

```

1 import argparse
2 import os
3 from datetime import datetime
4 from collections import Counter, defaultdict
5 import json
6 import pickle
7
8 import numpy as np
9 import torch
10 from torch import nn
11 from torchvision import datasets, transforms, models
12 import torchvision
13 import matplotlib.pyplot as plt
14 from torchsummary import summary
15 from tqdm import tqdm
16 import pandas as pd
17 from scipy.stats import pearsonr, spearmanr
18 import torch.nn.functional as F
19 from torch.nn.functional import softmax
20 import numpy
21
22 # =====
23 # Parsing the arguments
24 # =====
25 parser = argparse.ArgumentParser()
26 parser.add_argument("--test_run", type=int, required=True, help="Experiment run
    index")
27 parser.add_argument("--resnet_ckpt_path", type=str, required=True,
28                     help="Path to saved ResNet checkpoint (.pth)")
29 args = parser.parse_args()
30
31 test_run = args.test_run
32 resnet_ckpt_path = args.resnet_ckpt_path
33
34 # =====
35 # Device configuration
36 # =====
37 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
38 print("Device:", device)
39 print("test_run", test_run)
40 print("resnet_ckpt_path:", resnet_ckpt_path)
41
42 # =====
43 # DATA LOADING
44 # =====
45 transformations = transforms.Compose([
46     transforms.Resize(255),
47     transforms.CenterCrop(224),
48     transforms.ToTensor(),
49     transforms.Normalize(mean=[0.485, 0.456, 0.406],
50                          std=[0.229, 0.224, 0.225])
51 ])
52
53 base_data_dir =
54     f"/leonardo_work/IscrC_ArtLLMs/alitariqnagi_work/training_data_split_run_{test_run}"
55
56 train_set_mix = datasets.ImageFolder(
57     os.path.join(base_data_dir, "train"),
58     transform=transformations
59 )

```

```

59 val_set_mix = datasets.ImageFolder(
60     os.path.join(base_data_dir, "val"),
61     transform=transformations
62 )
63 test_set_mix = datasets.ImageFolder(
64     os.path.join(base_data_dir, "test"),
65     transform=transformations
66 )
67
68 print("Train size:", len(train_set_mix))
69 print("Val size:", len(val_set_mix))
70 print("Test size:", len(test_set_mix))
71
72 class_names = train_set_mix.classes
73 print("Classes:", class_names)
74
75 train_loader_mix = torch.utils.data.DataLoader(
76     train_set_mix, batch_size=32, shuffle=True, num_workers=8
77 )
78 val_loader_mix = torch.utils.data.DataLoader(
79     val_set_mix, batch_size=32, shuffle=False, num_workers=8
80 )
81 test_loader_mix = torch.utils.data.DataLoader(
82     test_set_mix, batch_size=32, shuffle=False, num_workers=8
83 )
84
85 # Optional: quick visualization
86 dataiter = iter(train_loader_mix)
87 images, labels = next(dataiter)
88
89
90 def imshow(img):
91     img = img / 2 + 0.5 # unnormalize
92     npimg = img.numpy()
93     plt.imshow(np.transpose(npimg, (1, 2, 0)))
94     plt.axis("off")
95
96
97 imshow(torchvision.utils.make_grid(images))
98 plt.show()
99
100 model = models.resnet50(weights='ResNet50_Weights.IMAGENET1K_V1')
101
102 for param in model.parameters():
103     param.requires_grad = True
104
105 model.fc = nn.Sequential(
106     nn.Linear(2048, 512),
107     nn.ReLU(),
108     nn.Dropout(0.2),
109     nn.Linear(512, len(class_names))
110 )
111
112 if torch.cuda.device_count() > 1:
113     print(f"Using DataParallel Mode with cuda count {torch.cuda.device_count()}")
114     model = torch.nn.DataParallel(model)
115
116 model.to(device)
117 print(summary(model, input_size=(3, 224, 224)))
118

```

```

119 # =====
120 # LOAD FINAL CHECKPOINT (NO TRAINING HERE)
121 # =====
122 final_ckpt_path = resnet_ckpt_path
123 print("Loading final model checkpoint:", final_ckpt_path)
124
125 # Load on CPU first (safer across GPU setups)
126 state_dict = torch.load(final_ckpt_path, map_location="cpu")
127
128 # If checkpoint was saved using DataParallel, keys will start with "module."
129 if any(k.startswith("module.") for k in state_dict.keys()):
130     print("[INFO] Detected DataParallel checkpoint. Stripping 'module.' prefix...")
131     state_dict = {k.replace("module.", "", 1): v for k, v in state_dict.items()}
132
133 model.load_state_dict(state_dict, strict=True)
134
135 model.to(device)
136 model.eval()
137 print("Model loaded.\n")
138
139 # =====
140 # CONFORMAL FUNCTIONS
141 # =====
142
143 # different values will be overwritten in the loop
144 alpha = 0.1
145
146
147 #####-----
148 # Calculation of Calibration Scores
149 # -----
150 def calibration_scores(model, data_loader, device):
151     model.eval()
152
153     scores_calibration_set = []
154
155     with torch.no_grad():
156         for X, y in tqdm(data_loader,
157 desc="#####Proceeding with the Calculation of
Calibration Scores#####"):
158             X = X.to(device)
159             y = y.to(device)
160
161             y = y.cpu().numpy()
162
163             pred_logits = model(X)
164
165             cal_smx = softmax(pred_logits, dim=1).cpu().numpy()
166             scores_per_batch = 1 - cal_smx[np.arange(y.shape[0]), y]
167
168             scores_calibration_set.extend(scores_per_batch)
169
170     scores_calibration_set = numpy.array(scores_calibration_set)
171
172     return scores_calibration_set
173
174
175 #####
176 #####

```

```

177
178 #####-----
179 # Calculation of Prediction Sets
180 # -----
181 def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha,
182 device):
183     model.eval()
184
185     prediction_sets = []
186     true_labels = []
187
188     n = len(calibration_scores_array)
189
190     q_level = np.ceil((n + 1) * (1 - alpha)) / n
191     qhat = np.quantile(calibration_scores_array, q_level, method='higher')
192
193     with torch.no_grad():
194         for X, y in tqdm(data_loader,
195 desc="#####Proceeding with the Calculation of
196 Testing Scores#####"):
197             X = X.to(device)
198             y = y.cpu().numpy()
199             pred_logits = model(X)
200
201             val_smx = softmax(pred_logits, dim=1)
202             val_smx = val_smx.cpu().numpy()
203
204             masks = val_smx >= (1 - qhat)
205
206             for _ in masks:
207                 prediction_sets.append(np.where(_)[0].tolist())
208
209                 true_labels.extend(y.tolist())
210
211     return prediction_sets, true_labels
212 #####
213 #####
214
215 #####-----
216 # EVALUATION
217 # -----
218 def evaluate(prediction_sets, labels):
219     number_of_correct_predictions = 0
220     for pred_set, true_label in zip(prediction_sets, labels):
221         if true_label in pred_set:
222             number_of_correct_predictions += 1
223     empirical_test_coverage = number_of_correct_predictions / len(labels)
224     average_prediction_set_size = np.mean([len(s) for s in prediction_sets])
225     return empirical_test_coverage, average_prediction_set_size
226
227 #####
228 #####
229
230
231 def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1,
class_names=None):

```

```

232 class_to_indices = defaultdict(list)
233 for idx, label in enumerate(true_labels):
234     class_to_indices[label].append(idx)
235
236 class_coverages = {}
237 for cls, indices in class_to_indices.items():
238     covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
239     coverage = covered / len(indices)
240     class_coverages[cls] = coverage
241
242 print("\n Feature-Stratified Coverage (by True Class Label):")
243 for cls, cov in sorted(class_coverages.items()):
244     name = class_names[cls] if class_names else str(cls)
245     print(f"   Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")
246
247 fsc_metric = min(class_coverages.values())
248 print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")
249
250 return fsc_metric, class_coverages
251
252
253 def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None,
254 title="Class-wise Coverage (FSC)":
255     classes = list(class_coverages.keys())
256     coverages = [class_coverages[c] for c in classes]
257
258     labels = [class_names[c] if class_names else str(c) for c in classes]
259
260     plt.figure(figsize=(12, 6))
261     bars = plt.bar(labels, coverages, color="pink", edgecolor="black")
262
263     target = 1 - alpha
264     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage ({target:.2f})")
265
266     for bar, cov in zip(bars, coverages):
267         if cov < target:
268             bar.set_color('salmon')
269
270     plt.xticks(rotation=45, ha="right")
271     plt.ylim(0, 1.05)
272     plt.ylabel("Coverage")
273     plt.title(title)
274     plt.legend()
275     plt.tight_layout()
276     filename = f"Class_wise_Coverage_(FSC)_{test_run}_{timestamp}.png"
277     plt.savefig(os.path.join(results_directory, filename))
278     plt.show()
279
280 def plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=0.1,
281 class_names=None,
282                                     title="Coverage vs Class Frequency"):
283     class_indices = sorted(class_coverages.keys())
284     coverages = [class_coverages[c] for c in class_indices]
285
286     total = len(true_labels)
287     counts = Counter(true_labels)
288     frequencies = [counts[c] / total for c in class_indices]

```

```

289     labels = [class_names[c] if class_names else str(c) for c in class_indices]
290     x = np.arange(len(class_indices))
291
292     fig, ax1 = plt.subplots(figsize=(12, 6))
293
294     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
295                    edgecolor='black')
296     ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
297                    alpha:.2f})")
298     ax1.set_ylim(0, 1.05)
299     ax1.set_ylabel("Coverage", color='pink')
300     ax1.tick_params(axis='y', labelcolor='pink')
301
302     for bar, cov in zip(bars1, coverages):
303         if cov < (1 - alpha):
304             bar.set_color('salmon')
305
306     ax2 = ax1.twinx()
307     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
308                    color='gray', alpha=0.5)
309     ax2.set_ylabel("Class Frequency", color='gray')
310     ax2.tick_params(axis='y', labelcolor='gray')
311
312     plt.xticks(x, labels, rotation=45, ha="right")
313     plt.title(title)
314     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
315     plt.tight_layout()
316     filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
317     plt.savefig(os.path.join(results_directory, filename))
318     plt.show()
319
320 def plot_coverage_vs_class_frequency_with_correlation(class_coverages, true_labels,
321 alpha=0.1, class_names=None,
322                                                    title="Coverage vs Class
323 Frequency"):
324     class_indices = sorted(class_coverages.keys())
325     coverages = [class_coverages[c] for c in class_indices]
326
327     total = len(true_labels)
328     counts = Counter(true_labels)
329     frequencies = [counts[c] / total for c in class_indices]
330
331     labels = [class_names[c] if class_names else str(c) for c in class_indices]
332     x = np.arange(len(class_indices))
333
334     pearson_corr, pearson_p = pearsonr(frequencies, coverages)
335     spearman_corr, spearman_p = spearmanr(frequencies, coverages)
336
337     print(f"\n Correlation between class frequency and coverage:")
338     print(f" - Pearson r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
339     print(f" - Spearman p = {spearman_corr:.3f}, p = {spearman_p:.3e}")
340
341     fig, ax1 = plt.subplots(figsize=(12, 6))
342
343     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
344                    edgecolor='black')
345     ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
346                    alpha:.2f})")
347     ax1.set_ylim(0, 1.05)

```

```

342     ax1.set_ylabel("Coverage", color='pink')
343     ax1.tick_params(axis='y', labelcolor='pink')
344
345     for bar, cov in zip(bars1, coverages):
346         if cov < (1 - alpha):
347             bar.set_color('salmon')
348
349     ax2 = ax1.twinx()
350     bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
351 color='gray', alpha=0.5)
352     ax2.set_ylabel("Class Frequency", color='gray')
353     ax2.tick_params(axis='y', labelcolor='gray')
354
355     plt.xticks(x, labels, rotation=45, ha="right")
356     plt.title(title)
357     fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
358     plt.tight_layout()
359     filename =
360     f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
361     plt.savefig(os.path.join(results_directory, filename))
362     plt.show()
363
364 def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1,
365                             bins=[[1], [2], [3, 4, 5], list(range(6, 100))]):
366     bin_groups = defaultdict(list)
367
368     for i, pred_set in enumerate(prediction_sets):
369         set_size = len(pred_set)
370         for bin_range in bins:
371             if set_size in bin_range:
372                 bin_groups[str(bin_range)].append(i)
373                 break
374
375     bin_coverages = {}
376     for bin_key, indices in bin_groups.items():
377         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
378         coverage = covered / len(indices) if indices else 0
379         bin_coverages[bin_key] = coverage
380
381     print("\n Size-Stratified Coverage (SSC):")
382     for bin_key, cov in bin_coverages.items():
383         print(f" Set Size {bin_key:10s}: Coverage = {cov:.3f} (Target ≥ {1 -
384 alpha:.2f})")
385
386     ssc = min(bin_coverages.values())
387     print(f"\n SSC Metric (min bin-wise coverage): {ssc:.3f}")
388     return ssc, bin_coverages
389
390 def plot_ssc_coverage(bin_coverages, alpha=0.1, title="Size-Stratified Coverage"):
391     bin_labels = list(bin_coverages.keys())
392     coverages = [bin_coverages[k] for k in bin_labels]
393
394     plt.figure(figsize=(10, 5))
395     bars = plt.bar(bin_labels, coverages, color='pink', edgecolor='black')
396
397     target = 1 - alpha
398     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
399 ({target:.2f})")

```

```

398
399     for bar, cov in zip(bars, coverages):
400         if cov < target:
401             bar.set_color('grey')
402
403     plt.xticks(rotation=45, ha='right')
404     plt.ylim(0, 1.05)
405     plt.ylabel("Coverage")
406     plt.title(title)
407     plt.legend()
408     plt.tight_layout()
409     filename = f"plot_ssc_coverage_{test_run}__{timestamp}.png"
410     plt.savefig(os.path.join(results_directory, filename))
411     plt.show()
412
413
414 def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None,
415                                     results_dir="", test_run="",
416                                     timestamp=""):
417     size_per_class = defaultdict(list)
418     for pred_set, true_label in zip(prediction_sets, true_labels):
419         size_per_class[true_label].append(len(pred_set))
420
421     sorted_classes = sorted(size_per_class.keys())
422     means = [np.mean(size_per_class[c]) for c in sorted_classes]
423     stds = [np.std(size_per_class[c]) for c in sorted_classes]
424     labels = [class_names[c] if class_names else str(c) for c in sorted_classes]
425
426     plt.figure(figsize=(12, 6))
427     plt.bar(labels, means, yerr=stds, capsize=5, color="lightgreen",
428            edgecolor="black")
429     plt.ylabel("Avg Prediction Set Size")
430     plt.xlabel("Class")
431     plt.title("Average Prediction Set Size by Class")
432     plt.xticks(rotation=45, ha="right")
433     plt.grid(axis="y", linestyle="--", alpha=0.5)
434     plt.tight_layout()
435
436     filename = f"prediction_set_size_by_class_{test_run}__{timestamp}.png"
437     plt.savefig(os.path.join(results_directory, filename))
438     plt.show()
439
440
441 def plot_calibration_score_distribution(calibration_scores, results_dir="",
442                                     test_run="", timestamp=""):
443     plt.figure(figsize=(10, 5))
444     plt.hist(calibration_scores, bins=30, color="pink", edgecolor="black", alpha=0.8)
445     plt.xlabel("Nonconformity Score (1 - P(True Class))")
446     plt.ylabel("Frequency")
447     plt.title("Calibration Score Distribution")
448     plt.grid(axis="y", linestyle="--", alpha=0.6)
449     plt.tight_layout()
450
451     filename = f"calibration_score_distribution_{test_run}__{timestamp}.png"
452     plt.savefig(os.path.join(results_directory, filename))
453
454     plt.show()
455
456 # =====

```



```

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

```

```

508         f.write("##### Results
#####\n")
509         f.write(f"test_run {test_run}\n")
510         f.write(f"Target coverage: {coverage_target:.3f}\n")
511         f.write(f"Alpha: {alpha:.3f}\n")
512         f.write(f"Coverage: {test_set_coverage:.3f}\n")
513         f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")
514
f.write("#####\n")
515
516     print("#####Results:#####")
517     print(f"test_run {test_run}")
518     print(f"Target coverage: {coverage_target:.3f}")
519     print(f"Alpha: {alpha:.3f}")
520     print(f"Coverage: {test_set_coverage:.3f}")
521     print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
522     print("#####")
523
524     set_sizes = [len(prediction_set) for prediction_set in prediction_sets]
525     counts = Counter(set_sizes)
526     total = sum(counts.values())
527     sizes = sorted(counts.keys())
528     frequencies = [counts[size] for size in sizes]
529     percentages = [100 * freq / total for freq in frequencies]
530
531     data = [{'size': len(prediction_set), 'correct': int(true in prediction_set)}
532             for prediction_set, true in zip(prediction_sets, true_labels)]
533     df = pd.DataFrame(data)
534     coverage_stats = df.groupby('size').agg({'correct': ['mean', 'sum', 'count']})
535     coverage_stats.columns = ['coverage', 'correct_count', 'total_count']
536     coverage_stats = coverage_stats.reindex(sizes, fill_value=0)
537
538     cumulative_correct = coverage_stats['correct_count'].cumsum()
539     cumulative_coverage = 100 * cumulative_correct / len(true_labels)
540     max_freq = max(frequencies)
541     cumulative_scaled = (cumulative_coverage / 100) * max_freq
542
543     fig, ax = plt.subplots(figsize=(13, 6))
544     bars = ax.bar(sizes, frequencies, color='pink', edgecolor='black',
label='Frequency', alpha=0.7)
545     ax.plot(sizes, cumulative_scaled, color='red', marker='o', linestyle='-',
546             alpha=0.4, linewidth=2, label='Cumulative Coverage')
547
548     for bar, size, pct in zip(bars, sizes, percentages):
549         height = bar.get_height()
550         count = counts[size]
551         coverage_val = coverage_stats.loc[size, 'coverage'] * 100
552         ax.annotate(f'{{count}} ({{pct:.1f}}%)\nCov: {coverage_val:.1f}%',
553                     xy=(bar.get_x() + bar.get_width() / 2, height / 2),
554                     ha='center', va='center', fontsize=9)
555
556     for bar, cum_cov in zip(bars, cumulative_coverage):
557         height = bar.get_height()
558         ax.annotate(f'CumCov: {cum_cov:.1f}%',
559                     xy=(bar.get_x() + bar.get_width() / 2, height),
560                     xytext=(0, 8),
561                     textcoords="offset points",
562                     ha='center', va='bottom', fontsize=9, color='darkorange')
563
564     mean_size = np.mean(set_sizes)

```

```

565     median_size = np.median(set_sizes)
566     ax.axvline(mean_size, color='gray', linestyle='--', label=f'Mean =
{mean_size:.2f}')
567     ax.axvline(median_size, color='orange', linestyle='--', label=f'Median =
{median_size:.2f}')
568
569     ax.set_xlabel("Prediction Set Size")
570     ax.set_ylabel("Frequency")
571     ax.set_title(f"Prediction Set Size Distribution with Coverage (Test Set), alpha=
{alpha:.2f}")
572     ax.set_xticks(sizes)
573     ax.grid(axis='y', linestyle='--', alpha=0.6)
574     ax.legend(loc='upper right')
575
576     plt.tight_layout()
577     plt.savefig(os.path.join(
578         results_directory,
579         f"Prediction_Set_Size_Distribution_with_Coverage_(Test_Set)_{test_run}_alpha_{alpha:
.2f}_{timestamp}.png"
580     ))
581     plt.show()
582
583     fsc_metric, per_class_cov = feature_stratified_coverage_by_class(
584         prediction_sets,
585         true_labels,
586         alpha=alpha,
587         class_names=class_names
588     )
589
590     fsc_metric, class_coverages = feature_stratified_coverage_by_class(
591         prediction_sets,
592         true_labels,
593         alpha=alpha,
594         class_names=class_names
595     )
596     plot_class_wise_coverage(class_coverages, alpha=alpha, class_names=class_names)
597
598     fsc_metric, class_coverages = feature_stratified_coverage_by_class(
599         prediction_sets,
600         true_labels,
601         alpha=alpha,
602         class_names=class_names
603     )
604     plot_coverage_vs_class_frequency(class_coverages, true_labels,
605                                     alpha=alpha, class_names=class_names)
606
607     plot_coverage_vs_class_frequency_with_correlation(
608         class_coverages=class_coverages,
609         true_labels=true_labels,
610         alpha=alpha,
611         class_names=class_names
612     )
613
614     ssc_metric, bin_coverages = size_stratified_coverage(
615         prediction_sets,
616         true_labels,
617         alpha=alpha,
618         bins=[[1], [2], [3], list(range(4, 100))]]
619     )

```

```

620
621     ssc_metric, bin_coverages = size_stratified_coverage(
622         prediction_sets,
623         true_labels,
624         alpha=alpha,
625         bins=[[1], [2], [3], list(range(4, 100))]
626     )
627     plot_ssc_coverage(bin_coverages, alpha=alpha)
628
629     plot_prediction_set_size_by_class(
630         prediction_sets=prediction_sets,
631         true_labels=true_labels,
632         class_names=class_names,
633         results_dir=results_directory,
634         test_run=test_run,
635         timestamp=timestamp
636     )
637
638     metrics = {
639         "test_run": test_run,
640         "timestamp": timestamp,
641         "alpha": alpha,
642         "target_coverage": coverage_target,
643         "test_set_coverage": test_set_coverage,
644         "average_prediction_set_size": avg_size,
645         "FSC": fsc_metric,
646         "SSC": ssc_metric,
647         "class_coverages": {str(k): float(v) for k, v in class_coverages.items()},
648         "bin_coverages": bin_coverages
649     }
650
651     with open(os.path.join(results_directory,
652         f"metrics_summary_{test_run}_alpha_{alpha:.2f}_{timestamp}.json"), "w") as f:
653         json.dump(metrics, f, indent=4)
654
655     all_alpha_metrics.append(metrics)
656
657     with open(os.path.join(base_results_root,
658         f"metrics_summary_all_alphas_{test_run}_{timestamp}.json"),
659         "w") as f:
660         json.dump(all_alpha_metrics, f, indent=4)
661
662     print("\nFinished multi-alpha conformal evaluation for alphas corresponding to
663         coverage 0.65-0.95.")
664     print("Results saved under:", base_results_root)
665

```