millennialPath = datasetPath/'Millennial'

    millennialNoMaskPath = millennialPath/'NoMask'
    millennialClothMaskPath = millennialPath/'Cloth'
    millennialN95MaskPath = millennialPath/'N95'
    millennialSurgicalMaskPath = millennialPath/'Surgical'
    millennialValveMaskPath = millennialPath/'N95Valve'

    testDF = pd.DataFrame()

    for imgPath in tqdm(list(millennialNoMaskPath.iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 0
        }, ignore_index=True)

    for imgPath in tqdm(list(millennialClothMaskPath.iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 1
        }, ignore_index=True)
    for imgPath in tqdm(list(millennialN95MaskPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 2
        }, ignore_index=True)

    for imgPath in tqdm(list(millennialSurgicalMaskPath.iterdir()), desc='surgical
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 3
        }, ignore_index=True)

    for imgPath in tqdm(list(millennialValveMaskPath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    predictions, actuals = torch.tensor([]), torch.tensor([])
    test_dfTest = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_dfTest, num_samples=1000, r
    data = iter(DataLoader(test_dfTest, batch_size=1000, num_workers=0, sampler=ra
    inputs,targets = data['image'], data['mask']
    targets = targets.flatten()


    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    predictions = torch.cat((predictions, output.flatten()), dim=0)
    actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(a

fold_result_test = predict()
fold_confusion_matrix_test = fold_result_test[0]
fold_result_test_metrics = fold_result_test[1:-1]
conf_mat(fold_confusion_matrix_test, classes)

print("Fold results for Millennial Group: ",fold_result_test_metrics)
```

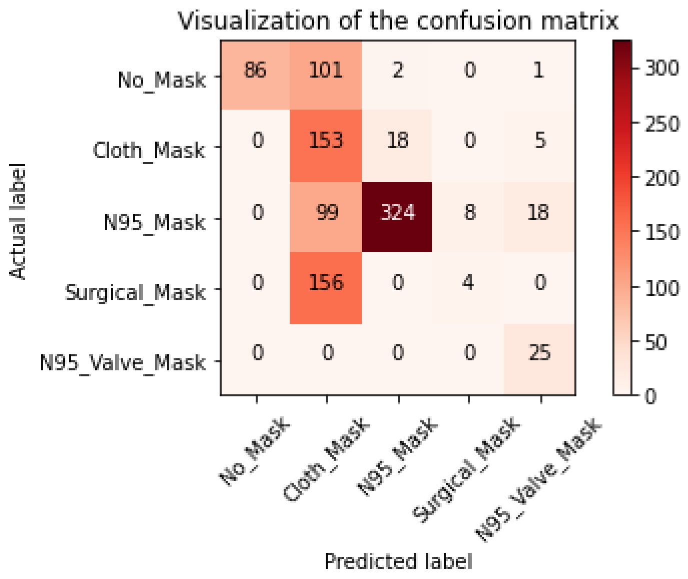
```
no: 100%|████████████████████████████████████
███| 142/142 [00:03<00:00, 41.95it/s]
cloth: 100%|█████████████████████████████████
███| 140/140 [00:03<00:00, 44.60it/s]
N95: 100%|███████████████████████████████████
███| 133/133 [00:02<00:00, 44.47it/s]
surgical: 100%|██████████████████████████████
███| 138/138 [00:03<00:00, 43.54it/s]
valve: 100%|█████████████████████████████████
███| 147/147 [00:03<00:00, 44.83it/s]
```

```
Confusion matrix without normalization
[[ 92  99   5   0   3]
 [  3 172   0   0   7]
 [  0  98  87   2   0]
 [  0 177   0  32   0]
 [  0   4   9   0 210]]
Fold results for Millennial Group:  (0.593, 0.8076512778212805, 0.593484245655289
8, 0.5822836223398076)
```



In [15]:
```
#AgeGroup - Boomer
from tqdm import tqdm
import matplotlib.pyplot as plt
import random

class_mapping = {
    0: "No_Mask",
    1: "Cloth_Mask",
    2: "N95_Mask",
    3: "Surgical_Mask",
    4: "N95_Valve_Mask"
}

def prepare_predict_df():
    datasetPath = dirPath/'testDataset/AgeGroup' #path to new test data. (not used
    boomerPath = datasetPath/'Boomer'

    boomerNoMaskPath = boomerPath/'NoMask'
    boomerClothMaskPath = boomerPath/'Cloth'
    boomerN95MaskPath = boomerPath/'N95'
    boomerSurgicalMaskPath = boomerPath/'Surgical'
    boomerValveMaskPath = boomerPath/'N95Valve'


    testDF = pd.DataFrame()

    for imgPath in tqdm(list(boomerNoMaskPath.iterdir()), desc='no'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 0
        }, ignore_index=True)

    for imgPath in tqdm(list(boomerClothMaskPath.iterdir()), desc='cloth'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 1
        }, ignore_index=True)

    for imgPath in tqdm(list(boomerN95MaskPath.iterdir()), desc='N95'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 2
        }, ignore_index=True)

    for imgPath in tqdm(list(boomerSurgicalMaskPath.iterdir()), desc='surgical'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 3
        }, ignore_index=True)

    for imgPath in tqdm(list(boomerValveMaskPath.iterdir()), desc='valve'):
        testDF = testDF.append({
            'image': str(imgPath),
            'mask': 4
        }, ignore_index=True)

    return mask_dataset(testDF)

def predict():
    predictions, actuals = torch.tensor([]), torch.tensor([])
    test_dfTest = prepare_predict_df()
    rand_sampler = torch.utils.data.RandomSampler(test_dfTest, num_samples=1000, r
    data = iter(DataLoader(test_dfTest, batch_size=1000, num_workers=0, sampler=ra
    inputs,targets = data['image'], data['mask']
    targets = targets.flatten()


    output = face_mask_detector_cnn(inputs)
    output = torch.argmax(output,axis=1)
    predictions = torch.cat((predictions, output.flatten()), dim=0)
    actuals = torch.cat((actuals, targets), dim=0)

    return (confusion_matrix(actuals.numpy(), predictions.numpy()),accuracy_score(

fold_result_test = predict()
fold_confusion_matrix_test = fold_result_test[0]
fold_result_test_metrics = fold_result_test[1:-1]
conf_mat(fold_confusion_matrix_test, classes)

print("Fold results for Boomer Group: ",fold_result_test_metrics)
```

```
no: 100%|█████████████████████████████████████████████████████████████████████████
███| 143/143 [00:03<00:00, 43.19it/s]
cloth: 100%|██████████████████████████████████████████████████████████████████████
███| 131/131 [00:02<00:00, 44.79it/s]
N95: 100%|████████████████████████████████████████████████████████████████████████
███| 301/301 [00:06<00:00, 44.02it/s]
surgical: 100%|███████████████████████████████████████████████████████████████████
███| 111/111 [00:02<00:00, 39.37it/s]
valve: 100%|██████████████████████████████████████████████████████████████████████
█████| 19/19 [00:00<00:00, 34.20it/s]
Confusion matrix without normalization
[[ 86 101   2   0   1]
 [  0 153  18   0   5]
 [  0  99 324   8  18]
 [  0 156   0   4   0]
 [  0   0   0   0  25]]
Fold results for Boomer Group:  (0.592, 0.6171974542089875, 0.6137106648479875, 0.
521848220179722)
```


Visualization of the confusion matrix
Actual label
Predicted label
No_Mask
Cloth_Mask
N95_Mask
Surgical_Mask
N95_Valve_Mask
86 101 2 0 1
0 153 18 0 5
0 99 324 8 18
0 156 0 4 0
0 0 0 0 25
300
250
200
150
100
50
0

```
In [ ]:
```