# A General Overview of Techniques and Visualizations

**Preamble**

Dominik Filliger & Noah Leuenberger

2024-11-14

# Table of contents

# 1 A General Overview of Techniques and Visualizations

Preamble

This booklet provides an accessible overview of our bachelor thesis project, focusing on fine-tuning pre-trained ECG and CMR encoders using diverse datasets and evaluating their adaptability for cardiovascular diagnostics. Our goal is to establish foundational representations of cardiovascular data that can be expanded upon in future research, ultimately improving diagnostic capabilities for diverse patient populations.

We explore how fine-tuning these pre-trained models with diverse data impacts their ability to capture nuanced, population-specific patterns, and whether they can effectively adapt to new contexts, such as varying health conditions, ethnic backgrounds, and socioeconomic profiles. Additionally, we present a modular, machine learning pipeline designed to support ongoing research and future integration of multimodal data.

# 2 Dataset - Automated Cardiac Diagnosis Challenge (ACDC)

The ACDC dataset is part of the Automated Cardiac Diagnosis Challenge, which provides comprehensive MRI data for studying cardiac function across several patient groups.

The ACDC dataset was created from real clinical exams acquired at the University Hospital of Dijon, France. The data is fully anonymized and complies with the local ethical regulations set by the hospital's ethical committee. The dataset captures several well-defined pathologies and provides enough cases to properly train machine learning models, as well as assess variations in key physiological parameters obtained from cine-MRI, such as **diastolic volume** and **ejection fraction**.

The dataset is composed of **150 exams** (each from a different patient) divided into **five evenly distributed subgroups** (four pathological groups and one healthy subject group). Each patient record also contains additional metadata like **weight, height**, and details regarding the **diastolic and systolic phases**.

## Study Population

The dataset includes **150 patients**, categorized into the following five subgroups:

1. **Normal subjects (NOR)** - 30 patients without known pathologies.

2. **Myocardial Infarction (MINF)** - 30 patients with left ventricular ejection fraction below 40% and abnormal contraction in several myocardial segments.
3. **Dilated Cardiomyopathy (DCM)** - 30 patients with an enlarged left ventricular volume (>100 mL/m²) and reduced ejection fraction (below 40%).
4. **Hypertrophic Cardiomyopathy (HCM)** - 30 patients with increased left ventricular mass (>110 g/m²), thickened myocardial segments (>15 mm in diastole), but with normal ejection fraction.
5. **Abnormal Right Ventricle (RV)** - 30 patients with either enlarged right ventricular volume (>110 mL/m²) or an ejection fraction of the right ventricle lower than 40%.

The groups are defined based on physiological parameters, such as **ventricular volumes**, **ejection fractions**, **local contractions**, **LV mass**, and the **maximum thickness** of the myocardium. The classification rules can be found in more detail in the relevant tab.

## Involved Systems

The MRI acquisitions were conducted over a six-year period using **two MRI scanners** with different magnetic field strengths:

- **1.5 T (Siemens Area, Siemens Medical Solutions, Germany)**
- **3.0 T (Siemens Trio Tim, Siemens Medical Solutions, Germany)**

Images were acquired in a **short-axis orientation**, covering the left ventricle from base to apex, with slice thicknesses of either **5 mm** or **8 mm**. The **spatial resolution** ranges between **1.37 to 1.68 mm²/pixel**, and **28 to 40 images** were taken to capture the complete or nearly complete cardiac cycle.

O. Bernard, A. Lalande, C. Zotti, F. Cervenansky, et al.
"Deep Learning Techniques for Automatic MRI Cardiac Multi-structures Segmentation and Diagnosis: Is the Problem Solved?"
IEEE Transactions on Medical Imaging, vol. 37, no. 11, pp. 2514-2525, Nov. 2018.
doi: 10.1109/TMI.2018.2837502

## 2.1 Loading the CMR Dataset

```
import sys
import os
from pathlib import Path

project_root = str(Path().absolute().parent.parent)
sys.path.append(project_root)

# unified dataset
from src.data.unified import UnifiedDataset
from src.data.dataset import DatasetModality
from src.visualization.cmr_viz import plot_processed_sample, plot_raw_sample, create_cardiac

data_root = Path(project_root) / "data"
```

Let's explore the ACDC dataset and take a closer look at its content.

```
acdc_data = UnifiedDataset(data_root, modality=DatasetModality.CMR, dataset_key="acdc")
acdc_data = acdc_data.raw_dataset

# Load first patient data
patient_record = acdc_data.load_record("patient001")

# Create patient data dictionary with all available metadata
patient_data = {
```

```
    "data": patient_record.data,
    "id": patient_record.id,
    "group": patient_record.target_labels,
    "ed_frame_idx": patient_record.metadata["ed_frame_idx"],
    "mid_frame_idx": patient_record.metadata["mid_frame_idx"],
    "es_frame_idx": patient_record.metadata["es_frame_idx"]
}

# Print metadata
print("\nPatient Metadata:")
for key, value in patient_record.metadata.items():
    print(f"{key}: {value}")
```

```
Patient Metadata:
ed_frame: 1
es_frame: 12
height: 184.0
weight: 95.0
nb_frames: 30
ed_frame_idx: 0
mid_frame_idx: 5
es_frame_idx: 11
nifti_path: training/patient001/patient001_4d.nii.gz
```

The dataset includes cardiac MRI data from multiple patients, each accompanied by relevant metadata. Here, we are focusing on one patient to get started.

## 2.2 Visualizing Key Cardiac Frames

We begin by examining the three key frames in the cardiac cycle: End-Diastolic (ED), Mid-Phase, and End-Systolic (ES).

```
import matplotlib.pyplot as plt

fig1 = plot_processed_sample(
    data=patient_data["data"],
    title=f'Patient {patient_data["id"]} - {patient_data["group"]}'
)
plt.show()
```

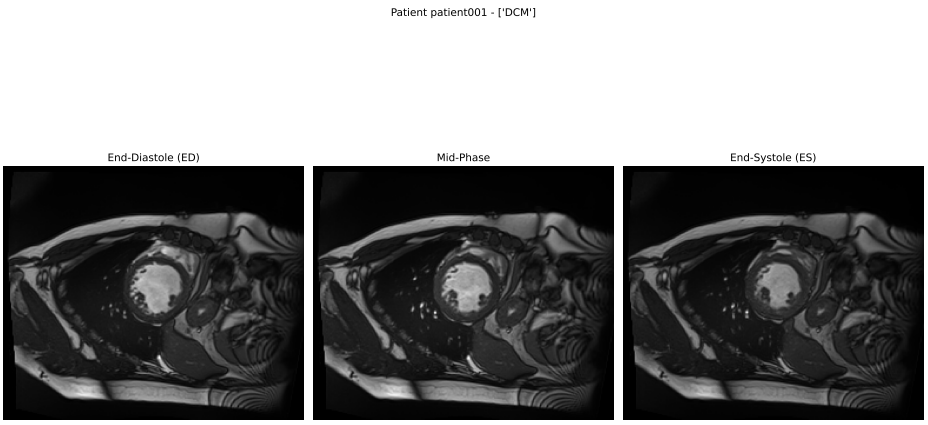



Figure 2.1: Key cardiac frames showing ED, Mid-Phase, and ES states.

The visualization in Figure 2.1 shows the heart's state at three critical phases. These states provide important snapshots of the heart's activity, giving insights into its pumping function.

## 2.3 Visualizing the Sequence of Frames

Since our data contains only three frames (ED, Mid-Phase, and ES), we can visualize them sequentially to observe the changes.

```
fig2 = plot_raw_sample(
    data=acdc_data._read_nifti(acdc_data.paths["raw"] / patient_record.metada
    frame_indices=[
        patient_record.metadata["ed_frame_idx"],
        patient_record.metadata["mid_frame_idx"],
        patient_record.metadata["es_frame_idx"]
    ],
    title=f'Patient {patient_data["id"]} - Cardiac Phases'
)
plt.show()
```

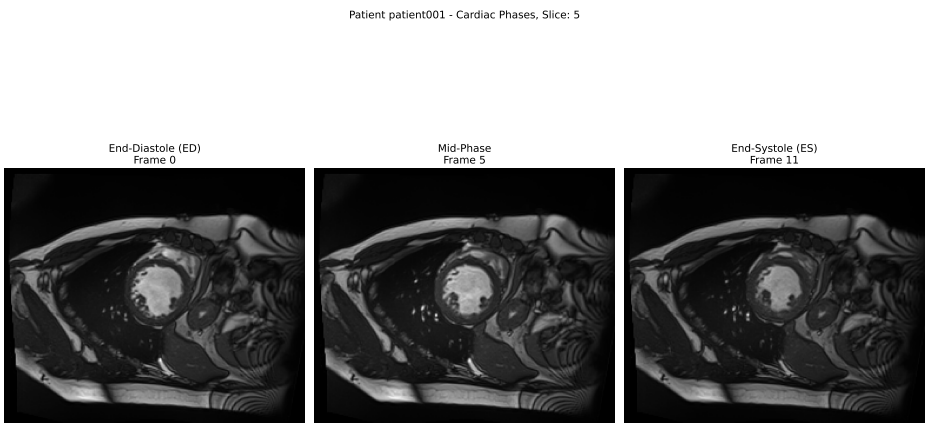



Figure 2.2: Sequential visualization of cardiac phases

This sequence shown in Figure 2.2 represents the heart's movement through the key phases. Observing these frames in order reveals the

changing shape and size of the heart chambers, which is vital for understanding cardiac function.

## 2.4 Cropping based on Turgut Biobank

The Turgut Biobank paper implements a custom manual cropping strategy with hard-coded values designed for their specific dataset. While this approach provides a standardized way to focus on cardiac regions, these fixed parameters may not be optimal for our ACDC dataset due to potential differences in image acquisition, patient positioning, and cardiac anatomical variations. Let's implement and evaluate this cropping approach across our key cardiac frames.

```
import numpy as np
import matplotlib.pyplot as plt

def apply_turgut_crop(image, img_size):
    """Apply Turgut Biobank cropping parameters to numpy array"""
    top = int(0.21 * img_size)
    left = int(0.325 * img_size)
    height = width = int(0.375 * img_size)

    # Ensure the image is large enough for cropping
    if image.shape[0] < top + height or image.shape[1] < left + width:
        raise ValueError("Image too small for specified crop dimensions")

    return image[top:top+height, left:left+width]

def normalize_image(img):
    """Normalize image to 0-1 range"""
    return (img - img.min()) / (img.max() - img.min())

# Get the NIFTI data and relevant frames
```

```
original_data = acdc_data._read_nifti(acdc_data.paths["raw"] / patient_recor
frames = [
    patient_record.metadata["ed_frame_idx"],
    patient_record.metadata["mid_frame_idx"],
    patient_record.metadata["es_frame_idx"]
]
frame_names = ['ED', 'Mid', 'ES']
mid_slice = original_data.shape[2] // 2

# Create figure with three pairs of images
fig, axes = plt.subplots(3, 2, figsize=(10, 15))
fig.suptitle('Original vs Turgut Cropped Images', fontsize=14)

# Process each frame
for idx, (frame, name) in enumerate(zip(frames, frame_names)):
    img = original_data[:, :, mid_slice, frame]
    img = normalize_image(img)

    try:
        cropped_img = apply_turgut_crop(img, img.shape[0])

        # Plot original
        axes[idx, 0].imshow(img, cmap='gray')
        axes[idx, 0].set_title(f'Original - {name}')
        axes[idx, 0].axis('off')

        # Plot cropped
        axes[idx, 1].imshow(cropped_img, cmap='gray')
        axes[idx, 1].set_title(f'Turgut Cropped - {name}')
        axes[idx, 1].axis('off')

    except ValueError as e:
        print(f"Error processing {name} frame: {e}")
```

```
plt.tight_layout()
plt.show()

# Print crop dimensions for reference
img_size = img.shape[0]
print(f"Original size: {img.shape}")
print(f"Cropped size: {cropped_img.shape}")
print(f"\nTurgut cropping parameters:")
print(f"Top: {int(0.21 * img_size)} pixels")
print(f"Left: {int(0.325 * img_size)} pixels")
print(f"Height/Width: {int(0.375 * img_size)} pixels")
```

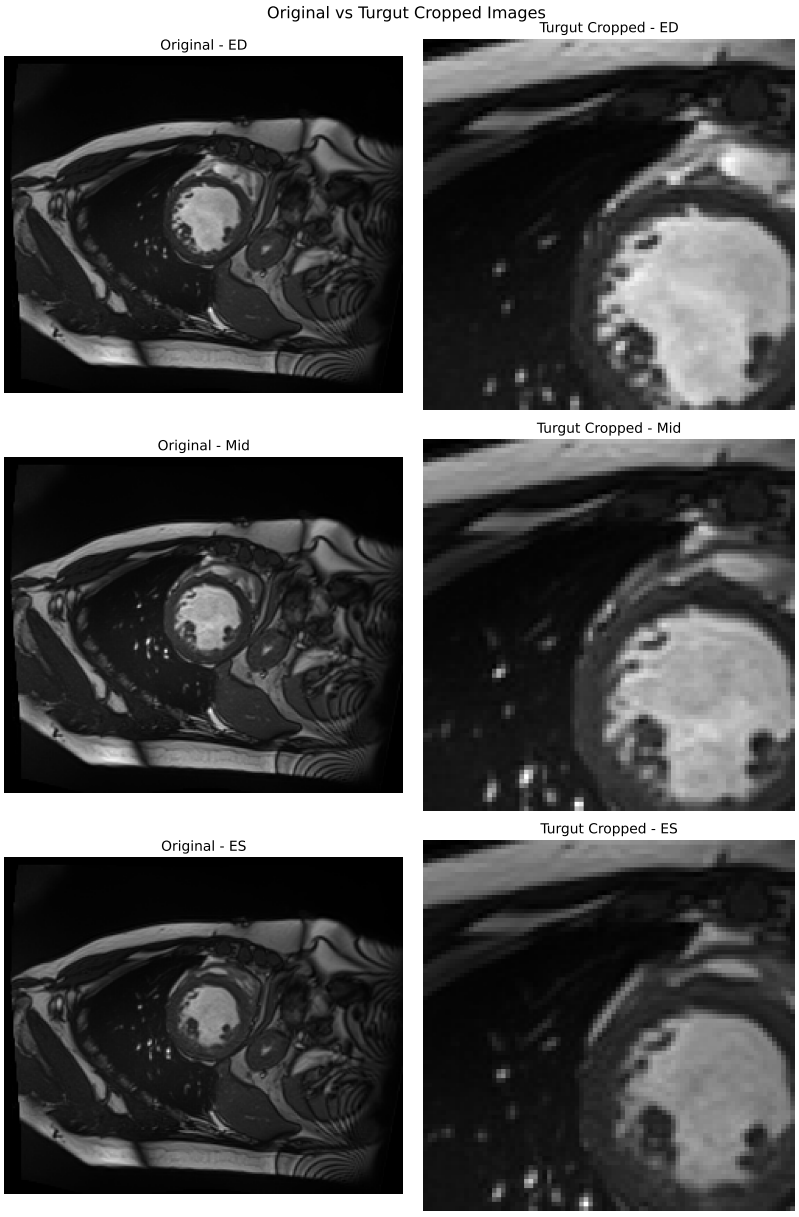



Figure 2.3: Comparison of original and Turgut-cropped cardiac images across ED, Mid, and ES frames

```
Original size: (216, 256)
Cropped size: (81, 81)

Turgut cropping parameters:
Top: 45 pixels
Left: 70 pixels
Height/Width: 81 pixels
```

The Turgut Biobank cropping uses fixed relative parameters to focus on the central region of the heart: - Starting at 21% from the top - Starting at 32.5% from the left - Taking a square region that is 37.5% of the original image size

By examining all three cardiac phases (ED, Mid, ES), we can better evaluate whether these fixed cropping parameters consistently capture the relevant cardiac structures in our ACDC dataset. This analysis helps determine if we need to adjust these parameters for our specific use case.

# 3 Dataset - 12-lead ECG for Arrhythmia Study

# 4 Introduction

The 12-lead ECG dataset is a comprehensive repository created by Chapman University, Shaoxing People's Hospital, and Ningbo First Hospital. It aims to facilitate research in arrhythmia detection and other cardiovascular studies. The dataset includes ECG signals collected from **45,152 patients**, all labeled by professional experts, with a **500 Hz** sampling rate. More details about the dataset can be found on the A large scale 12-lead electrocardiogram database for arrhythmia study website.

The ECG signals were collected as part of a clinical study to detect different types of **arrhythmias** and cardiovascular conditions. The dataset features ECGs in **WFDB** format, with both the raw data (`.mat` files) and the corresponding metadata (`.hea` files) containing information such as **age**, **gender**, **lead configuration**, and **SNOMED CT codes**.

# 5 Dataset Overview

Below are some key details of the study population:

- **Number of Patients**: 45,152
- **Sampling Rate**: 500 Hz
- **ECG Leads**: 12 (Standard leads)
- **Amplitude Unit**: Microvolt
- **Data Format**: WFDB (MAT and Header files)
- **SNOMED CT Codes**: Annotated for cardiovascular conditions

# 6 Visualizing 12-Lead ECG Signals

In this section, we visualize the complete 12-lead ECG signal for a sample patient. This overview allows us to examine the signal morphology across different leads and understand the overall quality of the data.

```
import sys
import os
from pathlib import Path
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

project_root = str(Path().absolute().parent.parent)
sys.path.append(project_root)

from src.data.unified import UnifiedDataset
from src.data.dataset import DatasetModality
from src.visualization.ecg_viz import plot_ecg_signals

data_root = Path(project_root) / "data"

arrhythmia_data = UnifiedDataset(data_root, modality=DatasetModality.ECG, dataset_key="arrhy
records = arrhythmia_data.get_all_record_ids()
metadata_store = arrhythmia_data.metadata_store

# create a DataFrame with metadata and labels
```

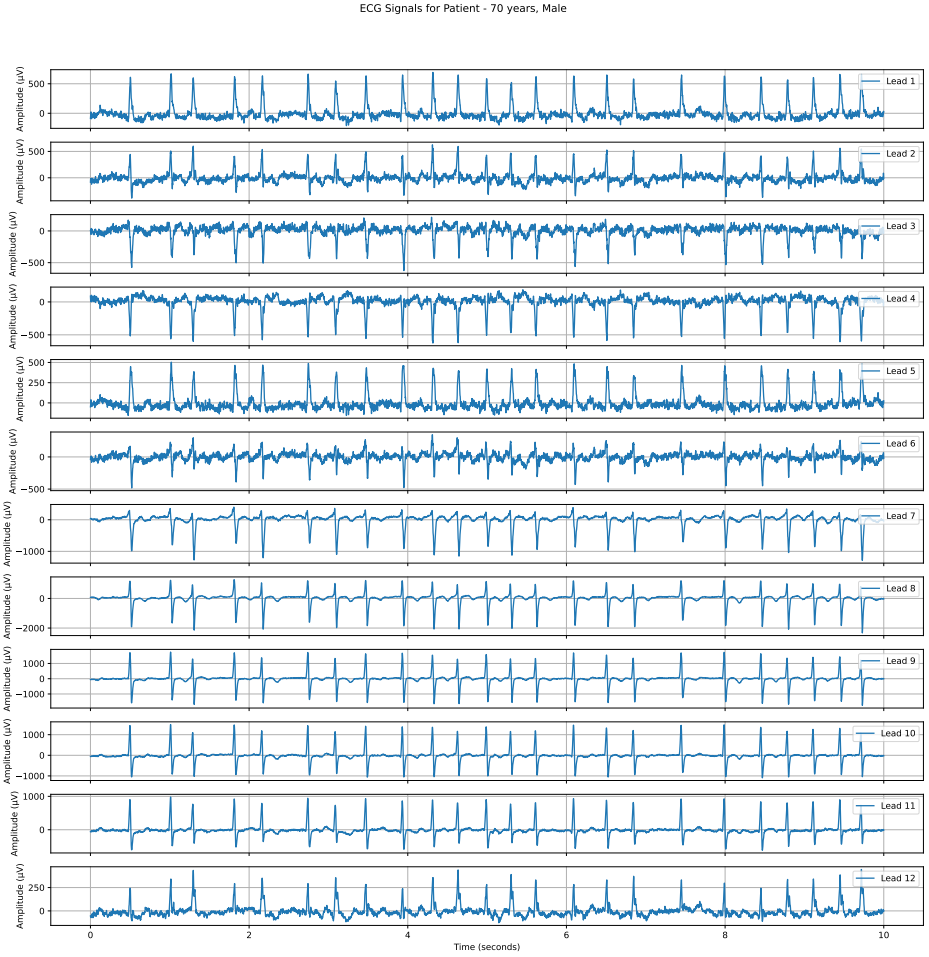
```
metadata_df = pd.DataFrame([{**metadata_store.get(record_id), 'record_id': re
metadata_df['labels'] = [arrhythmia_data[record_id].preprocessed_record.targe

print(f"Found {len(records)} patients")
metadata_df.head()
```

```
Found 44817 patients
```

| | age | is_male | labels_metadata | mat_file |
|---|---|---|---|---|
| 0 | 70.0 | True | [{'snomed_code': '164890007', 'acronyms': ['AF... | a-large-scale-12-lea |
| 1 | 52.0 | False | [{'snomed_code': '284470004', 'acronyms': ['AP... | a-large-scale-12-lea |
| 2 | 64.0 | True | [{'snomed_code': '426177001', 'acronyms': ['SB... | a-large-scale-12-lea |
| 3 | 60.0 | True | [{'snomed_code': '426177001', 'acronyms': ['SB... | a-large-scale-12-lea |
| 4 | 54.0 | False | [{'snomed_code': '426783006', 'acronyms': ['SR... | a-large-scale-12-lea |

Next, we visualize the raw 12-lead ECG signals for the first record in the dataset:

```
sample_record = arrhythmia_data[records[0]]
plot_ecg_signals(sample_record.raw_record.data, sample_record.raw_record.meta
```



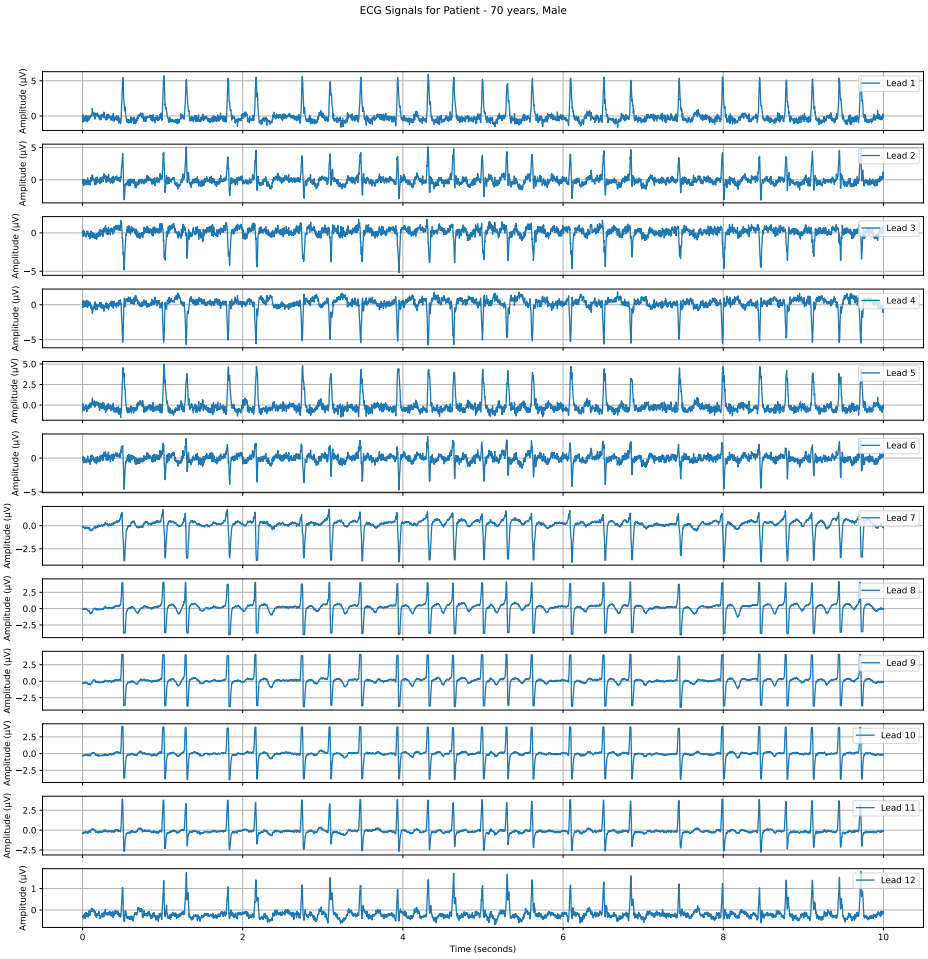

The figure above displays the **12-lead ECG signals** for a sample patient over **10 seconds**. Each lead provides a different perspective of the heart's electrical activity, offering comprehensive insight into the patient's cardiac health.

## 6.1 Comparing Raw and Preprocessed Signals

After preprocessing the ECG signals, we can view them in a more interpretable format. The preprocessing involves removing ALS baseline drift and normalizing the signals to have zero mean and unit variance. This step is critical for ensuring that subsequent models can learn meaningful patterns.

Display the preprocessed ECG signals:

```
plot_ecg_signals(sample_record.preprocessed_record.inputs, sample_record.prep
```




To quantify the changes introduced during preprocessing, we calculate the root mean square error (RMSE) between the raw and preprocessed signals. This metric provides a quantitative measure of the signal distortion due to preprocessing.

```
def calculate_rmse(signal1, signal2):
    return np.sqrt(np.mean((signal1 - signal2) ** 2))
```

```
raw_signal = sample_record.raw_record.data
preprocessed_signal = sample_record.preprocessed_record.inputs.numpy()

assert calculate_rmse(raw_signal, raw_signal) == 0.0, "RMSE with itself shoul

rmse = calculate_rmse(raw_signal, preprocessed_signal)
print(f"RMSE between raw and preprocessed signals: {rmse:.2f} microvolts")
```

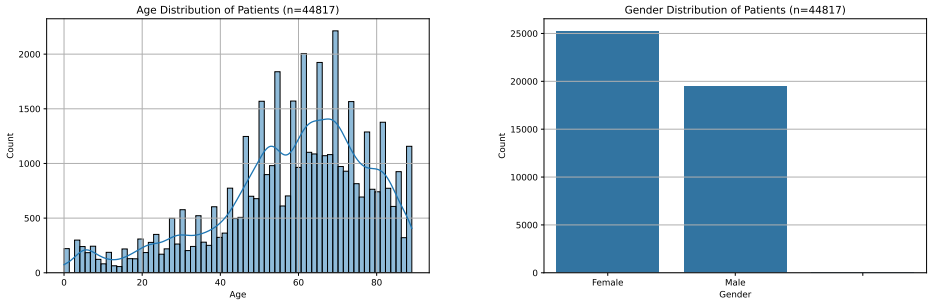
```
RMSE between raw and preprocessed signals: 211.13 microvolts
```

# 7 Patient Demographics and Condition Distribution

In this section, we examine the demographic characteristics of the patients in the dataset, including age and gender distribution. These plots provide insights into the population characteristics and help assess the generalizability of machine learning models trained on this data.

```
sample_size = len(metadata_df)
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Plot age distribution
sns.histplot(metadata_df["age"].dropna(), kde=True, ax=axes[0])
axes[0].set_title(f"Age Distribution of Patients (n={sample_size})")
axes[0].set_xlabel("Age")
axes[0].set_ylabel("Count")
axes[0].grid(True)

# Plot gender distribution
metadata_df["is_male"] = metadata_df["is_male"].fillna("Unknown")
sns.countplot(x="is_male", data=metadata_df, ax=axes[1])
axes[1].set_title(f"Gender Distribution of Patients (n={sample_size})")
axes[1].set_xlabel("Gender")
axes[1].set_xticklabels(["Female", "Male"])
axes[1].set_ylabel("Count")
axes[1].grid(True, axis="y")
```

```
plt.tight_layout()
plt.subplots_adjust(wspace=0.3)
plt.show()
```




The plots above illustrate the **age distribution** and **gender distribution** within the dataset, which are important for understanding the clinical diversity of the population.

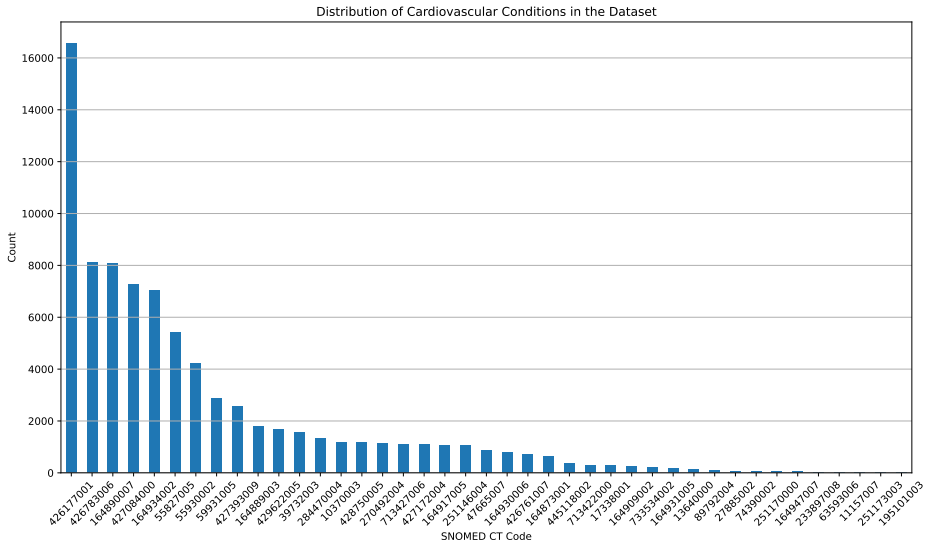
# 8 Condition Mapping and SNOMED CT Codes

The dataset provides detailed annotations for cardiovascular conditions using **SNOMED CT codes**. In this section, we load the mapping between these codes and their corresponding conditions and visualize the distribution of conditions across the dataset.

The following code plots a bar chart of the distribution of SNOMED CT codes:

```
# plot label distribution based on labels_metadata column which is a list of dictionaries
# each dictionary contains the following info:
#      def _build_metadata_mapping(self) -> Dict[str, dict]:
#         """Create unified metadata dictionary from merged data"""
#         return {
#             str(row["Snomed_CT"]): {
#                 "snomed_code": str(row["Snomed_CT"]),
#                 "acronyms": self._unique_values(
#                     row, ["Acronym Name_labeling", "Acronym Name_snomed"]
#                 ),
#                 "diagnosis_names": self._unique_values(row, ["Diagnosis", "Full Name"]),
#                 self.INT_CODE_META_KEY: row.get("Integration Code", "Unlabeled"),
#                 "integration_name": row.get("Integration Name", "Unlabeled"),
#                 "group": row.get("Group", "Unlabeled"),
#                 "comment": row.get("comment", ""),
#             }
#             for _, row in self.merged_df.iterrows()
```

```
#         }

fig, ax = plt.subplots(figsize=(15, 8))
metadata_df["labels_metadata"].explode().apply(lambda x: x["snomed_code"]).va
ax.set_title("Distribution of Cardiovascular Conditions in the Dataset")
ax.set_xlabel("SNOMED CT Code")
ax.set_ylabel("Count")
plt.xticks(rotation=45)
plt.grid(axis="y")
plt.show()
```



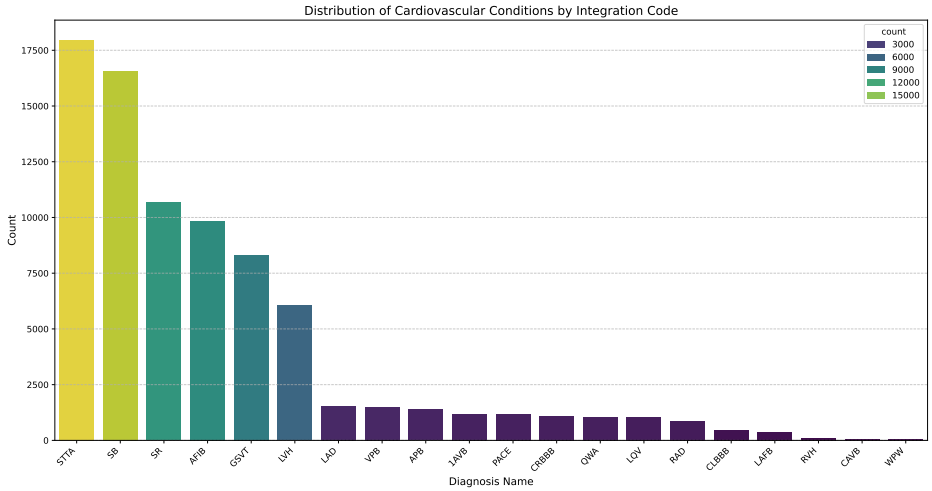

The above bar chart demonstrates that the distribution of cardiovascular conditions has a long tail, with some conditions being more prevalent than others. This information is crucial when developing machine learning models for condition detection and classification.

# 9 Distribution of Cardiovascular Conditions by Integration Code

To gain a more clinically interpretable view of the label distribution, we now visualize the distribution of conditions based on **Integration Codes**. The following code maps integration codes to their corresponding diagnosis names and produces a bar plot.

```
# Plot distribution of conditions based on integration codes
labels_metadata_exploded = metadata_df['labels_metadata'].explode()

# Create DataFrame from exploded metadata
labels_data = pd.json_normalize(labels_metadata_exploded)

# Count occurrences of each integration code
integration_counts = labels_data['integration_code'].value_counts().reset_index()
integration_counts.columns = ['integration_code', 'count']

# Get unique integration code to name mapping
integration_names = labels_data[['integration_code', 'integration_name']].drop_duplicates()

# Merge counts with names
integration_counts = integration_counts.merge(integration_names, on='integration_code', how=

# Sort by count for better visualization
integration_counts = integration_counts.sort_values('count', ascending=False)
```

```
# Create the plot
plt.figure(figsize=(15, 8))
sns.barplot(x='integration_name', y='count', data=integration_counts, palette
plt.title('Distribution of Cardiovascular Conditions by Integration Code', fo
plt.xlabel('Diagnosis Name', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.yticks(fontsize=10)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```




This bar plot shows the distribution of cardiovascular conditions by Integration Code, with human-readable diagnosis names on the x-axis. It highlights the most common conditions in the dataset while maintaining clarity through a clean layout and rotated labels.

# 10 Group Distribution and Label Merging Analysis

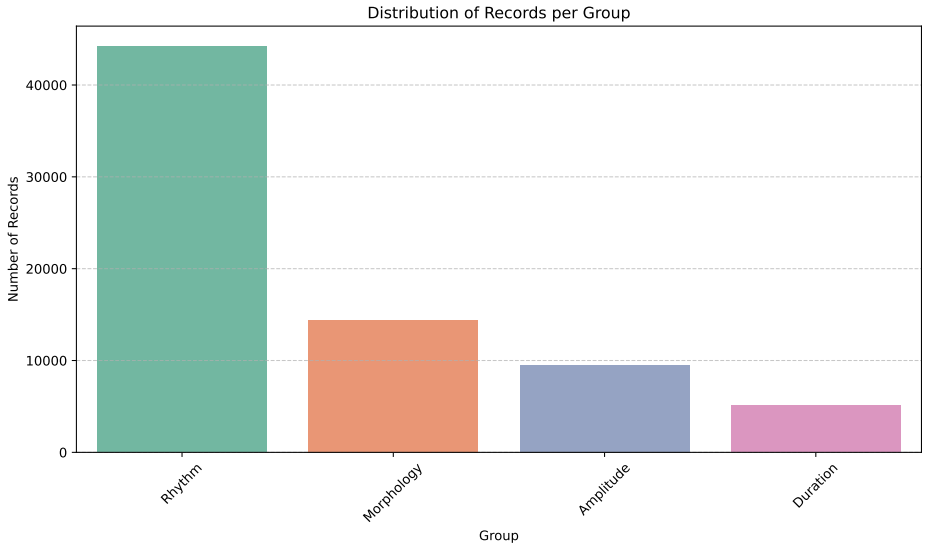
In the following sections, we analyze group-level patterns within the label metadata. We first visualize how many records belong to each group and then explore the combination patterns of groups across records. Finally, we compare the number of labels before and after applying our merging logic.

## 10.1 Visualizing Group Distribution Across Records

Each record may contain multiple labels that belong to one or more groups. We first extract the unique groups for each record so that if multiple labels belong to the same group, they are only counted once. The bar plot below shows the distribution of records per group.

```
def extract_unique_groups(label_metadata_list):
    if not isinstance(label_metadata_list, list):
        return []
    # Use the "group" key and strip spaces; fallback to "Unknown" if missing.
    groups = [d.get("group", "Unknown").strip() for d in label_metadata_list if d.get("group
    return list(set(groups))

metadata_df["unique_groups"] = metadata_df["labels_metadata"].apply(extract_unique_groups)

# Explode the unique_groups list so that each record appears once per group
```

```
group_exploded = metadata_df.explode("unique_groups")
group_counts = group_exploded["unique_groups"].value_counts().reset_index()
group_counts.columns = ["group", "record_count"]

plt.figure(figsize=(10, 6))
sns.barplot(x="group", y="record_count", data=group_counts, palette="Set2")
plt.title("Distribution of Records per Group")
plt.xlabel("Group")
plt.ylabel("Number of Records")
plt.xticks(rotation=45)
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.tight_layout()
plt.show()
```



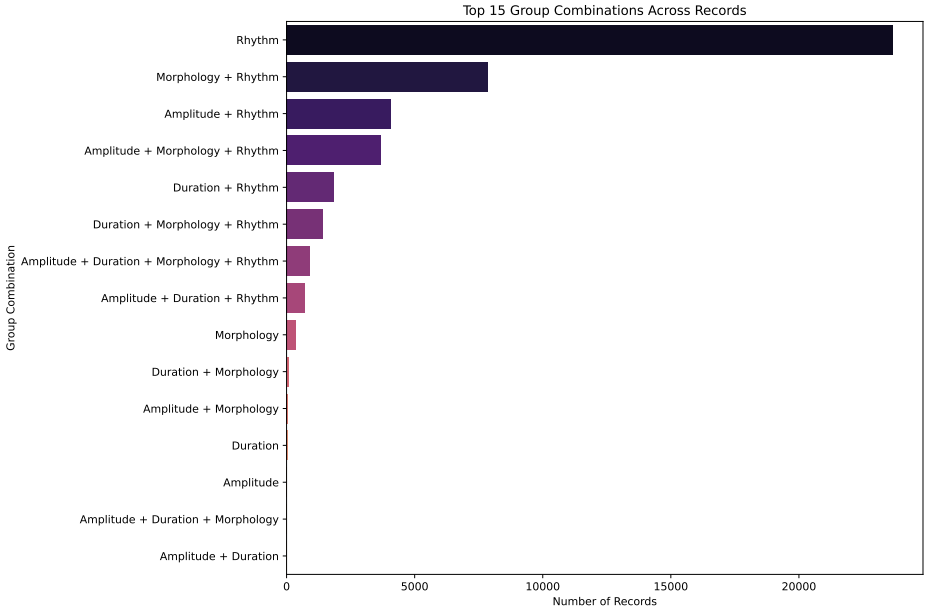
## 10.2 Analyzing Group Combination Patterns

Next, we analyze the frequency of different group combinations within each record. For every record, we sort and store the unique groups as a tuple. This allows us to count how often each combination occurs.

```
metadata_df["group_combo"] = metadata_df["unique_groups"].apply(lambda x: tuple(sorted(x)))
combo_counts = metadata_df["group_combo"].value_counts().reset_index()
combo_counts.columns = ["group_combination", "record_count"]

total_records = len(metadata_df)
combo_counts["percentage"] = 100 * combo_counts["record_count"] / total_records

print("Group Combination Frequencies (Top 10):")
print(combo_counts.head(10))

# Convert tuple to string for more readable labels in the plot
combo_counts["combo_str"] = combo_counts["group_combination"].apply(lambda x: " + ".join(x))

plt.figure(figsize=(12, 8))
sns.barplot(x="record_count", y="combo_str", data=combo_counts.head(15), palette="magma")
plt.title("Top 15 Group Combinations Across Records")
plt.xlabel("Number of Records")
plt.ylabel("Group Combination")
plt.tight_layout()
plt.show()
```

```
Group Combination Frequencies (Top 10):
                          group_combination  record_count  percentage
0                                  (Rhythm,)         23660   52.792467
1                      (Morphology, Rhythm)          7850   17.515675
2                       (Amplitude, Rhythm)          4080    9.103688
3           (Amplitude, Morphology, Rhythm)          3672    8.193319
```

| | | | |
|---|---|---|---|
| 4 | (Duration, Rhythm) | 1866 | 4.163599 |
| 5 | (Duration, Morphology, Rhythm) | 1423 | 3.175134 |
| 6 | (Amplitude, Duration, Morphology, Rhythm) | 919 | 2.050561 |
| 7 | (Amplitude, Duration, Rhythm) | 733 | 1.635540 |
| 8 | (Morphology,) | 371 | 0.827811 |
| 9 | (Duration, Morphology) | 75 | 0.167347 |




## 10.3 Within-Record Label Patterns: Same Group vs. Multiple Groups

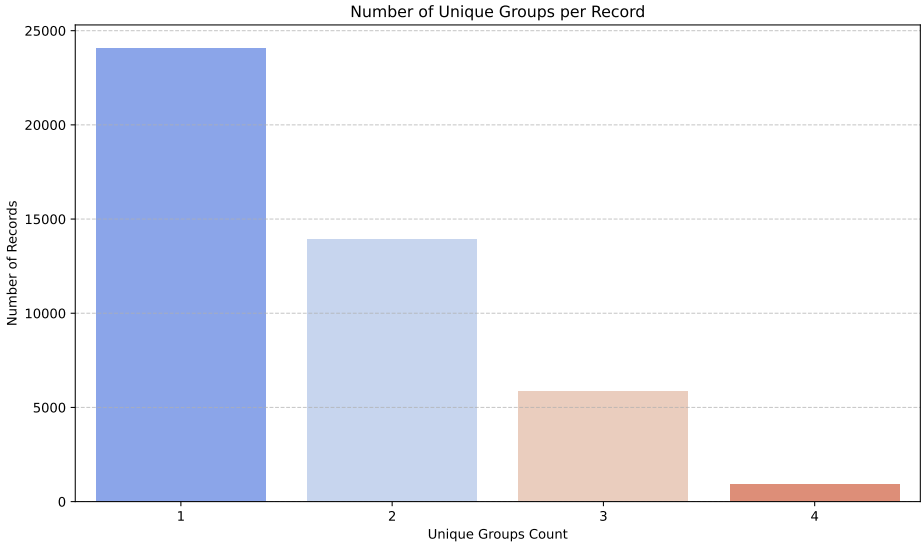
Here, we compare the total number of labels per record (raw count) with the number of unique groups (after merging labels within the same group). This comparison highlights records where multiple labels within the same group have been merged.

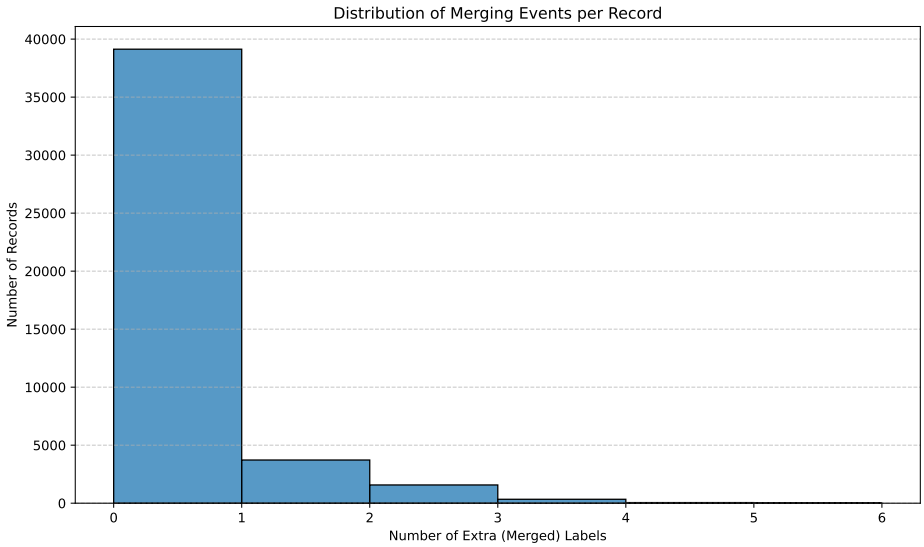
```
# Count total number of labels per record (raw, pre-merge)
metadata_df["total_labels"] = metadata_df["labels_metadata"].apply(lambda lst: len(lst) if i

# Count the number of unique groups per record (post merging)
metadata_df["num_unique_groups"] = metadata_df["unique_groups"].apply(len)

# The difference indicates how many extra labels per record have been merged
metadata_df["merging_applied"] = metadata_df.apply(lambda row: row["total_labels"] - row["nu

same_group_count = (metadata_df["total_labels"] == metadata_df["num_unique_groups"]).sum()
multiple_groups_count = (metadata_df["total_labels"] > metadata_df["num_unique_groups"]).sum

print(f"Records with all labels in different groups: {same_group_count}")
print(f"Records with duplicate labels in the same group (merging candidates): {multiple_grou

plt.figure(figsize=(10, 6))
sns.countplot(x="num_unique_groups", data=metadata_df, palette="coolwarm")
plt.title("Number of Unique Groups per Record")
plt.xlabel("Unique Groups Count")
plt.ylabel("Number of Records")
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 6))
sns.histplot(metadata_df["merging_applied"], bins=range(0, metadata_df["merging_applied"].ma
plt.title("Distribution of Merging Events per Record")
plt.xlabel("Number of Extra (Merged) Labels")
plt.ylabel("Number of Records")
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.tight_layout()
plt.show()
```

```
# In the subgroup merging logic, when a record has multiple labels in the sa
# only one label is kept (either the most common or the alphabetically first
# Here, we quantify the effect of this logic by comparing the total raw label
# with the merged labels (unique groups per record) in so called "merging ev

total_raw_labels = metadata_df["total_labels"].sum()
total_merged_labels = metadata_df["num_unique_groups"].sum()

print("Total number of labels (raw, pre-merge):", total_raw_labels)
print("Total number of labels (after merging to unique groups):", total_merge
print("Total number of merging events (labels removed):", total_raw_labels -

affected_records = (metadata_df["merging_applied"] > 0).sum()
print(f"Number of records affected by merging: {affected_records} out of {to

# Visualizing the effect on a sample of records
sample_records = metadata_df.sample(n=50, random_state=42).copy()
sample_records = sample_records.sort_index()

plt.figure(figsize=(12, 6))
plt.plot(sample_records.index, sample_records["total_labels"], "o-", label="
plt.plot(sample_records.index, sample_records["num_unique_groups"], "o-", la
plt.title("Comparison of Raw Label Count vs. Merged Label Count (Sample of R
plt.xlabel("Record Index (Sampled)")
plt.ylabel("Label Count")
plt.legend()
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.tight_layout()
plt.show()
```

```
Records with all labels in different groups: 39130
Records with duplicate labels in the same group (merging candidates): 5687
```

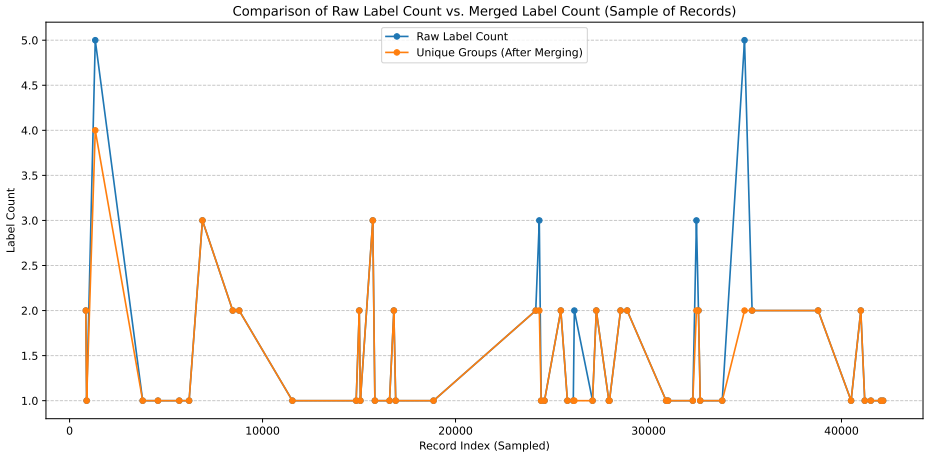
*10.3 Within-Record Label Patterns: Same Group vs. Multiple Groups*




```
Total number of labels (raw, pre-merge): 81336
```

```
Total number of labels (after merging to unique groups): 73224
Total number of merging events (labels removed): 8112
Number of records affected by merging: 5687 out of 44817 (12.69%)
```

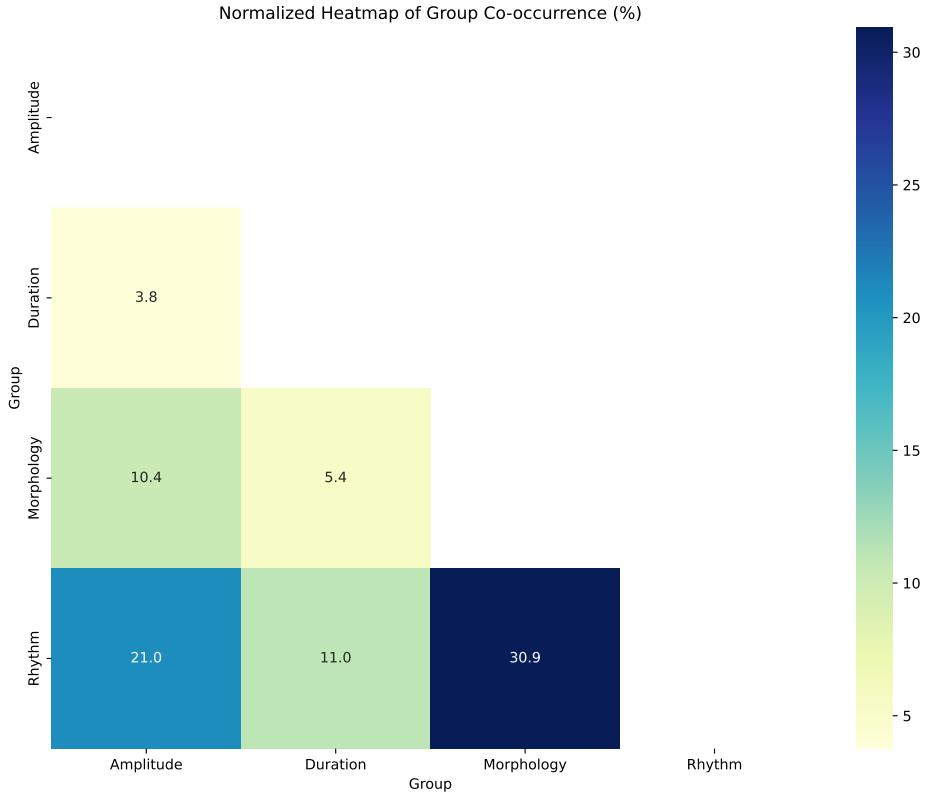



## 10.4 Heatmap of Group Co-occurrence

The heatmap below shows the normalized cross-correlation of group memberships. It displays the percentage of records that contain each pair of groups. For clarity, only the lower triangle of the matrix is shown.

```
all_groups = sorted(set(grp for groups in metadata_df["unique_groups"] for gr
co_occurrence = pd.DataFrame(0, index=all_groups, columns=all_groups)

for groups in metadata_df["unique_groups"]:
    for g1 in groups:
        for g2 in groups:
            co_occurrence.loc[g1, g2] += 1
```

```
# Normalize the co-occurrence counts to percentages (with respect to the total number of rec
co_occurrence_percent = (co_occurrence / len(metadata_df)) * 100

# Create a mask to hide the upper triangle of the heatmap
mask = np.triu(np.ones_like(co_occurrence_percent, dtype=bool))

plt.figure(figsize=(10, 8))
sns.heatmap(co_occurrence_percent, mask=mask, annot=True, fmt=".1f", cmap="YlGnBu")
plt.title("Normalized Heatmap of Group Co-occurrence (%)")
plt.xlabel("Group")
plt.ylabel("Group")
plt.tight_layout()
plt.show()
```


Normalized Heatmap of Group Co-occurrence (%)
Amplitude
Duration
Morphology
Rhythm
Group
3.8
10.4
5.4
21.0
11.0
30.9
Amplitude
Duration
Morphology
Rhythm
Group
30
25
20
15
10
5

# 11 Label Metadata Merging

In many records, multiple labels may belong to the same group. To simplify the analysis and improve consistency, we merge these labels based on a defined logic. For each group, if multiple labels are present, the merging function selects either the most common label (if available) or the alphabetically first label (based on the integration code). The following function encapsulates this merging logic.

```
# This function encapsulates the merging logic for a record's labels.
# For each group in the record, if multiple labels are present, it selects the most common l
# or otherwise the alphabetically first label (based on the integration code).
# Additionally, it uses the first available diagnosis name (from the "diagnosis_names" list)
#
# The function returns a dictionary mapping each group to a tuple (final_integration_code, f

def get_readable_label(meta):
    # Use the first diagnosis name if available; otherwise, fallback to the integration code
    names = meta.get("diagnosis_names")
    if isinstance(names, list) and len(names) > 0:
        return names[0]
    return meta.get("integration_code", "Unlabeled")

def merge_labels_for_record(label_metadata, most_common_label_by_group):
    """
    label_metadata: list of dictionaries, each containing keys 'group', 'integration_code',
    most_common_label_by_group: dict mapping group -> integration_code (most common across d
```

```
    Returns:
        dict: mapping group -> (final_integration_code, final_readable_label)
    """
    group_to_labels = {}
    for meta in label_metadata:
        group = meta.get("group", "Unknown").strip()
        code = meta.get("integration_code", "Unlabeled")
        readable = get_readable_label(meta)
        group_to_labels.setdefault(group, {})
        # Use a tuple (code, readable) to uniquely represent a label.
        group_to_labels[group][code] = readable

    merged = {}
    for group, labels in group_to_labels.items():
        # If the most common label is present, choose that.
        if most_common_label_by_group.get(group) in labels:
            chosen_code = most_common_label_by_group[group]
            chosen_readable = labels[chosen_code]
        else:
            # Otherwise, choose the label with the alphabetically first integ
            chosen_code = sorted(labels.keys())[0]
            chosen_readable = labels[chosen_code]
        merged[group] = (chosen_code, chosen_readable)
    return merged
```

## 11.1 Final Label Distribution Comparison (Raw vs. Merged)

Finally, we compare the raw label distributions with the final distributions after applying the merging logic. For each group, we normalize the counts to percentages and create side-by-side bar plots. This comparison helps illustrate the impact of the merging process on the dataset.

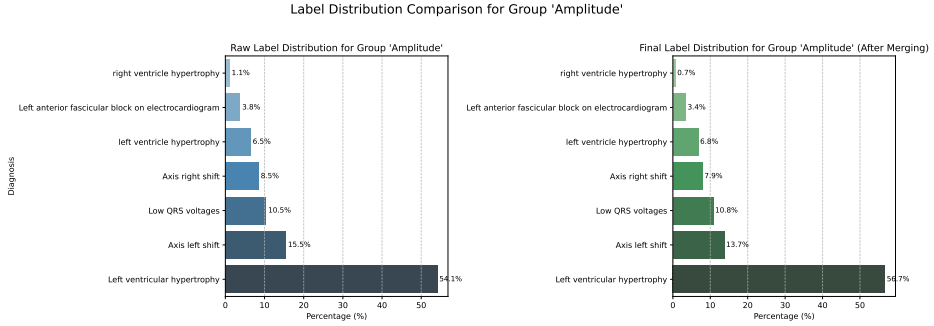
*11.1 Final Label Distribution Comparison (Raw vs. Merged)*

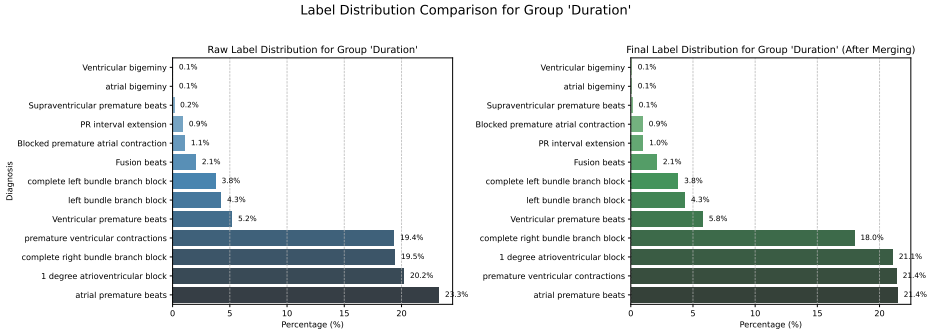
```
raw_label_counts = {}
for idx, row in metadata_df.iterrows():
    for meta in row["labels_metadata"]:
        group = meta.get("group", "Unknown").strip()
        # Use the first diagnosis name as the readable label.
        readable = get_readable_label(meta)
        raw_label_counts.setdefault(group, {})
        raw_label_counts[group][readable] = raw_label_counts[group].get(readable, 0) + 1

# Determine the most common integration code per group (based on raw counts) for merging log
most_common_label_by_group = {}
for group, counts in raw_label_counts.items():
    # Find the integration code corresponding to the highest count.
    # Since raw_label_counts is keyed by readable label, we need to recover the integration
    # For simplicity, we assume that the most common readable label corresponds uniquely to
    # To ensure consistency, we iterate over the records again.
    label_counter = {}
    for idx, row in metadata_df.iterrows():
        for meta in row["labels_metadata"]:
            if meta.get("group", "Unknown").strip() == group:
                code = meta.get("integration_code", "Unlabeled")
                label_counter[code] = label_counter.get(code, 0) + 1
    if label_counter:
        most_common_label_by_group[group] = max(label_counter, key=label_counter.get)
    else:
        most_common_label_by_group[group] = "Unlabeled"

# Now, simulate final (merged) label assignment across all records.
final_distribution = {}
for idx, row in metadata_df.iterrows():
    merged = merge_labels_for_record(row["labels_metadata"], most_common_label_by_group)
    for group, (code, readable) in merged.items():
        final_distribution.setdefault(group, {})
        final_distribution[group][readable] = final_distribution[group].get(readable, 0) + 1
```

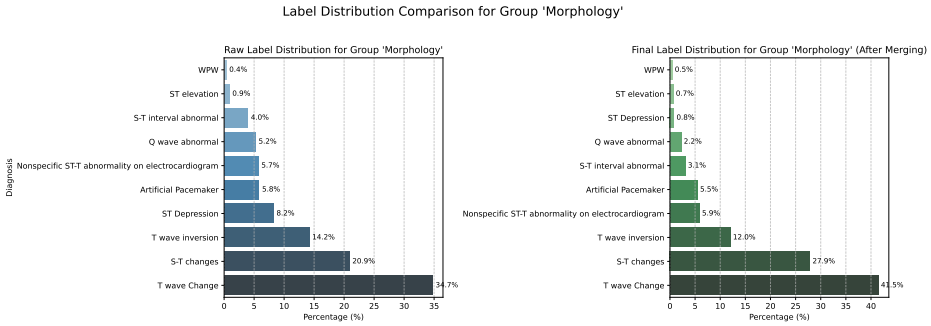
After computing the final label distributions, we plot side-by-side comparison plots for each group. The left plot shows the raw distribution, while the right plot displays the final distribution after merging.

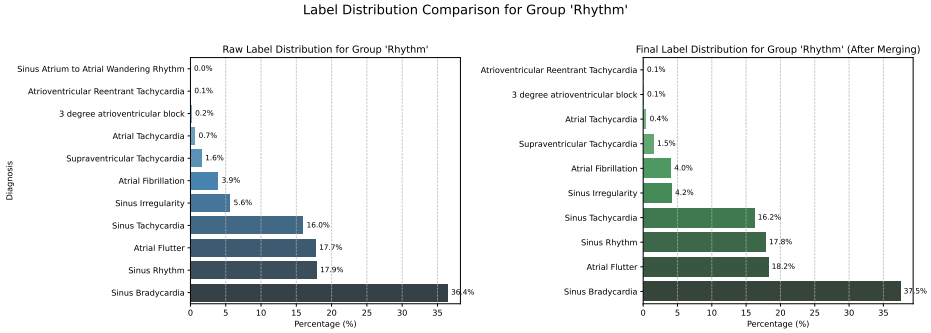
```
# For each group, create side-by-side comparison plots:
# Left: Raw distribution (normalized to percentage)
# Right: Final (merged) distribution (normalized to percentage)
groups_sorted = sorted(raw_label_counts.keys())

for group in groups_sorted:
    # Prepare raw data: Convert counts to percentages.
    raw_counts = raw_label_counts.get(group, {})
    total_raw = sum(raw_counts.values())
    raw_df = pd.DataFrame([
        {"diagnosis": diag, "count": cnt, "percentage": (cnt / total_raw) * 1
        for diag, cnt in raw_counts.items()
    ]).sort_values("percentage", ascending=True)  # sort ascending for horizo

    # Prepare final data: Convert counts to percentages.
    final_counts = final_distribution.get(group, {})
    total_final = sum(final_counts.values())
    final_df = pd.DataFrame([
        {"diagnosis": diag, "count": cnt, "percentage": (cnt / total_final) *
        for diag, cnt in final_counts.items()
    ]).sort_values("percentage", ascending=True)

    fig, axes = plt.subplots(1, 2, figsize=(16, 6))

    # Plot raw distribution using horizontal barplot
    sns.barplot(x="percentage", y="diagnosis", data=raw_df, ax=axes[0], palet
    axes[0].set_title(f"Raw Label Distribution for Group '{group}'")
    axes[0].set_xlabel("Percentage (%)")
    axes[0].set_ylabel("Diagnosis")
    axes[0].grid(axis="x", linestyle="--", alpha=0.7)
```

```
# Annotate each bar with its percentage
for patch in axes[0].patches:
    width = patch.get_width()
    y = patch.get_y() + patch.get_height() / 2
    axes[0].text(width + 0.5, y, f'{width:.1f}%', va='center', fontsize=9, color='black'

# Plot final (merged) distribution using horizontal barplot
sns.barplot(x="percentage", y="diagnosis", data=final_df, ax=axes[1], palette="Greens_d"
axes[1].set_title(f"Final Label Distribution for Group '{group}' (After Merging)")
axes[1].set_xlabel("Percentage (%)")
axes[1].set_ylabel("")
axes[1].grid(axis="x", linestyle="--", alpha=0.7)
# Annotate each bar with its percentage
for patch in axes[1].patches:
    width = patch.get_width()
    y = patch.get_y() + patch.get_height() / 2
    axes[1].text(width + 0.5, y, f'{width:.1f}%', va='center', fontsize=9, color='black'

plt.suptitle(f"Label Distribution Comparison for Group '{group}'", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```




Label Distribution Comparison for Group 'Duration'
Raw Label Distribution for Group 'Duration'
Final Label Distribution for Group 'Duration' (After Merging)
Diagnosis
Percentage (%)
Ventricular bigeminy 0.1%
atrial bigeminy 0.1%
Supraventricular premature beats 0.2%
PR interval extension 0.9%
Blocked premature atrial contraction 1.1%
Fusion beats 2.1%
complete left bundle branch block 3.8%
left bundle branch block 4.3%
Ventricular premature beats 5.2%
premature ventricular contractions 19.4%
complete right bundle branch block 19.5%
1 degree atrioventricular block 20.2%
atrial premature beats 23.3%
Blocked premature atrial contraction 0.9%
PR interval extension 1.0%
complete right bundle branch block 18.0%
1 degree atrioventricular block 21.1%
premature ventricular contractions 21.4%
atrial premature beats 21.4%


Label Distribution Comparison for Group 'Morphology'
Raw Label Distribution for Group 'Morphology'
Final Label Distribution for Group 'Morphology' (After Merging)
Diagnosis
Percentage (%)
WPW 0.4%
ST elevation 0.9%
S-T interval abnormal 4.0%
Q wave abnormal 5.2%
Nonspecific ST-T abnormality on electrocardiogram 5.7%
Artificial Pacemaker 5.8%
ST Depression 8.2%
T wave inversion 14.2%
S-T changes 20.9%
T wave Change 34.7%
WPW 0.5%
ST elevation 0.7%
ST Depression 0.8%
Q wave abnormal 2.2%
S-T interval abnormal 3.1%
Artificial Pacemaker 5.5%
Nonspecific ST-T abnormality on electrocardiogram 5.9%
T wave inversion 12.0%
S-T changes 27.9%
T wave Change 41.5%


Label Distribution Comparison for Group 'Rhythm'
Raw Label Distribution for Group 'Rhythm'
Final Label Distribution for Group 'Rhythm' (After Merging)
Diagnosis
Percentage (%)
Sinus Atrium to Atrial Wandering Rhythm 0.0%
Atrioventricular Reentrant Tachycardia 0.1%
3 degree atrioventricular block 0.2%
Atrial Tachycardia 0.7%
Supraventricular Tachycardia 1.6%
Atrial Fibrillation 3.9%
Sinus Irregularity 5.6%
Sinus Tachycardia 16.0%
Atrial Flutter 17.7%
Sinus Rhythm 17.9%
Sinus Bradycardia 36.4%
Atrioventricular Reentrant Tachycardia 0.1%
3 degree atrioventricular block 0.1%
Atrial Tachycardia 0.4%
Supraventricular Tachycardia 1.5%
Atrial Fibrillation 4.0%
Sinus Irregularity 4.2%
Sinus Tachycardia 16.2%
Sinus Rhythm 17.8%
Atrial Flutter 18.2%
Sinus Bradycardia 37.5%

## 11.2 Impact of Label Merging on Records by Group

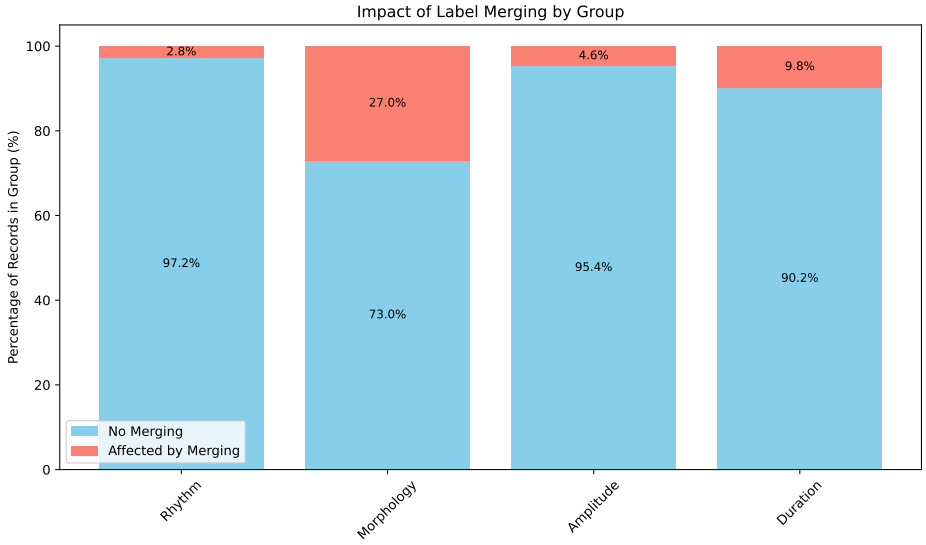
In many records, multiple labels may fall into the same group. In our merging logic, if a record has more than one label for a given group, these labels are merged into a single representative label. In this section, we quantify the impact of this merging process on a per-group basis. For each group, we compute the percentage of records that have multiple labels (and hence are impacted by merging) versus those that have only one label.

The stacked bar plot below shows, for each group, the percentage of records where merging is applied ("Affected by Merging") and those where no merging is needed ("No Merging"). The percentages are annotated on each bar segment. For clarification purposes, a record belong to a group if it has n >= 1 labels for that group. If n > 1, then merging is applied to obtain a single label. Thus, the "Affected by Merging" segment can also be looked at as the percentage of records where n > 1.

```
# Function to count the number of labels per group for a given record.
def count_labels_by_group(label_metadata):
    counts = {}
    if isinstance(label_metadata, list):
        for meta in label_metadata:
            group = meta.get("group", "Unknown").strip()
            counts[group] = counts.get(group, 0) + 1
    return counts

# For each record, count how many labels exist for each group.
metadata_df["group_label_counts"] = metadata_df["labels_metadata"].apply(count_labels_by_gro

# For each group, count records where merging is applied (more than one label) vs. not appli
group_stats = {}
for idx, row in metadata_df.iterrows():
    counts = row["group_label_counts"]
    for group, count in counts.items():
```

```
        if group not in group_stats:
            group_stats[group] = {"merged": 0, "not_merged": 0}
        if count > 1:
            group_stats[group]["merged"] += 1
        else:
            group_stats[group]["not_merged"] += 1

# Convert the group statistics to a DataFrame for plotting.
group_stats_df = pd.DataFrame([
    {
        "group": group,
        "merged": stats["merged"],
        "not_merged": stats["not_merged"],
        "total": stats["merged"] + stats["not_merged"]
    }
    for group, stats in group_stats.items()
])

# Calculate percentages for each group.
group_stats_df["pct_merged"] = 100 * group_stats_df["merged"] / group_stats_
group_stats_df["pct_not_merged"] = 100 * group_stats_df["not_merged"] / grou

# Sort groups for a cleaner plot.
group_stats_df = group_stats_df.sort_values("total", ascending=False)

# Create a stacked bar plot to visualize the impact of merging by group.
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the "No Merging" segment as the base.
bars_not = ax.bar(group_stats_df["group"], group_stats_df["pct_not_merged"],
                  label="No Merging", color="skyblue")

# Plot the "Merging Applied" segment on top.
bars_merge = ax.bar(group_stats_df["group"], group_stats_df["pct_merged"],
```

```
                    bottom=group_stats_df["pct_not_merged"], label="Affected by Merging", co

ax.set_ylabel("Percentage of Records in Group (%)")
ax.set_title("Impact of Label Merging by Group")
ax.legend()
plt.xticks(rotation=45)

# Annotate bar segments with the percentage values.
for rect in bars_not:
    height = rect.get_height()
    if height > 0:
        ax.text(rect.get_x() + rect.get_width()/2, rect.get_y() + height/2,
                f"{height:.1f}%", ha="center", va="center", fontsize=9)

for rect, base in zip(bars_merge, group_stats_df["pct_not_merged"]):
    height = rect.get_height()
    if height > 0:
        ax.text(rect.get_x() + rect.get_width()/2, base + height/2,
                f"{height:.1f}%", ha="center", va="center", fontsize=9)

plt.tight_layout()
plt.show()
```




The plot shows the per-group impact of label merging with clear percentage annotations on each segment, allowing you to see the exact proportion of records that required merging versus those that did not. We can see that especially the "Morphology" group has a high percentage of records where merging was applied (about 25%). For the other groups the percentage is lower.

## 11.3 Comparison of Label Combination Prevalence: Raw vs. Merged

In addition to the per-group merging impact, it is instructive to examine how label combination patterns change as a result of merging. In the raw data, a record may have duplicate group entries (e.g. `["A", "A", "B"]`), whereas after merging, each record contains only unique group labels (e.g. `["A", "B"]`). The following visualizations compare the prevalence of label combinations before and after merging.

*11.3 Comparison of Label Combination Prevalence: Raw vs. Merged*

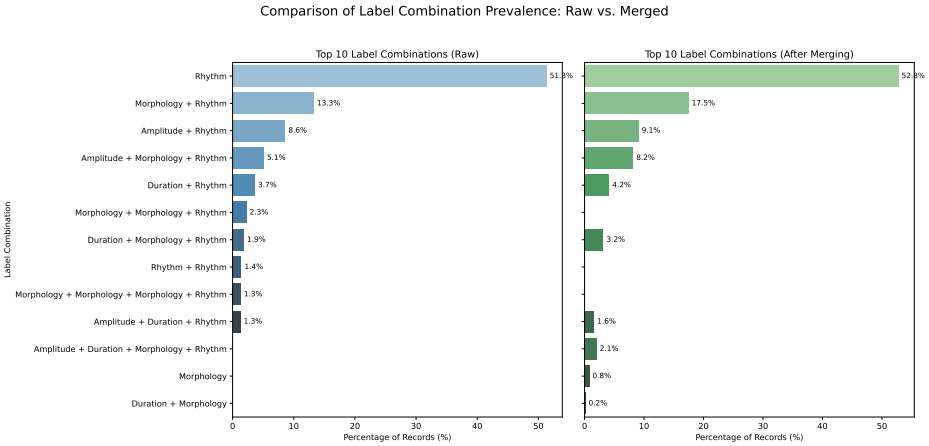
We create two new columns: one for the raw label combinations (including duplicates) and one for the merged (unique) combinations. To avoid issues with tuple-based categories, we convert each tuple into a string representation.

```
# Create a raw combination by collecting all group entries (including duplicates) per record
metadata_df["raw_combo"] = metadata_df["labels_metadata"].apply(
    lambda lst: tuple(sorted([meta.get("group", "Unknown").strip() for meta in lst]))
)

# The merged combination is already computed as unique groups.
metadata_df["merged_combo"] = metadata_df["unique_groups"].apply(lambda x: tuple(sorted(x)))

# Compute the frequency counts for both raw and merged combinations.
raw_combo_counts = metadata_df["raw_combo"].value_counts().reset_index()
raw_combo_counts.columns = ["combo", "count"]
merged_combo_counts = metadata_df["merged_combo"].value_counts().reset_index()
merged_combo_counts.columns = ["combo", "count"]

# Create string representations for the combinations.
raw_combo_counts["combo_str"] = raw_combo_counts["combo"].apply(lambda x: " + ".join(x) if i
merged_combo_counts["combo_str"] = merged_combo_counts["combo"].apply(lambda x: " + ".join(x

# Calculate percentages with respect to the total number of records.
total_records = len(metadata_df)
raw_combo_counts["percentage"] = 100 * raw_combo_counts["count"] / total_records
merged_combo_counts["percentage"] = 100 * merged_combo_counts["count"] / total_records
```

The side-by-side horizontal bar plots below show the top 10 most common label combinations before merging (raw) and after merging. Each bar is annotated with the percentage of records that have that combination.

```
fig, axes = plt.subplots(1, 2, figsize=(16, 8), sharey=True)
```

```
# Plot raw combination frequencies.
sns.barplot(x="percentage", y="combo_str", data=raw_combo_counts.head(10),
            ax=axes[0], palette="Blues_d")
axes[0].set_title("Top 10 Label Combinations (Raw)")
axes[0].set_xlabel("Percentage of Records (%)")
axes[0].set_ylabel("Label Combination")
for p in axes[0].patches:
    width = p.get_width()
    axes[0].text(width + 0.5, p.get_y() + p.get_height()/2,
                 f"{width:.1f}%", va="center", fontsize=9)

# Plot merged combination frequencies.
sns.barplot(x="percentage", y="combo_str", data=merged_combo_counts.head(10)
            ax=axes[1], palette="Greens_d")
axes[1].set_title("Top 10 Label Combinations (After Merging)")
axes[1].set_xlabel("Percentage of Records (%)")
axes[1].set_ylabel("")
for p in axes[1].patches:
    width = p.get_width()
    axes[1].text(width + 0.5, p.get_y() + p.get_height()/2,
                 f"{width:.1f}%", va="center", fontsize=9)

plt.suptitle("Comparison of Label Combination Prevalence: Raw vs. Merged", fo
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```




The plots show the top 10 most common label combinations before and after merging. The annotations on each bar segment provide a clear view of the percentage of records that have that combination. The merging process has effectively has left the set of labels per record as expected with unique group labels.

## 11.4 Diagnosis Co-occurrence Heatmaps (Using Diagnosis Names)

In addition to the previous visualizations based on group names, we now examine how diagnosis names co-occur across records. Because each label's metadata may contain multiple diagnosis names, we define a helper function that combines them into a single string (separated by `" / "`). We then compute two heatmaps: - One for the **raw diagnosis names** (from all labels per record), and
- One for the **merged diagnosis names** (after applying our merging logic).

### Raw Diagnosis Co-occurrence Heatmap

For the raw data, we extract the diagnosis name from each label and compute a co-occurrence matrix counting, for every pair of diagnosis names, in how many records they appear together.

```
# Define a helper function to combine diagnosis names from a label's metadata
def combined_diagnosis(meta):
    names = meta.get("diagnosis_names")
    if isinstance(names, list) and len(names) > 0:
        return " / ".join(sorted(set(names)))
    else:
        return meta.get("integration_code", "Unlabeled")

# For each record, compute the set of raw diagnosis names.
raw_diagnosis_sets = metadata_df["labels_metadata"].apply(
    lambda lst: set([combined_diagnosis(meta) for meta in lst]) if isinstance
)

# Get all unique diagnosis names from raw data.
all_raw_diagnoses = sorted({d for ds in raw_diagnosis_sets for d in ds})

# Initialize the co-occurrence matrix.
raw_co_occurrence = pd.DataFrame(0, index=all_raw_diagnoses, columns=all_raw_

# Populate the matrix: for each record, for each pair of diagnosis names, inc
for ds in raw_diagnosis_sets:
    for d1 in ds:
        for d2 in ds:
            raw_co_occurrence.loc[d1, d2] += 1

plt.figure(figsize=(12, 10))
sns.heatmap(raw_co_occurrence, annot=True, fmt="d", cmap="YlGnBu")
plt.title("Raw Diagnosis Co-occurrence Heatmap (Counts)")
```

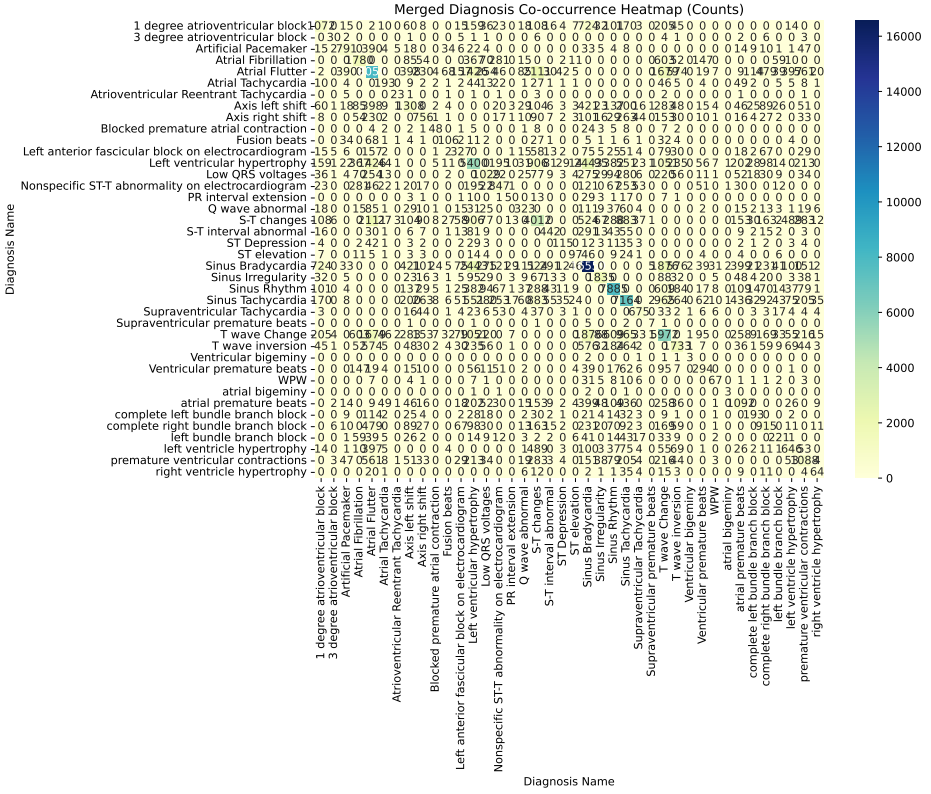
```
plt.xlabel("Diagnosis Name")
plt.ylabel("Diagnosis Name")
plt.tight_layout()
plt.show()
```


## Merged Diagnosis Co-occurrence Heatmap

For the merged labels, each record has at most one label per group. We use our previously defined merging function (`merge_labels_for_record`) and the dictionary `most_common_label_by_group` (computed earlier) to

obtain the merged labels. Then, we extract the diagnosis names from these merged results and compute their co-occurrence counts.

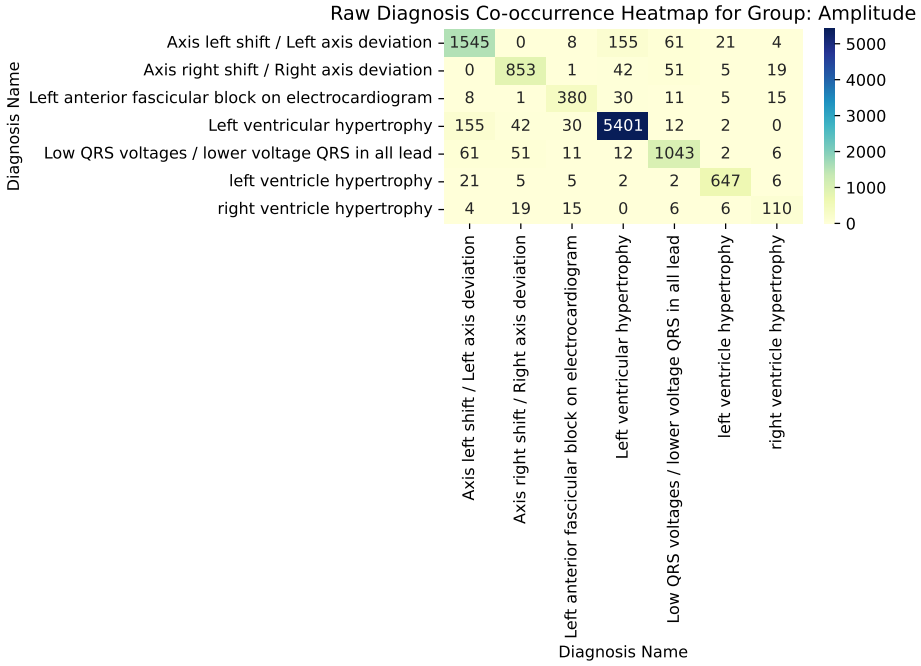
```
# Compute merged labels for each record (if not already computed).
metadata_df["merged_labels"] = metadata_df["labels_metadata"].apply(
    lambda lst: merge_labels_for_record(lst, most_common_label_by_group) if
)

# For each record, extract the set of merged diagnosis names.
merged_diagnosis_sets = metadata_df["merged_labels"].apply(
    lambda d: set(val[1] for val in d.values())
)

# Get all unique merged diagnosis names.
all_merged_diagnoses = sorted({d for ds in merged_diagnosis_sets for d in ds}

# Initialize the merged co-occurrence matrix.
merged_co_occurrence = pd.DataFrame(0, index=all_merged_diagnoses, columns=al

# Populate the matrix for merged diagnosis names.
for ds in merged_diagnosis_sets:
    for d1 in ds:
        for d2 in ds:
            merged_co_occurrence.loc[d1, d2] += 1

plt.figure(figsize=(12, 10))
sns.heatmap(merged_co_occurrence, annot=True, fmt="d", cmap="YlGnBu")
plt.title("Merged Diagnosis Co-occurrence Heatmap (Counts)")
plt.xlabel("Diagnosis Name")
plt.ylabel("Diagnosis Name")
plt.tight_layout()
plt.show()
```

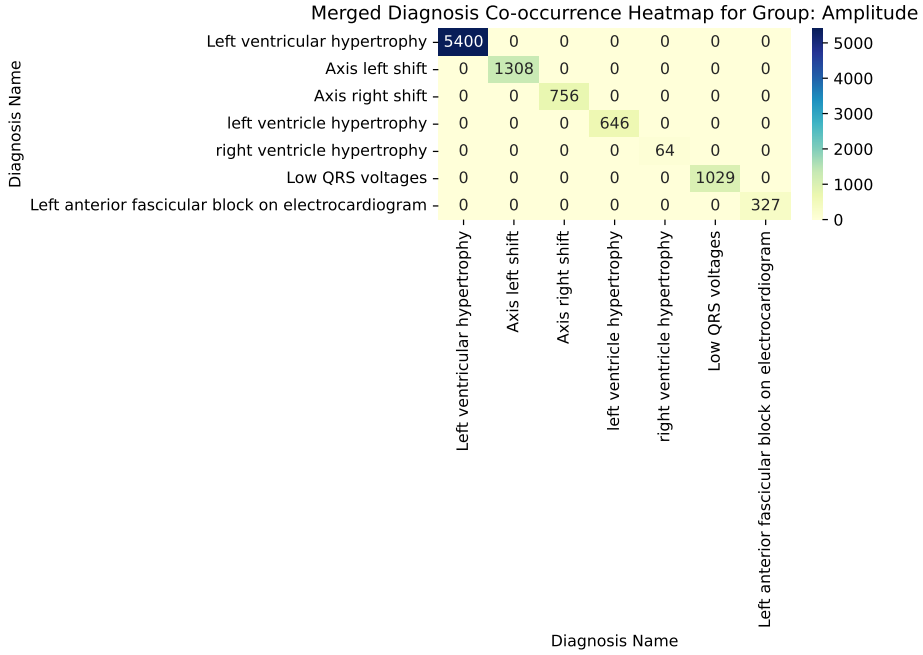


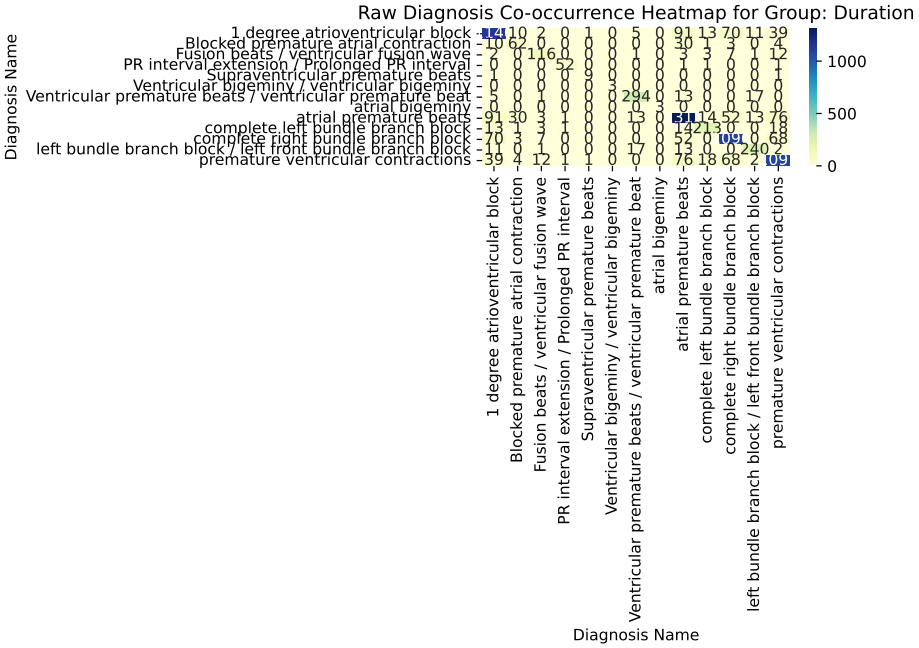

```
## Diagnosis Co-occurrence Heatmaps (Using Diagnosis Names) by Group

# Get list of all groups from the label metadata in the dataset.
all_groups = sorted({meta.get("group", "Unknown").strip()
                     for lst in metadata_df["labels_metadata"] if isinstance(lst, list)
                     for meta in lst})

# Define a helper function to combine diagnosis names from a label's metadata.
def combined_diagnosis(meta):
    names = meta.get("diagnosis_names")
    if isinstance(names, list) and len(names) > 0:
```

```
        # Combine multiple diagnosis names with a separator.
        return " / ".join(sorted(set(names)))
    else:
        return meta.get("integration_code", "Unlabeled")

# Loop over each group and compute separate heatmaps for raw and merged diagn
for grp in all_groups:
    ### RAW Diagnosis Co-occurrence for Group: grp
    # For each record, extract diagnosis names from raw labels that belong to
    raw_diag_sets = metadata_df["labels_metadata"].apply(
        lambda lst: set([combined_diagnosis(meta)
                         for meta in lst
                         if meta.get("group", "Unknown").strip() == grp])
                    if isinstance(lst, list) else set()
    )
    # Filter out records that don't have any diagnosis for this group.
    raw_diag_sets = raw_diag_sets[raw_diag_sets.apply(lambda s: len(s) > 0)]

    # Get all unique diagnosis names for this group.
    unique_raw_diag = sorted({d for s in raw_diag_sets for d in s})

    # Initialize the co-occurrence matrix for raw diagnosis names.
    raw_co_occ = pd.DataFrame(0, index=unique_raw_diag, columns=unique_raw_di

    # Populate the matrix: for each record, increment counts for every pair o
    for diag_set in raw_diag_sets:
        for d1 in diag_set:
            for d2 in diag_set:
                raw_co_occ.loc[d1, d2] += 1

    # Plot the heatmap for raw diagnosis co-occurrence for this group.
    plt.figure(figsize=(8, 6))
    sns.heatmap(raw_co_occ, annot=True, fmt="d", cmap="YlGnBu")
    plt.title(f"Raw Diagnosis Co-occurrence Heatmap for Group: {grp}")
```

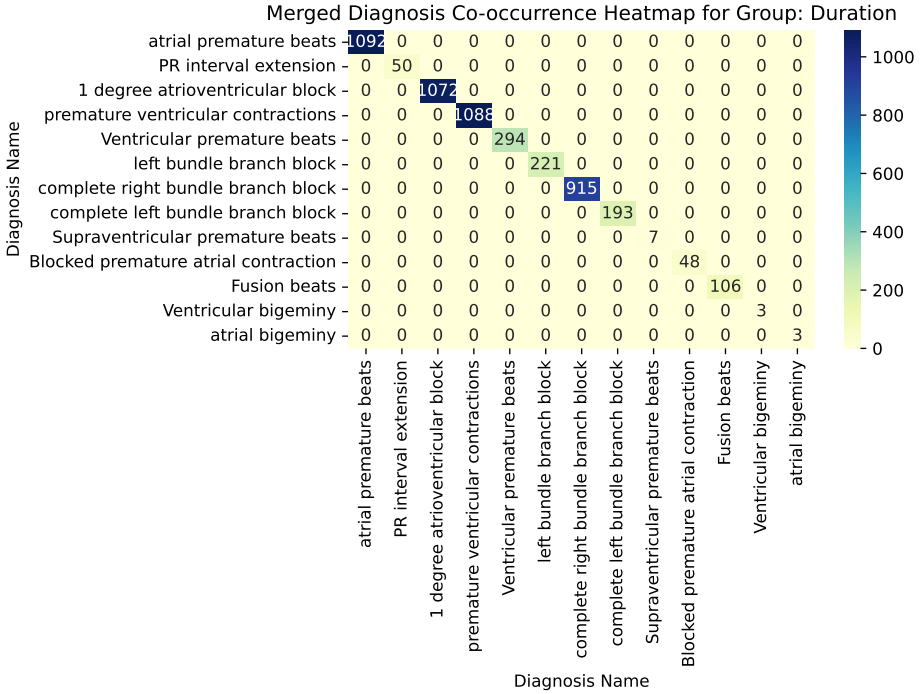
```
plt.xlabel("Diagnosis Name")
plt.ylabel("Diagnosis Name")
plt.tight_layout()
plt.show()

### Merged Diagnosis Co-occurrence for Group: grp
# For merged labels, each record has at most one label per group.
# Extract the merged diagnosis name for the current group.
merged_diag = metadata_df["merged_labels"].apply(
    lambda d: d.get(grp, (None, None))[1] if isinstance(d, dict) and grp in d else None
)
# Keep only non-None values.
merged_diag = merged_diag.dropna()

# In merged labels, since each record contributes only one diagnosis per group,
# the co-occurrence matrix will be diagonal (each record only "co-occurs" with itself).
unique_merged_diag = merged_diag.unique()
merged_counts = merged_diag.value_counts().sort_index()
merged_co_occ = pd.DataFrame(0, index=unique_merged_diag, columns=unique_merged_diag)
for diag in unique_merged_diag:
    merged_co_occ.loc[diag, diag] = merged_counts[diag]

# Plot the heatmap for merged diagnosis co-occurrence for this group.
plt.figure(figsize=(8, 6))
sns.heatmap(merged_co_occ, annot=True, fmt="d", cmap="YlGnBu")
plt.title(f"Merged Diagnosis Co-occurrence Heatmap for Group: {grp}")
plt.xlabel("Diagnosis Name")
plt.ylabel("Diagnosis Name")
plt.tight_layout()
plt.show()
```

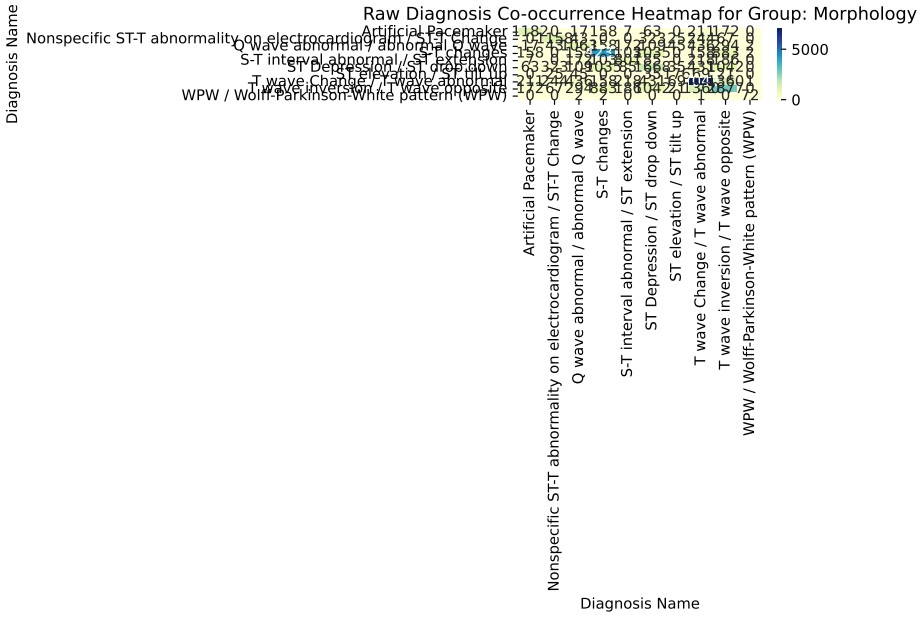

Raw Diagnosis Co-occurrence Heatmap for Group: Amplitude
Diagnosis Name
Axis left shift / Left axis deviation
Axis right shift / Right axis deviation
Left anterior fascicular block on electrocardiogram
Left ventricular hypertrophy
Low QRS voltages / lower voltage QRS in all lead
left ventricle hypertrophy
right ventricle hypertrophy
1545 0 8 155 61 21 4
0 853 1 42 51 5 19
8 1 380 30 11 5 15
155 42 30 5401 12 2 0
61 51 11 12 1043 2 6
21 5 5 2 2 647 6
4 19 15 0 6 6 110
5000
4000
3000
2000
1000
0
Diagnosis Name

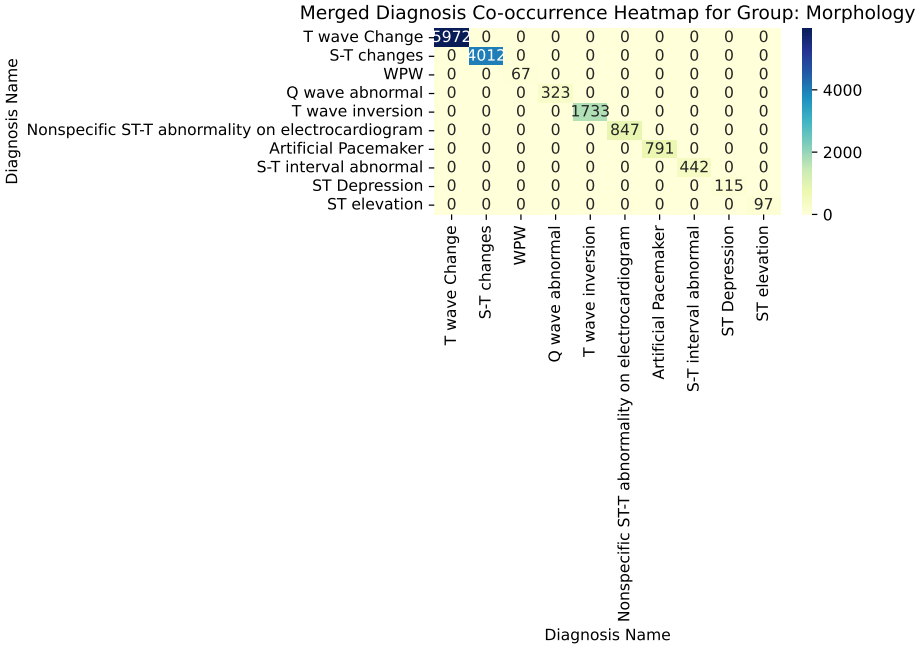

Merged Diagnosis Co-occurrence Heatmap for Group: Amplitude
Diagnosis Name
Left ventricular hypertrophy
Axis left shift
Axis right shift
left ventricle hypertrophy
right ventricle hypertrophy
Low QRS voltages
Left anterior fascicular block on electrocardiogram
5400 0 0 0 0 0 0
0 1308 0 0 0 0 0
0 0 756 0 0 0 0
0 0 0 646 0 0 0
0 0 0 0 64 0 0
0 0 0 0 0 1029 0
0 0 0 0 0 0 327
5000
4000
3000
2000
1000
0
Diagnosis Name

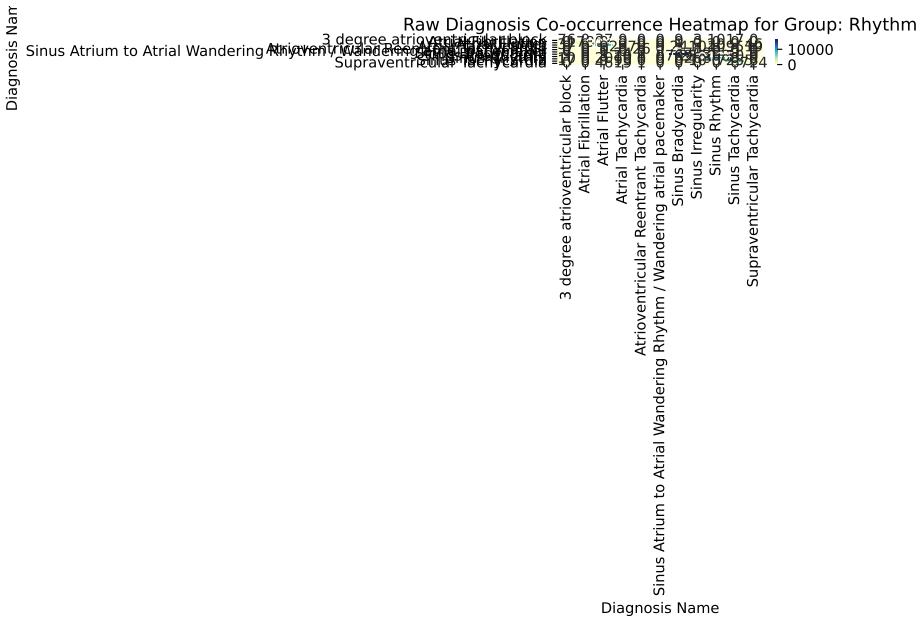

Raw Diagnosis Co-occurrence Heatmap for Group: Duration
Diagnosis Name
1 degree atrioventricular block
Blocked premature atrial contraction
Fusion beats / ventricular fusion wave
PR interval extension / Prolonged PR interval
Supraventricular premature beats
Ventricular bigeminy / ventricular bigeminy
Ventricular premature beats / ventricular premature beat
atrial bigeminy
atrial premature beats
complete left bundle branch block
complete right bundle branch block
left bundle branch block / left front bundle branch block
premature ventricular contractions
1000
500
0

## *11.4 Diagnosis Co-occurrence Heatmaps (Using Diagnosis Names)*

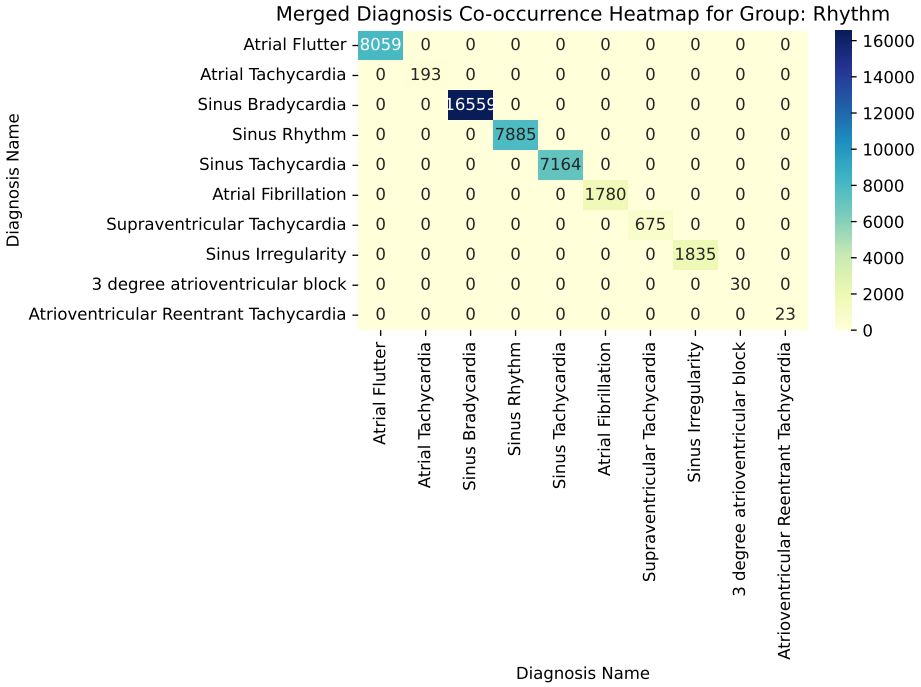



Raw Diagnosis Co-occurrence Heatmap for Group: Morphology
Diagnosis Name
Artificial Pacemaker
Nonspecific ST-T abnormality on electrocardiogram / ST-T Change
Q wave abnormal / abnormal Q wave
S-T changes
S-T interval abnormal / ST extension
ST Depression / ST drop down
ST elevation / ST tilt up
T wave Change / T wave abnormal
T wave inversion / T wave opposite
WPW / Wolff-Parkinson-White pattern (WPW)
5000
0


Merged Diagnosis Co-occurrence Heatmap for Group: Morphology
Diagnosis Name
T wave Change
S-T changes
WPW
Q wave abnormal
T wave inversion
Nonspecific ST-T abnormality on electrocardiogram
Artificial Pacemaker
S-T interval abnormal
ST Depression
ST elevation
5972 0 0 0 0 0 0 0 0 0
0 4012 0 0 0 0 0 0 0 0
0 0 67 0 0 0 0 0 0 0
0 0 0 323 0 0 0 0 0 0
0 0 0 0 1733 0 0 0 0 0
0 0 0 0 0 847 0 0 0 0
0 0 0 0 0 0 791 0 0 0
0 0 0 0 0 0 0 442 0 0
0 0 0 0 0 0 0 0 115 0
0 0 0 0 0 0 0 0 0 97
4000
2000
0
Diagnosis Name


Raw Diagnosis Co-occurrence Heatmap for Group: Rhythm
Diagnosis Name
Sinus Atrium to Atrial Wandering Rhythm / Wandering atrial pacemaker
Atrioventricular Reentrant Tachycardia
Atrial Tachycardia
Atrial Flutter
Atrial Fibrillation
3 degree atrioventricular block
Supraventricular Tachycardia
Sinus Tachycardia
Sinus Rhythm
Sinus Irregularity
Sinus Bradycardia
10000
0

## 11.4 Diagnosis Co-occurrence Heatmaps (Using Diagnosis Names)



# 12 Evaluation 1: Embedding Space Analysis (Pre-trained vs Fine-tuned)

In this notebook, we compare **Baseline (pre-trained)** vs. **Fine-tuned** embeddings on the full test set.
We highlight **single-labeled** records in color while the rest are shown in gray.

## 12.1 Imports

Here, we import Python standard libraries and our local project modules.

```
import sys
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import umap
from matplotlib.lines import Line2D
from matplotlib.patches import Patch
import matplotlib.patheffects as PathEffects
```

```
# Ensure project path is in sys.path
project_root = Path().absolute().parent.parent
sys.path.append(str(project_root))

# Our project modules
from src.visualization.embedding_viz import run_umap
from src.data.unified import UnifiedDataset
from src.data.dataset import DatasetModality

# Matplotlib style
plt.style.use("seaborn-v0_8-whitegrid")
plt.rcParams["figure.figsize"] = (16, 8)
plt.rcParams["font.size"] = 12
```

## 12.2 Color Palettes and Helpers

We define fixed colors for specific labels, as well as group-based color assignments.

```
SINGLE_COLOR_PALETTE = sns.color_palette("tab10", 11)

FIXED_LABEL_COLORS = {
    "SR": SINGLE_COLOR_PALETTE[0],     # Blue
    "AFIB": SINGLE_COLOR_PALETTE[1],   # Orange
    "SB": SINGLE_COLOR_PALETTE[2],     # Green
    "GSVT": SINGLE_COLOR_PALETTE[3],   # Red
    "PACE": SINGLE_COLOR_PALETTE[4],   # Purple
}

COLOR_PALETTES = {
    "Rhythm": SINGLE_COLOR_PALETTE,
    "Morphology": SINGLE_COLOR_PALETTE,
    "Duration": SINGLE_COLOR_PALETTE,
```

```
    "Amplitude": SINGLE_COLOR_PALETTE,
    "Other": SINGLE_COLOR_PALETTE,
}

def get_group_color_map(df_labels):
    """
    Generate a dict: group_label_map[group][label] -> color.
    df_labels must have 'integration_name' and 'group'.
    """
    # Get all unique labels
    all_labels = df_labels["integration_name"].unique()

    # Map each unique label to a color
    label_to_color = {}

    # First, use fixed colors for specific labels
    for label in all_labels:
        if label in FIXED_LABEL_COLORS:
            label_to_color[label] = FIXED_LABEL_COLORS[label]

    # Then assign colors to remaining labels
    color_idx = 0
    for label in all_labels:
        if label not in label_to_color:
            # Skip any colors used in FIXED_LABEL_COLORS
            while (color_idx < len(SINGLE_COLOR_PALETTE) and
                   any(SINGLE_COLOR_PALETTE[color_idx] == c
                       for c in FIXED_LABEL_COLORS.values())):
                color_idx += 1

            if color_idx < len(SINGLE_COLOR_PALETTE):
                label_to_color[label] = SINGLE_COLOR_PALETTE[color_idx]
                color_idx += 1
            else:
```

```
                # fallback color
                label_to_color[label] = (0.5, 0.5, 0.5)

    # Create the group structure
    group_label_map = {}
    for _, row in df_labels.iterrows():
        label = row["integration_name"]
        group = row["group"]
        if group not in group_label_map:
            group_label_map[group] = {}
        if label not in group_label_map[group]:
            group_label_map[group][label] = label_to_color[label]

    return group_label_map

# Distinct markers for each group
GROUP_MARKERS = {
    "Rhythm": "o",
    "Morphology": "s",
    "Duration": "^",
    "Amplitude": "D",
    "Other": "X",
}
```

## 12.3 Data Loading

Here, we load the **Arrhythmia (Chapman) test set** from our `UnifiedDataset`.
We then retrieve embeddings for **Baseline** and **Fine-Tuned** models.

```
print("Phase 1: UMAP on FULL data, highlighting single-labeled records.")
print("=" * 70)
```

```
arr_data = UnifiedDataset(
    Path(project_root) / "data", modality=DatasetModality.ECG, dataset_key="arrhythmia"
)
arr_splits = arr_data.get_splits()
arr_test_ids = arr_splits.get("test", [])

arr_md_store = arr_data.metadata_store

pretrained_embedding = "baseline"
finetuned_embedding = "fine_tuned_50"

records_info = []
emb_base_list = []
emb_ft_list = []

for rid in arr_test_ids:
    meta = arr_md_store.get(rid, {})
    labels_meta = meta.get("labels_metadata", [])

    try:
        emb_base = arr_data.get_embeddings(rid, embeddings_type=pretrained_embedding)
        emb_ft = arr_data.get_embeddings(rid, embeddings_type=finetuned_embedding)
    except Exception as e:
        print(f"Skipping {rid} (missing embeddings). Err: {e}")
        continue

    records_info.append(
        {"record_id": rid, "labels_meta": labels_meta, "n_labels": len(labels_meta)}
    )
    emb_base_list.append(emb_base)
    emb_ft_list.append(emb_ft)

if not records_info:
    print("No records found. Exiting.")
```

```
    sys.exit()

df_records = pd.DataFrame(records_info)
df_records["row_idx"] = df_records.index  # 0..N-1

# Stack embeddings
baseline_embeddings = np.vstack(emb_base_list)
finetuned_embeddings = np.vstack(emb_ft_list)

print(f"Total records loaded: {len(df_records)}")
print(" - Baseline shape:", baseline_embeddings.shape)
print(" - Fine-tuned shape:", finetuned_embeddings.shape)
```

```
Phase 1: UMAP on FULL data, highlighting single-labeled records.
======================================================================


Total records loaded: 6723
 - Baseline shape: (6723, 384)
 - Fine-tuned shape: (6723, 384)
```

## 12.4 Single-Labeled Subset

We now identify records that have **exactly one** label and store this subset separately.

```
mask_single = df_records["n_labels"] == 1
df_single = df_records[mask_single].copy()

df_single["integration_name"] = df_single["labels_meta"].apply(
    lambda lm: lm[0].get("integration_name", "unknown") if len(lm) == 1 else
)
df_single["group"] = df_single["labels_meta"].apply(
```

```
    lambda lm: lm[0].get("group", "Other") if len(lm) == 1 else "Other"
)

print("Single-labeled records:", len(df_single))
```

```
Single-labeled records: 3503
```

## 12.5 UMAP on the Full Dataset

We run UMAP on **all records** (both single- and multi-labeled) for a global view, then highlight single-labeled in the plot.

```
print("\nRunning UMAP (baseline & fine-tuned) on all records...")

umap_params = dict(n_neighbors=15, n_components=2, metric="euclidean", random_state=42)
baseline_umap = run_umap(baseline_embeddings, **umap_params)
finetuned_umap = run_umap(finetuned_embeddings, **umap_params)

print("UMAP finished.\n")

# Prepare color mapping for single-labeled points
single_labels = df_single[["integration_name", "group"]].drop_duplicates()
group_color_mapping = get_group_color_map(single_labels)
```

```
Running UMAP (baseline & fine-tuned) on all records...


UMAP finished.
```

## 12.6 Visualization

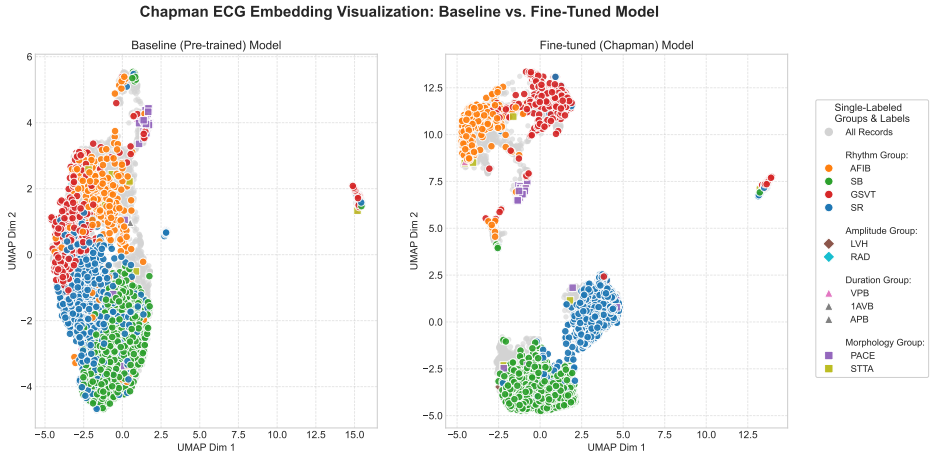
We create a **two-panel** figure comparing the **Baseline** vs. **Fine-tuned** spaces,
using gray for all records and colorful markers for single-labeled examples.

```
fig, axes = plt.subplots(1, 2, figsize=(16, 8))
fig.suptitle(
    "Chapman ECG Embedding Visualization: Baseline vs. Fine-Tuned Model",
    fontsize=18,
    fontweight="bold",
)

def plot_embedding(ax, emb_2d, title):
    ax.set_title(title, fontsize=14)

    # (1) All points in light gray
    ax.scatter(
        emb_2d[:, 0],
        emb_2d[:, 1],
        color="lightgray",
        edgecolor="none",
        s=40,
        alpha=0.6,
        label="All Records",
    )

    # (2) Overlay single-labeled points
    for row in df_single.itertuples():
        row_idx = row.row_idx
        label_name = getattr(row, "integration_name")
        group_name = getattr(row, "group")
```

```
            color = group_color_mapping[group_name].get(label_name, (0.5, 0.5, 0.5))
            marker = GROUP_MARKERS.get(group_name, "o")

            ax.scatter(
                emb_2d[row_idx, 0],
                emb_2d[row_idx, 1],
                c=[color],
                marker=marker,
                s=80,
                alpha=0.9,
                edgecolors="white",
                linewidth=0.5,
            )

    ax.set_xlabel("UMAP Dim 1", fontsize=12)
    ax.set_ylabel("UMAP Dim 2", fontsize=12)
    ax.grid(True, linestyle="--", alpha=0.5)

plot_embedding(axes[0], baseline_umap, "Baseline (Pre-trained) Model")
plot_embedding(axes[1], finetuned_umap, "Fine-tuned (Chapman) Model")

# Build legend
handles = []

# Handle for "All Records"
handles.append(
    Line2D(
        [0],
        [0],
        marker="o",
        color="lightgray",
        label="All Records",
        markersize=10,
        markeredgecolor="none",
```

```
        linewidth=0,
    )
)

# Group header + individual labels
unique_groups = single_labels["group"].unique()
for grp in unique_groups:
    handles.append(Patch(color="none", label=f"\n{grp} Group:"))
    grp_labels = single_labels[single_labels["group"] == grp]["integration_na
    mkr = GROUP_MARKERS.get(grp, "o")
    for lbl in grp_labels:
        c = group_color_mapping[grp].get(lbl, (0.5, 0.5, 0.5))
        handles.append(
            Line2D(
                [0],
                [0],
                marker=mkr,
                color="w",
                markerfacecolor=c,
                markersize=10,
                label=f"  {lbl}",
                linewidth=0,
            )
        )

legend = fig.legend(
    handles=handles,
    loc="center right",
    bbox_to_anchor=(1.05, 0.5),
    fontsize=11,
    frameon=True,
    fancybox=True,
    framealpha=0.95,
    title="Single-Labeled\nGroups & Labels",
```

```
    title_fontsize=12,
)

plt.savefig("baseline_vs_finetuned_embedding_visualization.png", dpi=150, bbox_inches="tight
plt.show()
```



# 13 Evaluation 2: Subcluster Analysis within Single-Label Groups

In this notebook, we analyze subclusters within specific **single-labeled** groups, focusing on dual-labeled records that share a primary label of interest.

## 13.1 Imports

```
import sys
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import umap
from matplotlib.lines import Line2D
from matplotlib.patches import Patch
from collections import defaultdict
from sklearn.cluster import KMeans, DBSCAN

# Ensure project path is in sys.path
project_root = Path().absolute().parent.parent
sys.path.append(str(project_root))

from src.visualization.embedding_viz import run_umap
```

```
from src.data.unified import UnifiedDataset
from src.data.dataset import DatasetModality

plt.style.use("seaborn-v0_8-whitegrid")
plt.rcParams["figure.figsize"] = (16, 8)
plt.rcParams["font.size"] = 12
```

## 13.2 Color Palettes and Helpers

```
SINGLE_COLOR_PALETTE = sns.color_palette("tab10", 11)
DUAL_COLOR_PALETTE = sns.color_palette("husl", 20)

FIXED_LABEL_COLORS = {
    "SR": SINGLE_COLOR_PALETTE[0],
    "AFIB": SINGLE_COLOR_PALETTE[1],
    "SB": SINGLE_COLOR_PALETTE[2],
    "GSVT": SINGLE_COLOR_PALETTE[3],
    "PACE": SINGLE_COLOR_PALETTE[4],
}

FIXED_SECONDARY_COLORS = {
    "STACH": (0.85, 0.37, 0.01),
    "SBRAD": (0.01, 0.66, 0.62),
    "SARRH": (0.58, 0.40, 0.74),
    "BIGU":  (0.17, 0.63, 0.17),
    "IVCD":  (0.84, 0.15, 0.16),
    "LAD":   (0.55, 0.35, 0.35),
    "RAD":   (0.94, 0.50, 0.50),
    "LVH":   (0.12, 0.47, 0.71),
    "RVH":   (0.68, 0.78, 0.91),
    "LNGQT": (0.46, 0.77, 0.35),
}
```

```
COLOR_PALETTES = {
    "Rhythm": SINGLE_COLOR_PALETTE,
    "Morphology": SINGLE_COLOR_PALETTE,
    "Duration": SINGLE_COLOR_PALETTE,
    "Amplitude": SINGLE_COLOR_PALETTE,
    "Other": SINGLE_COLOR_PALETTE,
}

GROUP_MARKERS = {
    "Rhythm": "o",
    "Morphology": "s",
    "Duration": "^",
    "Amplitude": "D",
    "Other": "X",
}

LABELS_OF_INTEREST = ["SR", "AFIB", "SB", "GSVT", "PACE"]
```

## 13.3 Downsampling, Outlier Removal, and Color-Mapping Utilities

```
def downsample_points(points, max_points=20, min_cluster_size=3, method="kmeans"):
    """
    Intelligently downsample points while preserving cluster structure.
    """
    if len(points) <= max_points:
        return points
    if method == "random":
        indices = np.random.choice(len(points), size=max_points, replace=False)
        return points[indices]
    elif method == "kmeans":
```

```
        n_clusters = min(max(min_cluster_size, len(points) // 5), max_points)
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        cluster_labels = kmeans.fit_predict(points)
        sampled_points = []
        for cluster_id in range(n_clusters):
            cluster_points = points[cluster_labels == cluster_id]
            if len(cluster_points) < 3:
                continue
            n_samples = max(1, int(max_points * len(cluster_points) / len(po
            idx = np.random.choice(len(cluster_points), size=min(n_samples,
            sampled_points.append(cluster_points[idx])
        if sampled_points:
            return np.vstack(sampled_points)
        else:
            return downsample_points(points, max_points, min_cluster_size, me
    elif method == "dbscan":
        dbscan = DBSCAN(eps=0.5, min_samples=min_cluster_size)
        cluster_labels = dbscan.fit_predict(points)
        sampled_points = []
        noise_points = points[cluster_labels == -1]
        if len(noise_points) > 0:
            n_noise_samples = min(max_points // 4, len(noise_points))
            if n_noise_samples > 0:
                idx = np.random.choice(len(noise_points), size=n_noise_sample
                sampled_points.append(noise_points[idx])
        unique_clusters = np.unique(cluster_labels)
        unique_clusters = unique_clusters[unique_clusters >= 0]
        for cluster_id in unique_clusters:
            cluster_points = points[cluster_labels == cluster_id]
            if len(cluster_points) < 3:
                continue
            n_samples = max(min_cluster_size, int(max_points * len(cluster_po
            idx = np.random.choice(len(cluster_points), size=min(n_samples,
            sampled_points.append(cluster_points[idx])
```

```
        if sampled_points:
            return np.vstack(sampled_points)
        else:
            return downsample_points(points, max_points, min_cluster_size, method="random")
    else:
        return downsample_points(points, max_points, min_cluster_size, method="random")

def remove_outliers_2d(points, z_thresh=3.0):
    """
    Removes outliers in 2D by z-scoring each dimension
    and discarding points beyond z_thresh in Euclidean distance.
    """
    if len(points) < 5:
        return points
    z_scores = np.abs((points - np.mean(points, axis=0)) / np.std(points, axis=0))
    z_dist = np.sqrt(np.sum(z_scores**2, axis=1))
    mask = z_dist < z_thresh
    return points[mask]

def get_group_color_map(df_labels):
    """
    Generate a dict: group_label_map[group][label] -> color.
    df_labels must have 'integration_name' and 'group'.
    """
    all_labels = df_labels["integration_name"].unique()
    label_to_color = {}
    for label in all_labels:
        if label in FIXED_LABEL_COLORS:
            label_to_color[label] = FIXED_LABEL_COLORS[label]
    color_idx = 0
    for label in all_labels:
        if label not in label_to_color:
            while color_idx < len(SINGLE_COLOR_PALETTE) and any(SINGLE_COLOR_PALETTE[color_i
                color_idx += 1
```

```
            if color_idx < len(SINGLE_COLOR_PALETTE):
                label_to_color[label] = SINGLE_COLOR_PALETTE[color_idx]
                color_idx += 1
            else:
                label_to_color[label] = (0.5, 0.5, 0.5)
    group_label_map = {}
    for _, row in df_labels.iterrows():
        label = row["integration_name"]
        group = row["group"]
        if group not in group_label_map:
            group_label_map[group] = {}
        if label not in group_label_map[group]:
            group_label_map[group][label] = label_to_color[label]
    return group_label_map
```

## 13.4 Data Loading & Metadata Extraction

```
print("Phase 2: Analyzing subclustering within single-label groups.")
arr_data = UnifiedDataset(Path(project_root) / "data", modality=DatasetModali
arr_splits = arr_data.get_splits()
arr_test_ids = arr_splits.get("test", [])
arr_md_store = arr_data.metadata_store

# Extract demographic metadata for all test records
demographic_data = {}
for rid in arr_test_ids:
    meta = arr_md_store.get(rid, {})
    age = meta.get("age", None)
    is_male = meta.get("is_male", None)
    demographic_data[rid] = {
        "age": age if isinstance(age, (int, float)) else None,
        "is_male": is_male if isinstance(is_male, bool) else None
```

```
    }
print(f"Extracted demographic data for {len(demographic_data)} records")

pretrained_embedding = "baseline"
finetuned_embedding = "fine_tuned_50"

def extract_extended_records_info(arr_test_ids, arr_md_store):
    records_info = []
    emb_base_list = []
    emb_ft_list = []
    for rid in arr_test_ids:
        meta = arr_md_store.get(rid, {})
        labels_meta = meta.get("labels_metadata", [])
        age = meta.get("age", None)
        is_male = meta.get("is_male", None)
        age = age if isinstance(age, (int, float)) else None
        is_male = is_male if isinstance(is_male, bool) else None
        try:
            emb_base = arr_data.get_embeddings(rid, embeddings_type=pretrained_embedding)
            emb_ft = arr_data.get_embeddings(rid, embeddings_type=finetuned_embedding)
        except Exception as e:
            print(f"Skipping {rid} (missing embeddings). Err: {e}")
            continue
        records_info.append({
            "record_id": rid,
            "labels_meta": labels_meta,
            "n_labels": len(labels_meta),
            "age": age,
            "is_male": is_male
        })
        emb_base_list.append(emb_base)
        emb_ft_list.append(emb_ft)
    return records_info, emb_base_list, emb_ft_list
```

```
records_info, emb_base_list, emb_ft_list = extract_extended_records_info(arr
if not records_info:
    print("No records found. Exiting.")
    sys.exit()

df_records = pd.DataFrame(records_info)
df_records["row_idx"] = df_records.index
baseline_embeddings = np.vstack(emb_base_list)
finetuned_embeddings = np.vstack(emb_ft_list)

print(f"Total records loaded: {len(df_records)}")
print(" - Baseline shape:", baseline_embeddings.shape)
print(" - Fine-tuned shape:", finetuned_embeddings.shape)
print(f" - With age data: {df_records['age'].notna().sum()}")
print(f" - With sex data: {df_records['is_male'].notna().sum()}")
```

```
Phase 2: Analyzing subclustering within single-label groups.
Extracted demographic data for 6723 records


Total records loaded: 6723
 - Baseline shape: (6723, 384)
 - Fine-tuned shape: (6723, 384)
 - With age data: 6720
 - With sex data: 6719
```

## 13.5 Single-Labeled and Dual-Labeled Records

```
mask_single = df_records["n_labels"] == 1
df_single = df_records[mask_single].copy()

df_single["integration_name"] = df_single["labels_meta"].apply(
    lambda lm: lm[0].get("integration_name", "unknown") if len(lm) == 1 else
```

```
)
df_single["group"] = df_single["labels_meta"].apply(
    lambda lm: lm[0].get("group", "Other") if len(lm) == 1 else "Other"
)
print("Single-labeled records:", len(df_single))

mask_dual = df_records["n_labels"] == 2
df_dual = df_records[mask_dual].copy()
dual_label_info = []

for _, row in df_dual.iterrows():
    labels_meta = row["labels_meta"]
    if len(labels_meta) != 2:
        continue
    label1 = labels_meta[0].get("integration_name", "unknown")
    label2 = labels_meta[1].get("integration_name", "unknown")
    group1 = labels_meta[0].get("group", "Other")
    group2 = labels_meta[1].get("group", "Other")

    primary_label = None
    secondary_label = None
    primary_group = None
    secondary_group = None

    if label1 in LABELS_OF_INTEREST:
        primary_label = label1
        secondary_label = label2
        primary_group = group1
        secondary_group = group2
    elif label2 in LABELS_OF_INTEREST:
        primary_label = label2
        secondary_label = label1
        primary_group = group2
        secondary_group = group1
```

```
    if primary_label:
        dual_label_info.append({
            "record_id": row["record_id"],
            "row_idx": row["row_idx"],
            "primary_label": primary_label,
            "primary_group": primary_group,
            "secondary_label": secondary_label,
            "secondary_group": secondary_group,
            "combo_label": f"{primary_label}+{secondary_label}"
        })

df_dual_filtered = pd.DataFrame(dual_label_info)
print(f"Dual-labeled records with one label in {LABELS_OF_INTEREST}:", len(di
```

```
Single-labeled records: 3503
Dual-labeled records with one label in ['SR', 'AFIB', 'SB', 'GSVT', 'PACE']:
```

## 13.6 UMAP on All Records

```
print("\nRunning UMAP (baseline & fine-tuned) on all records...")
umap_params = dict(n_neighbors=15, n_components=2, metric="euclidean", rando
baseline_umap = run_umap(baseline_embeddings, **umap_params)
finetuned_umap = run_umap(finetuned_embeddings, **umap_params)
print("UMAP finished.\n")

single_labels = df_single[["integration_name", "group"]].drop_duplicates()
group_color_mapping = get_group_color_map(single_labels)

def get_global_secondary_color_map(df_dual):
    all_secondary_labels = sorted(df_dual["secondary_label"].unique())
    secondary_color_map = {}
    for label in all_secondary_labels:
```

```
        if label in FIXED_SECONDARY_COLORS:
            secondary_color_map[label] = FIXED_SECONDARY_COLORS[label]
    remaining_labels = [l for l in all_secondary_labels if l not in secondary_color_map]
    remaining_colors = DUAL_COLOR_PALETTE[:len(remaining_labels)]
    for i, label in enumerate(remaining_labels):
        secondary_color_map[label] = remaining_colors[i]
    return secondary_color_map

global_secondary_color_map = get_global_secondary_color_map(df_dual_filtered)
primary_to_secondary = defaultdict(list)
for _, row in df_dual_filtered.iterrows():
    primary_to_secondary[row["primary_label"]].append(row["secondary_label"])
for primary in primary_to_secondary:
    primary_to_secondary[primary] = sorted(set(primary_to_secondary[primary]))
```

```
Running UMAP (baseline & fine-tuned) on all records...
```

```
UMAP finished.
```

## 13.7 Subcluster Plot for Each Primary Label

```
def plot_subclusters(ax, emb_2d, primary_label, title):
    ax.set_title(f"Label: {primary_label}", fontsize=14, fontweight='bold')
    primary_group = df_single[df_single["integration_name"] == primary_label]["group"].iloc[
    primary_color = group_color_mapping[primary_group][primary_label]
    single_mask = df_single["integration_name"] == primary_label
    single_rows = df_single[single_mask]
    if len(single_rows) == 0:
        ax.text(0.5, 0.5, f"No records with label {primary_label}",
                ha='center', va='center', transform=ax.transAxes)
        return
```

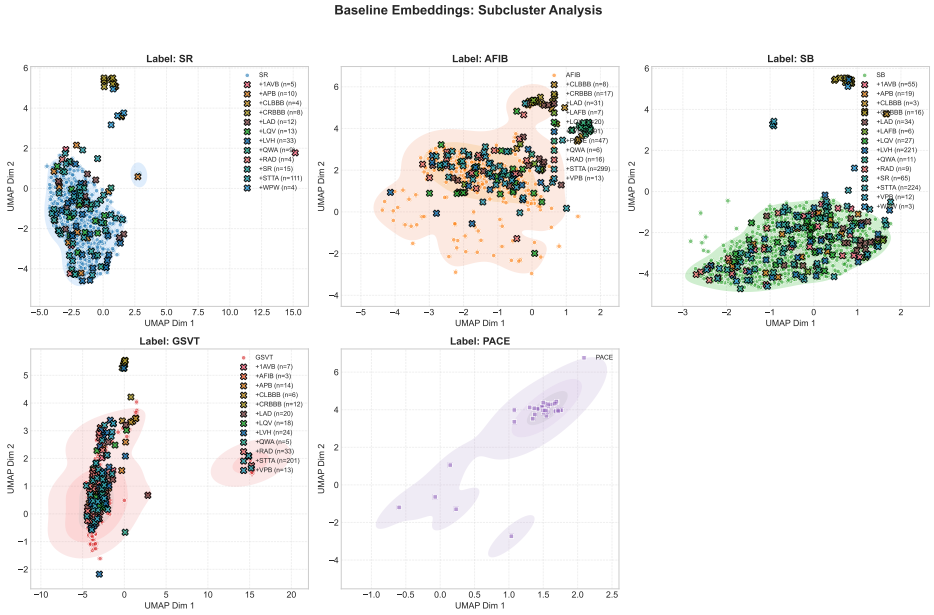
```
    primary_points = emb_2d[single_rows["row_idx"].values]
    clean_primary_points = remove_outliers_2d(primary_points, z_thresh=3.0)
    if len(clean_primary_points) >= 5:
        primary_df = pd.DataFrame(clean_primary_points, columns=["umap_x", "u
        sns.kdeplot(
            data=primary_df,
            x="umap_x",
            y="umap_y",
            fill=True,
            levels=4,
            alpha=0.25,
            color=primary_color,
            ax=ax,
            label=None
        )
        ax.scatter(
            primary_df["umap_x"],
            primary_df["umap_y"],
            c=[primary_color],
            marker=GROUP_MARKERS.get(primary_group, "o"),
            s=40,
            alpha=0.6,
            edgecolors="white",
            linewidth=0.3,
            label=f"{primary_label}"
        )
    dual_mask = df_dual_filtered["primary_label"] == primary_label
    if dual_mask.any():
        dual_rows = df_dual_filtered[dual_mask]
        secondary_labels = sorted(dual_rows["secondary_label"].unique())
        for secondary_label in secondary_labels:
            secondary_mask = dual_rows["secondary_label"] == secondary_label
            if not secondary_mask.any():
                continue
```

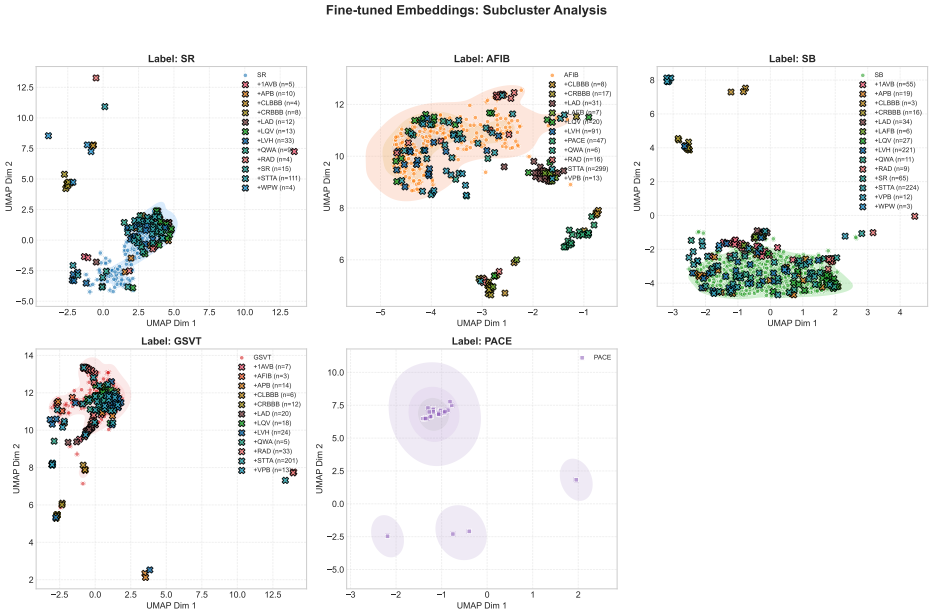
```
        secondary_points = emb_2d[dual_rows[secondary_mask]["row_idx"].values]
        if len(secondary_points) >= 3:
            max_points_per_secondary = 25
            secondary_points_sampled = downsample_points(
                secondary_points,
                max_points=max_points_per_secondary,
                min_cluster_size=3,
                method="kmeans"
            )
            secondary_color = global_secondary_color_map.get(secondary_label, (0.5, 0.5,
            ax.scatter(
                secondary_points_sampled[:, 0],
                secondary_points_sampled[:, 1],
                c=[secondary_color],
                marker="X",
                s=80,
                alpha=0.9,
                edgecolors="black",
                linewidth=0.5,
                label=f"+{secondary_label} (n={len(secondary_points)})"
            )
ax.set_xlabel("UMAP Dim 1", fontsize=12)
ax.set_ylabel("UMAP Dim 2", fontsize=12)
ax.grid(True, linestyle="--", alpha=0.3)
handles, labels = ax.get_legend_handles_labels()
if handles:
    ax.legend(handles=handles, labels=labels, loc="upper right", fontsize=9, framealpha=
```

## 13.8 Per-Label Subcluster Plots for Baseline and Fine-Tuned

```
for embedding_name, embedding_data in [("Baseline", baseline_umap), ("Fine-tu
    n_labels = len(LABELS_OF_INTEREST)
    n_cols = 3
    n_rows = (n_labels + n_cols - 1) // n_cols
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(18, 6*n_rows))
    fig.suptitle(f"{embedding_name} Embeddings: Subcluster Analysis", fontsiz
    axes = axes.flatten() if n_rows > 1 else [axes] if n_cols == 1 else axes
    for i, label in enumerate(LABELS_OF_INTEREST):
        if i < len(axes):
            plot_subclusters(axes[i], embedding_data, label, embedding_name)
    for j in range(n_labels, len(axes)):
        axes[j].set_visible(False)
    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show()
    plt.close()
```

## *13.8 Per-Label Subcluster Plots for Baseline and Fine-Tuned*






## 13.9 Extended Faceted Subcluster Plots

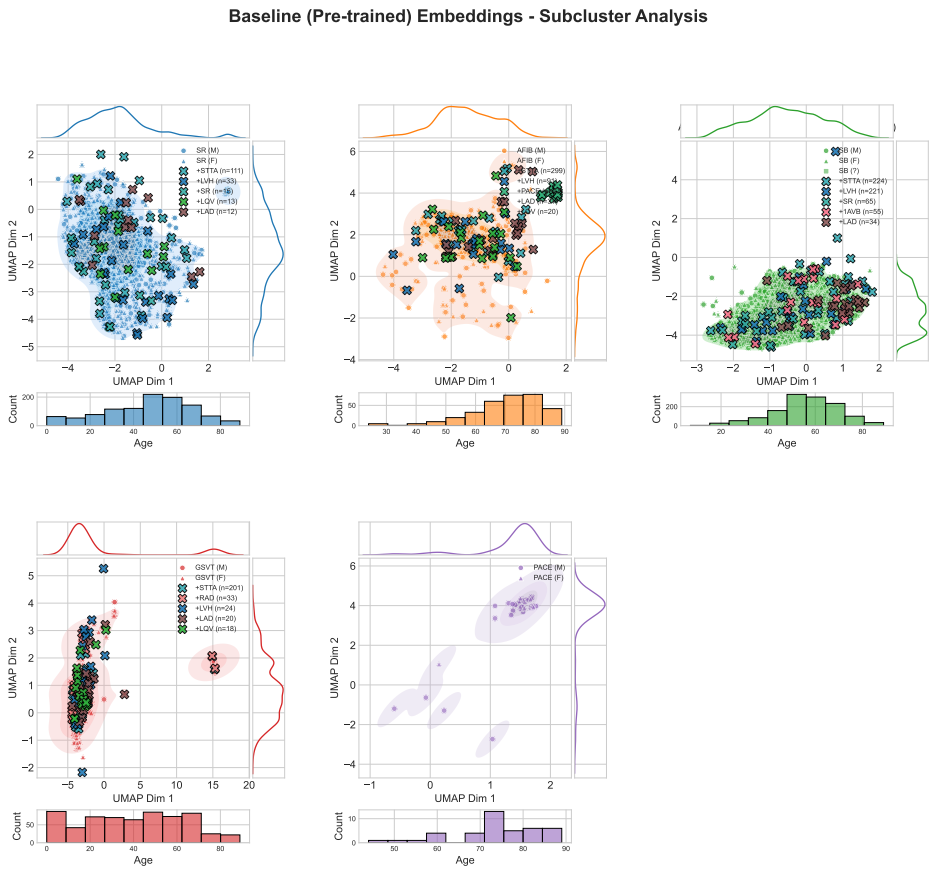
```
from mpl_toolkits.axes_grid1 import make_axes_locatable

def create_faceted_subcluster_plots(embedding_data, title, figsize=(18, 15)):
    n_labels = len(LABELS_OF_INTEREST)
    n_cols = 3
    n_rows = (n_labels + n_cols - 1) // n_cols
    fig = plt.figure(figsize=figsize)
    gs = fig.add_gridspec(n_rows, n_cols, hspace=0.3, wspace=0.3, height_rati
    fig.suptitle(f"{title}", fontsize=20, fontweight='bold', y=0.98)
    axes = []
    for i in range(n_rows):
        for j in range(n_cols):
```

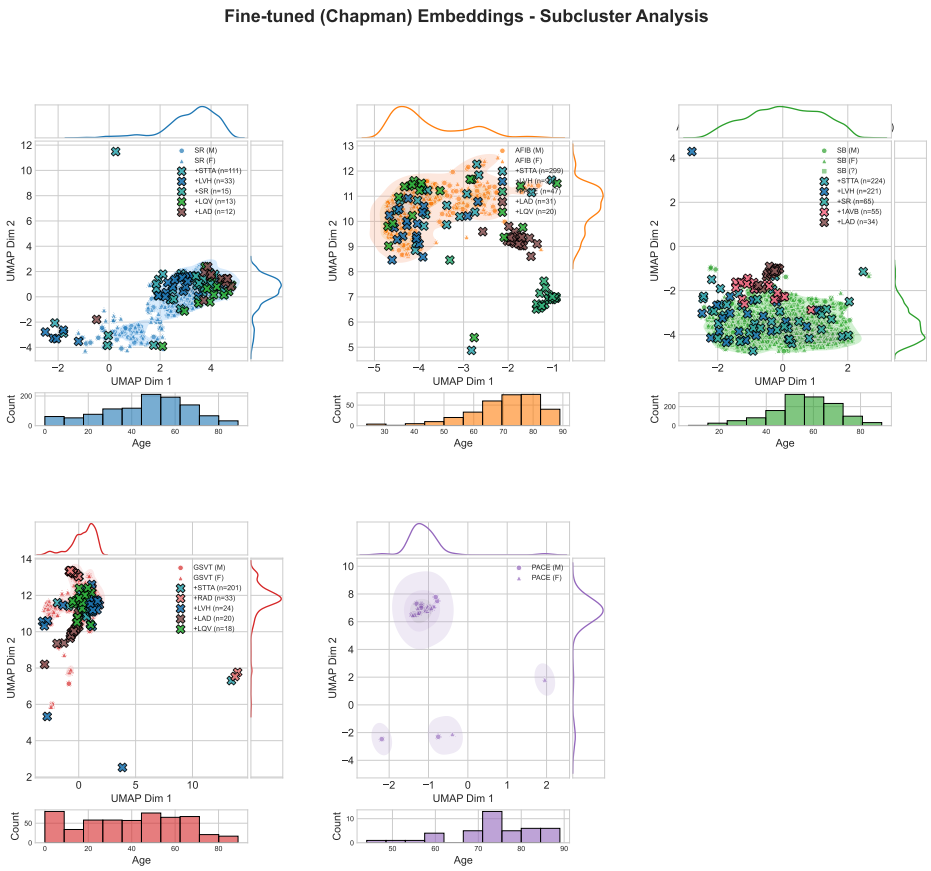
```
            idx = i * n_cols + j
            if idx < n_labels:
                axes.append(fig.add_subplot(gs[i, j]))
            else:
                ax = fig.add_subplot(gs[i, j])
                ax.set_visible(False)
                axes.append(ax)
    for i, primary_label in enumerate(LABELS_OF_INTEREST):
        if i >= len(axes):
            break
        ax = axes[i]
        primary_group = df_single[df_single["integration_name"] == primary_label]["group"].i
        primary_color = group_color_mapping[primary_group][primary_label]
        single_mask = df_single["integration_name"] == primary_label
        single_rows = df_single[single_mask]
        if len(single_rows) < 5:
            ax.text(0.5, 0.5, f"Not enough data for {primary_label}",
                    ha='center', va='center', transform=ax.transAxes)
            continue
        primary_points = embedding_data[single_rows["row_idx"].values]
        clean_primary_points = remove_outliers_2d(primary_points, z_thresh=3.0)
        primary_df = pd.DataFrame(clean_primary_points, columns=["umap_x", "umap_y"])
        primary_df["record_id"] = single_rows["record_id"].values[:len(primary_df)]
        primary_df["age"] = primary_df["record_id"].apply(lambda rid: demographic_data.get(r
        primary_df["is_male"] = primary_df["record_id"].apply(lambda rid: demographic_data.g
        valid_ages = primary_df["age"].dropna()
        age_stats = f"Age: {valid_ages.mean():.1f}±{valid_ages.std():.1f}" if len(valid_ages
        male_count = primary_df["is_male"].sum()
        female_count = (primary_df["is_male"] == False).sum()
        unknown_sex = primary_df["is_male"].isna().sum()
        sex_stats = f"Sex: {male_count}M/{female_count}F"
        if unknown_sex > 0:
            sex_stats += f" ({unknown_sex} unknown)"
        if "is_male" in primary_df.columns and not primary_df["is_male"].isna().all():
```

```
            male_df = primary_df[primary_df["is_male"] == True]
            if len(male_df) > 0:
                sns.scatterplot(
                    data=male_df, x="umap_x", y="umap_y",
                    color=primary_color, marker="o", s=40, alpha=0.7, ax=ax,
                )
            female_df = primary_df[primary_df["is_male"] == False]
            if len(female_df) > 0:
                sns.scatterplot(
                    data=female_df, x="umap_x", y="umap_y",
                    color=primary_color, marker="^", s=40, alpha=0.7, ax=ax,
                )
            unknown_df = primary_df[primary_df["is_male"].isna()]
            if len(unknown_df) > 0:
                sns.scatterplot(
                    data=unknown_df, x="umap_x", y="umap_y",
                    color=primary_color, marker="s", s=40, alpha=0.5, ax=ax,
                )
        else:
            sns.scatterplot(
                data=primary_df, x="umap_x", y="umap_y",
                color=primary_color, s=40, alpha=0.7, ax=ax, label=primary_la
            )
        if len(primary_df) >= 5:
            sns.kdeplot(
                data=primary_df, x="umap_x", y="umap_y",
                fill=True, levels=4, alpha=0.25, color=primary_color,
                bw_adjust=0.7, ax=ax
            )
        dual_mask = df_dual_filtered["primary_label"] == primary_label
        if dual_mask.any():
            dual_rows = df_dual_filtered[dual_mask]
            secondary_labels = sorted(dual_rows["secondary_label"].unique())
            top_secondaries = dual_rows["secondary_label"].value_counts().nla
```

```
    for secondary_label in top_secondaries:
        secondary_mask = dual_rows["secondary_label"] == secondary_label
        if not secondary_mask.any():
            continue
        secondary_points = embedding_data[dual_rows[secondary_mask]["row_idx"].value
        if len(secondary_points) >= 3:
            secondary_points_sampled = downsample_points(
                secondary_points, max_points=25, min_cluster_size=3, method="kmeans"
            )
            secondary_color = global_secondary_color_map.get(secondary_label, (0.5,
            ax.scatter(
                secondary_points_sampled[:, 0],
                secondary_points_sampled[:, 1],
                c=[secondary_color], marker="X", s=100,
                alpha=0.9, edgecolors="black", linewidth=0.5,
                label=f"+{secondary_label} (n={len(secondary_points)})"
            )
ax.legend(fontsize=8, loc='upper right', framealpha=0.7)
ax.set_xlabel("UMAP Dim 1")
ax.set_ylabel("UMAP Dim 2")
ax.set_aspect('auto')
ax.set_title(f"Label: {primary_label}\n{age_stats}, {sex_stats}", fontsize=12, pad=1
xlim = ax.get_xlim()
ylim = ax.get_ylim()
divider = make_axes_locatable(ax)
ax_top = divider.append_axes("top", size="15%", pad=0.05)
ax_top.tick_params(axis='both', which='both', labelbottom=False, labelleft=False)
if len(primary_df) >= 5:
    sns.kdeplot(data=primary_df, x="umap_x", color=primary_color, bw_adjust=0.7, ax=
    ax_top.set_xlim(xlim)
ax_top.set_yticks([])
ax_right = divider.append_axes("right", size="15%", pad=0.05)
ax_right.tick_params(axis='both', which='both', labelbottom=False, labelleft=False)
if len(primary_df) >= 5:
```

```
            sns.kdeplot(data=primary_df, y="umap_y", color=primary_color, bw
            ax_right.set_ylim(ylim)
        ax_right.set_xticks([])
        valid_ages = primary_df["age"].dropna()
        if valid_ages.size > 10:
            ax_age = divider.append_axes("bottom", size="15%", pad=0.5)
            sns.histplot(valid_ages, bins=10, color=primary_color, alpha=0.6
            ax_age.set_xlabel("Age")
            ax_age.set_ylabel("Count")
            ax_age.tick_params(labelsize=8)
    plt.tight_layout(rect=[0, 0, 1, 0.95], h_pad=0.2, w_pad=0.2)
    return fig

create_faceted_subcluster_plots(baseline_umap, "Baseline (Pre-trained) Embedd
plt.show()
plt.close()

create_faceted_subcluster_plots(finetuned_umap, "Fine-tuned (Chapman) Embeddi
plt.show()
plt.close()
```


Baseline (Pre-trained) Embeddings - Subcluster Analysis
UMAP Dim 1
UMAP Dim 2
Count
Age




## 13.10 Demographic Subcluster Plots

```
def create_demographic_subcluster_plots(embedding_data, title, figsize=(18, 2
    n_labels = len(LABELS_OF_INTEREST)
    n_cols = 3
    n_rows = 2 * ((n_labels + n_cols - 1) // n_cols)
    fig = plt.figure(figsize=figsize)
```

```
gs = fig.add_gridspec(n_rows, n_cols, hspace=0.3, wspace=0.3)
fig.suptitle(f"{title} - Demographic Analysis", fontsize=20, fontweight='bold', y=0.98)
axes = []
for i in range(n_rows):
    for j in range(n_cols):
        idx = i * n_cols + j
        if idx < 2 * n_labels:
            axes.append(fig.add_subplot(gs[i, j]))
        else:
            ax = fig.add_subplot(gs[i, j])
            ax.set_visible(False)
            axes.append(ax)
for i, primary_label in enumerate(LABELS_OF_INTEREST):
    if 2*i >= len(axes):
        break
    ax_sex = axes[2*i]
    ax_age = axes[2*i+1]
    ax_sex.set_title(f"{primary_label} - By Sex", fontsize=12, pad=10)
    ax_age.set_title(f"{primary_label} - By Age", fontsize=12, pad=10)
    primary_group = df_single[df_single["integration_name"] == primary_label]["group"].i
    primary_color = group_color_mapping[primary_group][primary_label]
    single_mask = df_single["integration_name"] == primary_label
    single_rows = df_single[single_mask]
    if len(single_rows) < 5:
        for ax_demo in [ax_sex, ax_age]:
            ax_demo.text(0.5, 0.5, f"Not enough data for {primary_label}",
                         ha='center', va='center', transform=ax_demo.transAxes)
        continue
    primary_points = embedding_data[single_rows["row_idx"].values]
    clean_primary_points = remove_outliers_2d(primary_points, z_thresh=3.0)
    primary_df = pd.DataFrame(clean_primary_points, columns=["umap_x", "umap_y"])
    primary_df["record_id"] = single_rows["record_id"].values[:len(primary_df)]
    primary_df["age"] = primary_df["record_id"].apply(lambda rid: demographic_data.get(r
    primary_df["is_male"] = primary_df["record_id"].apply(lambda rid: demographic_data.g
```

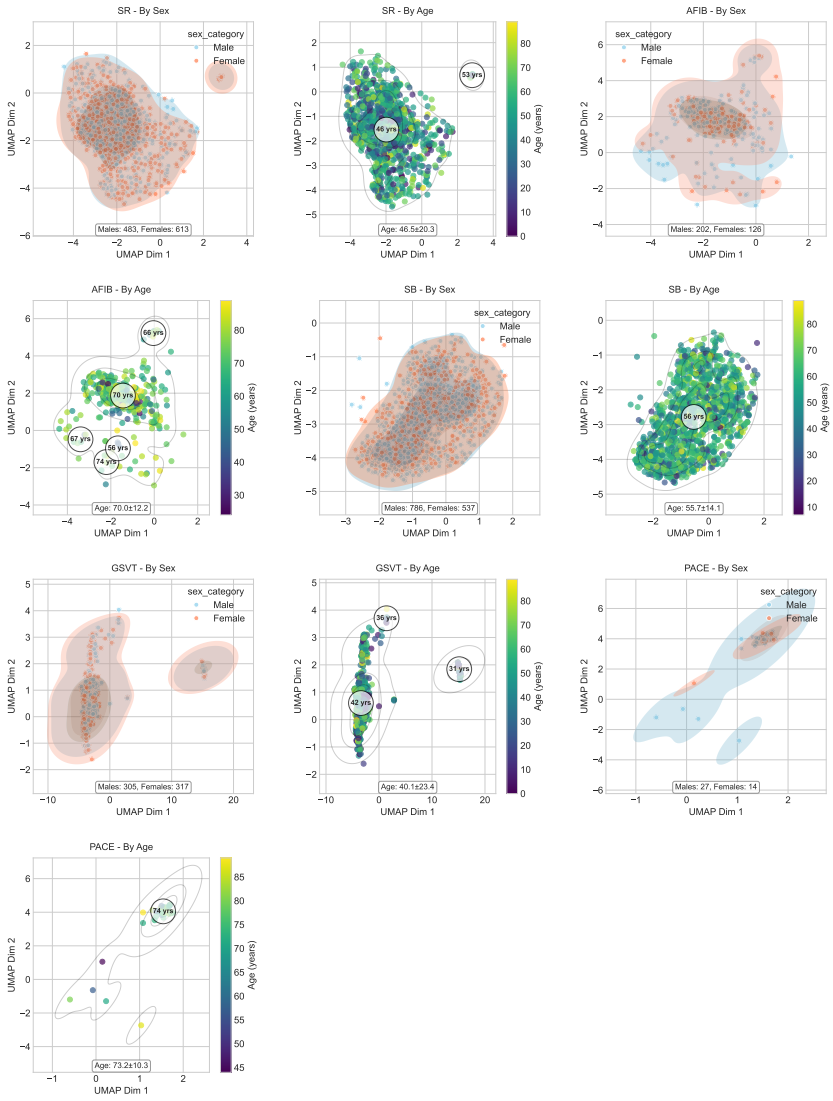
```
sex_df = primary_df.dropna(subset=["is_male"]).copy()
age_df = primary_df.dropna(subset=["age"]).copy()
if not sex_df.empty:
    sex_df["sex_category"] = sex_df["is_male"].apply(lambda x: "Male"
    sex_palette = {"Male": "skyblue", "Female": "coral"}
    sns.scatterplot(
        data=sex_df,
        x="umap_x",
        y="umap_y",
        hue="sex_category",
        palette=sex_palette,
        s=40, alpha=0.7, ax=ax_sex
    )
    for sex_cat in ["Male", "Female"]:
        sex_group = sex_df[sex_df["sex_category"] == sex_cat]
        if len(sex_group) >= 10:
            sns.kdeplot(
                data=sex_group, x="umap_x", y="umap_y",
                levels=3, alpha=0.3, fill=True,
                color=sex_palette[sex_cat], ax=ax_sex
            )
    male_count = sex_df["is_male"].sum()
    female_count = (sex_df["is_male"] == False).sum()
    sex_stats = f"Males: {male_count}, Females: {female_count}"
    ax_sex.annotate(
        sex_stats, xy=(0.5, 0.02), xycoords="axes fraction",
        ha="center", fontsize=10,
        bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", al
    )
else:
    ax_sex.text(0.5, 0.5, "No sex data available",
                ha='center', va='center', transform=ax_sex.transAxes)
if not age_df.empty:
    norm = plt.Normalize(age_df["age"].min(), age_df["age"].max())
```

```
sm = plt.cm.ScalarMappable(cmap="viridis", norm=norm)
sm.set_array([])
ax_age.scatter(
    age_df["umap_x"],
    age_df["umap_y"],
    c=age_df["age"],
    cmap="viridis",
    s=40,
    alpha=0.7
)
cbar = plt.colorbar(sm, ax=ax_age)
cbar.set_label("Age (years)")
if len(age_df) >= 10:
    sns.kdeplot(
        data=age_df, x="umap_x", y="umap_y",
        levels=4, alpha=0.2, linewidths=1,
        color="black", ax=ax_age
    )
age_stats = f"{age_df['age'].mean():.1f}±{age_df['age'].std():.1f}"
ax_age.annotate(
    f"Age: {age_stats}", xy=(0.5, 0.02), xycoords="axes fraction",
    ha="center", fontsize=10,
    bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alpha=0.8)
)
if len(age_df) >= 20:
    clustering = DBSCAN(eps=0.5, min_samples=5).fit(age_df[["umap_x", "umap_y"]]
    age_df["cluster"] = clustering.labels_
    if max(clustering.labels_) >= 0:
        cluster_ages = age_df.groupby("cluster")["age"].mean().to_dict()
        for cluster_id, cluster_age in cluster_ages.items():
            if cluster_id >= 0:
                cluster_points = age_df[age_df["cluster"] == cluster_id]
                center_x = cluster_points["umap_x"].mean()
                center_y = cluster_points["umap_y"].mean()
```

```
                            ax_age.annotate(
                                f"{cluster_age:.0f} yrs", xy=(center_x, cente
                                fontsize=9, fontweight="bold",
                                ha="center", va="center",
                                bbox=dict(boxstyle="circle,pad=0.3", fc="whit
                            )
        else:
            ax_age.text(0.5, 0.5, "No age data available",
                        ha='center', va='center', transform=ax_age.transAxes)
        ax_sex.set_xlabel("UMAP Dim 1")
        ax_sex.set_ylabel("UMAP Dim 2")
        ax_age.set_xlabel("UMAP Dim 1")
        ax_age.set_ylabel("UMAP Dim 2")
        ax_sex.set_aspect('auto')
        ax_age.set_aspect('auto')
    plt.tight_layout(rect=[0, 0, 1, 0.95])
    return fig

print("Generating demographic plots for baseline model...")
demographic_fig_baseline = create_demographic_subcluster_plots(
    baseline_umap, "Baseline Model - Demographic Subclustering Analysis"
)
plt.show()
plt.close()

print("Generating demographic plots for fine-tuned model...")
demographic_fig_finetuned = create_demographic_subcluster_plots(
    finetuned_umap, "Fine-tuned Model - Demographic Subclustering Analysis"
)
plt.show()
plt.close()
```

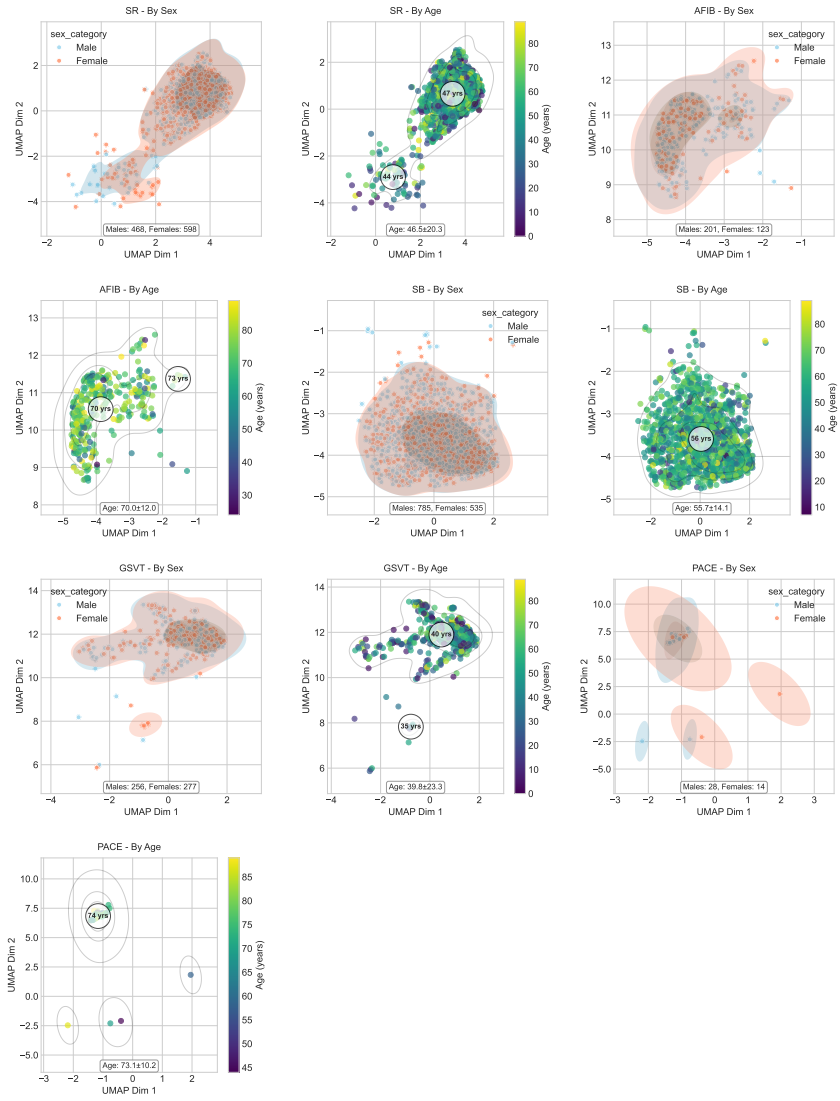
```
Generating demographic plots for baseline model...
```

**Baseline Model - Demographic Subclustering Analysis - Demographic Analysis**



```
Generating demographic plots for fine-tuned model...
```

**Fine-tuned Model - Demographic Subclustering Analysis - Demographic Analysis**



## 13.11 Statistical Analysis of Subclusters

```
print("\nStatistical Analysis of Subclusters:")
print("=" * 50)
single_counts = {}
dual_counts = {}
total_dual = 0

for primary_label in LABELS_OF_INTEREST:
    print(f"\nPrimary Label: {primary_label}")
    single_count = sum(df_single["integration_name"] == primary_label)
    single_counts[primary_label] = single_count
    print(f" - Single-labeled records: {single_count}")
    dual_mask = df_dual_filtered["primary_label"] == primary_label
    if not dual_mask.any():
        print(f" - No dual-labeled records with {primary_label}")
        dual_counts[primary_label] = 0
        continue
    dual_rows = df_dual_filtered[dual_mask]
    dual_count = len(dual_rows)
    dual_counts[primary_label] = dual_count
    total_dual += dual_count
    print(f" - Dual-labeled records: {dual_count}")
    secondary_counts = dual_rows["secondary_label"].value_counts()
    print(" - Secondary label distribution:")
    for label, count in secondary_counts.items():
        print(f"   * {label}: {count} records ({count/dual_count:.1%})")

print("\n\nSUMMARY OF FINDINGS")
print("=" * 50)
print(f"Total single-labeled records analyzed: {sum(single_counts.values())}"
print(f"Total dual-labeled records analyzed: {total_dual}")
```

```
for primary_label in LABELS_OF_INTEREST:
    single_count = single_counts[primary_label]
    dual_count = dual_counts[primary_label]
    total = single_count + dual_count
    if total > 0:
        print(f"\n{primary_label}:")
        print(f" - Single-labeled: {single_count} ({single_count/total:.1%})")
        print(f" - With secondary label: {dual_count} ({dual_count/total:.1%})")
```

```
Statistical Analysis of Subclusters:
==================================================

Primary Label: SR
 - Single-labeled records: 1134
 - Dual-labeled records: 232
 - Secondary label distribution:
   * STTA: 111 records (47.8%)
   * LVH: 33 records (14.2%)
   * SR: 15 records (6.5%)
   * LQV: 13 records (5.6%)
   * LAD: 12 records (5.2%)
   * APB: 10 records (4.3%)
   * QWA: 9 records (3.9%)
   * CRBBB: 8 records (3.4%)
   * 1AVB: 5 records (2.2%)
   * RAD: 4 records (1.7%)
   * WPW: 4 records (1.7%)
   * CLBBB: 4 records (1.7%)
   * PACE: 2 records (0.9%)
   * VPB: 2 records (0.9%)

Primary Label: AFIB
```

```
 - Single-labeled records: 341
 - Dual-labeled records: 556
 - Secondary label distribution:
   * STTA: 299 records (53.8%)
   * LVH: 91 records (16.4%)
   * PACE: 47 records (8.5%)
   * LAD: 31 records (5.6%)
   * LQV: 20 records (3.6%)
   * CRBBB: 17 records (3.1%)
   * RAD: 16 records (2.9%)
   * VPB: 13 records (2.3%)
   * CLBBB: 8 records (1.4%)
   * LAFB: 7 records (1.3%)
   * QWA: 6 records (1.1%)
   * RVH: 1 records (0.2%)

Primary Label: SB
 - Single-labeled records: 1338
 - Dual-labeled records: 707
 - Secondary label distribution:
   * STTA: 224 records (31.7%)
   * LVH: 221 records (31.3%)
   * SR: 65 records (9.2%)
   * 1AVB: 55 records (7.8%)
   * LAD: 34 records (4.8%)
   * LQV: 27 records (3.8%)
   * APB: 19 records (2.7%)
   * CRBBB: 16 records (2.3%)
   * VPB: 12 records (1.7%)
   * QWA: 11 records (1.6%)
   * RAD: 9 records (1.3%)
   * LAFB: 6 records (0.8%)
   * WPW: 3 records (0.4%)
   * CLBBB: 3 records (0.4%)
```

```
   * PACE: 2 records (0.3%)

Primary Label: GSVT
 - Single-labeled records: 626
 - Dual-labeled records: 361
 - Secondary label distribution:
   * STTA: 201 records (55.7%)
   * RAD: 33 records (9.1%)
   * LVH: 24 records (6.6%)
   * LAD: 20 records (5.5%)
   * LQV: 18 records (5.0%)
   * APB: 14 records (3.9%)
   * VPB: 13 records (3.6%)
   * CRBBB: 12 records (3.3%)
   * 1AVB: 7 records (1.9%)
   * CLBBB: 6 records (1.7%)
   * QWA: 5 records (1.4%)
   * AFIB: 3 records (0.8%)
   * PACE: 1 records (0.3%)
   * SR: 1 records (0.3%)
   * RVH: 1 records (0.3%)
   * LAFB: 1 records (0.3%)
   * GSVT: 1 records (0.3%)

Primary Label: PACE
 - Single-labeled records: 45
 - Dual-labeled records: 4
 - Secondary label distribution:
   * LVH: 2 records (50.0%)
   * VPB: 1 records (25.0%)
   * STTA: 1 records (25.0%)


SUMMARY OF FINDINGS
```

```
================================================
Total single-labeled records analyzed: 3484
Total dual-labeled records analyzed: 1860

SR:
 - Single-labeled: 1134 (83.0%)
 - With secondary label: 232 (17.0%)

AFIB:
 - Single-labeled: 341 (38.0%)
 - With secondary label: 556 (62.0%)

SB:
 - Single-labeled: 1338 (65.4%)
 - With secondary label: 707 (34.6%)

GSVT:
 - Single-labeled: 626 (63.4%)
 - With secondary label: 361 (36.6%)

PACE:
 - Single-labeled: 45 (91.8%)
 - With secondary label: 4 (8.2%)
```

# 14 Phase 3 - Cross-Domain Notebook

In this notebook, we integrate **Arrhythmia (Chapman)** and **PTB-XL** ECG embeddings to analyze cross-domain performance of both the **Baseline** and **Fine-tuned** models.

## 14.1 Imports and Configuration

```
import sys
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.lines import Line2D
from matplotlib.patches import Patch

project_root = Path().absolute().parent.parent
sys.path.append(str(project_root))

from src.visualization.embedding_viz import run_umap
from src.data.unified import UnifiedDataset
from src.data.dataset import DatasetModality
```

```
plt.rcParams["figure.figsize"] = (16, 8)
plt.rcParams["font.size"] = 12
```

## 14.2 Data Loading: Arrhythmia & PTB-XL

```
arr_data = UnifiedDataset(
    project_root / "data", modality=DatasetModality.ECG, dataset_key="arrhytl
)
ptbxl_data = UnifiedDataset(
    project_root / "data", modality=DatasetModality.ECG, dataset_key="ptbxl"
)

arr_splits = arr_data.get_splits()
ptbxl_splits = ptbxl_data.get_splits()

arr_test_ids = arr_splits.get("test", [])
ptbxl_test_ids = ptbxl_splits.get("test", [])

arr_md_store = arr_data.metadata_store
ptbxl_md_store = ptbxl_data.metadata_store

pretrained_embedding = "baseline"
finetuned_embedding = "fine_tuned_50"
```

## 14.3 Mapping and Label Definitions

```
PTBXL_ARR_MAP = {
    "AFIB": "AFIB",
    "AFLT": "AFIB",
    "SR":   "SR",
    "SARRH": "SR",
```

```
    "SBRAD": "SB",
    "PACE":  "PACE",
    "STACH": "GSVT",
    "SVARR": "GSVT",
    "SVTAC": "GSVT",
    "PSVT":  "GSVT",
}

LABELS_OF_INTEREST = ["AFIB", "GSVT", "SB", "SR", "PACE"]

LABEL_COLOR_MAP = {
    "SR":   sns.color_palette("tab10")[0],  # Blue
    "AFIB": sns.color_palette("tab10")[1],  # Orange
    "SB":   sns.color_palette("tab10")[2],  # Green
    "GSVT": sns.color_palette("tab10")[3],  # Red
    "PACE": sns.color_palette("tab10")[4],  # Purple
}

label2color = LABEL_COLOR_MAP
```

## 14.4 Arrhythmia (Phase 1) Embeddings & Metadata

```
arr_records_info = []
emb_base_list_arr = []
emb_ft_list_arr = []

for rid in arr_test_ids:
    meta = arr_md_store.get(rid, {})
    labels_meta = meta.get("labels_metadata", [])
    try:
        emb_base = arr_data.get_embeddings(rid, embeddings_type=pretrained_embedding)
        emb_ft = arr_data.get_embeddings(rid, embeddings_type=finetuned_embedding)
```

```
    except Exception:
        continue
    arr_records_info.append({"record_id": rid, "labels_meta": labels_meta})
    emb_base_list_arr.append(emb_base)
    emb_ft_list_arr.append(emb_ft)

df_arr = pd.DataFrame(arr_records_info)
df_arr["row_idx"] = df_arr.index

baseline_arr = np.vstack(emb_base_list_arr)
finetuned_arr = np.vstack(emb_ft_list_arr)
```

## 14.5 Treat “PACE” as Rhythm

```
def find_rhythm_labels(labels_meta):
    """
    Return a list of integration_names that are group="Rhythm" or "PACE".
    """
    rhythm_names = []
    for lm in labels_meta:
        integration_name = lm.get("integration_name", "")
        group = lm.get("group", "")
        if group == "Rhythm" or integration_name == "PACE":
            rhythm_names.append(integration_name)
    return rhythm_names

df_arr["rhythm_labels"] = df_arr["labels_meta"].apply(find_rhythm_labels)
df_arr["n_rhythm_labels"] = df_arr["rhythm_labels"].apply(len)

def get_single_label_of_interest(row):
    if row["n_rhythm_labels"] == 1:
        lbl = row["rhythm_labels"][0]
```

```
        if lbl in LABELS_OF_INTEREST:
            return lbl
    return None

df_arr["single_rhythm_label"] = df_arr.apply(get_single_label_of_interest, axis=1)
```

## 14.6 PTB-XL Embeddings & Metadata

```
ptbxl_records_info = []
emb_base_list_ptb = []
emb_ft_list_ptb = []

for rid in ptbxl_test_ids:
    meta = ptbxl_md_store.get(rid, {})
    scp_codes = meta.get("scp_codes", {})
    scp_statements = meta.get("scp_statements", {})

    valid_rhythm_codes = []
    for code_key in scp_codes:
        if code_key in PTBXL_ARR_MAP:
            code_info = scp_statements.get(code_key, {})
            if code_info.get("rhythm", 0.0) == 1.0:
                valid_rhythm_codes.append(code_key)

    if len(valid_rhythm_codes) == 1:
        code_key = valid_rhythm_codes[0]
        mapped_label = PTBXL_ARR_MAP[code_key]
        try:
            emb_base = ptbxl_data.get_embeddings(rid, embeddings_type=pretrained_embedding)
            emb_ft = ptbxl_data.get_embeddings(rid, embeddings_type=finetuned_embedding)
        except Exception:
            continue
```

```
        ptbxl_records_info.append(
            {"record_id": rid, "ptbxl_code": code_key, "mapped_label": mapped
        )
        emb_base_list_ptb.append(emb_base)
        emb_ft_list_ptb.append(emb_ft)

df_ptbxl = pd.DataFrame(ptbxl_records_info)
df_ptbxl["row_idx"] = df_ptbxl.index

baseline_ptb = np.vstack(emb_base_list_ptb)
finetuned_ptb = np.vstack(emb_ft_list_ptb)
```

## 14.7 UMAP on Combined Arrhythmia + PTB-XL

```
umap_params = dict(n_neighbors=15, n_components=2, metric="euclidean", random

def combine_and_umap(arr_emb, ptb_emb):
    combined = np.vstack([arr_emb, ptb_emb])
    combined_umap = run_umap(combined, **umap_params)
    coords_arr = combined_umap[: len(arr_emb)]
    coords_ptb = combined_umap[len(arr_emb) :]
    return coords_arr, coords_ptb

arr_baseline_umap, ptb_baseline_umap = combine_and_umap(baseline_arr, baselin
arr_finetuned_umap, ptb_finetuned_umap = combine_and_umap(finetuned_arr, fine

df_arr_baseline = df_arr.copy()
df_arr_baseline["umap_x"] = arr_baseline_umap[:, 0]
df_arr_baseline["umap_y"] = arr_baseline_umap[:, 1]

df_arr_finetuned = df_arr.copy()
```

```
df_arr_finetuned["umap_x"] = arr_finetuned_umap[:, 0]
df_arr_finetuned["umap_y"] = arr_finetuned_umap[:, 1]

df_ptbxl_baseline = df_ptbxl.copy()
df_ptbxl_baseline["umap_x"] = ptb_baseline_umap[:, 0]
df_ptbxl_baseline["umap_y"] = ptb_baseline_umap[:, 1]

df_ptbxl_finetuned = df_ptbxl.copy()
df_ptbxl_finetuned["umap_x"] = ptb_finetuned_umap[:, 0]
df_ptbxl_finetuned["umap_y"] = ptb_finetuned_umap[:, 1]
```

## 14.8 Outlier Removal Helper

```
def remove_outliers_2d(points, z_thresh=3.0):
    if len(points) == 0:
        return points
    mean = np.mean(points, axis=0)
    std = np.std(points, axis=0)
    std[std < 1e-12] = 1e-12
    z_scores = (points - mean) / std
    dist = np.sqrt((z_scores**2).sum(axis=1))
    return points[dist <= z_thresh]
```

## 14.9 Joint Plot with Marginal KDEs

```
def create_joint_plot(df_arr, df_ptb, title, figsize=(12, 8)):
    arr_data = df_arr[["umap_x", "umap_y"]].copy()
    arr_data["dataset"] = "Chapman"

    ptb_data = df_ptb[["umap_x", "umap_y"]].copy()
    ptb_data["dataset"] = "PTB-XL"
```

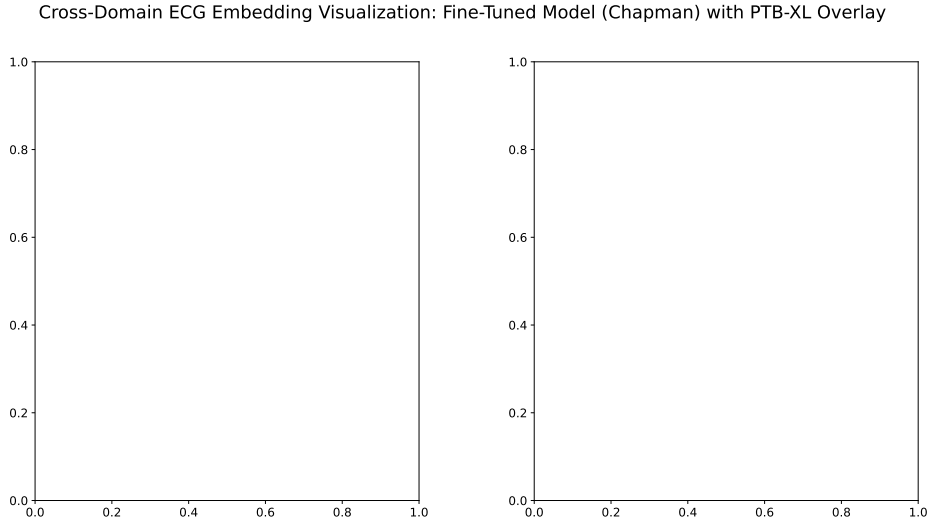
```
combined_data = pd.concat([arr_data, ptb_data])

g = sns.JointGrid(
    data=combined_data, x="umap_x", y="umap_y", height=figsize[0], ratio=
)

g.ax_joint.scatter(
    df_arr["umap_x"],
    df_arr["umap_y"],
    color="lightgray",
    s=20,
    alpha=0.6,
    label="Chapman (all)",
)

for lbl in LABELS_OF_INTEREST:
    sub = df_arr.loc[df_arr["single_rhythm_label"] == lbl, ["umap_x", "um
    print(len(sub))
    if len(sub) < 5:
        continue
    sub = remove_outliers_2d(sub, z_thresh=3.0 if lbl != "PACE" else 1.0)
    if len(sub) < 5:
        continue
    sub_df = pd.DataFrame(sub, columns=["umap_x", "umap_y"])
    sns.kdeplot(
        data=sub_df,
        x="umap_x",
        y="umap_y",
        fill=True,
        levels=4,
        alpha=0.25,
        color=label2color.get(lbl, "red"),
        ax=g.ax_joint,
```

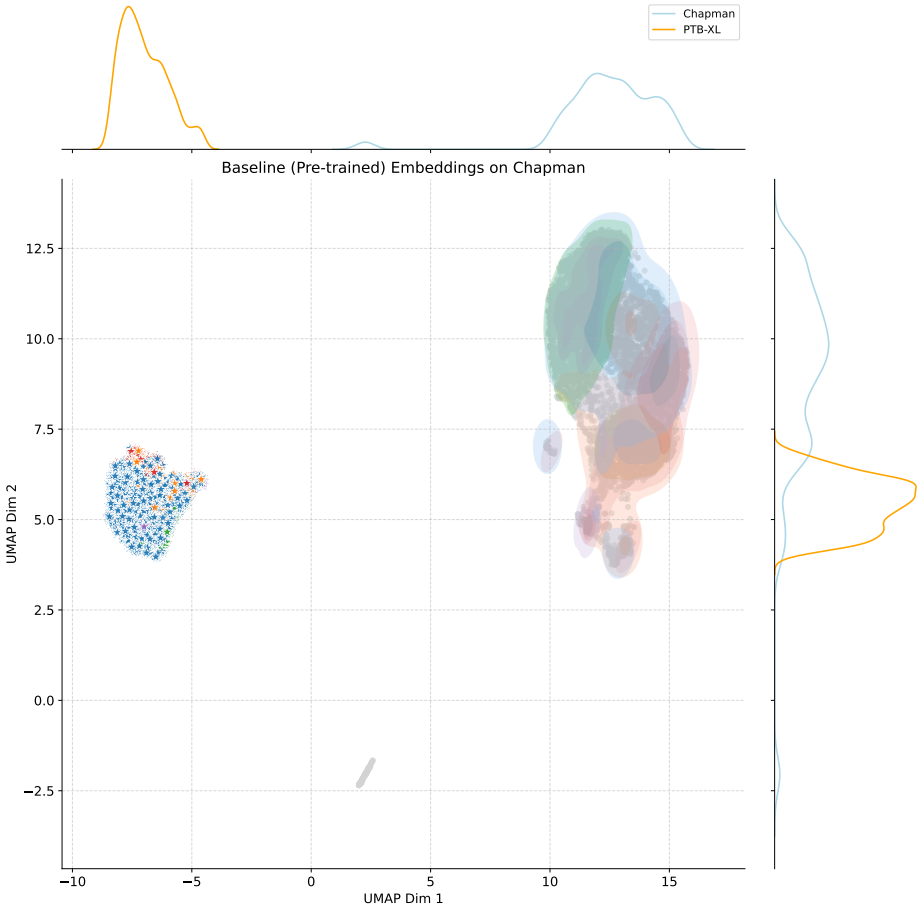
```
        label=None,
    )

g.ax_joint.scatter(
    df_ptb["umap_x"],
    df_ptb["umap_y"],
    c=[label2color.get(m, "black") for m in df_ptb["mapped_label"]],
    marker="*",
    s=120,
    edgecolors="white",
    linewidth=0.6,
    alpha=0.9,
    label="PTB-XL (single-rhythm, overlay)",
)

sns.kdeplot(data=arr_data, x="umap_x", color="lightblue", ax=g.ax_marg_x, label="Chapmar
sns.kdeplot(data=ptb_data, x="umap_x", color="orange", ax=g.ax_marg_x, label="PTB-XL")
sns.kdeplot(data=arr_data, y="umap_y", color="lightblue", ax=g.ax_marg_y)
sns.kdeplot(data=ptb_data, y="umap_y", color="orange", ax=g.ax_marg_y)

g.ax_marg_x.legend(fontsize=10)

g.ax_joint.set_xlabel("UMAP Dim 1")
g.ax_joint.set_ylabel("UMAP Dim 2")
g.ax_joint.set_title(title, fontsize=14)
g.ax_joint.grid(True, linestyle="--", alpha=0.5)

return g
```

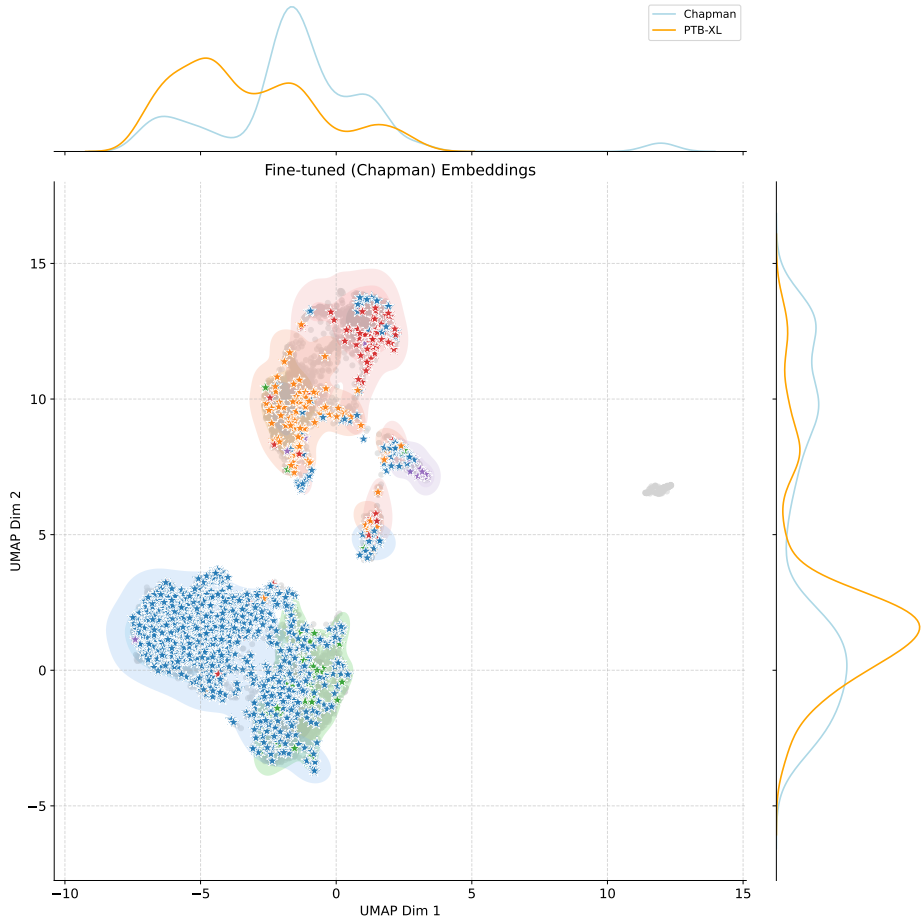
## 14.10 Create Two Joint Plots for Baseline vs. Fine-Tuned

```
fig, axes = plt.subplots(1, 2, figsize=(16, 8))
plt.subplots_adjust(wspace=0.3)
fig.suptitle(
    "Cross-Domain ECG Embedding Visualization: Fine-Tuned Model (Chapman) wit
    fontsize=18,
)

g1 = create_joint_plot(
    df_arr_baseline, df_ptbxl_baseline, f"Baseline (Pre-trained) Embeddings c
)
g2 = create_joint_plot(
    df_arr_finetuned, df_ptbxl_finetuned, f"Fine-tuned (Chapman) Embeddings"
)

g1.savefig("baseline_joint_plot.png")
g2.savefig("finetuned_joint_plot.png")
```

```
1345
1243
2300
1459
61
1345
1243
2300
1459
61
```


Cross-Domain ECG Embedding Visualization: Fine-Tuned Model (Chapman) with PTB-XL Overlay
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Chapman
PTB-XL
Baseline (Pre-trained) Embeddings on Chapman
UMAP Dim 2
UMAP Dim 1
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−10
−5
0
5
10
15




## 14.11 Plot Density Contours in a 1x2 Grid

```
def plot_density_contours(ax, df_arr_plot, df_ptb_plot, panel_title, ignore_outliers=True):
    ax.set_title(panel_title, fontsize=14)
    ax.scatter(
```

```
        df_arr_plot["umap_x"],
        df_arr_plot["umap_y"],
        color="lightgray",
        s=20,
        alpha=0.6,
        label="Chapman (all)",
    )
    for lbl in LABELS_OF_INTEREST:
        sub = df_arr_plot.loc[df_arr_plot["single_rhythm_label"] == lbl, ["u
        print(len(sub))
        if len(sub) < 5:
            continue
        if ignore_outliers:
            sub = remove_outliers_2d(sub, z_thresh=3.0 if lbl != "PACE" else
        if len(sub) < 5:
            continue
        sub_df = pd.DataFrame(sub, columns=["umap_x", "umap_y"])
        sns.kdeplot(
            data=sub_df,
            x="umap_x",
            y="umap_y",
            fill=True,
            levels=4,
            alpha=0.25,
            color=label2color.get(lbl, "red"),
            ax=ax,
            label=None,
        )
    ax.scatter(
        df_ptb_plot["umap_x"],
        df_ptb_plot["umap_y"],
        c=[label2color.get(m, "black") for m in df_ptb_plot["mapped_label"]]
        marker="*",
        s=120,
```

```
        edgecolors="white",
        linewidth=0.6,
        alpha=0.9,
        label="PTB-XL (single-rhythm, overlay)",
    )
    ax.set_xlabel("UMAP Dim 1")
    ax.set_ylabel("UMAP Dim 2")
    ax.grid(True, linestyle="--", alpha=0.5)

plot_density_contours(
    axes[0],
    df_arr_baseline,
    df_ptbxl_baseline,
    f"Baseline (Pre-trained) Embeddings on Chapman",
)
plot_density_contours(
    axes[1],
    df_arr_finetuned,
    df_ptbxl_finetuned,
    f"Fine-Tuned (Chapman) Embeddings",
)

plt.tight_layout(rect=[0, 0.2, 1, 0.97])
```

```
1345
1243
2300
1459
61
1345
1243
2300
1459
61
```

```
<Figure size 4800x2400 with 0 Axes>
```

## 14.12 Legend Construction

```
general_handles = [
    Line2D([0], [0], marker="o", color="lightgray", markersize=8, linewidth=0
    Line2D([0], [0], marker="*", color="black", markersize=15, linewidth=0),
]
general_labels = ["Chapman (all)", "PTB-XL (single-rhythm, overlay)"]

contour_handles = [Patch(color="none")]
contour_labels = ["Chapman Single-Label (Rhythm) Contours:"]

for lbl in LABELS_OF_INTEREST:
    clr = label2color.get(lbl, "black")
    contour_handles.append(Patch(facecolor=clr, edgecolor="none", alpha=0.25)
    contour_labels.append(f"{lbl} (Chapman)")

ptbxl_handles = [Patch(color="none")]
ptbxl_labels = ["PTB-XL Single-Rhythm (Mapped Labels):"]

ptbxl_labels_present = sorted(df_ptbxl["mapped_label"].unique())
for lbl in ptbxl_labels_present:
    clr = label2color.get(lbl, "black")
    ptbxl_handles.append(Line2D([0], [0], marker="*", color=clr, markersize=1
    ptbxl_labels.append(f"{lbl}")

legend1 = fig.legend(
    general_handles,
    general_labels,
    loc="lower left",
    bbox_to_anchor=(0.2, -0.04),
```

```
    frameon=False,
    fontsize=9,
    handletextpad=0.5,
)

legend2 = fig.legend(
    contour_handles,
    contour_labels,
    loc="lower center",
    bbox_to_anchor=(0.5, -0.14),
    frameon=False,
    fontsize=9,
    handletextpad=0.5,
)

legend3 = fig.legend(
    ptbxl_handles,
    ptbxl_labels,
    loc="lower right",
    bbox_to_anchor=(0.8, -0.14),
    frameon=False,
    fontsize=9,
    handletextpad=0.5,
)

legend_frame = plt.Rectangle(
    (0.15, -0.14),
    0.7,
    0.18,
    transform=fig.transFigure,
    fill=False,
    edgecolor="gray",
    linewidth=1,
)
```

```
fig.patches.append(legend_frame)

plt.tight_layout(rect=[0, 0.15, 1, 0.97])
plt.savefig("cross_domain_embedding_visualization.png", dpi=150, bbox_inches=
plt.show()
```

```
<Figure size 4800x2400 with 0 Axes>
```

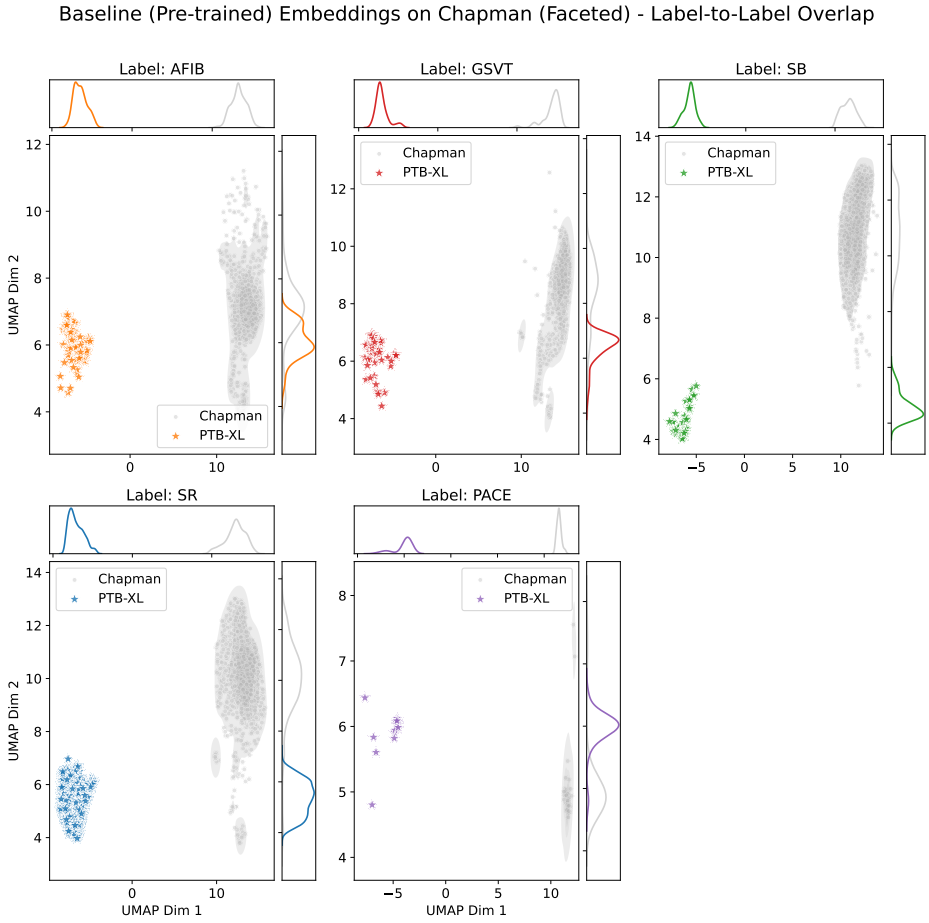
## 14.13 Faceted Joint Plots per Label

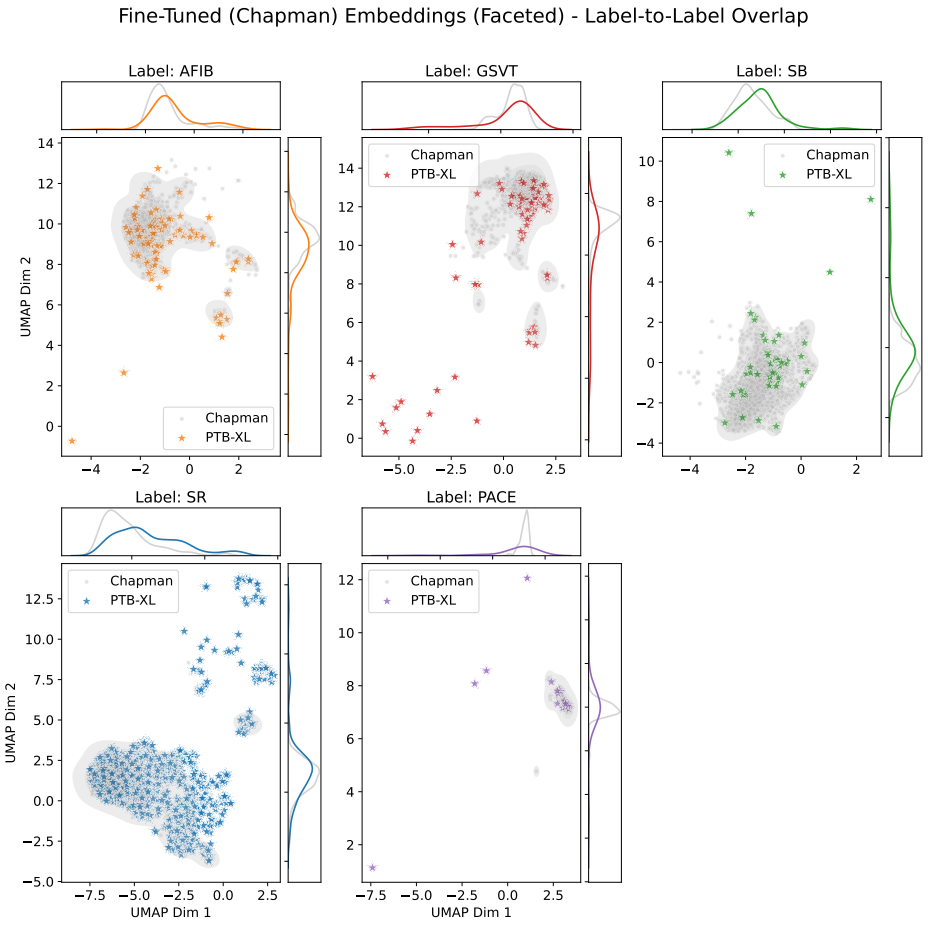
```
from mpl_toolkits.axes_grid1 import make_axes_locatable

def create_faceted_joint_plots(df_arr, df_ptb, title, figsize=(12, 12)):
    n_labels = len(LABELS_OF_INTEREST)
    n_cols = 3
    n_rows = (n_labels + n_cols - 1) // n_cols
    fig, axes = plt.subplots(n_rows, n_cols, figsize=figsize)
    fig.suptitle(f"{title} - Label-to-Label Overlap", fontsize=18)
    axes = axes.flatten() if n_rows > 1 else axes

    for i, lbl in enumerate(LABELS_OF_INTEREST):
        if i >= len(axes):
            break
        ax = axes[i]
        arr_lbl = df_arr[df_arr["single_rhythm_label"] == lbl].copy()
        ptb_lbl = df_ptb[df_ptb["mapped_label"] == lbl].copy()
        if len(arr_lbl) < 5 or len(ptb_lbl) < 2:
            ax.text(0.5, 0.5, f"Not enough data", ha="center", va="center", t
            continue

        arr_points = remove_outliers_2d(arr_lbl[["umap_x", "umap_y"]].values
                                        z_thresh=3.0 if lbl != "PACE" else 1
```

```
if len(arr_points) >= 5:
    arr_lbl_clean = pd.DataFrame(arr_points, columns=["umap_x", "umap_y"])
else:
    arr_lbl_clean = arr_lbl[["umap_x", "umap_y"]]

sns.scatterplot(
    data=arr_lbl_clean,
    x="umap_x",
    y="umap_y",
    color="lightgray",
    s=20,
    alpha=0.6,
    ax=ax,
    label="Chapman",
)

sns.scatterplot(
    data=ptb_lbl,
    x="umap_x",
    y="umap_y",
    color=label2color.get(lbl, "black"),
    marker="*",
    s=120,
    edgecolors="white",
    linewidth=0.6,
    alpha=0.8,
    ax=ax,
    label="PTB-XL",
)

if len(arr_lbl_clean) >= 5:
    sns.kdeplot(
        data=arr_lbl_clean,
        x="umap_x",
```

```
            y="umap_y",
            fill=True,
            levels=4,
            alpha=0.25,
            color="lightgray",
            ax=ax,
            label=None,
        )

    ax.set_xlabel("UMAP Dim 1" if i >= len(axes) - n_cols else "")
    ax.set_ylabel("UMAP Dim 2" if i % n_cols == 0 else "")

    divider = make_axes_locatable(ax)
    ax_top = divider.append_axes("top", size="15%", pad=0.1)
    ax_top.tick_params(axis="both", which="both", labelbottom=False, labe
    if len(arr_lbl_clean) >= 5:
        sns.kdeplot(data=arr_lbl_clean, x="umap_x", color="lightgray", ax
    if len(ptb_lbl) >= 2:
        sns.kdeplot(data=ptb_lbl, x="umap_x", color=label2color.get(lbl,
    ax_top.set_title(f"Label: {lbl}", fontsize=14, pad=5)
    ax_top.set_ylabel("")
    ax_top.set_yticks([])

    ax_right = divider.append_axes("right", size="15%", pad=0.1)
    ax_right.tick_params(axis="both", which="both", labelbottom=False, la
    if len(arr_lbl_clean) >= 5:
        sns.kdeplot(data=arr_lbl_clean, y="umap_y", color="lightgray", ax
    if len(ptb_lbl) >= 2:
        sns.kdeplot(data=ptb_lbl, y="umap_y", color=label2color.get(lbl,
    ax_right.set_xlabel("")
    ax_right.set_xticks([])

    if i == 0:
        ax.legend(loc="lower right")
```

```
    for j in range(i + 1, len(axes)):
        axes[j].set_visible(False)

    plt.tight_layout(rect=[0, 0, 1, 0.97])
    return fig

create_faceted_joint_plots(
    df_arr_baseline,
    df_ptbxl_baseline,
    "Baseline (Pre-trained) Embeddings on Chapman (Faceted)",
    figsize=(12, 12),
)

create_faceted_joint_plots(
    df_arr_finetuned,
    df_ptbxl_finetuned,
    "Fine-Tuned (Chapman) Embeddings (Faceted)",
    figsize=(12, 12),
)

plt.savefig("baseline_faceted_joint_plots.png", dpi=150, bbox_inches="tight")
plt.savefig("finetuned_faceted_joint_plots.png", dpi=150, bbox_inches="tight")
```


Baseline (Pre-trained) Embeddings on Chapman (Faceted) - Label-to-Label Overlap
Label: AFIB
Label: GSVT
Label: SB
Label: SR
Label: PACE
Chapman
PTB-XL
UMAP Dim 1
UMAP Dim 2


Fine-Tuned (Chapman) Embeddings (Faceted) - Label-to-Label Overlap
Label: AFIB
Label: GSVT
Label: SB
Label: SR
Label: PACE
Chapman
PTB-XL
UMAP Dim 1
UMAP Dim 2