

```
1 import argparse
2 import os
3 from datetime import datetime
4 from pathlib import Path
5 from collections import Counter, defaultdict
6 import json
7 import pickle
8
9
10 import numpy as np
11 import torch
12 import torchvision
13 import matplotlib.pyplot as plt
14 from torch import nn
15 from torchvision import transforms
16 from torchinfo import summary
17 from tqdm import tqdm
18 from scipy.stats import pearsonr, spearmanr
19 import torch.nn.functional as F
20 from torch.nn.functional import softmax
21 import numpy
22
23 # =====
24 # Parsing the arguments-----
25 # =====
26 parser = argparse.ArgumentParser()
27 parser.add_argument("--test_run", type=int, required=True, help="Experiment run index")
28 parser.add_argument("--vit_ckpt_path", type=str, required=True, help="Path to saved ViT checkpoint (.pth)")
29 parser.add_argument(
30     "--data_dir",
31     type=str,
32     required=True,
33     help="Path to dataset root containing train/, val/, and test/ folders"
34 )
35 args = parser.parse_args()
36
37 test_run = args.test_run
38 vit_ckpt_path = args.vit_ckpt_path
39
40 # =====
41 # Device configuration
42 # =====
43 device = "cuda" if torch.cuda.is_available() else "cpu"
44 print("#####")
45 print(f"Vision Transformer Conformal Prediction For Test Run: {test_run}")
46 print(f"The device for running the experiment is {device}")
47 print("#####")
48
49 # Basic version prints
50 print(f"torch version: {torch.__version__}")
51 print(f"torchvision version: {torchvision.__version__}")
52
53 # =====
54 # Dataset paths and summary
55 # =====
56 DATA_DIRECTORY = args.data_dir #f"/leonardo_work/IscrC_SKIDD-AI/alitariqnagi_work/training_data_split_run_{test_run}"
57 main_folder = DATA_DIRECTORY
```

```

58 splits = ['train', 'test', 'val']
59
60 summary_dict = {}
61 for split in splits:
62     split_path = os.path.join(main_folder, split)
63     total_files = 0
64     summary_dict[split] = {}
65
66     if not os.path.exists(split_path):
67         print(f"Directory not found: {split_path}")
68         continue
69
70     for disease in os.listdir(split_path):
71         disease_path = os.path.join(split_path, disease)
72         if not os.path.isdir(disease_path):
73             continue
74
75         num_files = len([
76             f for f in os.listdir(disease_path)
77             if os.path.isfile(os.path.join(disease_path, f))
78         ])
79
80         summary_dict[split][disease] = num_files
81         total_files += num_files
82
83     print(f"\n{split.upper()} - Total files: {total_files}")
84     for disease, count in summary_dict[split].items():
85         print(f" {disease}: {count} files")
86
87 train_dir = f"{DATA_DIRECTORY}/train"
88 val_dir = f"{DATA_DIRECTORY}/val"
89 test_dir = f"{DATA_DIRECTORY}/test"
90
91 # =====
92 # Data transforms and loaders
93 # =====
94 IMG_SIZE = 224
95 manual_transforms = transforms.Compose([
96     transforms.Resize((IMG_SIZE, IMG_SIZE)),
97     transforms.ToTensor(),
98 ])
99 print(f"Manually created transforms: {manual_transforms}")
100
101 from going_modular import data_setup
102 from going_modular.helper_functions import set_seeds
103 set_seeds()
104
105 BATCH_SIZE = 32
106
107 # Create an initial dataloader to get class_names
108 train_dataloader, _, class_names = data_setup.create_dataloaders(
109     train_dir=train_dir,
110     test_dir=val_dir,
111     transform=manual_transforms,
112     batch_size=BATCH_SIZE
113 )
114
115 print(f"The class names are: {class_names}")
116
117 # =====

```

```

118 # ViT model loading checkpoint
119 # =====
120 pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT
121 pretrained_vit =
122     torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)
123 # Replace classifier head
124 pretrained_vit.heads = nn.Linear(in_features=768,
125                                 out_features=len(class_names)).to(device)
126 summary(model=pretrained_vit,
127           input_size=(32, 3, 224, 224),
128           col_names=["input_size", "output_size", "num_params", "trainable"],
129           col_width=20,
130           row_settings=["var_names"])
131
132 # Get transforms from ViT weights
133 pretrained_vit_transforms = pretrained_vit_weights.transforms()
134 print(pretrained_vit_transforms)
135
136 from going_modular.data_setup import create_dataloaders_no_shuffle
137 val_dataloader_pretrained, test_dataloader_pretrained, class_names =
138     create_dataloaders_no_shuffle(
139         train_dir=val_dir,
140         test_dir=test_dir,
141         transform=pretrained_vit_transforms,
142         batch_size=32
143     )
144 # =====
145 # Load saved ViT checkpoint
146 # =====
147 print(f"[INFO] Loading pretrained ViT weights from: {vit_ckpt_path}")
148
149 # Handling DataParallel if multiple GPUs
150 if torch.cuda.device_count() > 1:
151     print(f"[INFO] Using DataParallel on {torch.cuda.device_count()} GPUs")
152     pretrained_vit = torch.nn.DataParallel(pretrained_vit)
153
154 pretrained_vit.to(device)
155
156 state_dict = torch.load(vit_ckpt_path, map_location=device)
157 try:
158     pretrained_vit.load_state_dict(state_dict)
159 except RuntimeError as e:
160     print("[WARNING] Error loading state_dict, possibly DataParallel/non-DataParallel
161 mismatch.")
162     print("Error was:", e)
163     raise
164
165 pretrained_vit.eval()
166 print("[INFO] ViT checkpoint loaded successfully.\n")
167 # =====
168 # Conformal prediction functions (unchanged)
169 # =====
170
171 # alpha will be overridden with different values in the loop
172 alpha = 0.1
173

```

```

174 def calibration_scores(model, data_loader, device):
175     model.eval()
176     scores_calibration_set = []
177
178     with torch.no_grad():
179         for X, y in tqdm(data_loader,
180                           desc="# ##### Proceeding with the Calculation of
Calibration Scores#####"):
180             X = X.to(device)
181             y = y.to(device)
182
183             y = y.cpu().numpy()
184             pred_logits = model(X)
185             cal_smx = softmax(pred_logits, dim=1).cpu().numpy()
186             scores_per_batch = 1 - cal_smx[np.arange(y.shape[0])], y
187             scores_calibration_set.extend(scores_per_batch)
188
189
190     scores_calibration_set = numpy.array(scores_calibration_set)
191     return scores_calibration_set
192
193
194 def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha,
device):
195     model.eval()
196     prediction_sets = []
197     true_labels = []
198
199     n = len(calibration_scores_array)
200     q_level = np.ceil((n + 1) * (1 - alpha)) / n
201     qhat = np.quantile(calibration_scores_array, q_level, method='higher')
202
203     with torch.no_grad():
204         for X, y in tqdm(data_loader,
205                           desc="# ##### Proceeding with the Calculation of
Testing Scores#####"):
206             X = X.to(device)
207             y = y.cpu().numpy()
208             pred_logits = model(X)
209
210             val_smx = softmax(pred_logits, dim=1).cpu().numpy()
211             masks = val_smx >= (1 - qhat)
212
213             for m in masks:
214                 prediction_sets.append(np.where(m)[0].tolist())
215
216             true_labels.extend(y.tolist())
217
218     return prediction_sets, true_labels
219
220
221 def evaluate(prediction_sets, labels):
222     number_of_correct_predictions = 0
223     for pred_set, true_label in zip(prediction_sets, labels):
224         if true_label in pred_set:
225             number_of_correct_predictions += 1
226     empirical_test_coverage = number_of_correct_predictions / len(labels)
227     average_prediction_set_size = np.mean([len(s) for s in prediction_sets])
228     return empirical_test_coverage, average_prediction_set_size
229
230

```

```

231 def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1,
232     class_names=None):
233     class_to_indices = defaultdict(list)
234     for idx, label in enumerate(true_labels):
235         class_to_indices[label].append(idx)
236
237     class_coverages = {}
238     for cls, indices in class_to_indices.items():
239         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
240         coverage = covered / len(indices)
241         class_coverages[cls] = coverage
242
243     print("\n Feature-Stratified Coverage (by True Class Label):")
244     for cls, cov in sorted(class_coverages.items()):
245         name = class_names[cls] if class_names else str(cls)
246         print(f" Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")
247
248     fsc_metric = min(class_coverages.values())
249     print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")
250
251     return fsc_metric, class_coverages
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278 def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None,
279                             title="Class-wise Coverage (FSC)"):
280     classes = list(class_coverages.keys())
281     coverages = [class_coverages[c] for c in classes]
282     labels = [class_names[c] if class_names else str(c) for c in classes]
283
284     plt.figure(figsize=(12, 6))
285     bars = plt.bar(labels, coverages, color="pink", edgecolor="black")
286
287     target = 1 - alpha
288     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
289 ({target:.2f})")
290
291     for bar, cov in zip(bars, coverages):
292         if cov < target:
293             bar.set_color('salmon')
294
295     plt.xticks(rotation=45, ha="right")
296     plt.ylim(0, 1.05)
297     plt.ylabel("Coverage")
298     plt.title(title)
299     plt.legend()
300     plt.tight_layout()
301     filename = f"Class_wise_Coverage_(FSC)_{test_run}_{timestamp}.png"
302     plt.savefig(os.path.join(results_directory, filename))
303     plt.show()
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
889

```

```

287     labels = [class_names[c] if class_names else str(c) for c in class_indices]
288     x = np.arange(len(class_indices))
289
290     fig, ax1 = plt.subplots(figsize=(12, 6))
291
292     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
293                      edgecolor='black')
293     ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
294     alpha:.2f})")
294     ax1.set_ylim(0, 1.05)
295     ax1.set_ylabel("Coverage", color='pink')
296     ax1.tick_params(axis='y', labelcolor='pink')
297
298     for bar, cov in zip(bars1, coverages):
299         if cov < (1 - alpha):
300             bar.set_color('salmon')
301
302     ax2 = ax1.twinx()
303     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
304                      color='gray', alpha=0.5)
304     ax2.set_ylabel("Class Frequency", color='gray')
305     ax2.tick_params(axis='y', labelcolor='gray')
306
307     plt.xticks(x, labels, rotation=45, ha="right")
308     plt.title(title)
309     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
310     plt.tight_layout()
311     filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
312     plt.savefig(os.path.join(results_directory, filename))
313     plt.show()
314
315
316 def plot_coverage_vs_class_frequency_with_correlation(class_coverages, true_labels,
317                                                       alpha=0.1, class_names=None,
318                                                       title="Coverage vs Class
319                                                       Frequency"):
320     class_indices = sorted(class_coverages.keys())
321     coverages = [class_coverages[c] for c in class_indices]
322
323     total = len(true_labels)
324     counts = Counter(true_labels)
325     frequencies = [counts[c] / total for c in class_indices]
326
327     labels = [class_names[c] if class_names else str(c) for c in class_indices]
328     x = np.arange(len(class_indices))
329
330     pearson_corr, pearson_p = pearsonr(frequencies, coverages)
331     spearman_corr, spearman_p = spearmanr(frequencies, coverages)
332
333     print(f"\n Correlation between class frequency and coverage:")
334     print(f" - Pearson r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
335     print(f" - Spearman p = {spearman_corr:.3f}, p = {spearman_p:.3e}")
336
337     fig, ax1 = plt.subplots(figsize=(12, 6))
338
338     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
339                      edgecolor='black')
339     ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -

```

```

340     ax1.set_ylabel("Coverage", color='pink')
341     ax1.tick_params(axis='y', labelcolor='pink')
342
343     for bar, cov in zip(bars1, coverages):
344         if cov < (1 - alpha):
345             bar.set_color('salmon')
346
347     ax2 = ax1.twinx()
348     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
349                      color='gray', alpha=0.5)
350     ax2.set_ylabel("Class Frequency", color='gray')
351     ax2.tick_params(axis='y', labelcolor='gray')
352
353     plt.xticks(x, labels, rotation=45, ha="right")
354     plt.title(title)
355     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
356     plt.tight_layout()
357     filename =
358         f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
359     plt.savefig(os.path.join(results_directory, filename))
360     plt.show()
361
362 def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1,
363                               bins=[[1], [2], [3, 4, 5], list(range(6, 100))]):
364     bin_groups = defaultdict(list)
365
366     for i, pred_set in enumerate(prediction_sets):
367         set_size = len(pred_set)
368         for bin_range in bins:
369             if set_size in bin_range:
370                 bin_groups[str(bin_range)].append(i)
371                 break
372
373     bin_coverages = {}
374     for bin_key, indices in bin_groups.items():
375         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
376         coverage = covered / len(indices) if indices else 0
377         bin_coverages[bin_key] = coverage
378
379     print("\n Size-Stratified Coverage (SSC):")
380     for bin_key, cov in bin_coverages.items():
381         print(f" Set Size {bin_key}:10s: Coverage = {cov:.3f} (Target ≥ {1 -
alpha:.2f})")
382
383     ssc = min(bin_coverages.values())
384     print(f"\n SSC Metric (min bin-wise coverage): {ssc:.3f}")
385     return ssc, bin_coverages
386
387 def plot_ssc_coverage(bin_coverages, alpha=0.1, title="Size-Stratified Coverage"):
388     bin_labels = list(bin_coverages.keys())
389     coverages = [bin_coverages[k] for k in bin_labels]
390
391     plt.figure(figsize=(10, 5))
392     bars = plt.bar(bin_labels, coverages, color='pink', edgecolor='black')
393
394     target = 1 - alpha
395     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
({target:.2f})")

```

```

396
397     for bar, cov in zip(bars, coverages):
398         if cov < target:
399             bar.set_color('grey')
400
401     plt.xticks(rotation=45, ha='right')
402     plt.ylim(0, 1.05)
403     plt.ylabel("Coverage")
404     plt.title(title)
405     plt.legend()
406     plt.tight_layout()
407     filename = f"plot_ssc_coverage_{test_run}_{timestamp}.png"
408     plt.savefig(os.path.join(results_directory, filename))
409     plt.show()
410
411
412 def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None,
413                                         results_dir="", test_run="", timestamp=""):
414     size_per_class = defaultdict(list)
415     for pred_set, true_label in zip(prediction_sets, true_labels):
416         size_per_class[true_label].append(len(pred_set))
417
418     sorted_classes = sorted(size_per_class.keys())
419     means = [np.mean(size_per_class[c]) for c in sorted_classes]
420     stds = [np.std(size_per_class[c]) for c in sorted_classes]
421     labels = [class_names[c] if class_names else str(c) for c in sorted_classes]
422
423     plt.figure(figsize=(12, 6))
424     plt.bar(labels, means, yerr=stds, capsized=5, color="lightgreen",
425             edgecolor="black")
426     plt.ylabel("Avg Prediction Set Size")
427     plt.xlabel("Class")
428     plt.title("Average Prediction Set Size by Class")
429     plt.xticks(rotation=45, ha="right")
430     plt.grid(axis="y", linestyle="--", alpha=0.5)
431     plt.tight_layout()
432
433     filename = f"prediction_set_size_by_class_{test_run}_{timestamp}.png"
434     plt.savefig(os.path.join(results_directory, filename))
435     plt.show()
436
437 def plot_calibration_score_distribution(calibration_scores, results_dir="", test_run="", timestamp ""):
438     plt.figure(figsize=(10, 5))
439     plt.hist(calibration_scores, bins=30, color="pink", edgecolor="black", alpha=0.8)
440     plt.xlabel("Nonconformity Score (1 - P(True Class))")
441     plt.ylabel("Frequency")
442     plt.title("Calibration Score Distribution")
443     plt.grid(axis='y', linestyle='--', alpha=0.6)
444     plt.tight_layout()
445
446     filename = f"calibration_score_distribution_{test_run}_{timestamp}.png"
447     plt.savefig(os.path.join(results_directory, filename))
448     plt.show()
449
450
451 # =====
452 # Multiple alpha values conformal evaluation

```

```

453 # =====
454 timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
455
456 base_results_root =
457     f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}"
458 os.makedirs(base_results_root, exist_ok=True)
459 print("Base conformal results directory:", base_results_root)
460
461 print("\n===== Computing calibration scores on validation set =====\n")
462 results_directory = base_results_root
463 scores_calibration = calibration_scores(
464     model=pretrained_vit,
465     data_loader=val_dataloader_pretrained,
466     device=device
467 )
468 np.save(
469     os.path.join(base_results_root,
470                 f"calibration_scores_experiment_run_{test_run}_{timestamp}.npy"),
471     scores_calibration
472 )
473 with open(
474     os.path.join(base_results_root,
475                 f"scores_calibration_run_{test_run}_{timestamp}.pkl"),
476     "wb"
477 ) as f:
478     pickle.dump(scores_calibration, f)
479
480 plot_calibration_score_distribution(
481     calibration_scores=scores_calibration,
482     results_dir=base_results_root,
483     test_run=test_run,
484     timestamp=timestamp
485 )
486
487 coverage_levels = np.arange(0.65, 0.95 + 1e-9, 0.05)
488 all_alpha_metrics = []
489
490 for coverage_target in coverage_levels:
491     alpha = 1.0 - coverage_target
492
493     print("=====")
494     print(f" Target coverage: {coverage_target:.2f} | alpha = {alpha:.2f}")
495     print("=====\\n")
496
497 cov_tag = int(round(coverage_target * 100))
498 results_directory = os.path.join(base_results_root, f"coverage_{cov_tag}")
499 os.makedirs(results_directory, exist_ok=True)
500
501 prediction_sets, true_labels = predict_with_conformal_sets(
502     model=pretrained_vit,
503     data_loader=test_dataloader_pretrained,
504     calibration_scores_array=scores_calibration,
505     alpha=alpha,
506     device=device
507 )
508     test_set_coverage, avg_size = evaluate(prediction_sets=prediction_sets,
509     labels=true_labels)
510     log_path = os.path.join(

```

```

511     results_directory,
512     f"logs_experiment_run_{test_run}_alpha_{alpha:.2f}_{timestamp}.txt"
513 )
514 with open(log_path, "w") as f:
515     f.write("##### Results\n")
516     f.write(f"test_run {test_run}\n")
517     f.write(f"Target coverage: {coverage_target:.3f}\n")
518     f.write(f"Alpha: {alpha:.3f}\n")
519     f.write(f"Coverage: {test_set_coverage:.3f}\n")
520     f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")
521
522 f.write("#####\n")
523 print("#####Results#####")
524 print(f"test_run {test_run}")
525 print(f"Target coverage: {coverage_target:.3f}")
526 print(f"Alpha: {alpha:.3f}")
527 print(f"Coverage: {test_set_coverage:.3f}")
528 print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
529 print("#####")
530
531 with open(
532     os.path.join(results_directory,
533                 f"prediction_data_experiment_run_{test_run}_{timestamp}.pkl"),
534     "wb"
535 ) as f:
536     pickle.dump(
537         {
538             "prediction_sets": prediction_sets,
539             "true_labels": true_labels
540         },
541         f
542     )
543
544 # FSC & related plots
545 fsc_metric, class_coverages = feature_stratified_coverage_by_class(
546     prediction_sets,
547     true_labels,
548     alpha=alpha,
549     class_names=class_names
550 )
551 plot_class_wise_coverage(class_coverages, alpha=alpha, class_names=class_names)
552 plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=alpha,
553                                   class_names=class_names)
554 plot_coverage_vs_class_frequency_with_correlation(
555     class_coverages=class_coverages,
556     true_labels=true_labels,
557     alpha=alpha,
558     class_names=class_names
559 )
560
561 # SSC & related plots
562 ssc_metric, bin_coverages = size_stratified_coverage(
563     prediction_sets,
564     true_labels,
565     alpha=alpha,
566     bins=[[1], [2], [3], list(range(4, 100))])
567 plot_ssc_coverage(bin_coverages, alpha=alpha)

```

```

568
569     # Prediction set size by class
570     plot_prediction_set_size_by_class(
571         prediction_sets=prediction_sets,
572         true_labels=true_labels,
573         class_names=class_names,
574         results_dir=results_directory,
575         test_run=test_run,
576         timestamp=timestamp
577     )
578
579     metrics = {
580         "test_run": test_run,
581         "timestamp": timestamp,
582         "alpha": alpha,
583         "target_coverage": coverage_target,
584         "test_set_coverage": test_set_coverage,
585         "average_prediction_set_size": avg_size,
586         "FSC": fsc_metric,
587         "SSC": ssc_metric,
588         "class_coverages": {str(k): float(v) for k, v in class_coverages.items()},
589         "bin_coverages": bin_coverages
590     }
591
592     with open(os.path.join(results_directory,
593         f"metrics_summary_{test_run}_alpha_{alpha:.2f}_{timestamp}.json"), "w") as f:
594         json.dump(metrics, f, indent=4)
595
596     all_alpha_metrics.append(metrics)
597
598     with open(os.path.join(base_results_root,
599                     f"metrics_summary_all_alphas_{test_run}_{timestamp}.json"),
600 "w") as f:
601         json.dump(all_alpha_metrics, f, indent=4)
602
603 print("\nFinished multi-alpha conformal evaluation for coverage 0.65-0.95 (step
604 0.05).")
605 print("All conformal results saved under:", base_results_root)
606

```