Display of surface renderings of one case from part 1.c

A red and pink object

Description automatically generated with medium confidence

Boxplots of overall results across the 10 cases from 1.d-1.f

A graph of a diagram

Description automatically generated with medium confidence

Table of Wilcoxon signed-rank test p-values and significant differences between the different results in 3.c, comparing accuracy among the 3 raters and majority vote

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Tables of the overall confusion matrices and sensitivity/specificity values from part 2.

A screenshot of a computer

Description automatically generated

Code:

Surface. py

def surfDistances(self, mesh2):  
 *"""  
 Calculate surface distances from mesh1 to mesh2, integrating tqdm for progress tracking.  
 """* triangles = np.array(  
 [mesh2.verts[face\_indices] for face\_indices in mesh2.faces]) # Shape (M, 3, 3)  
  
 # Placeholder for the actual vectorized computation  
 distances = vectorized\_distance\_computation(self.verts, triangles)  
  
 # Step 3: Minimization  
 min\_distances = np.min(distances, axis=1) # Assuming distances is of shape (N, M)  
 mean\_distance = np.mean(min\_distances)  
 max\_distance = np.max(min\_distances)  
  
 return mean\_distance, max\_distance, None, None  
  
def pointsetDistance(self, t1s):  
 *"""  
 Calculate point set distances between two sets of points, including Mean Absolute Point Set Distance (MAPD),  
 Hausdorff Point Distance (HPD), and points of interest related to HPD.  
  
 :param gts: Ground Truth Set, with points as an Nx3 numpy array.  
 :param t1s: Target Set 1, with points as an Mx3 numpy array.  
 :return: MAPD from gts to t1s, HPD from gts to t1s, and points of interest.  
 """* # Build a KD-tree for the vertices of mesh2  
 tree = KDTree(t1s.verts)  
  
 # Query the KD-tree for the closest point in mesh2 for each vertex in mesh1  
 distances, \_ = tree.query(self.verts)  
  
 # Calculate MASD and HD  
 MASDg1 = np.mean(distances)  
  
 HDg1 = np.max(distances)  
  
 return None, MASDg1, HDg1

similarityMatrix.py

import nrrd  
  
from Project.surface import \*  
from Project4.confusionMatrix import \*  
from Project4.metricFunctions import \*  
from Project4.plotStatistics import \*  
  
# Assuming the base directory and a dictionary of patients are defined  
bsdir = '/Users/leonslaptop/Desktop/2024 Spring/ECE 3892/data/'  
  
# Lists to store metrics for each rater and the majority vote  
metrics = {  
 'volume': {'t1': [], 't2': [], 't3': [], 'mv': []},  
 'dice': {'t1': [], 't2': [], 't3': [], 'mv': []},  
 'mean\_surface\_distance': {'t1': [], 't2': [], 't3': [], 'mv': []},  
 'hausdorff\_distance': {'t1': [], 't2': [], 't3': [], 'mv': []}  
}  
  
# Initialize dictionaries to hold the cumulative confusion matrices for each rater  
cumulative\_matrices = {  
 't1': confusionMatrix(np.array([]), np.array([])),  
 't2': confusionMatrix(np.array([]), np.array([])),  
 't3': confusionMatrix(np.array([]), np.array([]))  
}  
  
pts = {0: '0522c0001', 1: '0522c0002', 2: '0522c0003', 3: '0522c0009', 4: '0522c0013',  
 5: '0522c0014', 6: '0522c0017', 7: '0522c0057', 8: '0522c0070', 9: '0522c0077'}  
  
# Loop through the first 10 cases  
for pt\_id, pt in pts.items():  
 gt\_path = f'{bsdir}{pt}/structures/mandible.nrrd'  
 gt, hdr = nrrd.read(gt\_path)  
 voxsz = [hdr['space directions'][0][0], hdr['space directions'][1][1],  
 hdr['space directions'][2][2]]  
  
 # Load segmentations  
 t1, \_ = nrrd.read(f'{bsdir}{pt}/structures/target1.nrrd')  
 t2, \_ = nrrd.read(f'{bsdir}{pt}/structures/target2.nrrd')  
 t3, \_ = nrrd.read(f'{bsdir}{pt}/structures/target3.nrrd')  
  
 # Majority vote  
 mv = np.sum([t1, t2, t3], axis=0) > 1.5  
  
 # Create surfaces for each segmentation  
 surfaces = {  
 'gt': surface(),  
 't1': surface(),  
 't2': surface(),  
 't3': surface(),  
 'mv': surface()  
 }  
 surfaces['gt'].createSurfaceFromVolume(gt, voxsz, 0.5)  
 surfaces['t1'].createSurfaceFromVolume(t1, voxsz, 0.5)  
 surfaces['t2'].createSurfaceFromVolume(t2, voxsz, 0.5)  
 surfaces['t3'].createSurfaceFromVolume(t3, voxsz, 0.5)  
 surfaces['mv'].createSurfaceFromVolume(mv.astype(int), voxsz, 0.5)  
  
 # VTK Visualization (Optional: Uncomment to use if myVtkWin is configured correctly)  
 win = myVtkWin()  
 win.addSurf(surfaces['t1'].verts, surfaces['t1'].faces, opacity=0.5)  
 win.addSurf(surfaces['t2'].verts, surfaces['t2'].faces, color=[1, 1, 0], opacity=0.5)  
 win.addSurf(surfaces['t3'].verts, surfaces['t3'].faces, color=[1, 0, 1], opacity=0.5)  
 # win.start()  
  
 # Calculate metrics for each rater and the majority vote  
 segmentations = {'t1': t1, 't2': t2, 't3': t3, 'mv': mv.astype(int)}  
  
 for rater, seg in segmentations.items():  
 dice\_score = dice\_coefficient(gt, seg)  
 metrics['dice'][rater].append(dice\_score)  
  
 volume = surfaces[rater].volume()  
 metrics['volume'][rater].append(volume)  
  
 MASD\_gt\_rater, HD\_gt\_rater, \_, \_ = surfaces['gt'].surfDistances(surfaces[rater])  
 MASD\_rater\_gt, HD\_rater\_gt, \_, \_ = surfaces[rater].surfDistances(surfaces['gt'])  
 mean\_surface\_distance = (MASD\_gt\_rater + MASD\_rater\_gt) / 2  
 hausdorff\_distance = max(HD\_gt\_rater, HD\_rater\_gt)  
  
 metrics['mean\_surface\_distance'][rater].append(mean\_surface\_distance)  
 metrics['hausdorff\_distance'][rater].append(hausdorff\_distance)  
  
 print(f"{pt} DONE!")  
  
 # Temporarily create confusion matrices for current patient and rater to update cumulative matrices  
 for rater, segmentation in [('t1', t1), ('t2', t2), ('t3', t3)]:  
 current\_matrix = confusionMatrix(gt, segmentation)  
 cumulative\_matrices[rater].TP += current\_matrix.TP  
 cumulative\_matrices[rater].FP += current\_matrix.FP  
 cumulative\_matrices[rater].FN += current\_matrix.FN  
 cumulative\_matrices[rater].TN += current\_matrix.TN  
  
# Calculate and print sensitivity and specificity for each rater  
sensitivity\_specificity = {}  
  
# After looping through all patients, calculate and print out the overall metrics for each rater  
for rater, matrix in cumulative\_matrices.items():  
 print(f"Confusion matrix for {rater}:")  
 matrix.print() # This now uses the print method of the confusionMatrix class  
  
 sensitivity = matrix.sensitivity()  
 specificity = matrix.specificity()  
  
 print(f"{rater}: Sensitivity = {sensitivity:.4f}, Specificity = {specificity:.4f}")  
 print(f"Dice: {matrix.dice()}\n")  
  
plot\_metrics(metrics, "Segmentation Metrics Across 10 Cases")  
  
# Perform and print Wilcoxon signed-rank test results  
perform\_wilcoxon\_test(metrics, 'volume')  
perform\_wilcoxon\_test(metrics, 'dice')  
perform\_wilcoxon\_test(metrics, 'mean\_surface\_distance')  
perform\_wilcoxon\_test(metrics, 'hausdorff\_distance')  
  
overall\_metrics = calculate\_overall\_metrics(cumulative\_matrices)  
  
# Print the overall confusion matrices and sensitivity/specificity values  
for rater, metrics in overall\_metrics.items():  
 print(f"Rater: {rater}")  
 print("Confusion Matrix:")  
 cumulative\_matrices[rater].print()  
 print(f"Sensitivity: {metrics['Sensitivity']:.3f}")  
 print(f"Specificity: {metrics['Specificity']:.3f}\n")

confusionmatrix.py

import numpy as np  
  
  
class confusionMatrix:  
 def \_\_init\_\_(self, gt, pred):  
 *"""Initialize confusion matrix with ground truth and prediction."""* self.TP = np.sum((gt == 1) & (pred == 1))  
 self.FP = np.sum((gt == 0) & (pred == 1))  
 self.TN = np.sum((gt == 0) & (pred == 0))  
 self.FN = np.sum((gt == 1) & (pred == 0))  
  
 def print(self):  
 *"""Print the confusion matrix."""* print("\t\t\tPredicted Positive\t|\tPredicted Negative")  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
 print(f"Actual Positive\t| {self.TP}\t\t| {self.FN}")  
 print(f"Actual Negative\t| {self.FP}\t\t| {self.TN}\n")  
  
 def sensitivity(self):  
 *"""Calculate and return sensitivity (recall)."""* return self.TP / (self.TP + self.FN) if (self.TP + self.FN) > 0 else 0  
  
 def specificity(self):  
 *"""Calculate and return specificity."""* return self.TN / (self.TN + self.FP) if (self.TN + self.FP) > 0 else 0  
  
 def dice(self):  
 *"""Calculate and return the Dice coefficient."""* return (2 \* self.TP) / (2 \* self.TP + self.FP + self.FN)

metricfunction.py

import numpy as np  
from tqdm import tqdm  
  
  
def dice\_coefficient(gt, pred):  
 *"""  
 Calculate the Dice coefficient, a measure of set similarity.  
  
 Parameters:  
 - gt: Ground truth binary segmentation mask as a numpy array.  
 - pred: Predicted binary segmentation mask as a numpy array.  
  
 Returns:  
 - dice: Dice coefficient as a float.  
 """* # Calculate intersection and union  
 intersection = np.logical\_and(gt, pred).sum()  
 gt\_sum = gt.sum() + pred.sum()  
  
 # Calculate Dice coefficient. Add a small epsilon to avoid division by zero.  
 dice = (2. \* intersection + 1e-6) / (gt\_sum + 1e-6)  
  
 return dice  
  
  
def vectorized\_distance\_computation(points, triangles):  
 *"""  
 Computes the minimum distance from each point to any of the given triangles, integrating decision making  
 for on-surface or off-surface projection.  
  
 Args:  
 points (np.array): Array of shape (N, 3) containing N points in 3D space.  
 triangles (np.array): Array of shape (M, 3, 3) representing M triangles, each defined by 3 vertices.  
 norms (np.array): Array of shape (M, 3) containing the normal vectors for M triangles.  
  
 Returns:  
 np.array: Array of shape (N,) containing the minimum distance from each point to the closest triangle.  
 """* N = len(points)  
 M = len(triangles)  
  
 distances = np.inf \* np.ones((N, M))  
  
 # Calculate normals for each triangle  
 edge1 = triangles[:, 1, :] - triangles[:, 0, :]  
 edge2 = triangles[:, 2, :] - triangles[:, 0, :]  
 norms = np.cross(edge1, edge2)  
 norms\_magnitude = np.linalg.norm(norms, axis=1, keepdims=True)  
 norms = norms / norms\_magnitude # Normalize the normals  
  
 # Iterate over each triangle with progress updates  
 for j in tqdm(range(M), desc="Computing distances to triangles"):  
 triangle = triangles[j]  
 norm = norms[j]  
  
 # Compute projected points and check if inside or outside the triangle  
 projected\_points, is\_inside = project\_and\_check(points, triangle, norm)  
  
 # Initialize an array to store the minimum distances for this triangle  
 min\_distances\_for\_triangle = np.zeros(N)  
  
 if np.any(is\_inside):  
 points\_inside = points[is\_inside]  
 projected\_points\_inside = projected\_points[is\_inside]  
 distances\_to\_plane = calculate\_distances\_from\_projected\_points(points\_inside,  
 projected\_points\_inside)  
 min\_distances\_for\_triangle[is\_inside] = distances\_to\_plane  
  
 # Compute distances for points projecting off the surface (edges and vertices)  
 if np.any(~is\_inside):  
 points\_outside = points[~is\_inside]  
 edge\_distances = vectorized\_distance\_to\_edges(points\_outside, triangle)  
 min\_distances\_for\_triangle[~is\_inside] = np.min(edge\_distances, axis=1)  
  
 # Update the distances matrix for this triangle  
 distances[:, j] = min\_distances\_for\_triangle  
  
 return distances  
  
  
def calculate\_distances\_from\_projected\_points(points, projected\_points):  
 *"""  
 Computes the distances from points to their projections on the plane.  
  
 Args:  
 points (np.array): Array of shape (N, 3) containing N points in 3D space.  
 projected\_points (np.array): Array of shape (N, 3) containing the projections of points onto a plane.  
  
 Returns:  
 np.array: Distances from each point to its projection on the plane.  
 """* # Calculate the vector differences between points and their projections  
 vector\_differences = points - projected\_points  
  
 # Compute the distances as the norm of these vector differences  
 distances = np.linalg.norm(vector\_differences, axis=1)  
  
 return distances  
  
  
def project\_and\_check(points, triangle, norm):  
 # Calculate the vector from the triangle's first vertex to the points  
 v0 = triangle[0]  
 vectors\_to\_points = points - v0  
  
 # Calculate the distance from the points to the triangle plane  
 distance\_to\_plane = np.dot(vectors\_to\_points, norm)  
  
 # Project points onto the plane  
 projected\_points = points - np.outer(distance\_to\_plane, norm)  
  
 # Check if the projected points are inside the triangle  
 is\_inside = is\_point\_inside\_triangle(projected\_points, triangle)  
  
 return projected\_points, is\_inside  
  
  
def is\_point\_inside\_triangle(pts, tri):  
 # Barycentric technique to check if point is inside the triangle  
 v0, v1, v2 = tri[0], tri[1], tri[2]  
 v0v1 = v1 - v0  
 v0v2 = v2 - v0  
  
 # Prepare points  
 v0p = pts - v0[np.newaxis, :] # Vector from v0 to each point  
  
 # Compute dot products  
 dot00 = np.dot(v0v2, v0v2)  
 dot01 = np.dot(v0v1, v0v2)  
 dot11 = np.dot(v0v1, v0v1)  
 dot02 = np.einsum('ij,j->i', v0p, v0v2) # Dot product of v0p vectors with v0v2  
 dot12 = np.einsum('ij,j->i', v0p, v0v1) # Dot product of v0p vectors with v0v1  
  
 # Compute barycentric coordinates  
 invDenom = 1.0 / (dot00 \* dot11 - dot01 \* dot01)  
 u = (dot11 \* dot02 - dot01 \* dot12) \* invDenom  
 v = (dot00 \* dot12 - dot01 \* dot02) \* invDenom  
  
 # Check if point is in triangle  
 inside = (u >= 0) & (v >= 0) & (u + v <= 1)  
  
 return inside  
  
  
def vectorized\_distance\_to\_edges(points, triangle):  
 *"""  
 Computes the vectorized distance from points to the edges of a triangle.  
  
 Args:  
 points (np.array): Array of shape (N, 3) containing N points.  
 triangle (np.array): Array of shape (3, 3) representing a triangle's vertices.  
  
 Returns:  
 np.array: Array of shape (N, 3) containing distances from each point to each of the triangle's edges.  
 """* # Calculate edge vectors  
 edges = np.array(  
 [triangle[1] - triangle[0], triangle[2] - triangle[1], triangle[0] - triangle[2]])  
  
 # Vector from each vertex to points  
 p\_to\_vertices = np.array([points - triangle[0], points - triangle[1], points - triangle[2]])  
  
 # Calculate projection coefficients for all edges  
 coefficients = np.einsum('ijk,ik->ij', p\_to\_vertices, edges) / (  
 np.linalg.norm(edges, axis=1)[:, np.newaxis] \*\* 2)  
  
 # Clip coefficients to [0, 1] range  
 coefficients\_clipped = np.clip(coefficients, 0, 1)  
  
 # Ensure coefficients are correctly broadcastable to (3, 8671, 3) for multiplication with edges  
 coefficients\_reshaped = coefficients\_clipped[:, :, np.newaxis]  
  
 # Broadcast triangle for addition: reshape triangle for compatibility  
 triangle\_broadcast = triangle[:, np.newaxis, :] # Reshape to (3, 1, 3) for broadcasting  
  
 # Now apply the reshaped coefficients to edges and add the broadcasted triangle vertices  
 projections = triangle\_broadcast + coefficients\_reshaped \* edges[:, np.newaxis, :]  
  
 # Calculate distances from points to projected points  
 distances = np.sqrt(np.sum((points[np.newaxis, :, :] - projections) \*\* 2, axis=2))  
  
 # Stack distances into a single array  
 distances = distances.T  
  
 return distances

plotstatistics.py

import matplotlib.pyplot as plt  
import numpy as np  
from scipy.stats import wilcoxon  
  
  
# Assuming dice\_scores\_all is a dictionary with rater keys ('t1', 't2', 't3', 'mv') and lists of Dice scores as values  
# Example:  
# dice\_scores\_all = {  
# 't1': [0.9, 0.92, 0.93, ...], # Dice scores for rater 1 across all cases  
# 't2': [0.91, 0.9, 0.92, ...], # Dice scores for rater 2 across all cases  
# 't3': [0.88, 0.89, 0.9, ...], # Dice scores for rater 3 across all cases  
# 'mv': [0.94, 0.95, 0.96, ...], # Dice scores for majority vote across all cases  
# }  
  
# Visualization with boxplots for each metric  
def plot\_metrics(metric\_data, title):  
 fig, axs = plt.subplots(1, len(metric\_data), figsize=(20, 5))  
 for ax, (metric\_name, values) in zip(axs, metric\_data.items()):  
 ax.boxplot(values.values(), labels=values.keys())  
 ax.set\_title(f'{title}: {metric\_name}')  
 ax.set\_ylabel(metric\_name)  
 ax.grid(True)  
 plt.tight\_layout()  
 plt.show()  
  
  
# Function to compare metrics using Wilcoxon signed-rank test  
def compare\_metrics(metric\_data):  
 raters = list(metric\_data.keys())  
 p\_values\_table = np.zeros((len(raters), len(raters)), dtype=float)  
  
 # Fill the table with p-values  
 for i, rater1 in enumerate(raters):  
 for j, rater2 in enumerate(raters):  
 if i < j: # Avoid redundant comparisons  
 stat, p\_value = wilcoxon(metric\_data[rater1], metric\_data[rater2])  
 p\_values\_table[i, j] = p\_value  
 p\_values\_table[j, i] = p\_value # Symmetric matrix  
 elif i == j:  
 p\_values\_table[i, j] = np.nan # NaN for comparisons with themselves  
  
 return p\_values\_table, raters  
  
  
# Function to perform Wilcoxon signed-rank test and print results  
def perform\_wilcoxon\_test(metrics, metric\_name):  
 raters = list(metrics[metric\_name].keys())  
 num\_raters = len(raters)  
 p\_values\_table = np.empty((num\_raters, num\_raters))  
 p\_values\_table[:] = np.NaN # Initialize with NaN  
  
 # Perform Wilcoxon signed-rank test between pairs of raters for the specified metric  
 for i in range(num\_raters):  
 for j in range(i + 1, num\_raters):  
 scores\_i = metrics[metric\_name][raters[i]]  
 scores\_j = metrics[metric\_name][raters[j]]  
 stat, p\_value = wilcoxon(scores\_i, scores\_j)  
 p\_values\_table[i, j] = p\_value  
 p\_values\_table[j, i] = p\_value # Symmetric matrix  
  
 # Print the table of p-values  
 print(f"Wilcoxon signed-rank test p-values for {metric\_name}:")  
 print("\t" + "\t".join(raters))  
 for i, rater in enumerate(raters):  
 print(f"{rater}\t" + "\t".join(  
 ["{:.3f}".format(p) if not np.isnan(p) else "NaN" for p in p\_values\_table[i]]))  
  
 # Identify significant differences  
 alpha = 0.05  
 print(f"\nSignificant differences for {metric\_name} (p < {alpha}):")  
 for i in range(num\_raters):  
 for j in range(i + 1, num\_raters):  
 if p\_values\_table[i, j] < alpha:  
 print(f"Between {raters[i]} and {raters[j]}: YES (p = {p\_values\_table[i, j]:.3f})")  
 # else:  
 # print(f"Between {raters[i]} and {raters[j]}: NO (p = {p\_values\_table[i, j]:.3f})")  
  
  
# Assuming cumulative\_matrices is filled as before, calculate sensitivity and specificity for each rater  
def calculate\_overall\_metrics(cumulative\_matrices):  
 results = {}  
 for rater, cm in cumulative\_matrices.items():  
 sensitivity = cm.sensitivity()  
 specificity = cm.specificity()  
 results[rater] = {'Sensitivity': sensitivity, 'Specificity': specificity}  
 return results