

Lib/evaluation.py

```
1 # =====
2 # Ground-truth metrics: Dice and clDice
3 # =====
4
5 import numpy as np
6 import vtk
7 import slicer
8
9 from scipy.spatial.distance import dice as _diceDissimilarity
10 from scipy.ndimage import label as ndlabel, distance_transform_edt
11
12 from skimage.morphology import skeletonize
13 from scipy.ndimage import convolve
14 from scipy.spatial.distance import pdist
15
16 # from skan import Skeleton, summarize
17
18
19 # clDICE from MIT licensed https://github.com/jocpae/clDice
20 from clDice.clDice_metric.clDice import clDice
21
22 # TODO: Look at Pyvane for tortuosity
23 # License under CollaboratiVessel/Lib/pyvane/LICENSE.txt
24
25 # Implemented via claude following an expansion to the DICE and clDICE metrics
26 def _confusionStats(pred, gt):
27     """Confusion-matrix counts plus derived rates.
28
29     `gt` is the reference. Useful when the prediction over- or
30     under-segments: precision/recall split tells direction, IoU is a
31     stricter overlap than Dice, volume_ratio is pred/gt size in voxels.
32     """
33     pred = pred.astype(bool, copy=False)
```

```
34 gt = gt.astype(bool, copy=False)
35 tp = int(np.logical_and(pred, gt).sum())
36 fp = int(np.logical_and(pred, ~gt).sum())
37 fn = int(np.logical_and(~pred, gt).sum())
38 tn = int(np.logical_and(~pred, ~gt).sum())
39 p_n = tp + fp
40 g_n = tp + fn
41 return {
42     "tp": tp, "fp": fp, "fn": fn, "tn": tn,
43     "precision": tp / p_n if p_n > 0 else float("nan"),
44     "recall": tp / g_n if g_n > 0 else float("nan"),
45     "specificity": tn / (tn + fp) if (tn + fp) > 0 else float("nan"),
46     "iou": tp / (tp + fp + fn) if (tp + fp + fn) > 0 else float("nan"),
47     "volume_ratio": p_n / g_n if g_n > 0 else float("nan"),
48 }
49
50
51 # Implemented via claude following an expansion to the DICE and cLDICE metrics
52 def _centerlineStats(mask, mean_spacing_mm, referenceMask=None):
53     """Obtain the statistics of a prediction in vessel length
54     Skeletonise and return total / per-branch length.
55     DANGER: assumes (near) isotropic voxels -> in testing they were 6x6x6µm
56
57     A 'branch' here is a connected component of the 3D skeleton under
58     26-neighbour connectivity (cheap proxy for centerline segments).
59     Length is approximated by skeleton voxel count x mean axis spacing
60
61     If a referenceMask is provided, returns 'matching_length_mm':
62     the length of skeleton voxels that fall inside the reference/GT
63     Discounts centerline picked up in false-positive
64     regions – important when comparing methods that over-segment.
65     """
66     out = {"length_mm": 0.0, "n_branches": 0,
67           "mean_branch_length_mm": float("nan")}
68     if referenceMask is not None:
69         out["matching_length_mm"] = 0.0
```

```
70     if not mask.any():
71         return out
72     skel = skeletonize(
73         mask.astype(np.uint8)).astype(bool)
74     total = float(skel.sum()) * mean_spacing_mm
75     _labeled, n = ndlabel(skel,
76                          structure=np.ones((3, 3, 3), dtype=np.uint8))
77     out["length_mm"] = total
78     out["n_branches"] = int(n)
79     out["mean_branch_length_mm"] = total / n if n > 0 else float("nan")
80     if referenceMask is not None:
81         matching = float((skel & referenceMask.astype(bool)).sum()) * mean_spacing_mm
82         out["matching_length_mm"] = matching
83     return out
84
85
86
87 def perVesselStats(predMask, gtMask, spacing_mm):
88     """For each connected component in predMask, compute volume, length, and
89     the list of GT component IDs it overlaps with (the "correspondence").
90
91     A predicted vessel "corresponds" to a GT vessel if they share at least
92     one voxel. The output is a list of GT component IDs per prediction –
93     one ID for 1:1 match, multiple IDs when the prediction bridges what
94     GT considers separate vessels, empty list when the prediction is FP.
95
96     Args:
97         predMask: 3D binary array (prediction or threshold result).
98         gtMask: 3D binary array (ground truth, OR another reference like
99                thresholding when no GT is available).
100         spacing_mm: (x, y, z) voxel size in mm.
101
102     Returns:
103         List of dicts, one per predMask component:
104             vessel_id, volume_voxels, length_mm, overlap_voxels,
105             corresponding_ids (list of int), n_corresponding (int)
```

```
106     """
107
108     if not predMask.any():
109         return []
110
111     spacing_arr = np.array(spacing_mm)
112     mean_spacing = float(np.mean(spacing_arr))
113
114     pred_labeled, n_pred = ndlabel(predMask)
115     if n_pred == 0:
116         return []
117
118     # Label the reference (GT or threshold) so we can identify which reference
119     # vessel each prediction component overlaps with.
120     if gtMask is not None and gtMask.any():
121         gt_labeled, _ = ndlabel(gtMask)
122     else:
123         gt_labeled = np.zeros_like(pred_labeled)
124
125     # Updated?
126     # if gtMask is not None and gtMask.any():
127     #     overlap_voxels = int((comp_mask & gtMask).sum())
128     # else:
129     #     overlap_voxels = 0
130
131     # Skeletonize the whole prediction once, then index per component
132     pred_skel = skeletonize(predMask.astype(np.uint8)).astype(bool)
133
134     results = []
135     for vid in range(1, n_pred + 1):
136         comp_mask = (pred_labeled == vid)
137         if not comp_mask.any():
138             continue
139
140         comp_skel = comp_mask & pred_skel
141         # NOTE: voxel count x mean spacing for length can undercount
```

```
142     # But we are leveraging that all vessels follow a similar profile
143     # i.e. that there are no huge balls
144     length_mm     = float(comp_skel.sum() * mean_spacing)
145     volume_vox    = int(comp_mask.sum())
146
147     # Which GT (or reference) component IDs does this prediction touch?
148     overlap_vals = gt_labeled[comp_mask]
149     gt_ids_here  = overlap_vals[overlap_vals > 0]
150     corresponding_ids = sorted(set(int(x) for x in gt_ids_here))
151     overlap_voxels    = int(gt_ids_here.size)
152
153     results.append({
154         "vessel_id":      vid,
155         "volume_voxels":  volume_vox,
156         "length_mm":      length_mm,
157         "overlap_voxels": overlap_voxels,
158         "corresponding_ids": ";".join(str(c) for c in corresponding_ids),
159         "n_corresponding": len(corresponding_ids),
160     })
161
162     return results
163
164 # # Based on https://github.com/chcomin/pyvane
165 # # as in https://pubmed.ncbi.nlm.nih.gov/35620183/
166 # # Helped implementation using claude
167 # def perVesselStats(mask, gtMask, spacing_mm):
168 #     """For each connected component in `mask`, return per-vessel metrics
169 #     using skan for skeleton analysis.
170 #     """
171
172 #     if not mask.any():
173 #         return []
174
175 #     labeled, n_comp = ndlabel(mask)
176 #     if n_comp == 0:
177 #         return []
```

```
178
179 #     skel = skeletonize(mask.astype(np.uint8)).astype(bool)
180 #     dist = distance_transform_edt(mask, sampling=spacing_mm)
181
182 #     results = []
183 #     for vid in range(1, n_comp + 1):
184 #         comp_mask = (labeled == vid)
185 #         comp_skel = comp_mask & skel
186 #         if not comp_skel.any():
187 #             continue
188
189 #         # Use skan for length and tortuosity
190 #         try:
191 #             sk_obj = Skeleton(comp_skel.astype(np.uint8), spacing=spacing_mm)
192 #             df = summarize(sk_obj)
193 #             # branch-distance is the path length along the skeleton (mm)
194 #             # euclidean-distance is the straight-line endpoint distance (mm)
195 #             total_length = float(df["branch-distance"].sum())
196
197 #             # Tortuosity averaged across branches, weighted by branch length
198 #             tort_per_branch = df["branch-distance"] / df["euclidean-distance"].replace(0, np.nan)
199
200 #             mean_tortuosity = float(
201 #                 (tort_per_branch * df["branch-distance"]).sum()
202 #                 / df["branch-distance"].sum()
203 #                 ) if df["branch-distance"].sum() > 0 else float("nan")
204
205 #         except Exception as e:
206 #             print(f"Error in vessel length {e}")
207 #             # Fallback for single-voxel or degenerate skeletons skan can't handle
208 #             total_length = float(comp_skel.sum() * np.mean(spacing_mm))
209 #             mean_tortuosity = float("nan")
210
211 #     volume_vox = int(comp_mask.sum())
212 #     radii = dist[comp_skel]
213 #     mean_radius_mm = float(np.mean(radii)) if radii.size > 0 else 0.0
```

```
214
215 #         overlap_mask = comp_mask & gtMask
216 #         overlap_voxels = int(overlap_mask.sum())
217 #         gt_overlap_length_mm = float(
218 #             (comp_skel & gtMask).sum() * np.mean(spacing_mm))
219
220 #         results.append({
221 #             "vessel_id":          vid,
222 #             "length_mm":          total_length,
223 #             "volume_voxels":      volume_vox,
224 #             "mean_radius_mm":     mean_radius_mm,
225 #             "overlap_voxels":     overlap_voxels,
226 #             "gt_overlap_length_mm": gt_overlap_length_mm,
227 #             "tortuosity":         mean_tortuosity,
228 #         })
229
230 #     return results
231
232
233 def computeDice(pred, gt):
234     """DICE from scipy"""
235     pred = pred.astype(bool, copy=False)
236     gt = gt.astype(bool, copy=False)
237     if not pred.any() and not gt.any():
238         return 1.0
239     # scipy returns the Dice dissimilarity (1 - DSC) on flattened bool arrays.
240     return 1.0 - float(_diceDissimilarity(pred.ravel(), gt.ravel()))
241
242
243 # cLDICE from Lib
244 def computeCLDice(pred, gt):
245     pred = pred.astype(bool, copy=False)
246     gt = gt.astype(bool, copy=False)
247     # Wrapper for empty or full edge case
248     if not pred.any() and not gt.any():
249         return 1.0
```

```
250     if not pred.any() or not gt.any():
251         return 0.0
252
253     result = clDice(pred, gt)
254     # Catch another empty case
255     if not np.isfinite(result):
256         return 0.0
257     return float(result)
258
259
260 # Get the chunk location as encoded in DICOM - necessary because segmentations are
261 # subregions of the main region due to time limitations
262 def _chunkOriginInSourceIJK(chunkVolumeNode, sourceVolumeNode):
263     rasOrigin = chunkVolumeNode.GetOrigin()
264     m = vtk.vtkMatrix4x4()
265     sourceVolumeNode.GetRASToIJKMatrix(m)
266     h = m.MultiplyPoint([rasOrigin[0], rasOrigin[1], rasOrigin[2], 1.0])
267     return int(round(h[0])), int(round(h[1])), int(round(h[2]))
268
269
270 # Evaluation made with assistane from Claude to handle cases:
271 # - Empty chunk
272 # - Dislocated chunk
273 # - Multiple chunks
274 # GT is also trimmed to the size of the prediction to avoid large empty zones
275 #
276 # When 'threshold' is provided, three comparisons are produced per chunk:
277 # 1) predicted vs GT (the pipeline's output, always computed)
278 # 2) threshold vs GT (simple grey-value thresholding baseline)
279 # 3) predicted vs threshold (pipeline output against the baseline)
280 # The threshold mask is all points over the greyval thresh for the full volume
281
282 def evaluateAgainstChunks(chunkVolumeNodes, sourceVolumeNode,
283                           predictedMask, gt_margin=5, threshold=None):
284     results = []
285     src_shape = predictedMask.shape
```

```
286
287 # Source array is needed for per-chunk intensity stats (median/std over
288 # GT region) and for the optional threshold mask.
289 src_arr = slicer.util.arrayFromVolume(sourceVolumeNode)
290 if src_arr.ndim == 4:
291     src_arr = src_arr[..., 0]
292
293 # Build the binary threshold mask
294 thresholdMask = None
295 src_min = src_max = None
296 if threshold is not None:
297     thresholdMask = (src_arr > float(threshold))
298     src_min = float(src_arr.min())
299     src_max = float(src_arr.max())
300 # Shape sanity check: thresholdMask must align with predictedMask
301 # so the same crop indices apply to both.
302 if thresholdMask.shape != predictedMask.shape:
303     thresholdMask = None
304
305
306 # Iterate over multiple chunks, but in practice we do 1 chunk
307 # Continue used to skip over empty chunks
308 for node in chunkVolumeNodes:
309     arr = slicer.util.arrayFromVolume(node)
310     if arr.ndim == 4:
311         arr = arr[..., 0]
312     gt_full = (arr > 0)
313
314 # Empty gt
315 if not gt_full.any():
316     results.append({
317         "name": node.GetName(),
318         "dice": float("nan"), "cldice": float("nan"),
319         "gt_voxels": 0, "pred_voxels": 0,
320         "shape": tuple(gt_full.shape),
321         "note": "annotation has no positive voxels",
```

```
322         })
323         continue
324
325
326     zs, ys, xs = np.where(gt_full)
327     c_z0 = max(int(zs.min()) - gt_margin, 0)
328     c_z1 = min(int(zs.max()) + 1 + gt_margin, gt_full.shape[0])
329     c_y0 = max(int(ys.min()) - gt_margin, 0)
330     c_y1 = min(int(ys.max()) + 1 + gt_margin, gt_full.shape[1])
331     c_x0 = max(int(xs.min()) - gt_margin, 0)
332     c_x1 = min(int(xs.max()) + 1 + gt_margin, gt_full.shape[2])
333
334
335     gt_block = gt_full[c_z0:c_z1, c_y0:c_y1, c_x0:c_x1]
336
337     i_off, j_off, k_off = _chunkOriginInSourceIJK(
338         node, sourceVolumeNode
339     )
340     ks = max(c_z0 + k_off, 0)
341     js = max(c_y0 + j_off, 0)
342     xs_ = max(c_x0 + i_off, 0)
343     ke = min(c_z0 + k_off + (c_z1 - c_z0), src_shape[0])
344     je = min(c_y0 + j_off + (c_y1 - c_y0), src_shape[1])
345     xe = min(c_x0 + i_off + (c_x1 - c_x0), src_shape[2])
346
347     # Dislocated gt
348     if ks >= ke or js >= je or xs_ >= xe:
349         results.append({
350             "name": node.GetName(),
351             "dice": float("nan"), "cldice": float("nan"),
352             "gt_voxels": int(gt_block.sum()), "pred_voxels": 0,
353             "shape": tuple(gt_block.shape),
354             "note": "GT region falls outside the source volume",
355         })
356     continue
357
```

```
358 pred_block = predictedMask[ks:ke, js:je, xs_:xe]
359 gt_trim = gt_block[
360     (ks - (c_z0 + k_off)):(ks - (c_z0 + k_off)) + (ke - ks),
361     (js - (c_y0 + j_off)):(js - (c_y0 + j_off)) + (je - js),
362     (xs_ - (c_x0 + i_off)):(xs_ - (c_x0 + i_off)) + (xe - xs_),
363 ]
364
365 d = computeDice(pred_block, gt_trim)
366 c = computeClDice(pred_block, gt_trim)
367 conf_pg = _confusionStats(pred_block, gt_trim)
368
369 mean_sp = float(np.mean(sourceVolumeNode.GetSpacing()))
370
371 # Source intensity over the GT-positive voxels of this chunk:
372 # tells us what vessels look like in the raw data (useful for
373 # picking a sensible threshold value).
374 src_block = src_arr[ks:ke, js:je, xs_:xe]
375 gt_intensities = src_block[gt_trim]
376 if gt_intensities.size > 0:
377     median_intensity = float(np.median(gt_intensities))
378     std_intensity = float(np.std(gt_intensities))
379 else:
380     median_intensity = float("nan")
381     std_intensity = float("nan")
382
383 entry = {
384     "name": node.GetName(),
385     "dice": d, "cldice": c,
386     "precision": conf_pg["precision"],
387     "recall": conf_pg["recall"],
388     "specificity": conf_pg["specificity"],
389     "iou": conf_pg["iou"],
390     "volume_ratio": conf_pg["volume_ratio"],
391     "tp": conf_pg["tp"], "fp": conf_pg["fp"],
392     "fn": conf_pg["fn"], "tn": conf_pg["tn"],
393     "gt_voxels": conf_pg["tp"] + conf_pg["fn"],
```

```
394     "pred_voxels": conf_pg["tp"] + conf_pg["fp"],
395     "median_intensity": median_intensity,
396     "std_intensity": std_intensity,
397     "shape": tuple(gt_trim.shape),
398     "source_ijk": (xs_, js, ks, xe, je, ke),
399     # Centerline stats with referenceMask=gt_trim so we also get
400     # matching_length_mm (skeleton n GT) – discounts FP centerline.
401     "pred_centerline": _centerlineStats(pred_block, mean_sp,
402                                         referenceMask=gt_trim),
403     "gt_centerline": _centerlineStats(gt_trim, mean_sp),
404 }
405
406 if thresholdMask is not None:
407     thr_block = thresholdMask[ks:ke, js:je, xs_:xe]
408     entry["threshold_value"] = float(threshold)
409     entry["thr_voxels"] = int(thr_block.sum())
410     entry["dice_thr_vs_gt"] = computeDice(thr_block, gt_trim)
411     entry["cldice_thr_vs_gt"] = computeClDice(thr_block, gt_trim)
412     entry["dice_pred_vs_thr"] = computeDice(pred_block, thr_block)
413     entry["cldice_pred_vs_thr"] = computeClDice(pred_block, thr_block)
414     entry["thr_vs_gt"] = _confusionStats(thr_block, gt_trim)
415     entry["pred_vs_thr"] = _confusionStats(pred_block, thr_block)
416     entry["thr_centerline"] = _centerlineStats(thr_block, mean_sp,
417                                                referenceMask=gt_trim)
418
419 results.append(entry)
420
421 valid = [r for r in results if not np.isnan(r.get("dice", float("nan")))]
422
423 def _meanOver(seq):
424     seq = [x for x in seq if x is not None and not (isinstance(x, float) and np.isnan(x))]
425     return float(np.mean(seq)) if seq else float("nan")
426
427 summary = {
428     "per_chunk": results,
429     "n": len(valid),
```

```
430     "mean_dice":         _meanOver(r["dice"]         for r in valid),
431     "mean_cldice":       _meanOver(r["cldice"]       for r in valid),
432     "mean_iou":          _meanOver(r["iou"]          for r in valid),
433     "mean_recall":       _meanOver(r["recall"]       for r in valid),
434     "mean_precision":    _meanOver(r["precision"]    for r in valid),
435     "mean_volume_ratio": _meanOver(r["volume_ratio"] for r in valid),
436     "mean_pred_length_mm": _meanOver(r["pred_centerline"]["length_mm"] for r in valid),
437     "mean_gt_length_mm":  _meanOver(r["gt_centerline"]["length_mm"]  for r in valid),
438 }
439
440 if thresholdMask is not None:
441     thr_valid = [r for r in valid if "dice_thr_vs_gt" in r]
442     summary["threshold_value"] = float(threshold)
443     summary["total_thr_voxels"] = int(thresholdMask.sum())
444     summary["src_value_range"] = (src_min, src_max)
445     summary["mean_dice_thr_vs_gt"] = _meanOver(r["dice_thr_vs_gt"] for r in thr_valid)
446     summary["mean_cldice_thr_vs_gt"] = _meanOver(r["cldice_thr_vs_gt"] for r in thr_valid)
447     summary["mean_dice_pred_vs_thr"] = _meanOver(r["dice_pred_vs_thr"] for r in thr_valid)
448     summary["mean_cldice_pred_vs_thr"] = _meanOver(r["cldice_pred_vs_thr"] for r in thr_valid)
449     summary["mean_iou_thr_vs_gt"] = _meanOver(r["thr_vs_gt"]["iou"] for r in thr_valid)
450     summary["mean_volume_ratio_thr_vs_gt"] = _meanOver(r["thr_vs_gt"]["volume_ratio"] for r in thr_valid)
451     summary["mean_recall_thr_vs_gt"] = _meanOver(r["thr_vs_gt"]["recall"] for r in thr_valid)
452     summary["mean_precision_thr_vs_gt"] = _meanOver(r["thr_vs_gt"]["precision"] for r in thr_valid)
453     summary["mean_iou_pred_vs_thr"] = _meanOver(r["pred_vs_thr"]["iou"] for r in thr_valid)
454     summary["mean_thr_length_mm"] = _meanOver(r["thr_centerline"]["length_mm"] for r in thr_valid)
455 return summary
456
```