Rater Agreement Analysis

Table of contents

1 Code to calculate agreement between Monique and Reyan **1**

2 1 Annotation Framework: Multi-Label, Order-Invariant Agreement **2**

2.1 1.1 Context and purpose . . . 2
2.2 1.2 Visit-level agreement classes . . . 2
2.3 1.3 Set-similarity metrics . . . 2
2.4 1.4 Three-rater adjudication logic . . . 3

3 2 Agreement Beyond Chance (Chance-Corrected Metrics) **3**

3.1 2.1 Binary expansion per category . . . 3
3.2 2.2 Why Gwet's AC1 . . . 3
3.3 2.3 Interpretation . . . 4
3.4 Agreement with NLP Pipeline . . . 34
3.5 Draft text: validation metrics (edit as needed) . . . 46

1 Code to calculate agreement between Monique and Reyan

Rationale: Quantify inter-rater reliability for multi-label RVC assignments to establish label quality.

Refactor note: This notebook is now structured for deterministic, fail-fast execution while preserving the original agreement estimands.

2 1 Annotation Framework: Multi-Label, Order-Invariant Agreement

2.1 1.1 Context and purpose

Each emergency-department visit receives up to five Reason-for-Visit (RFV1–RFV5) labels drawn from the **17 top-level NHAMCS RVC groups** defined in the *MIMIC IV Chief Complaint → NHAMCS Reason-for-Visit Reviewer Manual v0.3*. These represent patient-stated reasons, not clinician diagnoses.

Our goal is to quantify **inter-rater agreement** among multiple human annotators who each output an **unordered subset** of these groups for every visit. For analysis we ignore RFV ordering—treating the set (S_{jr} = {categories assigned by rater r to visit j}) as a simple set of 0–5 elements.

2.2 1.2 Visit-level agreement classes

For any two raters A and B:

Class	Definition	Intuition
Full	(S_A = S_B)	Complete category match
Partial	(S_A S_B) and (S_A S_B)	Share at least one category
None	(S_A S_B =)	Disjoint label sets

For three raters, - **Full** = all identical; - **Partial** = any overlap between any pair; - **None** = pairwise disjoint.

2.3 1.3 Set-similarity metrics

Because order is meaningless, comparisons use **set-based overlaps**:

Metric	Formula	Notes
Jaccard	(	S_A S_B
Overlap Coefficient	(	S_A S_B
F (Set)	(2	S_A S_B
Micro-F (Set)	(2	S_A S_B

These capture graded similarity beyond binary full/none counts.

2.4 1.4 Three-rater adjudication logic

When a third reviewer adjudicates: - Compute the distribution of *full / partial / none* across all three. - For disagreements between raters 1 and 2, evaluate whether rater 3: - matches either rater, - matches their union or intersection, - introduces any new category, or - provides a subset of the union.

This quantifies how the final "reference standard" was reached.

3 2 Agreement Beyond Chance (Chance-Corrected Metrics)

3.1 2.1 Binary expansion per category

Convert the multi-label sets into a binary decision matrix over the 17 canonical RVC groups: for each visit × rater × category, record 1 if the category is present. This yields parallel binary classification tasks where standard chance-corrected statistics apply.

3.2 2.2 Why Gwet's AC1

Cohen's is sensitive to marginal prevalence—especially when most visits lack a given category (the *kappa paradox*). Gwet's AC1 provides a more stable estimate of "agreement beyond chance" in such sparse settings.

For each category g with m raters:

$$A_j = \frac{\binom{n_j^+}{2} + \binom{m-n_j^+}{2}}{\binom{m}{2}}, \quad P_o = \text{mean}(A_j), \quad p = \frac{1}{Nm} \sum_{j,r} X_{jr}^{(g)}$$

$$\text{AC1} = \frac{P_o - 2p(1-p)}{1 - 2p(1-p)}$$

Report **per-category AC1** and **macro-average** across categories. Also compute **pairwise** (for comparison) and **percent agreement** to provide a full view.

Metric	What it captures	Typical reporting

3.3 2.3 Interpretation

Metric	What it captures	Typical reporting
Percent agreement	Raw consistency, ignores chance	Always include
Cohen's	Chance-corrected for two raters	Supplemental
Gwet's AC1	Chance-corrected, robust to imbalance	Primary statistic
Multi-rater AC1	Extension of AC1 to m > 2	Summary across all raters

High percent agreement with high AC1 (0.9–1.0) implies strong, reproducible labeling of the RVC groups defined in the Reviewer Manual v0.3.

Rationale: Define set-based agreement metrics and chance-corrected statistics appropriate for multi-label labels.

```
#
 ↪  ------------------------------------------------------------------------
# Configuration cell
#
# Purpose:
# 1) Keep all file paths and run settings in one obvious place.
# 2) Make it easy for collaborators to re-run this notebook without editing
#    downstream analysis logic.
#
 ↪  ------------------------------------------------------------------------

from __future__ import annotations

import os
import sys
import warnings
from pathlib import Path

from dotenv import load_dotenv

# Load optional .env values (for example WORK_DIR) if present.
load_dotenv()
```

```
# Suppress benign openpyxl extension warning emitted by validated annotation
 ↪  sheets.
warnings.filterwarnings(
    "ignore",
    message="Data Validation extension is not supported and will be removed",
    category=UserWarning,
)

# WORK_DIR anchors all relative paths. If not provided, default to the
# current working directory so the notebook still runs in a local clone.
WORK_DIR = Path(os.getenv("WORK_DIR", Path.cwd())).expanduser().resolve()
SRC_DIR = WORK_DIR / "src"
if SRC_DIR.exists() and str(SRC_DIR) not in sys.path:
    sys.path.insert(0, str(SRC_DIR))

CANONICAL_NLP_FILENAME = "MIMICIV all with CC_with_NLP.xlsx"


def resolve_rater_nlp_input_path(work_dir: Path, input_filename: str | None =
 ↪  None) -> Path:
    filename = input_filename or CANONICAL_NLP_FILENAME
    input_path = (work_dir / "MIMIC tabular data" /
 ↪  filename).expanduser().resolve()
    if not input_path.exists():
        raise FileNotFoundError(
            "Expected rater NLP input workbook was not found at "
            f"{input_path}. Run the classifier notebook first or set "
            "RATER_NLP_INPUT_FILENAME."
        )
    return input_path

RATER_ANNOTATION_PATH = os.getenv("RATER_ANNOTATION_PATH")
RATER_NLP_INPUT_FILENAME = os.getenv("RATER_NLP_INPUT_FILENAME")

annotation_default_path = WORK_DIR / "Annotation/Final 2025-10-14 Annotation
 ↪  Sample.xlsx"
annotation_path = (
    Path(RATER_ANNOTATION_PATH).expanduser().resolve()
    if RATER_ANNOTATION_PATH
    else annotation_default_path
)
nlp_path = resolve_rater_nlp_input_path(
```

```
    WORK_DIR,
    RATER_NLP_INPUT_FILENAME if RATER_NLP_INPUT_FILENAME else None,
)

# Centralized run configuration. Edit these values when sharing the
# notebook across environments or workbook versions.
CONFIG = {
    "annotation_path": annotation_path,
    "annotation_sheet": "cohort_cc_sample",
    "category_sheet": "Data",
    "category_column": "RVC Categories",
    "nlp_path": nlp_path,
    "nlp_sheet": 0,
    "nlp_min_sim": None,
    "raters_output_dir": WORK_DIR / "Annotation/Full Annotations/Agreement
     ↪  Metrics",
    "nlp_output_dir": WORK_DIR / "annotation_agreement_outputs_nlp",
}

ANNOTATION_PATH = Path(CONFIG["annotation_path"])
ANNOTATION_SHEET = CONFIG["annotation_sheet"]
CATEGORY_SHEET = CONFIG["category_sheet"]
CATEGORY_COLUMN = CONFIG["category_column"]
NLP_PATH = Path(CONFIG["nlp_path"])
NLP_SHEET = CONFIG["nlp_sheet"]
NLP_MIN_SIM = CONFIG["nlp_min_sim"]
RATERS_OUTPUT_DIR = Path(CONFIG["raters_output_dir"])
NLP_OUTPUT_DIR = Path(CONFIG["nlp_output_dir"])

# Fail fast if required input workbooks are missing. This prevents subtle
# downstream errors that are harder for new users to debug.
for path in (ANNOTATION_PATH, NLP_PATH):
    if not path.exists():
        raise FileNotFoundError(f"Required input file not found: {path}")

# Ensure output folders exist before writing artifacts.
RATERS_OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
NLP_OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

# Echo active configuration for provenance in notebook logs.
print(f"WORK_DIR: {WORK_DIR}")
print(f"Annotation workbook: {ANNOTATION_PATH}")
```

```
print(f"NLP workbook (canonical default: {CANONICAL_NLP_FILENAME}):
 ↪  {NLP_PATH}")
print(f"Raters output dir: {RATERS_OUTPUT_DIR}")
print(f"NLP output dir: {NLP_OUTPUT_DIR}")
```

```
WORK_DIR: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Proje
CC-NLP
Annotation workbook: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Res
CC-NLP/Annotation/Final 2025-10-14 Annotation Sample.xlsx
NLP workbook (canonical default: MIMICIV all with CC_with_NLP.xlsx): /Users/blocke/Box Sync/R
CC-NLP/MIMIC tabular data/MIMICIV all with CC_with_NLP.xlsx
Raters output dir: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Resea
CC-NLP/Annotation/Full Annotations/Agreement Metrics
NLP output dir: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research
CC-NLP/annotation_agreement_outputs_nlp
```

```
#
 ↪  ------------------------------------------------------------------------------
# Shared helper functions used by both analyses below.
#
# These helpers intentionally keep I/O checks, label processing, and metric
# calculations separate so trainees can reason about each step in isolation.
#
 ↪  ------------------------------------------------------------------------------

from collections import Counter
from typing import Dict, List, Set, Tuple

import numpy as np
import pandas as pd

import hashlib

from hypercap_cc_nlp.pipeline_audit import collect_run_manifest

RVC_CODE_TO_LABEL: dict[str, str] = {
    "RVC-INJ": "Injuries & adverse effects",
    "RVC-SYM-RESP": "Symptom - Respiratory",
    "RVC-SYM-CIRC": "Symptom - Circulatory",
    "RVC-SYM-NERV": "Symptom - Nervous",
    "RVC-SYM-DIG": "Symptom - Digestive",
    "RVC-SYM-GU": "Symptom - Genitourinary",
```

```
    "RVC-SYM-MSK": "Symptom - Musculoskeletal",
    "RVC-SYM-SKIN": "Symptom - Skin/Hair/Nails",
    "RVC-SYM-EYE": "Symptom - Eye/Ear",
    "RVC-SYM-GEN": "Symptom - General",
    "RVC-SYM-PSY": "Symptom - Psychological",
    "RVC-DIS": "Diseases (patient-stated)",
    "RVC-TEST": "Abnormal test result",
    "RVC-DIAG": "Diagnostic/Screening/Preventive",
    "RVC-TREAT": "Treatment/Medication",
    "RVC-ADMIN": "Administrative",
    "RVC-UNCL": "Uncodable/Unknown",
}
R3_NLP_TARGET_SAMPLE_N = 160
R3_NLP_GATE_POLICY = "warn_below_target_fail_on_zero"


def normalize_join_keys(df: pd.DataFrame, key_cols: list[str]) ->
 ↪  pd.DataFrame:
    validate_required_columns(df, key_cols, "rater join input")
    normalized = df.copy()
    for key in key_cols:
        normalized[key] = pd.to_numeric(normalized[key],
 ↪  errors="coerce").astype("Int64")
    return normalized


def _validate_unique_keys(df: pd.DataFrame, key_cols: list[str], *, context:
 ↪  str) -> None:
    duplicates = int(df.duplicated(subset=key_cols).sum())
    if duplicates:
        raise ValueError(
            f"{context} has {duplicates} duplicate rows for keys {key_cols}.
             ↪  "
            "Expected unique keys before merge."
        )


def _anti_join_keys(left_df: pd.DataFrame, right_df: pd.DataFrame, key_cols:
 ↪  list[str]) -> pd.DataFrame:
    right_keys = right_df[key_cols].drop_duplicates()
    only_left = (
        left_df[key_cols]
        .drop_duplicates()
```

```
        .merge(right_keys, on=key_cols, how="left", indicator=True)
        .loc[lambda frame: frame["_merge"].eq("left_only"), key_cols]
        .sort_values(key_cols)
        .reset_index(drop=True)
    )
    return only_left


def select_join_key_columns(
    df_r3: pd.DataFrame,
    df_nlp: pd.DataFrame,
) -> tuple[list[str], str]:
    preferred_candidates: list[tuple[list[str], str]] = [
        (["ed_stay_id"], "ed_stay_id"),
        (["hadm_id", "subject_id"], "hadm_id_subject_id"),
        (["hadm_id"], "hadm_id"),
    ]
    for key_cols, strategy in preferred_candidates:
        if not set(key_cols).issubset(df_r3.columns) or not
         ↪  set(key_cols).issubset(df_nlp.columns):
            continue
        r3_nonnull = df_r3[key_cols].notna().all(axis=1)
        nlp_nonnull = df_nlp[key_cols].notna().all(axis=1)
        if int(r3_nonnull.sum()) == 0 or int(nlp_nonnull.sum()) == 0:
            continue
        return key_cols, strategy
    raise ValueError(
        "Unable to find a valid join key strategy. Expected shared non-null
         ↪  keys from "
        "['ed_stay_id'] or ['hadm_id','subject_id'] or ['hadm_id']."
    )


def build_join_key_inventory(
    df_r3: pd.DataFrame,
    df_nlp: pd.DataFrame,
    key_cols: list[str],
    *,
    strategy: str,
) -> pd.DataFrame:
    normalized_r3 = normalize_join_keys(df_r3, key_cols)
    normalized_nlp = normalize_join_keys(df_nlp, key_cols)
    r3_keys = normalized_r3[key_cols].drop_duplicates()
```

```
    nlp_keys = normalized_nlp[key_cols].drop_duplicates()
    overlap_keys = r3_keys.merge(nlp_keys, on=key_cols, how="inner")
    rows = [
        {
            "source": "adjudicated_r3",
            "join_key_strategy": strategy,
            "join_key_columns": "|".join(key_cols),
            "rows": int(len(normalized_r3)),
            "unique_keys": int(len(r3_keys)),
        },
        {
            "source": "nlp_output",
            "join_key_strategy": strategy,
            "join_key_columns": "|".join(key_cols),
            "rows": int(len(normalized_nlp)),
            "unique_keys": int(len(nlp_keys)),
        },
        {
            "source": "key_overlap",
            "join_key_strategy": strategy,
            "join_key_columns": "|".join(key_cols),
            "rows": int(len(overlap_keys)),
            "unique_keys": int(len(overlap_keys)),
        },
    ]
    return pd.DataFrame(rows)


def build_r3_nlp_join_audit(
    df_r3: pd.DataFrame,
    df_nlp: pd.DataFrame,
    key_cols: list[str],
    *,
    target_sample_n: int = R3_NLP_TARGET_SAMPLE_N,
    gate_policy: str = R3_NLP_GATE_POLICY,
    key_strategy: str = "hadm_id_subject_id",
) -> tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame, dict[str, object]]:
    normalized_r3 = normalize_join_keys(df_r3, key_cols)
    normalized_nlp = normalize_join_keys(df_nlp, key_cols)

    _validate_unique_keys(normalized_r3, key_cols, context="R3 source")
    _validate_unique_keys(normalized_nlp, key_cols, context="NLP source")
```

```
    matched = normalized_r3.merge(normalized_nlp, on=key_cols, how="inner")
    _validate_unique_keys(matched, key_cols, context="R3/NLP joined output")

    unmatched_adjudicated = _anti_join_keys(normalized_r3, normalized_nlp,
↪   key_cols)
    unmatched_nlp = _anti_join_keys(normalized_nlp, normalized_r3, key_cols)

    r3_rows = int(len(normalized_r3))
    nlp_rows = int(len(normalized_nlp))
    matched_rows = int(len(matched))
    unmatched_adjudicated_rows = int(len(unmatched_adjudicated))
    unmatched_nlp_rows = int(len(unmatched_nlp))

    audit: dict[str, object] = {
        "key_columns": key_cols,
        "key_strategy": key_strategy,
        "target_sample_n": int(target_sample_n),
        "gate_policy": gate_policy,
        "r3_rows": r3_rows,
        "nlp_rows": nlp_rows,
        "matched_rows": matched_rows,
        "unmatched_adjudicated_rows": unmatched_adjudicated_rows,
        "unmatched_nlp_rows": unmatched_nlp_rows,
        "matched_rate_vs_adjudicated": float(matched_rows / r3_rows) if
         ↪  r3_rows else None,
        "unmatched_rate_vs_adjudicated": (
            float(unmatched_adjudicated_rows / r3_rows) if r3_rows else None
        ),
        "unmatched_rate_vs_nlp": float(unmatched_nlp_rows / nlp_rows) if
         ↪  nlp_rows else None,
    }

    if matched_rows == 0:
        raise ValueError(
            "R3/NLP join produced zero matched rows after key normalization.
             ↪  "
            "Check key consistency between annotation and NLP workbooks."
        )

    if matched_rows < int(target_sample_n):
        audit["coverage_status"] = "warning"
        audit["join_interpretation"] = "below_target_overlap_nonzero"
        audit["severity"] = "warning"
```

```
    else:
        audit["coverage_status"] = "ok"
        audit["join_interpretation"] = "target_overlap_met"
        audit["severity"] = "info"

    return matched, unmatched_adjudicated, unmatched_nlp, audit


def hash_join_keys(
    df: pd.DataFrame,
    *,
    key_cols: list[str],
    hash_col: str = "key_hash",
) -> pd.DataFrame:
    normalized = normalize_join_keys(df, key_cols)
    _validate_unique_keys(normalized, key_cols, context="join-key hash
 ↪  input")
    key_frame = normalized[key_cols].drop_duplicates().copy()

    def _hash_row(row: pd.Series) -> str:
        material = "|".join(
            "" if pd.isna(row[column]) else str(int(row[column])) for column
             ↪  in key_cols
        )
        return hashlib.sha256(material.encode("utf-8")).hexdigest()

    key_frame[hash_col] = key_frame.apply(_hash_row, axis=1)
    return key_frame[[hash_col]].sort_values(hash_col).reset_index(drop=True)


def render_latex_longtable(
    table_df: pd.DataFrame,
    *,
    caption: str,
    label: str,
    landscape: bool = False,
    index: bool = True,
) -> str:
    latex_text = table_df.to_latex(
        index=index,
        escape=False,
        longtable=True,
        caption=caption,
```

```
        label=label,
    )
    if landscape:
        latex_text = "\\begin{landscape}\\n" + latex_text +
↪   "\\n\\end{landscape}\\n"
    return latex_text


# --- Input/sheet validation helpers
↪   ----------------------------------------

def read_sheet_checked(xlsx_path: Path, sheet_name: str | int) ->
↪   pd.DataFrame:
    """Read an Excel sheet with explicit validation for sheet existence."""
    if not xlsx_path.exists():
        raise FileNotFoundError(f"Excel file not found: {xlsx_path}")

    xls = pd.ExcelFile(xlsx_path)
    if isinstance(sheet_name, str):
        if sheet_name not in xls.sheet_names:
            raise ValueError(
                f"Sheet '{sheet_name}' not found in {xlsx_path}. Available:
                ↪  {xls.sheet_names}"
            )
    elif isinstance(sheet_name, int):
        if sheet_name < 0 or sheet_name >= len(xls.sheet_names):
            raise ValueError(
                f"Sheet index {sheet_name} is out of range for {xlsx_path}. "
                f"Valid range: 0..{len(xls.sheet_names) - 1}"
            )
    else:
        raise TypeError(f"Unsupported sheet type: {type(sheet_name)}")

    return pd.read_excel(xls, sheet_name=sheet_name)


def validate_required_columns(df: pd.DataFrame, required: List[str], df_name:
↪   str) -> None:
    missing = sorted(set(required).difference(df.columns))
    if missing:
        raise KeyError(f"{df_name} is missing required columns: {missing}")
```

```
def validate_unique_keys(df: pd.DataFrame, keys: List[str], df_name: str) ->
 ↪  None:
    validate_required_columns(df, keys, df_name)
    duplicates = int(df.duplicated(subset=keys).sum())
    if duplicates:
        raise ValueError(
            f"{df_name} has {duplicates} duplicate rows for key columns
             ↪  {keys}. "
            "Expected unique keys before merge."
        )


# --- Label normalization and extraction helpers
 ↪  ------------------------------

def normalize_label(value: object) -> str | None:
    """Normalize punctuation/spacing so category strings match reliably."""
    if pd.isna(value):
        return None
    text = str(value).strip()
    if not text:
        return None
    # Normalize dash variants so semantically identical labels map together
    # (e.g., patient-stated vs patient-stated).
    text = text.replace(" - ", " - ").replace("-", "-").replace("-", "-")
    text = " ".join(text.split())
    return text


RVC_LABEL_TO_CODE: dict[str, str] = {
    normalize_label(label): code
    for code, label in RVC_CODE_TO_LABEL.items()
    if normalize_label(label) is not None
}
RVC_LABEL_TO_CODE.update(
    {
        "Diseases (patient-stated diagnosis)": "RVC-DIS",
        "Diseases (patient-stated diagnosis)": "RVC-DIS",
        "Diseases (patient-stated)": "RVC-DIS",
        "Diseases (patient stated)": "RVC-DIS",
        "Uncodable / Unknown": "RVC-UNCL",
        "Uncodable/Unknown": "RVC-UNCL",
    }
```

```
)


def canonicalize_rvc_code(value: object) -> str | None:
    normalized = normalize_label(value)
    if normalized is None:
        return None
    upper = normalized.upper()
    if upper in RVC_CODE_TO_LABEL:
        return upper
    return RVC_LABEL_TO_CODE.get(normalized)


def canonical_label_for_code(code: object) -> str:
    normalized_code = str(code).strip().upper()
    return RVC_CODE_TO_LABEL.get(normalized_code, normalized_code)


def load_categories_from_sheet(
    xlsx_path: Path,
    sheet: str = "Data",
    column: str = "RVC Categories",
) -> List[str]:
    cat_df = read_sheet_checked(xlsx_path, sheet)
    validate_required_columns(cat_df, [column], f"{xlsx_path.name}:{sheet}")

    categories: list[str] = []
    unmapped_categories: list[str] = []
    for raw_value in cat_df[column].dropna().astype(str).tolist():
        code = canonicalize_rvc_code(raw_value)
        if code is None:
            unmapped_categories.append(str(raw_value))
            continue
        categories.append(code)

    if not categories:
        raise ValueError(
            f"No categories found in column '{column}' of sheet '{sheet}'
             ↪  ({xlsx_path})."
        )
    if unmapped_categories:
        sample = sorted(set(unmapped_categories))[:10]
        raise ValueError(
```

```
            "Unmapped canonical categories found in category sheet: "
            f"{sample}. Update RVC label-to-code mapping."
        )

    if len(set(categories)) != len(categories):
        duplicates = [c for c, n in Counter(categories).items() if n > 1]
        raise ValueError(f"Duplicate canonical categories found:
         ↪ {duplicates}")

    return categories


def extract_rater_sets(df: pd.DataFrame, rater: int, max_slots: int = 5) ->
 ↪ List[Set[str]]:
    """Return one set of assigned categories per row for a numbered rater."""
    return extract_rater_sets_from_df(df, rater_col_prefix=f"annot{rater}",
     ↪ max_slots=max_slots)


def extract_rater_sets_from_df(
    df: pd.DataFrame,
    rater_col_prefix: str,
    max_slots: int = 5,
) -> List[Set[str]]:
    """Return one set of categories per row for columns like
     ↪ <prefix>_rvs1_cat..rvsN_cat."""
    cols = [
        f"{rater_col_prefix}_rvs{i}_cat"
        for i in range(1, max_slots + 1)
        if f"{rater_col_prefix}_rvs{i}_cat" in df.columns
    ]
    if not cols:
        raise KeyError(
            f"No columns found for prefix '{rater_col_prefix}'. "
            f"Expected columns like '{rater_col_prefix}_rvs1_cat'."
        )

    sets: List[Set[str]] = []
    for _, row in df[cols].iterrows():
        labels: Set[str] = set()
        for col in cols:
            code = canonicalize_rvc_code(row[col])
            if code:
```

```
                labels.add(code)
        sets.append(labels)
    return sets


def extract_nlp_sets_from_df(
    df_nlp: pd.DataFrame,
    max_slots: int = 5,
    min_sim: float | None = None,
) -> List[Set[str]]:
    """Build predicted category sets from RFVn_name columns with optional
     ↪ RFVn_sim threshold."""
    name_cols = [f"RFV{i}_name" for i in range(1, max_slots + 1) if
 ↪ f"RFV{i}_name" in df_nlp.columns]
    sim_cols = [f"RFV{i}_sim" for i in range(1, max_slots + 1) if
 ↪ f"RFV{i}_sim" in df_nlp.columns]
    # Similarity thresholding is optional. If min_sim is None, keep all
    # predicted labels. If set, require RFVn_sim >= min_sim.
    use_sim = min_sim is not None and len(sim_cols) == len(name_cols)

    sets: List[Set[str]] = []
    for _, row in df_nlp.iterrows():
        labels: Set[str] = set()
        for i in range(1, max_slots + 1):
            name_col = f"RFV{i}_name"
            if name_col not in df_nlp.columns:
                continue
            code = canonicalize_rvc_code(row[name_col])
            if not code:
                continue

            if use_sim:
                sim_col = f"RFV{i}_sim"
                sim_val = row.get(sim_col, np.nan)
                if pd.isna(sim_val) or float(sim_val) < float(min_sim):
                    continue
            labels.add(code)
        sets.append(labels)
    return sets


def build_label_mapping_audit(
    df_r3: pd.DataFrame,
```

```
    df_nlp: pd.DataFrame,
    *,
    r3_label_cols: list[str],
    nlp_label_cols: list[str],
) -> pd.DataFrame:
    rows: list[dict[str, object]] = []
    for source_name, frame, cols in (
        ("adjudicated_r3", df_r3, r3_label_cols),
        ("nlp_output", df_nlp, nlp_label_cols),
    ):
        for col in cols:
            if col not in frame.columns:
                continue
            series = frame[col].dropna().astype(str)
            if series.empty:
                continue
            for raw_label, row_count in
             ↪ series.value_counts(dropna=False).items():
                normalized_label = normalize_label(raw_label)
                canonical_code = canonicalize_rvc_code(raw_label)
                rows.append(
                    {
                        "source": source_name,
                        "column": col,
                        "raw_label": raw_label,
                        "normalized_label": normalized_label,
                        "canonical_code": canonical_code,
                        "canonical_label":
                         ↪  canonical_label_for_code(canonical_code)
                        if canonical_code
                        else None,
                        "mapped": bool(canonical_code),
                        "row_count": int(row_count),
                    }
                )
    if not rows:
        return pd.DataFrame(
            columns=[
                "source",
                "column",
                "raw_label",
                "normalized_label",
                "canonical_code",
```

```
                "canonical_label",
                "mapped",
                "row_count",
            ]
        )
    return (
        pd.DataFrame(rows)
        .sort_values(["source", "mapped", "row_count", "raw_label"],
↪   ascending=[True, True, False, True])
        .reset_index(drop=True)
    )


# --- Set-level agreement metrics (order-invariant multi-label)
↪   ----------------

def jaccard(a: Set[str], b: Set[str]) -> float:
    union_size = len(a | b)
    return 1.0 if union_size == 0 else len(a & b) / union_size


def overlap_coeff(a: Set[str], b: Set[str]) -> float:
    min_size = min(len(a), len(b))
    # Convention: when both sets are empty, treat overlap as perfect (1.0)
    # because both raters made the same decision: assign nothing.
    return (
        1.0
        if min_size == 0 and len(a) == 0 and len(b) == 0
        else (0.0 if min_size == 0 else len(a & b) / min_size)
    )


def f1_set(a: Set[str], b: Set[str]) -> float:
    denom = len(a) + len(b)
    return 1.0 if denom == 0 else 2 * len(a & b) / denom


def classify_three(a: Set[str], b: Set[str], c: Set[str]) -> str:
    if a == b == c:
        return "full"
    return "partial" if (a & b) or (a & c) or (b & c) else "none"
```

```python
def micro_f1_raters(a_sets: List[Set[str]], b_sets: List[Set[str]]) -> float:
    tp = sum(len(a & b) for a, b in zip(a_sets, b_sets))
    denom = sum(len(a) + len(b) for a, b in zip(a_sets, b_sets))
    return 1.0 if denom == 0 else 2 * tp / denom


def summarize_set_agreement(
    a_sets: List[Set[str]],
    b_sets: List[Set[str]],
) -> Tuple[pd.DataFrame, pd.Series]:
    recs = []
    for a, b in zip(a_sets, b_sets):
        # Visit-level class labels support intuitive interpretation for
        # non-statistical readers (exact match, partial overlap, no overlap).
        recs.append(
            {
                "exact": int(a == b),
                "partial": int((a != b) and (len(a & b) > 0)),
                "none": int(len(a & b) == 0),
                "jaccard": jaccard(a, b),
                "overlap": overlap_coeff(a, b),
                "f1_set": f1_set(a, b),
                "len_a": len(a),
                "len_b": len(b),
                "len_inter": len(a & b),
                "len_union": len(a | b),
            }
        )

    out = pd.DataFrame(recs)
    summary = pd.Series(
        {
            "N_items": len(out),
            "exact_rate": out["exact"].mean(),
            "partial_rate": out["partial"].mean(),
            "none_rate": out["none"].mean(),
            "mean_jaccard": out["jaccard"].mean(),
            "mean_overlap": out["overlap"].mean(),
            "mean_f1_set": out["f1_set"].mean(),
            "micro_f1_set": micro_f1_raters(a_sets, b_sets),
            "mean_len_a": out["len_a"].mean(),
            "mean_len_b": out["len_b"].mean(),
        }
```

```
    )
    return out, summary


def summarize_three_way(
    r1_sets: List[Set[str]],
    r2_sets: List[Set[str]],
    r3_sets: List[Set[str]],
) -> pd.Series:
    labels = [
        classify_three(a, b, c)
        for a, b, c in zip(r1_sets, r2_sets, r3_sets)
    ]
    n_items = len(labels)
    counts = Counter(labels)
    return pd.Series(
        {
            "N_items": n_items,
            "full_rate": counts.get("full", 0) / n_items,
            "partial_rate": counts.get("partial", 0) / n_items,
            "none_rate": counts.get("none", 0) / n_items,
        }
    )


def adjudication_resolution(
    r1_sets: List[Set[str]],
    r2_sets: List[Set[str]],
    r3_sets: List[Set[str]],
) -> pd.Series:
    recs = []
    for a, b, c in zip(r1_sets, r2_sets, r3_sets):
        if a == b:
            continue
        union_set = a | b
        inter_set = a & b
        recs.append(
            {
                "r3_equals_r1": int(c == a),
                "r3_equals_r2": int(c == b),
                "r3_equals_union": int(c == union_set),
                "r3_equals_intersection": int((len(inter_set) > 0) and (c ==
                 ↪  inter_set)),
```

```
                "r3_introduces_new": int(len(c - union_set) > 0),
                "r3_subset_of_union": int(c <= union_set),
            }
        )

    if not recs:
        return pd.Series({"N_disagreements": 0})

    out = pd.DataFrame(recs)
    return pd.Series(
        {
            "N_disagreements": len(out),
            "r3_equals_r1_rate": out["r3_equals_r1"].mean(),
            "r3_equals_r2_rate": out["r3_equals_r2"].mean(),
            "r3_equals_union_rate": out["r3_equals_union"].mean(),
            "r3_equals_intersection_rate":
             ↪  out["r3_equals_intersection"].mean(),
            "r3_introduces_new_rate": out["r3_introduces_new"].mean(),
            "r3_subset_of_union_rate": out["r3_subset_of_union"].mean(),
        }
    )


# --- Binary one-vs-rest encoding for chance-corrected metrics
 ↪  -----------------

def flatten_binary_decisions(
    raters_sets: Dict[str, List[Set[str]]],
    categories: List[str],
) -> Dict[str, np.ndarray]:
    n = len(next(iter(raters_sets.values())))
    k = len(categories)
    idx: Dict[str, int] = {}
    for i, category in enumerate(categories):
        normalized_category = normalize_label(category)
        if normalized_category is None:
            continue
        if normalized_category in idx:
            raise ValueError(
                "Duplicate category after normalization in binary expansion:
                 ↪  "
                f"{category}"
            )
```

```
        idx[normalized_category] = i

    out: Dict[str, np.ndarray] = {}
    for rater, sets in raters_sets.items():
        mat = np.zeros((n, k), dtype=int)
        for j, labels in enumerate(sets):
            for lab in labels:
                lab = normalize_label(lab)
                if lab in idx:
                    mat[j, idx[lab]] = 1
        out[rater] = mat
    return out


def flatten_binary_decisions_single(
    sets: List[Set[str]],
    categories: List[str],
) -> np.ndarray:
    idx: Dict[str, int] = {}
    for i, category in enumerate(categories):
        normalized_category = normalize_label(category)
        if normalized_category is None:
            continue
        if normalized_category in idx:
            raise ValueError(
                "Duplicate category after normalization in binary expansion:
                ↪  "
                f"{category}"
            )
        idx[normalized_category] = i
    mat = np.zeros((len(sets), len(categories)), dtype=int)
    for j, labels in enumerate(sets):
        for lab in labels:
            lab = normalize_label(lab)
            if lab in idx:
                mat[j, idx[lab]] = 1
    return mat


# --- Chance-corrected agreement metrics
↪  ----------------------------------------

def pairwise_binary_agreement_stats(
```

```
    m1: np.ndarray,
    m2: np.ndarray,
    categories: List[str] | None = None,
    include_confusion_counts: bool = False,
) -> pd.DataFrame:
    """Compute per-category agreement, kappa, and AC1 for two raters.

    We convert the multi-label task into one binary task per category
    (present vs absent), then compute chance-corrected metrics.
    """

    n, k = m1.shape
    rows = []
    for c in range(k):
        y1 = m1[:, c]
        y2 = m2[:, c]
        tp = int(((y1 == 1) & (y2 == 1)).sum())
        tn = int(((y1 == 0) & (y2 == 0)).sum())
        fp = int(((y1 == 0) & (y2 == 1)).sum())
        fn = int(((y1 == 1) & (y2 == 0)).sum())
        total = tp + tn + fp + fn

        if total == 0:
            if include_confusion_counts:
                rows.append(
                    {
                        "category_ix": c,
                        "category": categories[c] if categories is not None
                        ↪  else c,
                        "N": 0,
                        "tp": tp,
                        "tn": tn,
                        "fp": fp,
                        "fn": fn,
                        "prevalence_r3": np.nan,
                        "prevalence_nlp": np.nan,
                        "percent_agreement": np.nan,
                        "cohen_kappa": np.nan,
                        "gwet_ac1": np.nan,
                    }
                )
            else:
                rows.append(
```

```
                    {
                        "category": c,
                        "N": 0,
                        "percent_agreement": np.nan,
                        "cohen_kappa": np.nan,
                        "gwet_ac1": np.nan,
                    }
                )
            continue

        # Observed agreement (Po): proportion of matching binary decisions.
        po = (tp + tn) / total
        p1 = y1.mean()
        p2 = y2.mean()
        # Cohen kappa expected agreement under independent prevalence.
        pe_kappa = p1 * p2 + (1 - p1) * (1 - p2)
        kappa = (po - pe_kappa) / (1 - pe_kappa) if (1 - pe_kappa) != 0 else
↪  np.nan
        # AC1 expected agreement uses pooled prevalence (pbar).
        pbar = (p1 + p2) / 2
        pe_ac1 = 2 * pbar * (1 - pbar)
        ac1 = (po - pe_ac1) / (1 - pe_ac1) if (1 - pe_ac1) != 0 else np.nan

        if include_confusion_counts:
            rows.append(
                {
                    "category_ix": c,
                    "category": categories[c] if categories is not None else
                     ↪  c,
                    "N": total,
                    "tp": tp,
                    "tn": tn,
                    "fp": fp,
                    "fn": fn,
                    "prevalence_r3": p1,
                    "prevalence_nlp": p2,
                    "percent_agreement": po,
                    "cohen_kappa": kappa,
                    "gwet_ac1": ac1,
                }
            )
        else:
            rows.append(
```

```
                {
                    "category": c,
                    "N": total,
                    "percent_agreement": po,
                    "cohen_kappa": kappa,
                    "gwet_ac1": ac1,
                }
            )

    return pd.DataFrame(rows)


def multirater_ac1_per_category(
    matrices: List[np.ndarray],
) -> pd.DataFrame:
    """Compute multi-rater AC1 per category using average pairwise
    ↪ agreement."""

    m = len(matrices)
    n, k = matrices[0].shape
    stack = np.stack(matrices, axis=0)
    denom_pairs = m * (m - 1) / 2

    rows = []
    for c in range(k):
        m_c = stack[:, :, c]
        npos = m_c.sum(axis=0)
        nneg = m - npos
        # For each visit, Aj is the proportion of agreeing rater pairs
        # (both positive or both negative) for this category.
        a_j = (npos * (npos - 1) / 2 + nneg * (nneg - 1) / 2) / denom_pairs
        po = a_j.mean()
        p = npos.sum() / (n * m)
        pe = 2 * p * (1 - p)
        ac1 = (po - pe) / (1 - pe) if (1 - pe) != 0 else np.nan
        rows.append({"category": c, "percent_agreement": po, "gwet_ac1": ac1,
↪   "prevalence": p})

    return pd.DataFrame(rows)


# Summarize per-category chance-corrected results into macro and micro views.
def summarize_kappa_ac1(df_pw: pd.DataFrame) -> pd.Series:
```

```
    out = {
        "macro_cohen_kappa": df_pw["cohen_kappa"].mean(),
        "macro_gwet_ac1": df_pw["gwet_ac1"].mean(),
        "macro_percent_agreement": df_pw["percent_agreement"].mean(),
    }

    # Prefer confusion-matrix micro agreement when counts are available.
    if {"tp", "tn", "N"}.issubset(df_pw.columns):
        out["micro_percent_agreement"] = (df_pw["tp"].sum() +
↪   df_pw["tn"].sum()) / df_pw["N"].sum()
    else:
        out["micro_percent_agreement"] = (df_pw["N"] *
↪   df_pw["percent_agreement"]).sum() / df_pw["N"].sum()

    return pd.Series(out)


# --- Assertion helpers used to fail fast during notebook runs
↪   ------------------

def assert_rate_bounds(series: pd.Series, label: str) -> None:
    non_null = series.dropna()
    if non_null.empty:
        return
    min_val = float(non_null.min())
    max_val = float(non_null.max())
    if min_val < 0 or max_val > 1:
        raise AssertionError(f"{label} must be in [0, 1], got [{min_val},
        ↪  {max_val}]")


def assert_expected_columns(df: pd.DataFrame, required: List[str], label:
↪   str) -> None:
    missing = sorted(set(required).difference(df.columns))
    if missing:
        raise AssertionError(f"{label} missing expected columns: {missing}")

#
↪   ------------------------------------------------------------------------
# Analysis A: Human rater agreement (R1, R2, R3)
#
# Workflow:
# 1) Load validated annotation/category inputs.
```

```
# 2) Convert each visit to an unordered set of categories per rater.
# 3) Compute set-level metrics and chance-corrected binary metrics.
# 4) Run assertions, then write reproducible output artifacts.
#
 ↪  ------------------------------------------------------------------------

# Canonical category names define the fixed category universe used for
# binary one-vs-rest metrics.
CANONICAL_CATS = load_categories_from_sheet(
    ANNOTATION_PATH,
    sheet=CATEGORY_SHEET,
    column=CATEGORY_COLUMN,
)

df = read_sheet_checked(ANNOTATION_PATH, ANNOTATION_SHEET)

# Validate required rater columns up front so missing data fails early.
key_cols = ["hadm_id", "subject_id"]
required_rater_cols = key_cols + [
    f"annot{r}_rvs{i}_cat"
    for r in (1, 2, 3)
    for i in range(1, 6)
]
validate_required_columns(df, required_rater_cols, "Annotation sheet")
validate_unique_keys(df, key_cols, "Annotation sheet")

r1_sets = extract_rater_sets(df, 1)
r2_sets = extract_rater_sets(df, 2)
r3_sets = extract_rater_sets(df, 3)

assert len(r1_sets) == len(df), "R1 set count must equal annotation row
 ↪  count"
assert len(r2_sets) == len(df), "R2 set count must equal annotation row
 ↪  count"
assert len(r3_sets) == len(df), "R3 set count must equal annotation row
 ↪  count"

# Pairwise set-level agreement captures visit-level overlap/exactness.
pair12_df, pair12_summary = summarize_set_agreement(r1_sets, r2_sets)
pair13_df, pair13_summary = summarize_set_agreement(r1_sets, r3_sets)
pair23_df, pair23_summary = summarize_set_agreement(r2_sets, r3_sets)

three_summary = summarize_three_way(r1_sets, r2_sets, r3_sets)
```

```
# Chance-corrected metrics require binary matrices by category.
raters_sets = {"r1": r1_sets, "r2": r2_sets, "r3": r3_sets}
bin_mats = flatten_binary_decisions(raters_sets, CANONICAL_CATS)

for rater_name, mat in bin_mats.items():
    assert mat.shape == (len(df), len(CANONICAL_CATS)), (
        f"{rater_name} binary matrix shape mismatch: expected "
        f"({len(df)}, {len(CANONICAL_CATS)}), got {mat.shape}"
    )

pw12 = pairwise_binary_agreement_stats(bin_mats["r1"], bin_mats["r2"])
pw13 = pairwise_binary_agreement_stats(bin_mats["r1"], bin_mats["r3"])
pw23 = pairwise_binary_agreement_stats(bin_mats["r2"], bin_mats["r3"])

category_code_map = {i: c for i, c in enumerate(CANONICAL_CATS)}
category_label_map = {i: canonical_label_for_code(c) for i, c in
 ↪  enumerate(CANONICAL_CATS)}
for df_pw in (pw12, pw13, pw23):
    df_pw["category_code"] = df_pw["category"].map(category_code_map)
    df_pw["category_name"] = df_pw["category"].map(category_label_map)

multi_ac1 = multirater_ac1_per_category([bin_mats["r1"], bin_mats["r2"],
 ↪  bin_mats["r3"]])
multi_ac1["category_code"] = multi_ac1["category"].map(category_code_map)
multi_ac1["category_name"] = multi_ac1["category"].map(category_label_map)

pair12_chance = summarize_kappa_ac1(pw12)
pair13_chance = summarize_kappa_ac1(pw13)
pair23_chance = summarize_kappa_ac1(pw23)

multi_macro_ac1 = multi_ac1["gwet_ac1"].mean()
multi_macro_agree = multi_ac1["percent_agreement"].mean()

# Deterministic validation checks
# These assertions guard against silent failures (shape drift, out-of-range
# rates, and missing expected columns).
for summary_name, summary in (
    ("pair12_summary", pair12_summary),
    ("pair13_summary", pair13_summary),
    ("pair23_summary", pair23_summary),
):
```

```
    assert int(summary["N_items"]) == len(df), f"{summary_name} N_items
     ↪ mismatch"
    assert_rate_bounds(summary[["exact_rate", "partial_rate", "none_rate"]],
 ↪ f"{summary_name} rates")
    assert_rate_bounds(summary[["mean_jaccard", "mean_overlap",
 ↪ "mean_f1_set", "micro_f1_set"]], f"{summary_name} similarity metrics")

assert_rate_bounds(three_summary[["full_rate", "partial_rate", "none_rate"]],
 ↪ "three_summary rates")
assert int(three_summary["N_items"]) == len(df), "three_summary N_items
 ↪ mismatch"

for label, df_pw in (("pw12", pw12), ("pw13", pw13), ("pw23", pw23)):
    assert_expected_columns(
        df_pw,
        ["category", "N", "percent_agreement", "cohen_kappa", "gwet_ac1",
 ↪ "category_code", "category_name"],
        label,
    )
    assert len(df_pw) == len(CANONICAL_CATS), f"{label} row count mismatch
     ↪ with categories"
    assert_rate_bounds(df_pw["percent_agreement"], f"{label}
 ↪ percent_agreement")
    assert_rate_bounds(df_pw["gwet_ac1"], f"{label} gwet_ac1")

assert_expected_columns(
    multi_ac1,
    ["category", "percent_agreement", "gwet_ac1", "prevalence",
 ↪ "category_code", "category_name"],
    "multi_ac1",
)
assert len(multi_ac1) == len(CANONICAL_CATS), "multi_ac1 row count mismatch
 ↪ with categories"
assert_rate_bounds(multi_ac1["percent_agreement"], "multi_ac1
 ↪ percent_agreement")
assert_rate_bounds(multi_ac1["gwet_ac1"], "multi_ac1 gwet_ac1")

# Output filenames are intentionally unchanged to preserve downstream
# manuscript/report references.
# Outputs
pair12_df.to_csv(RATERS_OUTPUT_DIR / "pair_R1_R2_set_metrics.csv",
 ↪ index=False)
```

```
pair13_df.to_csv(RATERS_OUTPUT_DIR / "pair_R1_R3_set_metrics.csv",
 ↪  index=False)
pair23_df.to_csv(RATERS_OUTPUT_DIR / "pair_R2_R3_set_metrics.csv",
 ↪  index=False)

pw12.to_csv(RATERS_OUTPUT_DIR / "pair_R1_R2_binary_stats.csv", index=False)
pw13.to_csv(RATERS_OUTPUT_DIR / "pair_R1_R3_binary_stats.csv", index=False)
pw23.to_csv(RATERS_OUTPUT_DIR / "pair_R2_R3_binary_stats.csv", index=False)

multi_ac1.to_csv(RATERS_OUTPUT_DIR / "all3_multirater_ac1_by_category.csv",
 ↪  index=False)

adj_summary = adjudication_resolution(r1_sets, r2_sets, r3_sets)

summary_text = f"""
=== Pairwise set-level (order-invariant) ===

R1 vs R2:
{pair12_summary.to_string()}

R1 vs R3:
{pair13_summary.to_string()}

R2 vs R3:
{pair23_summary.to_string()}

=== Three-rater set-level (full/partial/none) ===
{three_summary.to_string()}

=== Chance-corrected (binary per category) ===
R1 vs R2:
{pair12_chance.to_string()}

R1 vs R3:
{pair13_chance.to_string()}

R2 vs R3:
{pair23_chance.to_string()}

=== Multi-rater AC1 (3 raters) ===
macro_gwet_ac1={multi_macro_ac1:.4f}
macro_percent_agreement={multi_macro_agree:.4f}
```

```
=== Adjudication (R3) for R1 R2 ===
{adj_summary.to_string()}
"""

(RATERS_OUTPUT_DIR / "summary.txt").write_text(summary_text)

print(summary_text)
print("\nOutputs written to:", RATERS_OUTPUT_DIR.resolve())
```

```
=== Pairwise set-level (order-invariant) ===

R1 vs R2:
N_items         160.000000
exact_rate        0.806250
partial_rate      0.131250
none_rate         0.062500
mean_jaccard      0.856250
mean_overlap      0.882292
mean_f1_set       0.877440
micro_f1_set      0.873950
mean_len_a        1.493750
mean_len_b        1.481250

R1 vs R3:
N_items         160.000000
exact_rate        0.850000
partial_rate      0.075000
none_rate         0.075000
mean_jaccard      0.879167
mean_overlap      0.892708
mean_f1_set       0.891190
micro_f1_set      0.897275
mean_len_a        1.493750
mean_len_b        1.487500

R2 vs R3:
N_items         160.000000
exact_rate        0.831250
partial_rate      0.112500
none_rate         0.056250
mean_jaccard      0.876042
mean_overlap      0.900000
```

```
mean_f1_set       0.894583
micro_f1_set      0.892632
mean_len_a        1.481250
mean_len_b        1.487500

=== Three-rater set-level (full/partial/none) ===
N_items         160.00000
full_rate         0.75625
partial_rate      0.23125
none_rate         0.01250

=== Chance-corrected (binary per category) ===
R1 vs R2:
macro_cohen_kappa          0.734781
macro_gwet_ac1             0.972273
macro_percent_agreement    0.977941
micro_percent_agreement    0.977941

R1 vs R3:
macro_cohen_kappa          0.801703
macro_gwet_ac1             0.976768
macro_percent_agreement    0.981985
micro_percent_agreement    0.981985

R2 vs R3:
macro_cohen_kappa          0.779458
macro_gwet_ac1             0.976298
macro_percent_agreement    0.981250
micro_percent_agreement    0.981250

=== Multi-rater AC1 (3 raters) ===
macro_gwet_ac1=0.9751
macro_percent_agreement=0.9804

=== Adjudication (R3) for R1 R2 ===
N_disagreements                31.000000
r3_equals_r1_rate               0.483871
r3_equals_r2_rate               0.387097
r3_equals_union_rate            0.096774
r3_equals_intersection_rate     0.000000
r3_introduces_new_rate          0.096774
r3_subset_of_union_rate         0.903226
```

```
Outputs written to: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Reso
CC-NLP/Annotation/Full Annotations/Agreement Metrics
```

3.4 Agreement with NLP Pipeline

Rationale: Compare adjudicated human labels with NLP outputs to assess model agreement and error modes.

```
#
 ↪  ------------------------------------------------------------------------
# Analysis B: Adjudicator (R3) vs NLP agreement
#
# We intentionally mirror Analysis A so outputs are directly comparable:
# same set-level metrics and same binary chance-corrected framework.
#
 ↪  ------------------------------------------------------------------------

import json

# Step 1: Load adjudicator labels and validate keys/columns.
df = read_sheet_checked(ANNOTATION_PATH, ANNOTATION_SHEET)
r3_key_candidates = ["hadm_id", "subject_id", "ed_stay_id"]
r3_label_cols = [
    c for c in df.columns
    if c.startswith("annot3_rvs") and c.endswith("_cat")
]
r3_present_keys = [column for column in r3_key_candidates if column in
 ↪  df.columns]
r3_cols = r3_present_keys + r3_label_cols
validate_required_columns(df, r3_cols, "Annotation sheet for R3")
df_r3 = df[r3_cols].copy()

# Step 2: Load NLP outputs and validate key columns.
df_nlp = read_sheet_checked(NLP_PATH, NLP_SHEET)
nlp_key_candidates = ["hadm_id", "subject_id", "ed_stay_id"]
nlp_name_cols = [f"RFV{i}_name" for i in range(1, 6)]
present_name_cols = [c for c in nlp_name_cols if c in df_nlp.columns]
nlp_present_keys = [column for column in nlp_key_candidates if column in
 ↪  df_nlp.columns]
validate_required_columns(df_nlp, nlp_present_keys, "NLP sheet")
if not present_name_cols:
    raise KeyError("NLP file must contain RFV*_name columns.")
```

```
r3_join_keys, join_key_strategy = select_join_key_columns(df_r3, df_nlp)
validate_unique_keys(df_r3, r3_join_keys, "Annotation sheet for R3")
validate_unique_keys(df_nlp, r3_join_keys, "NLP sheet")
key_inventory = build_join_key_inventory(
    df_r3,
    df_nlp,
    r3_join_keys,
    strategy=join_key_strategy,
)
label_mapping_audit = build_label_mapping_audit(
    df_r3,
    df_nlp,
    r3_label_cols=r3_label_cols,
    nlp_label_cols=present_name_cols,
)
unmapped_label_count =
 ↪  int((~label_mapping_audit["mapped"].fillna(False)).sum())
if unmapped_label_count > 0:
    print(
        "WARNING: Unmapped category labels detected in R3/NLP mapping
         ↪  audit:",
        unmapped_label_count,
    )

# Step 3: Build key-normalized join plus explicit overlap audit.
df_join, unmatched_adjudicated, unmatched_nlp, join_audit =
 ↪  build_r3_nlp_join_audit(
    df_r3,
    df_nlp,
    r3_join_keys,
    target_sample_n=R3_NLP_TARGET_SAMPLE_N,
    gate_policy=R3_NLP_GATE_POLICY,
    key_strategy=join_key_strategy,
)

n_before = len(df_r3)
n_after = int(join_audit["matched_rows"])
n_dropped = int(join_audit["unmatched_adjudicated_rows"])
n_unmatched_nlp = int(join_audit["unmatched_nlp_rows"])

assert n_after > 0, "Join produced zero rows; check key consistency between
 ↪  annotation and NLP workbooks."
```

```
assert n_after <= n_before, "Inner join should not increase rows when both
 ↪  sides have unique keys."

def _fmt_rate(value: float | None) -> str:
    return "NA" if value is None else f"{value:.1%}"

print(
    f"Joined R3 to NLP on {join_audit['key_columns']}: "
    f"{n_after} matched
     ↪  ({_fmt_rate(join_audit['matched_rate_vs_adjudicated'])}); "
    f"{n_dropped} adjudicated rows had no NLP match "
    f"({_fmt_rate(join_audit['unmatched_rate_vs_adjudicated'])}); "
    f"{n_unmatched_nlp} NLP rows had no adjudicated match "
    f"({_fmt_rate(join_audit['unmatched_rate_vs_nlp'])})."
)
join_severity = str(join_audit.get("severity", "warning")).lower()
join_interpretation = str(join_audit.get("join_interpretation",
 ↪  "below_target_overlap_nonzero"))
coverage_status = str(join_audit.get("coverage_status", "warning"))
target_sample_n = int(join_audit.get("target_sample_n",
 ↪  R3_NLP_TARGET_SAMPLE_N))
if coverage_status == "warning":
    print(
        "WARNING: R3/NLP overlap is below target sample size "
        f"(matched={n_after}, target={target_sample_n}); metrics remain valid
         ↪  on matched rows."
    )
if n_dropped > 0 or n_unmatched_nlp > 0:
    if join_severity == "info":
        print(
            "INFO: Non-identical overlap is expected for subset benchmarking
             ↪  "
            f"({join_interpretation}). Metrics are computed on matched rows
             ↪  only."
        )
    else:
        print(
            "WARNING: Non-identical overlap between adjudicated and NLP
             ↪  cohorts "
            f"({join_interpretation}). Metrics are computed on matched rows
             ↪  only."
        )
```

```
# Step 4: Convert joined rows to unordered category sets for both sources.
r3_sets_join = extract_rater_sets_from_df(df_join, rater_col_prefix="annot3")
nlp_sets = extract_nlp_sets_from_df(df_join, min_sim=NLP_MIN_SIM)

assert len(r3_sets_join) == len(df_join), "R3 joined set count mismatch"
assert len(nlp_sets) == len(df_join), "NLP set count mismatch"

# Step 5: Set-level agreement gives intuitive visit-level similarity.
pair_df, pair_summary = summarize_set_agreement(r3_sets_join, nlp_sets)
assert int(pair_summary["N_items"]) == len(df_join), "R3/NLP summary N_items
 ↪  mismatch"
assert_rate_bounds(pair_summary[["exact_rate", "partial_rate", "none_rate"]],
 ↪  "R3/NLP set rates")
assert_rate_bounds(pair_summary[["mean_jaccard", "mean_overlap",
 ↪  "mean_f1_set", "micro_f1_set"]], "R3/NLP set similarity metrics")

# Step 6: Record category-level differences per visit for error analysis.
def list_diff(a: Set[str], b: Set[str]) -> Tuple[List[str], List[str]]:
    return sorted(a - b), sorted(b - a)

missed = []
added = []
for a, b in zip(r3_sets_join, nlp_sets):
    m, ad = list_diff(a, b)
    missed.append("; ".join(m))
    added.append("; ".join(ad))

visit_metrics = df_join[r3_join_keys].copy()
visit_metrics["exact"] = pair_df["exact"]
visit_metrics["partial"] = pair_df["partial"]
visit_metrics["none"] = pair_df["none"]
visit_metrics["jaccard"] = pair_df["jaccard"]
visit_metrics["overlap"] = pair_df["overlap"]
visit_metrics["f1_set"] = pair_df["f1_set"]
visit_metrics["r3_size"] = pair_df["len_a"]
visit_metrics["nlp_size"] = pair_df["len_b"]
visit_metrics["missed_by_nlp"] = missed
visit_metrics["added_by_nlp"] = added

assert_expected_columns(
    visit_metrics,
    r3_join_keys
    + [
```

```
        "exact",
        "partial",
        "none",
        "jaccard",
        "overlap",
        "f1_set",
        "r3_size",
        "nlp_size",
        "missed_by_nlp",
        "added_by_nlp",
    ],
    "visit_metrics",
)
assert len(visit_metrics) == len(df_join), "visit_metrics row count mismatch"

# Step 7: Compute per-category chance-corrected metrics on binary matrices.
m_r3 = flatten_binary_decisions_single(r3_sets_join, CANONICAL_CATS)
m_nlp = flatten_binary_decisions_single(nlp_sets, CANONICAL_CATS)

assert m_r3.shape == (len(df_join), len(CANONICAL_CATS)), "R3 binary matrix
 ↪  shape mismatch"
assert m_nlp.shape == (len(df_join), len(CANONICAL_CATS)), "NLP binary matrix
 ↪  shape mismatch"

pw_stats = pairwise_binary_agreement_stats(
    m_r3,
    m_nlp,
    categories=CANONICAL_CATS,
    include_confusion_counts=True,
)
pw_stats["category_code"] = pw_stats["category"].astype("string")
pw_stats["category_label"] =
 ↪  pw_stats["category_code"].map(canonical_label_for_code)
chance_summary = summarize_kappa_ac1(pw_stats)

assert_expected_columns(
    pw_stats,
    [
        "category_ix",
        "category",
        "category_code",
        "category_label",
        "N",
```

```
        "tp",
        "tn",
        "fp",
        "fn",
        "prevalence_r3",
        "prevalence_nlp",
        "percent_agreement",
        "cohen_kappa",
        "gwet_ac1",
    ],
    "pw_stats",
)
assert len(pw_stats) == len(CANONICAL_CATS), "pw_stats row count mismatch
 ↪  with categories"
assert_rate_bounds(pw_stats["percent_agreement"], "pw_stats
 ↪  percent_agreement")
assert_rate_bounds(pw_stats["gwet_ac1"], "pw_stats gwet_ac1")

# Fail-fast consistency checks: binary per-category tables must reflect
# non-exact set-level disagreement and non-empty label prevalence.
non_exact_visit_n = int(
    (pd.to_numeric(visit_metrics["exact"],
 ↪  errors="coerce").fillna(0).eq(0)).sum()
)
binary_disagreement_n = int(
    pd.to_numeric(pw_stats["fp"], errors="coerce").fillna(0).sum()
    + pd.to_numeric(pw_stats["fn"], errors="coerce").fillna(0).sum()
)
if non_exact_visit_n > 0 and binary_disagreement_n == 0:
    raise ValueError(
        "R3-vs-NLP binary expansion is degenerate: set-level disagreements
         ↪  exist "
        f"(non_exact_visit_n={non_exact_visit_n}) but fp+fn==0 across
         ↪  categories."
    )

non_empty_set_visit_n = int(
    (
        pd.to_numeric(visit_metrics["r3_size"],
 ↪  errors="coerce").fillna(0).gt(0)
        | pd.to_numeric(visit_metrics["nlp_size"],
         ↪  errors="coerce").fillna(0).gt(0)
    ).sum()
```

```
)
category_prevalence_nonzero_n = int(
    (
        pd.to_numeric(pw_stats["prevalence_r3"],
 ↪  errors="coerce").fillna(0).gt(0)
        | pd.to_numeric(pw_stats["prevalence_nlp"],
         ↪  errors="coerce").fillna(0).gt(0)
    ).sum()
)
if non_empty_set_visit_n > 0 and category_prevalence_nonzero_n == 0:
    raise ValueError(
        "R3-vs-NLP binary expansion has zero category prevalence despite
         ↪  non-empty "
        f"label sets (non_empty_set_visit_n={non_empty_set_visit_n})."
    )

# Step 8: Write outputs using stable filenames for downstream workflows.
visit_metrics.to_csv(NLP_OUTPUT_DIR / "R3_vs_NLP_set_metrics_by_visit.csv",
 ↪  index=False)
pw_stats.to_csv(NLP_OUTPUT_DIR / "R3_vs_NLP_binary_stats_by_category.csv",
 ↪  index=False)
unmatched_adjudicated.to_csv(
    NLP_OUTPUT_DIR / "R3_vs_NLP_unmatched_adjudicated_keys.csv",
    index=False,
)
unmatched_nlp.to_csv(
    NLP_OUTPUT_DIR / "R3_vs_NLP_unmatched_nlp_keys.csv",
    index=False,
)
(NLP_OUTPUT_DIR / "R3_vs_NLP_join_audit.json").write_text(
    json.dumps(join_audit, indent=2)
)
key_inventory.to_csv(
    NLP_OUTPUT_DIR / "R3_vs_NLP_key_inventory.csv",
    index=False,
)
label_mapping_audit.to_csv(
    NLP_OUTPUT_DIR / "R3_vs_NLP_label_mapping_audit.csv",
    index=False,
)

matched_key_hashes = hash_join_keys(df_join[r3_join_keys],
 ↪  key_cols=r3_join_keys)
```

```
matched_key_hashes.to_csv(
    NLP_OUTPUT_DIR / "R3_vs_NLP_matched_key_hashes.csv",
    index=False,
)

input_manifest = {
    "annotation_path": str(ANNOTATION_PATH),
    "annotation_sheet": ANNOTATION_SHEET,
    "nlp_path": str(NLP_PATH),
    "nlp_sheet": NLP_SHEET,
    "row_counts": {
        "annotation_r3_rows": int(len(df_r3)),
        "nlp_rows": int(len(df_nlp)),
        "matched_rows": int(len(df_join)),
        "unmatched_adjudicated_rows": int(len(unmatched_adjudicated)),
        "unmatched_nlp_rows": int(len(unmatched_nlp)),
    },
    "key_columns": r3_join_keys,
    "key_strategy": join_key_strategy,
    "join_audit": join_audit,
    "matched_key_hash_count": int(len(matched_key_hashes)),
    "unmapped_label_count": unmapped_label_count,
}
(NLP_OUTPUT_DIR / "R3_vs_NLP_input_manifest.json").write_text(
    json.dumps(input_manifest, indent=2)
)

rater_run_manifest = collect_run_manifest(
    WORK_DIR,
    run_id=f"rater_{pd.Timestamp.now().strftime('%Y%m%d_%H%M%S')}",
)
rater_run_manifest["stage"] = "rater"
rater_run_manifest["inputs"] = {
    "annotation_path": str(ANNOTATION_PATH),
    "annotation_sheet": ANNOTATION_SHEET,
    "nlp_path": str(NLP_PATH),
    "nlp_sheet": NLP_SHEET,
}
rater_run_manifest["join_audit"] = join_audit
rater_run_manifest["outputs"] = {
    "summary_path": str(NLP_OUTPUT_DIR / "R3_vs_NLP_summary.txt"),
    "join_audit_path": str(NLP_OUTPUT_DIR / "R3_vs_NLP_join_audit.json"),
```

```
    "key_inventory_path": str(NLP_OUTPUT_DIR /
     ↪  "R3_vs_NLP_key_inventory.csv"),
    "label_mapping_audit_path": str(NLP_OUTPUT_DIR /
     ↪  "R3_vs_NLP_label_mapping_audit.csv"),
    "input_manifest_path": str(NLP_OUTPUT_DIR /
     ↪  "R3_vs_NLP_input_manifest.json"),
    "matched_key_hashes_path": str(NLP_OUTPUT_DIR /
     ↪  "R3_vs_NLP_matched_key_hashes.csv"),
}
(NLP_OUTPUT_DIR / "R3_vs_NLP_run_manifest.json").write_text(
    json.dumps(rater_run_manifest, indent=2)
)

summary_lines = []
summary_lines.append("=== R3 vs NLP join audit ===")
summary_lines.append(pd.Series(join_audit).to_string())
summary_lines.append("=== R3 (adjudicator) vs NLP: set-level
 ↪  (order-invariant) ===")
summary_lines.append(pair_summary.to_string())
summary_lines.append("\n=== R3 vs NLP: chance-corrected (binary per category)
 ↪  ===")
summary_lines.append(chance_summary.to_string())
summary_text = "\n\n".join(summary_lines)

(NLP_OUTPUT_DIR / "R3_vs_NLP_summary.txt").write_text(summary_text)

print(summary_text)
print("\nArtifacts written to:", NLP_OUTPUT_DIR.resolve())
```

```
Joined R3 to NLP on ['hadm_id', 'subject_id']: 33 matched (20.6%); 127 adjudicated rows had r
WARNING: R3/NLP overlap is below target sample size (matched=33, target=160); metrics remain
WARNING: Non-identical overlap between adjudicated and NLP cohorts (below_target_overlap_nonz
=== R3 vs NLP join audit ===

key_columns                                [hadm_id, subject_id]
key_strategy                                  hadm_id_subject_id
target_sample_n                                              160
gate_policy                    warn_below_target_fail_on_zero
r3_rows                                                      160
nlp_rows                                                    7152
matched_rows                                                  33
unmatched_adjudicated_rows                                   127
unmatched_nlp_rows                                          7119
```

```
matched_rate_vs_adjudicated                              0.20625
unmatched_rate_vs_adjudicated                            0.79375
unmatched_rate_vs_nlp                                   0.995386
coverage_status                                          warning
join_interpretation                 below_target_overlap_nonzero
severity                                                 warning

=== R3 (adjudicator) vs NLP: set-level (order-invariant) ===

N_items          33.000000
exact_rate        0.818182
partial_rate      0.060606
none_rate         0.121212
mean_jaccard      0.848485
mean_overlap      0.863636
mean_f1_set       0.857576
micro_f1_set      0.876404
mean_len_a        1.333333
mean_len_b        1.363636


=== R3 vs NLP: chance-corrected (binary per category) ===

macro_cohen_kappa          0.678707
macro_gwet_ac1             0.975872
macro_percent_agreement    0.980392
micro_percent_agreement    0.980392

Artifacts written to: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Re
CC-NLP/annotation_agreement_outputs_nlp
```

```
join_audit_table = pd.DataFrame([join_audit])
pair_summary_table =
 ↪  pair_summary.to_frame(name="value").reset_index().rename(
    columns={"index": "metric"}
)
chance_summary_table = chance_summary.reset_index().rename(columns={"index":
 ↪  "metric"})

print(
    render_latex_longtable(
        join_audit_table,
```

```
        caption="R3 vs NLP join audit summary.",
        label="tab:r3_nlp_join_audit",
        landscape=True,
        index=False,
    )
)
print(
    render_latex_longtable(
        pair_summary_table,
        caption="R3 vs NLP set-level agreement summary.",
        label="tab:r3_nlp_set_summary",
        index=False,
    )
)
print(
    render_latex_longtable(
        chance_summary_table,
        caption="R3 vs NLP chance-corrected category-level agreement
↪  summary.",
        label="tab:r3_nlp_chance_summary",
        landscape=True,
        index=False,
    )
)
```

```
\begin{landscape}\n\begin{longtable}{llrlrrrrrrrrlll}
\caption{R3 vs NLP join audit summary.} \label{tab:r3_nlp_join_audit} \\
\toprule
key_columns & key_strategy & target_sample_n & gate_policy & r3_rows & nlp_rows & matched_rou
\midrule
\endfirsthead
\caption[]{R3 vs NLP join audit summary.} \\
\toprule
key_columns & key_strategy & target_sample_n & gate_policy & r3_rows & nlp_rows & matched_rou
\midrule
\endhead
\midrule
\multicolumn{15}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
['hadm_id', 'subject_id'] & hadm_id_subject_id & 160 & warn_below_target_fail_on_zero & 160 &
```

```
\end{longtable}
\n\end{landscape}\n
\begin{longtable}{lr}
\caption{R3 vs NLP set-level agreement summary.} \label{tab:r3_nlp_set_summary} \\
\toprule
metric & value \\
\midrule
\endfirsthead
\caption[]{R3 vs NLP set-level agreement summary.} \\
\toprule
metric & value \\
\midrule
\endhead
\midrule
\multicolumn{2}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
N_items & 33.000000 \\
exact_rate & 0.818182 \\
partial_rate & 0.060606 \\
none_rate & 0.121212 \\
mean_jaccard & 0.848485 \\
mean_overlap & 0.863636 \\
mean_f1_set & 0.857576 \\
micro_f1_set & 0.876404 \\
mean_len_a & 1.333333 \\
mean_len_b & 1.363636 \\
\end{longtable}

\begin{landscape}\n\begin{longtable}{lr}
\caption{R3 vs NLP chance-corrected category-level agreement summary.} \label{tab:r3_nlp_cha
\toprule
metric & 0 \\
\midrule
\endfirsthead
\caption[]{R3 vs NLP chance-corrected category-level agreement summary.} \\
\toprule
metric & 0 \\
\midrule
\endhead
\midrule
```

```
\multicolumn{2}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
macro_cohen_kappa & 0.678707 \\
macro_gwet_ac1 & 0.975872 \\
macro_percent_agreement & 0.980392 \\
micro_percent_agreement & 0.980392 \\
\end{longtable}
\n\end{landscape}\n
```

3.5 Draft text: validation metrics (edit as needed)

For multi-label RFV assignment, we treat each visit's labels as an unordered set and summarize agreement with set-based metrics (exact/partial/none, Jaccard, overlap, and set-F1). We report both mean set-F1 (average per visit) and micro set-F1 (pooled over all labels). For chance-corrected agreement, we expand the multi-label task into 17 one-vs-rest binary decisions (one per NHAMCS top-level category) and compute per-category Cohen's kappa and Gwet's AC1; macro-averages across categories are reported, and a multi-rater AC1 is computed across all three reviewers. Model-vs-adjudicator agreement uses the same set-level metrics and per-category binary stats, enabling per-category performance reporting.

Rationale: Provide manuscript-ready wording for the methods and validation sections.

```
#
 ↪  ------------------------------------------------------------------------
# Analysis C: Per-category summary tables used for manuscript-style reporting
#
 ↪  ------------------------------------------------------------------------


# This helper pools pairwise reviewer tables and reports mean metrics by
# category, which is easier to read than three separate pair tables.
def summarize_per_category_across_pairs(
    pw_list: List[pd.DataFrame],
    pair_names: List[str],
) -> pd.DataFrame:
    frames = []
    for frame, name in zip(pw_list, pair_names):
        tmp = frame.copy()
        tmp["pair"] = name
        if "category_name" not in tmp.columns:
```

```
            tmp["category_name"] = tmp.get("category")
        frames.append(tmp)

    all_pw = pd.concat(frames, ignore_index=True)
    summary = (
        all_pw.groupby("category_name", as_index=False)
        .agg(
            mean_kappa=("cohen_kappa", "mean"),
            mean_ac1=("gwet_ac1", "mean"),
            mean_percent_agreement=("percent_agreement", "mean"),
            N=("N", "sum"),
        )
        .sort_values("mean_ac1")
    )
    return summary


# Defensive check: this cell should run only after the analysis cells above.
for name in ("pw12", "pw13", "pw23", "pw_stats"):
    if name not in globals():
        raise RuntimeError(
            f"Expected variable '{name}' is missing. Run the analysis cells
             ↪  above first."
        )

per_cat_rater_summary = summarize_per_category_across_pairs(
    [pw12, pw13, pw23],
    ["R1_R2", "R1_R3", "R2_R3"],
)
assert_expected_columns(
    per_cat_rater_summary,
    ["category_name", "mean_kappa", "mean_ac1", "mean_percent_agreement",
 ↪  "N"],
    "per_cat_rater_summary",
)
per_cat_rater_summary.to_csv(RATERS_OUTPUT_DIR /
 ↪  "per_category_rater_agreement_summary.csv", index=False)

# Sorting by AC1 surfaces lowest-agreement categories first for targeted
# qualitative review.
per_cat_r3_nlp = pw_stats.copy().sort_values("gwet_ac1")
assert_expected_columns(
    per_cat_r3_nlp,
```

```
    [
        "category_ix",
        "category",
        "N",
        "tp",
        "tn",
        "fp",
        "fn",
        "prevalence_r3",
        "prevalence_nlp",
        "percent_agreement",
        "cohen_kappa",
        "gwet_ac1",
    ],
    "per_cat_r3_nlp",
)
assert per_cat_r3_nlp["gwet_ac1"].is_monotonic_increasing, "R3 vs NLP table
 ↪  should be sorted by gwet_ac1"
per_cat_r3_nlp.to_csv(NLP_OUTPUT_DIR /
 ↪  "R3_vs_NLP_binary_stats_by_category.csv", index=False)

# Quick glance: lowest agreement categories
_display_cols = ["category", "N", "percent_agreement", "cohen_kappa",
 ↪  "gwet_ac1"]
print("Lowest AC1 categories (R3 vs NLP):")
print(per_cat_r3_nlp[_display_cols].head(5).to_string(index=False))
```

```
Lowest AC1 categories (R3 vs NLP):
    category  N  percent_agreement  cohen_kappa  gwet_ac1
RVC-SYM-NERV 33           0.909091     0.741514  0.859873
    RVC-UNCL 33           0.939394    -0.031250  0.935610
   RVC-TREAT 33           0.939394    -0.031250  0.935610
 RVC-SYM-GEN 33           0.969697     0.840580  0.962606
     RVC-INJ 33           0.969697     0.840580  0.962606
```