```python
import os
from datetime import datetime

# For this notebook to run with updated APIs, we need torch 1.12+ and torchvision 0.13+
try:
    import torch
    import torchvision
    assert int(torch.__version__.split(".")[1]) >= 12 or int(torch.__version__.split(".")[0]) == 2, "torch version should be 1.12+"
    assert int(torchvision.__version__.split(".")[1]) >= 13, "torchvision version should be 0.13+"
    print(f"torch version: {torch.__version__}")
    print(f"torchvision version: {torchvision.__version__}")
except:
    print(f"[INFO] torch/torchvision versions not as required, installing nightly versions.")
    #!pip3 install -U torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
    import torch
    import torchvision
    print(f"torch version: {torch.__version__}")
    print(f"torchvision version: {torchvision.__version__}")


import os
import shutil
import random
import matplotlib.pyplot as plt
import torch
import torchvision
from torch import nn
from torchvision import transforms
try:
    from torchinfo import summary
except:
    print("[INFO] Couldn't find torchinfo... installing it.")
    #!pip install -q torchinfo
    from torchinfo import summary
from pathlib import Path
import os



test_run = 7
EPOCHS = 200


print("#####################################################################")
print("#####################################################################")
print("#####################################################################")
print(f"Vision Transformer Experiment For Test Run: {test_run}")
print(f"The Number of Epochs are: {EPOCHS}")
print("#####################################################################")
print("#####################################################################")
print("#####################################################################")


device = "cuda" if torch.cuda.is_available() else "cpu"
print("#####################################################################")
```

```python
print(f"The device for running the experiment is {device}")
print("###################################################################")






# Path to the dataset folder
main_folder = f"/../../training_data_split_run_{test_run}"
splits = ['train', 'test', 'val']

summary = {}

for split in splits:
    split_path = os.path.join(main_folder, split)
    total_files = 0
    summary[split] = {}


    if not os.path.exists(split_path):
        print(f"Directory not found: {split_path}")
        continue

    for disease in os.listdir(split_path):
        disease_path = os.path.join(split_path, disease)

        if not os.path.isdir(disease_path):
            continue


        num_files = len([
            f for f in os.listdir(disease_path)
            if os.path.isfile(os.path.join(disease_path, f))
        ])

        summary[split][disease] = num_files
        total_files += num_files

    print(f"\n{split.upper()} - Total files: {total_files}")
    for disease, count in summary[split].items():
        print(f"  {disease}: {count} files")


train_dir = f"/../../training_data_split_run_{test_run}/train"
test_dir = f"/../../training_data_split_run_{test_run}/val"

# Create image size (from Table 3 in the ViT paper)
IMG_SIZE = 224

# Create transform pipeline manually
manual_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
])
print(f"Manually created transforms: {manual_transforms}")

from going_modular import data_setup

```

```
BATCH_SIZE = 32 # this is lower than the ViT paper but it's because we're starting small

# Create data loaders
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=manual_transforms, # use manually created transforms
    batch_size=BATCH_SIZE
)

train_dataloader, test_dataloader, class_names


print(f"The class names are: {class_names}")


from going_modular.helper_functions import set_seeds
set_seeds()


pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT

# 2. Setup a ViT model instance with pretrained weights
pretrained_vit = torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)

# 3. Freeze the base parameters
#for parameter in pretrained_vit.parameters():
#    parameter.requires_grad = False

# 4. Change the classifier head (set the seeds to ensure same initialization with linear head)
#set_seeds()
pretrained_vit.heads = nn.Linear(in_features=768, out_features=len(class_names)).to(device)
# pretrained_vit


from torchinfo import summary

# Print a summary using torchinfo (uncomment for actual output)
summary(model=pretrained_vit,
        input_size=(32, 3, 224, 224), # (batch_size, color_channels, height, width)
        # col_names=["input_size"], # uncomment for smaller output
        col_names=["input_size", "output_size", "num_params", "trainable"],
        col_width=20,
        row_settings=["var_names"]
)


# Get automatic transforms from pretrained ViT weights
pretrained_vit_transforms = pretrained_vit_weights.transforms()
print(pretrained_vit_transforms)

train_dataloader_pretrained, test_dataloader_pretrained, class_names = data_setup.create_dataloaders(train_dir=train_dir,

                test_dir=test_dir,
```

```
                transform=pretrained_vit_transforms,
                batch_size=32)

from going_modular import engine

# Create optimizer and loss function
#optimizer = torch.optim.Adam(params=pretrained_vit.parameters(),
#                             lr=1e-3)
optimizer = torch.optim.Adam(params=pretrained_vit.parameters(),
                             lr=1e-5)


loss_fn = torch.nn.CrossEntropyLoss()


import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Wrap model for multi-GPU if available
if torch.cuda.device_count() > 1:
    print(f"[INFO] Using DataParallel on {torch.cuda.device_count()} GPUs")
    #pretrained_vit = torch.nn.DataParallel(pretrained_vit)
    pretrained_vit = torch.nn.DataParallel(pretrained_vit)

pretrained_vit.to(device)


# Train the classifier head of the pretrained ViT feature extractor model
#set_seeds()

from going_modular.helper_functions import plot_loss_curves
import os
from datetime import datetime
import numpy
import pickle
import numpy as np


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

import time
start_time = time.time()
pretrained_vit_results = engine.train(model=pretrained_vit,
                                      train_dataloader=train_dataloader_pretrained,
                                      test_dataloader=test_dataloader_pretrained,
                                      optimizer=optimizer,
                                      loss_fn=loss_fn,
                                      epochs=EPOCHS,
                                      device=device,
                                      test_run = test_run,
                                      results_directory=results_directory
                                      )
training_time = time.time() - start_time
print(f"The total time for training is {training_time:.2f} seconds")
```

```
225
226
227
228
229 plot_loss_curves(pretrained_vit_results,
    save_path=results_directory+f"/loss_accuracy_curve_{test_run}_{timestamp}.png")
230
231
232 # Save the model
233 from going_modular import utils
234
235 utils.save_model(model=pretrained_vit,
236                  target_dir=results_directory,
237
    model_name=f"Pretrained_vit_scratch_training_test_run_{test_run}_save_after_200_epoch
    s.pth")
238
239
240 from torchinfo import summary
241 # Print a summary using torchinfo (uncomment for actual output)
242 summary(model=pretrained_vit,
243         input_size=(32, 3, 224, 224), # (batch_size, color_channels, height, width)
244         # col_names=["input_size"], # uncomment for smaller output
245         col_names=["input_size", "output_size", "num_params", "trainable"],
246         col_width=20,
247         row_settings=["var_names"]
248 )
249
250
251 from pathlib import Path
252
253 # Get the model size in bytes then convert to megabytes
254 pretrained_vit_model_size =
    Path(results_directory+f"/Pretrained_vit_scratch_training_test_run_{test_run}_save_af
    ter_200_epochs.pth").stat().st_size // (1024*1024) # division converts bytes to
    megabytes (roughly)
255 print(f"Pretrained ViT feature extractor model size: {pretrained_vit_model_size} MB")
256
257
258
259 # Setup directory paths to train and test images
260 val_dir =
    f"/leonardo_work/IscrC_ArtLLMs/alitariqnagi_work/training_data_split_run_{test_run}/v
    al"
261 test_dir =
    f"/leonardo_work/IscrC_ArtLLMs/alitariqnagi_work/training_data_split_run_{test_run}/t
    est"
262
263 from going_modular import data_setup
264 from going_modular.data_setup import create_dataloaders_no_shuffle
265 # Setup dataloaders  create_dataloaders_no_shuffle
266 val_dataloader_pretrained, test_dataloader_pretrained, class_names =
    data_setup.create_dataloaders_no_shuffle(train_dir=val_dir,
267
                     test_dir=test_dir,
268
                     transform=pretrained_vit_transforms,
269
                     batch_size=32)
270
```

```
271
272 ##############################################################################################################################
273 import numpy as np
274 import torch
275 import torch.nn.functional as F
276 from torch.nn.functional import softmax
277 import numpy
278 from tqdm import tqdm
279
280 alpha = 0.1
281
282
283 ##########################----------------------------------------Calculation of Calibration Scores-----------------------------------------------
    ########################
284 def calibration_scores(model, data_loader, device):
285
286     model.eval()
287
288     scores_calibration_set = []
289
290     with torch.no_grad():
291         for X, y in tqdm(data_loader, desc="###################Proceeding with the Calculation of Calibration Scores###################"):
292             X = X.to(device)
293             y = y.to(device)
294
295             y = y.cpu().numpy()
296
297             pred_logits = model(X)
298
299             # cal_smx = model(calib_X).softmax(dim=1 ).numpy()
300             cal_smx = softmax(pred_logits, dim=1).cpu().numpy()
301
302             # cal_scores = 1-cal_smx[np . arange(n),cal_labels]
303             scores_per_batch = 1 - cal_smx[np.arange(y.shape[0]), y]
304
305             scores_calibration_set.extend(scores_per_batch)
306
307     scores_calibration_set = numpy.array(scores_calibration_set)
308
309     return scores_calibration_set
310 ##############################################################################################################################
    ############################################################
311
312
313 ##########################----------------------------------------Calculation of Prediction Sets--------------------------#####################
314 def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha, device):
315
316
317     model.eval()
318
319
320     prediction_sets = []
321     true_labels = []
322
323     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=pretrained_vit, data_loader=val_dataloader_pretrained, device=device)
prediction_sets, true_labels = predict_with_conformal_sets(model=pretrained_vit, data_loader=test_dataloader_pretrained, 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)



with open(f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}/logs_experiment_run_{test_run}_{timestamp}.txt", "w") as f:
    f.write("############################ Results ###################################\n")
    f.write(f"Coverage: {test_set_coverage:.3f}\n")
    f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")

 f.write("##############################################################################\n")
print(f"logs file in directory Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}")


print("############################Results:###################################")
print(f"Coverage: {test_set_coverage:.3f}")
print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
print("##############################################################################")

np.save(f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}/calibration_scores_experiment_run_{test_run}_{timestamp}.npy", scores_calibration)

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



##############################################TO BE CHANGED#################################################################
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
```

```
import pandas as pd

set_sizes = [len(prediction_set) for prediction_set in prediction_sets]
counts = Counter(set_sizes)
total = sum(counts.values())
sizes = sorted(counts.keys())
frequencies = [counts[size] for size in sizes]
percentages = [100 * freq / total for freq in frequencies]

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"))
```

```
476
477
478
479
480 plt.show()
481
482
483
484 ##################################################script#############################
    ##############################################
485 from collections import Counter
486 import matplotlib.pyplot as plt
487 from collections import defaultdict
488 import numpy as np
489 from scipy.stats import pearsonr, spearmanr
490
491
492 def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1,
    class_names=None):
493
494
495     class_to_indices = defaultdict(list)
496     for idx, label in enumerate(true_labels):
497         class_to_indices[label].append(idx)
498
499     class_coverages = {}
500     for cls, indices in class_to_indices.items():
501         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
502         coverage = covered / len(indices)
503         class_coverages[cls] = coverage
504
505     print("\n Feature-Stratified Coverage (by True Class Label):")
506     for cls, cov in sorted(class_coverages.items()):
507         name = class_names[cls] if class_names else str(cls)
508         print(f"  Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")
509
510     fsc_metric = min(class_coverages.values())
511     print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")
512
513     return fsc_metric, class_coverages
514
515
516 fsc_metric, per_class_cov = feature_stratified_coverage_by_class(
517     prediction_sets,
518     true_labels,
519     alpha=alpha,
520     class_names=class_names
521 )
522
523
524 def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None,
    title="Class-wise Coverage (FSC)"):
525
526     classes = list(class_coverages.keys())
527     coverages = [class_coverages[c] for c in classes]
528
529     # Get display names
530     labels = [class_names[c] if class_names else str(c) for c in classes]
531
532     # 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')

    # Twin y-axis for frequencies
    ax2 = ax1.twinx()
    bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency", color='gray', alpha=0.5)
    ax2.set_ylabel("Class Frequency", color='gray')
    ax2.tick_params(axis='y', labelcolor='gray')

    # X-axis
    plt.xticks(x, labels, rotation=45, ha="right")

    # Title and legend
    plt.title(title)
    fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
    plt.tight_layout()
    filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


# Step 1: Compute coverage by class
fsc_metric, class_coverages = feature_stratified_coverage_by_class(
    prediction_sets,
    true_labels,
    alpha=alpha,
    class_names=class_names
)

# Step 2: Compare against class frequency
plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=alpha, class_names=class_names)


def plot_coverage_vs_class_frequency_with_correlation(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]

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

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

    # Correlation analysis
    pearson_corr, pearson_p = pearsonr(frequencies, coverages)
```

```
    spearman_corr, spearman_p = spearmanr(frequencies, coverages)

    print(f"\n Correlation between class frequency and coverage:")
    print(f"  - Pearson  r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
    print(f"  - Spearman ρ = {spearman_corr:.3f}, p = {spearman_p:.3e}")

    # Plotting
    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 classes
    for bar, cov in zip(bars1, coverages):
        if cov < (1 - alpha):
            bar.set_color('salmon')

    # Frequency bars on second axis
    ax2 = ax1.twinx()
    bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency", color='gray', alpha=0.5)
    ax2.set_ylabel("Class Frequency", color='gray')
    ax2.tick_params(axis='y', labelcolor='gray')

    # X-axis and titles
    plt.xticks(x, labels, rotation=45, ha="right")
    plt.title(title)
    fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
    plt.tight_layout()
    filename = f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
    plt.savefig(os.path.join(results_directory, filename))
    plt.show()


plot_coverage_vs_class_frequency_with_correlation(
    class_coverages=class_coverages,
    true_labels=true_labels,
    alpha=alpha,
    class_names=class_names
)



def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1, bins=[[1], [2], [3, 4, 5], list(range(6, 100))]):
    """
    Computes size-stratified coverage (SSC): coverage within prediction set size bins.

    Args:
        prediction_sets (List[List[int]]): output sets from conformal_predict
        true_labels (List[int]): ground-truth labels
        alpha (float): target error level (e.g., 0.1)
        bins (List[List[int]]): list of prediction set size groups
```

```
    Returns:
        ssc (float): min group coverage (worst-case)
        bin_coverages (dict): {bin_range: coverage}
    """
    bin_groups = defaultdict(list)

    # 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})")
```

```
757
758     # Highlight low-coverage bins
759     for bar, cov in zip(bars, coverages):
760         if cov < target:
761             bar.set_color('grey')
762
763     plt.xticks(rotation=45, ha='right')
764     plt.ylim(0, 1.05)
765     plt.ylabel("Coverage")
766     plt.title(title)
767     plt.legend()
768     plt.tight_layout()
769     filename = f"plot_ssc_coverage_{test_run}__{timestamp}.png"
770     plt.savefig(os.path.join(results_directory, filename))
771     plt.show()
772
773
774 ssc_metric, bin_coverages = size_stratified_coverage(
775     prediction_sets,
776     true_labels,
777     alpha=alpha,
778     bins=[[1], [2], [3], list(range(4, 100))]
779 )
780
781 plot_ssc_coverage(bin_coverages, alpha=alpha)
782
783
784
785 def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None,
    results_dir="", test_run="", timestamp=""):
786
787     # Group prediction set sizes by true class
788     size_per_class = defaultdict(list)
789     for pred_set, true_label in zip(prediction_sets, true_labels):
790         size_per_class[true_label].append(len(pred_set))
791
792     # Sort classes
793     sorted_classes = sorted(size_per_class.keys())
794     means = [np.mean(size_per_class[c]) for c in sorted_classes]
795     stds = [np.std(size_per_class[c]) for c in sorted_classes]
796     labels = [class_names[c] if class_names else str(c) for c in sorted_classes]
797
798     # Plot
799     plt.figure(figsize=(12, 6))
800     plt.bar(labels, means, yerr=stds, capsize=5, color="lightgreen",
    edgecolor="black")
801     plt.ylabel("Avg Prediction Set Size")
802     plt.xlabel("Class")
803     plt.title("Average Prediction Set Size by Class")
804     plt.xticks(rotation=45, ha="right")
805     plt.grid(axis="y", linestyle="--", alpha=0.5)
806     plt.tight_layout()
807
808     # Save
809     filename = f"prediction_set_size_by_class_{test_run}__{timestamp}.png"
810     plt.savefig(os.path.join(results_directory, filename))
811     plt.show()
812
813
814 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
)




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)




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

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