```python
import numpy as np
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
from torchvision import datasets, transforms, models
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt
import imageio
from torch.optim import lr_scheduler
from torchsummary import summary

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device

test_run = 7

print("test_run",test_run)


# Specify transforms using torchvision.transforms as transforms library
transformations = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

# Split the data into training, validation and datasets
# import splitfolders
# import os

# path = 'F:\skin_disease_data' #'mix-data'
# print(os.listdir(path))
# splitfolders.ratio(path, output="mix_data_split", seed=1337, ratio=(.6, .2, .2))

train_set_mix = datasets.ImageFolder(f"/../../training_data_split_run_{test_run}/train", transform=transformations)
val_set_mix = datasets.ImageFolder(f"/../../training_data_split_run_{test_run}/val", transform=transformations)
test_set_mix = datasets.ImageFolder(f"/../../training_data_split_run_{test_run}/test", transform=transformations)



print(len(train_set_mix))
print(len(val_set_mix))
print(len(test_set_mix))


class_names  = train_set_mix.classes
class_names

train_loader_mix = torch.utils.data.DataLoader(train_set_mix, batch_size=32, shuffle=True, num_workers=8)
```

```
val_loader_mix = torch.utils.data.DataLoader(val_set_mix, batch_size =32, shuffle=False, num_workers=8)
test_loader_mix = torch.utils.data.DataLoader(test_set_mix, batch_size =32, shuffle=False, num_workers=8)


# get some random training images
dataiter = iter(train_loader_mix)
images, labels = next(dataiter)


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
imshow(torchvision.utils.make_grid(images))


model = models.resnet50(weights='ResNet50_Weights.IMAGENET1K_V1')




for param in model.parameters():
    param.requires_grad = True #False #Set True to train the whole network
# Creating final fully connected Layer that accorting to the no of classes we require
model.fc = nn.Sequential(nn.Linear(2048, 512),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(512, len(class_names)))#,
                                 #nn.LogSoftmax(dim=1))



#print(summary(model, input_size=(3, 224, 224)))
# Loss and optimizer
criterion = nn.CrossEntropyLoss()

# variable learning rate for different layers and usingg cosine annealing warm restarts
optimizer = optim.SGD([
        {'params': model.conv1.parameters(), 'lr':1e-4},
        {'params': model.bn1.parameters(),   'lr': 1e-4},
        {'params': model.layer1.parameters(), 'lr':1e-4},
        {'params': model.layer2.parameters(),'lr':1e-4},
        {'params': model.layer3.parameters(),'lr':1e-3},
        {'params': model.layer4.parameters() ,'lr':1e-3},
        {'params': model.fc.parameters(), 'lr': 1e-2}   # the classifier needs to learn weights faster
    ], lr=0.001, weight_decay=0.0005)




if torch.cuda.device_count() > 1:
    print(f"Using DataParallel Mode with cuda count {torch.cuda.device_count()}" )
    model = torch.nn.DataParallel(model)


```

```
model.to(device)


print(summary(model, input_size=(3, 224, 224)))



# Restarts the learning rate after every 5 epoch
scheduler = lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer,
        T_0= 5,
        T_mult= 1,
    )


epochs = 200
best_acc = 0.0
iters = len(train_loader_mix)
patience = 500
patience_counter = patience
best_val_loss = np.inf


train_loss, val_loss = [], []

import time


print("-----starting training--------")
for epoch in range (epochs):
    start_time = time.time()

    #logger.warning(f"Epoch started: {epoch}")
    train_loss_epoch = 0.0
    valid_loss_epoch = 0.0
    accuracy = 0.0

    # Trainin_the_ model
    model.train()
    correct_predictions_train = 0
    total_samples_train = 0


    for i, sample in enumerate (train_loader_mix):
        inputs, labels = sample
        # Move data to GPU
        inputs, labels = inputs.to(device), labels.to(device)
        # Clear Optimizers
        optimizer.zero_grad()
        # Forward Pass
        logps = model.forward(inputs)
        # Loss
        loss = criterion(logps, labels)
        # Backprop (Calculate Gradients)
        loss.backward()
        # Adjust parameters based on gradients
        optimizer.step()
        scheduler.step(epoch + i / iters) # if using cosine annealing warm restarts
        # Add the loss to the  running loss
        train_loss_epoch += loss.item() * inputs.size(0)
```

```
170
171         # Calculate training accuracy
172         _, predicted = torch.max(logps, 1) #_, train_preds = torch.max(logps, 1)
173         correct_predictions_train += torch.sum(predicted == labels).item()#
correct_train += torch.sum(train_preds == labels).item()
174         total_samples_train += labels.size(0) #total_train += labels.size(0)
175
176     train_accuracy = correct_predictions_train / total_samples_train# train_accuracy
= correct_train / total_train
177
178
179
180
181
182     # Update the learning rate scheduler
183     #scheduler.step()
184
185     # Validation
186     model.eval ()
187     correct_predictions = 0
188     total_samples = 0
189     # Tell torch not to calculate gradients
190     with torch.no_grad ():
191         for inputs, labels in val_loader_mix:
192             # Move to device
193             inputs, labels = inputs.to (device), labels.to(device)
194             # Forward pass
195             output = model.forward(inputs)
196             # Calculate Loss
197             val_loss_batch = criterion(output, labels)
198             # Add loss to the validation set's running loss
199             valid_loss_epoch += val_loss_batch.item() * inputs.size(0)
200
201             # Since our model outputs a LogSoftmax, find the real
202             _, predicted = torch.max(output, 1)
203             correct_predictions += torch.sum(predicted == labels).item()
204             total_samples += labels.size(0)
205         accuracy = correct_predictions / total_samples
206
207
208     # Early Stopping
209     if valid_loss_epoch < best_val_loss:
210         best_val_loss = valid_loss_epoch
211         patience_counter = 0
212         torch.save(model.state_dict(),
f'best_model_resnet_modified_300_epochs_28_7_2025_train_accuracy_print_test_run_{test
_run}.pth')
213         print(f"Saving the model
best_model_resnet_modified_300_epochs_28_7_2025_train_accuracy_print_test_run_{test_r
un}.pth at epoch {epoch}")
214     else:
215         patience_counter -= 1
216         if patience_counter == 0:
217             print('Early stopping')
218             break
219
220
221
222     # Get the average loss for the  epoch
223     train_loss_epoch /= len(train_loader_mix.dataset)
```

```
    valid_loss_epoch /= len(val_loader_mix.dataset)

    # Append the loss and accuracy
    train_loss.append(train_loss_epoch)
    val_loss.append(valid_loss_epoch)



    epoch_time = time.time() - start_time
    # Print out the information
    print(f'Epoch {epoch + 1},'
          f' Training Loss: {train_loss_epoch:.6f},'
          f' Training Accuracy: {train_accuracy:.6f},'
          f' Validation Loss: {valid_loss_epoch:.6f},'
          f' Accuracy: {accuracy:.6f}'
          f"Time: {epoch_time:.2f} seconds"
    )

#logger.warning(f"Training finished")

print(f"Saving the model
best_model_resnet_modified_300_epochs_28_7_2025_train_accuracy_print_test_run_{test_r
un}_save_after_300_epochs.pth")
torch.save(model.state_dict(),
f'best_model_resnet_modified_300_epochs_28_7_2025_train_accuracy_print_test_run_{test
_run}_save_after_300_epochs.pth')


# Plotting the training and validation loss
plt.plot(range(1, epochs+1), train_loss, label='Training loss')
plt.plot(range(1, epochs+1), val_loss, label='Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title(f'Training and Validation Loss_{test_run}')
plt.legend()
plt.savefig(f"training_loss_curves_test_run_{test_run}.png")
plt.show()

#print(f'Best Validation Accuracy: {best_acc:.6f}')


##############################################################################################
#########################################################################
import numpy as np
import torch
import torch.nn.functional as F
from torch.nn.functional import softmax
import numpy
from tqdm import tqdm

alpha = 0.1


##########################----------------------------------------Calculation of
Calibration Scores-------------------------------------------------
########################
def calibration_scores(model, data_loader, device):

    model.eval()

```

```
    scores_calibration_set = []

    with torch.no_grad():
        for X, y in tqdm(data_loader, desc="##################Proceeding with the Calculation of Calibration Scores##################"):
            X = X.to(device)
            y = y.to(device)

            y = y.cpu().numpy()

            pred_logits = model(X)

            # cal_smx = model(calib_X).softmax(dim=1 ).numpy()
            cal_smx = softmax(pred_logits, dim=1).cpu().numpy()

            # cal_scores = 1-cal_smx[np . arange(n),cal_labels]
            scores_per_batch = 1 - cal_smx[np.arange(y.shape[0]), y]

            scores_calibration_set.extend(scores_per_batch)

    scores_calibration_set = numpy.array(scores_calibration_set)

    return scores_calibration_set
#############################################################################################################


##########################----------------------------------Calculation of Prediction Sets--------------------------#########################
def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha, device):


    model.eval()


    prediction_sets = []
    true_labels = []

    n = len(calibration_scores_array)

    #get adjusted quantile
    #q_level = np . ceil((n+1 ) *(1-alpha)) /n
    q_level = np.ceil((n + 1) * (1 - alpha)) / n

    # qhat = np . quantile(calibration_scores_array, q_level, method='higher')
    qhat = np.quantile(calibration_scores_array, q_level, method='higher')

    with torch.no_grad():
        for X, y in tqdm(data_loader, desc="##################Proceeding with the Calculation of Testing Scores##################"):

            X = X.to(device)
            y = y.cpu().numpy()
            pred_logits = model(X)

            # cal_smx = model(calib_X).softmax(dim=1 ).numpy()
            val_smx = softmax(pred_logits, dim=1)
            val_smx = val_smx.cpu().numpy()

```

```
            # prediction_sets = val_smx >= (1-qhat)
            masks = val_smx >= (1 - qhat)

            for _ in masks:
                prediction_sets.append(np.where(_)[0].tolist())

            true_labels.extend(y.tolist())

    return prediction_sets, true_labels
################################################################################################################################


##########################----------------------------------------EVALUATION----------------------------------------##########################
def evaluate(prediction_sets, labels):
    number_of_correct_predictions = 0
    for pred_set, true_label in zip(prediction_sets, labels):
        if true_label in pred_set:
            number_of_correct_predictions += 1
    empirical_test_coverage = number_of_correct_predictions / len(labels)
    average_prediction_set_size = np.mean([len(s) for s in prediction_sets])
    return empirical_test_coverage, average_prediction_set_size
################################################################################################################################

# ----------------------- Run Full Conformal Prediction -----------------------
scores_calibration = calibration_scores(model=model, data_loader=val_loader_mix, device=device)
prediction_sets, true_labels = predict_with_conformal_sets(model=model, data_loader=test_loader_mix, calibration_scores_array=scores_calibration, \
                                                           alpha=alpha, device=device)
test_set_coverage, avg_size = evaluate(prediction_sets=prediction_sets, labels=true_labels)

import os
from datetime import datetime
import numpy
import pickle
import numpy as np


timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
os.makedirs(f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}", exist_ok=True)

results_directory = f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}"

import pickle

with open(f"{results_directory}/scores_calibration_run_{test_run}__{timestamp}.pkl", "wb") as f:
    pickle.dump(scores_calibration, f)


# with open("scores_calibration.pkl", "rb") as f:
#     scores_calibration = pickle.load(f)

```

```
382
383
384 with
    open(f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}/logs
    _experiment_run_{test_run}_{timestamp}.txt", "w") as f:
385     f.write("########################## Results
    ##################################\n")
386     f.write(f"Coverage: {test_set_coverage:.3f}\n")
387     f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")
388
     f.write("#########################################################################\n")
389 print(f"logs file in directory
    Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}")
390
391
392 print("##########################Results:###################################")
393 print(f"Coverage: {test_set_coverage:.3f}")
394 print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
395 print("#########################################################################")
396
397 np.save(f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}/c
    alibration_scores_experiment_run_{test_run}_{timestamp}.npy", scores_calibration)
398
399 with open(f"
    {results_directory}/prediction_data_experiment_run_{test_run}__{timestamp}.pkl",
    "wb") as f:
400     pickle.dump({
401         "prediction_sets": prediction_sets,
402         "true_labels": true_labels
403     }, f)
404
405
406 # import numpy as np
407 # import pickle
408
409 # # Load calibration scores
410 # scores = np.load("conformal_results/scores_calibration_2025-08-02_14-30-00.npy")
411
412 # # Load prediction sets and labels
413 # with open("conformal_results/prediction_data_2025-08-02_14-30-00.pkl", "rb") as f:
414 #     data = pickle.load(f)
415 #     prediction_sets = data["prediction_sets"]
416 #     true_labels = data["true_labels"]
417
418
419 ##############################################TO BE
    CHANGED#################################################################################
    #############
420 import numpy as np
421 import matplotlib.pyplot as plt
422 from collections import Counter
423 import pandas as pd
424
425 set_sizes = [len(prediction_set) for prediction_set in prediction_sets]
426 counts = Counter(set_sizes)
427 total = sum(counts.values())
428 sizes = sorted(counts.keys())
429 frequencies = [counts[size] for size in sizes]
430 percentages = [100 * freq / total for freq in frequencies]
431
```

```
data = [{'size': len(prediction_set), 'correct': int(true in prediction_set)} for prediction_set, true in zip(prediction_sets, true_labels)]
df = pd.DataFrame(data)
coverage_stats = df.groupby('size').agg({'correct': ['mean', 'sum', 'count']})
coverage_stats.columns = ['coverage', 'correct_count', 'total_count']
coverage_stats = coverage_stats.reindex(sizes, fill_value=0)

cumulative_correct = coverage_stats['correct_count'].cumsum()
cumulative_coverage = 100 * cumulative_correct / len(true_labels)
max_freq = max(frequencies)
cumulative_scaled = (cumulative_coverage / 100) * max_freq

fig, ax = plt.subplots(figsize=(13, 6))
bars = ax.bar(sizes, frequencies, color='pink', edgecolor='black', label='Frequency', alpha =0.7)
ax.plot(sizes, cumulative_scaled, color='red', marker='o', linestyle='-', alpha=0.4, linewidth=2, label='Cumulative Coverage')

for bar, size, pct in zip(bars, sizes, percentages):
    height = bar.get_height()
    count = counts[size]
    coverage = coverage_stats.loc[size, 'coverage'] * 100
    ax.annotate(f'{count} ({pct:.1f}%)\nCov: {coverage:.1f}%',
                xy=(bar.get_x() + bar.get_width() / 2, height / 2),
                ha='center', va='center', fontsize=9)

for bar, cum_cov in zip(bars, cumulative_coverage):
    height = bar.get_height()
    ax.annotate(f'CumCov: {cum_cov:.1f}%',
                xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 8),
                textcoords="offset points",
                ha='center', va='bottom', fontsize=9, color='darkorange')

mean_size = np.mean(set_sizes)
median_size = np.median(set_sizes)
ax.axvline(mean_size, color='gray', linestyle='--', label=f'Mean = {mean_size:.2f}')
ax.axvline(median_size, color='orange', linestyle='--', label=f'Median = {median_size:.2f}')

ax.set_xlabel("Prediction Set Size")
ax.set_ylabel("Frequency")
ax.set_title("Prediction Set Size Distribution with Coverage (Test Set)")
ax.set_xticks(sizes)
ax.grid(axis='y', linestyle='--', alpha=0.6)
ax.legend(loc='upper right')

plt.tight_layout()
plt.savefig(os.path.join(results_directory, f"Prediction_Set_Size_Distribution_with_Coverage_(Test_Set)_{test_run}__{timestamp}.png"))




plt.show()


```

```
##################################################script#############################
#############################################
from collections import Counter
import matplotlib.pyplot as plt
from collections import defaultdict
import numpy as np
from scipy.stats import pearsonr, spearmanr


def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1,
class_names=None):


    class_to_indices = defaultdict(list)
    for idx, label in enumerate(true_labels):
        class_to_indices[label].append(idx)

    class_coverages = {}
    for cls, indices in class_to_indices.items():
        covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
        coverage = covered / len(indices)
        class_coverages[cls] = coverage

    print("\n Feature-Stratified Coverage (by True Class Label):")
    for cls, cov in sorted(class_coverages.items()):
        name = class_names[cls] if class_names else str(cls)
        print(f"  Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")

    fsc_metric = min(class_coverages.values())
    print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")

    return fsc_metric, class_coverages


fsc_metric, per_class_cov = feature_stratified_coverage_by_class(
    prediction_sets,
    true_labels,
    alpha=alpha,
    class_names=class_names
)


def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None,
title="Class-wise Coverage (FSC)"):

    classes = list(class_coverages.keys())
    coverages = [class_coverages[c] for c in classes]

    # Get display names
    labels = [class_names[c] if class_names else str(c) for c in classes]

    # Plot
    plt.figure(figsize=(12, 6))
    bars = plt.bar(labels, coverages, color="pink", edgecolor="black")

    # Target line
    target = 1 - alpha
    plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
({target:.2f})")

```

```
    # Highlight underperforming classes
    for bar, cov in zip(bars, coverages):
        if cov < target:
            bar.set_color('salmon')

    plt.xticks(rotation=45, ha="right")
    plt.ylim(0, 1.05)
    plt.ylabel("Coverage")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    filename = f"Class_wise_Coverage_(FSC)_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()

fsc_metric, class_coverages = feature_stratified_coverage_by_class(
    prediction_sets,
    true_labels,
    alpha=alpha,
    class_names=class_names
)

plot_class_wise_coverage(class_coverages, alpha=alpha, class_names=class_names)




def plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=0.1,
class_names=None, title="Coverage vs Class Frequency"):



    class_indices = sorted(class_coverages.keys())
    coverages = [class_coverages[c] for c in class_indices]


    total = len(true_labels)
    counts = Counter(true_labels)
    frequencies = [counts[c] / total for c in class_indices]


    labels = [class_names[c] if class_names else str(c) for c in class_indices]
    x = np.arange(len(class_indices))

    fig, ax1 = plt.subplots(figsize=(12, 6))


    bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
edgecolor='black')
    ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
alpha:.2f})")
    ax1.set_ylim(0, 1.05)
    ax1.set_ylabel("Coverage", color='pink')
    ax1.tick_params(axis='y', labelcolor='pink')

    # Highlight under-covered bars
    for bar, cov in zip(bars1, coverages):
        if cov < (1 - alpha):
            bar.set_color('salmon')
```

```
598
599     # Twin y-axis for frequencies
600     ax2 = ax1.twinx()
601     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
color='gray', alpha=0.5)
602     ax2.set_ylabel("Class Frequency", color='gray')
603     ax2.tick_params(axis='y', labelcolor='gray')
604
605     # X-axis
606     plt.xticks(x, labels, rotation=45, ha="right")
607
608     # Title and legend
609     plt.title(title)
610     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
611     plt.tight_layout()
612     filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
613     plt.savefig(os.path.join(results_directory, filename))
614     plt.show()
615
616
617 # Step 1: Compute coverage by class
618 fsc_metric, class_coverages = feature_stratified_coverage_by_class(
619     prediction_sets,
620     true_labels,
621     alpha=alpha,
622     class_names=class_names
623 )
624
625 # Step 2: Compare against class frequency
626 plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=alpha,
class_names=class_names)
627
628
629 def plot_coverage_vs_class_frequency_with_correlation(class_coverages, true_labels,
alpha=0.1, class_names=None, title="Coverage vs Class Frequency"):
630
631
632
633     class_indices = sorted(class_coverages.keys())
634     coverages = [class_coverages[c] for c in class_indices]
635
636     # Compute frequencies
637     total = len(true_labels)
638     counts = Counter(true_labels)
639     frequencies = [counts[c] / total for c in class_indices]
640
641     # Class labels
642     labels = [class_names[c] if class_names else str(c) for c in class_indices]
643     x = np.arange(len(class_indices))
644
645     # Correlation analysis
646     pearson_corr, pearson_p = pearsonr(frequencies, coverages)
647     spearman_corr, spearman_p = spearmanr(frequencies, coverages)
648
649     print(f"\n Correlation between class frequency and coverage:")
650     print(f"  - Pearson  r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
651     print(f"  - Spearman ρ = {spearman_corr:.3f}, p = {spearman_p:.3e}")
652
653     # Plotting
654     fig, ax1 = plt.subplots(figsize=(12, 6))
```

```
655
656     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
    edgecolor='black')
657     ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
    alpha:.2f})")
658     ax1.set_ylim(0, 1.05)
659     ax1.set_ylabel("Coverage", color='pink')
660     ax1.tick_params(axis='y', labelcolor='pink')
661
662     # Highlight under-covered classes
663     for bar, cov in zip(bars1, coverages):
664         if cov < (1 - alpha):
665             bar.set_color('salmon')
666
667     # Frequency bars on second axis
668     ax2 = ax1.twinx()
669     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
    color='gray', alpha=0.5)
670     ax2.set_ylabel("Class Frequency", color='gray')
671     ax2.tick_params(axis='y', labelcolor='gray')
672
673     # X-axis and titles
674     plt.xticks(x, labels, rotation=45, ha="right")
675     plt.title(title)
676     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
677     plt.tight_layout()
678     filename =
    f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
679     plt.savefig(os.path.join(results_directory, filename))
680     plt.show()
681
682
683 plot_coverage_vs_class_frequency_with_correlation(
684     class_coverages=class_coverages,
685     true_labels=true_labels,
686     alpha=alpha,
687     class_names=class_names
688 )
689
690
691
692 def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1, bins=[[1], [2],
    [3, 4, 5], list(range(6, 100))]):
693     """
694     Computes size-stratified coverage (SSC): coverage within prediction set size
    bins.
695
696     Args:
697         prediction_sets (List[List[int]]): output sets from conformal_predict
698         true_labels (List[int]): ground-truth labels
699         alpha (float): target error level (e.g., 0.1)
700         bins (List[List[int]]): list of prediction set size groups
701
702     Returns:
703         ssc (float): min group coverage (worst-case)
704         bin_coverages (dict): {bin_range: coverage}
705     """
706     bin_groups = defaultdict(list)
707
708     # Assign examples to bins based on set size
```

```
    for i, pred_set in enumerate(prediction_sets):
        set_size = len(pred_set)
        for bin_range in bins:
            if set_size in bin_range:
                bin_groups[str(bin_range)].append(i)
                break

    bin_coverages = {}
    for bin_key, indices in bin_groups.items():
        covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
        coverage = covered / len(indices) if indices else 0
        bin_coverages[bin_key] = coverage

    print("\n Size-Stratified Coverage (SSC):")
    for bin_key, cov in bin_coverages.items():
        print(f"  Set Size {bin_key:10s}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")

    ssc = min(bin_coverages.values())
    print(f"\n SSC Metric (min bin-wise coverage): {ssc:.3f}")
    return ssc, bin_coverages


ssc_metric, bin_coverages = size_stratified_coverage(
    prediction_sets,
    true_labels,
    alpha=alpha,
    bins=[[1], [2], [3], list(range(4, 100))]
)



def plot_ssc_coverage(bin_coverages, alpha=0.1, title="Size-Stratified Coverage"):
    """
    Plots coverage per prediction set size bin.

    Args:
        bin_coverages (dict): mapping from bin label (e.g. '[1]', '[2, 3]') to coverage
        alpha (float): target error level
        title (str): plot title
    """
    bin_labels = list(bin_coverages.keys())
    coverages = [bin_coverages[k] for k in bin_labels]

    plt.figure(figsize=(10, 5))
    bars = plt.bar(bin_labels, coverages, color='pink', edgecolor='black')

    # Target line
    target = 1 - alpha
    plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage ({target:.2f})")

    # Highlight low-coverage bins
    for bar, cov in zip(bars, coverages):
        if cov < target:
            bar.set_color('grey')

    plt.xticks(rotation=45, ha='right')
    plt.ylim(0, 1.05)
```

```
    plt.ylabel("Coverage")
    plt.title(title)
    plt.legend()
    plt.tight_layout()
    filename = f"plot_ssc_coverage_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


ssc_metric, bin_coverages = size_stratified_coverage(
    prediction_sets,
    true_labels,
    alpha=alpha,
    bins=[[1], [2], [3], list(range(4, 100))]
)

plot_ssc_coverage(bin_coverages, alpha=alpha)



def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None,
results_dir="", test_run="", timestamp=""):

    # Group prediction set sizes by true class
    size_per_class = defaultdict(list)
    for pred_set, true_label in zip(prediction_sets, true_labels):
        size_per_class[true_label].append(len(pred_set))

    # Sort classes
    sorted_classes = sorted(size_per_class.keys())
    means = [np.mean(size_per_class[c]) for c in sorted_classes]
    stds = [np.std(size_per_class[c]) for c in sorted_classes]
    labels = [class_names[c] if class_names else str(c) for c in sorted_classes]

    # Plot
    plt.figure(figsize=(12, 6))
    plt.bar(labels, means, yerr=stds, capsize=5, color="lightgreen",
edgecolor="black")
    plt.ylabel("Avg Prediction Set Size")
    plt.xlabel("Class")
    plt.title("Average Prediction Set Size by Class")
    plt.xticks(rotation=45, ha="right")
    plt.grid(axis="y", linestyle="--", alpha=0.5)
    plt.tight_layout()

    # Save
    filename = f"prediction_set_size_by_class_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


plot_prediction_set_size_by_class(
    prediction_sets=prediction_sets,
    true_labels=true_labels,
    class_names=class_names,
    results_dir=results_directory,
    test_run=test_run,
    timestamp=timestamp
)

```

```python


def plot_calibration_score_distribution(calibration_scores, results_dir="", test_run="", timestamp=""):

    plt.figure(figsize=(10, 5))
    plt.hist(calibration_scores, bins=30, color="pink", edgecolor="black", alpha=0.8)
    plt.xlabel("Nonconformity Score (1 - P(True Class))")
    plt.ylabel("Frequency")
    plt.title("Calibration Score Distribution")
    plt.grid(axis='y', linestyle='--', alpha=0.6)
    plt.tight_layout()

    filename = f"calibration_score_distribution_{test_run}__{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))

    plt.show()


plot_calibration_score_distribution(
    calibration_scores=scores_calibration,
    results_dir=results_directory,
    test_run=test_run,
    timestamp=timestamp
)


import json

metrics = {
    "test_run": test_run,
    "timestamp": timestamp,
    "alpha": alpha,
    "test_set_coverage": test_set_coverage,
    "average_prediction_set_size": avg_size,
    "FSC": fsc_metric,
    "SSC": ssc_metric,
    "class_coverages": {str(k): float(v) for k, v in class_coverages.items()},
    "bin_coverages": bin_coverages
}

with open(os.path.join(results_directory, f"metrics_summary_{test_run}_{timestamp}.json"), "w") as f:
    json.dump(metrics, f, indent=4)




################################################################################################################################
```