Hypercap CC NLP Classifier

Table of contents

1 Workbook for MIMIC Hypercapnia Presenting Chief Concern Analysis

This notebook is intentionally self-contained and deterministic. It performs NLP-based mapping from free-text ED chief complaints to NHAMCS top-level RVC categories, with strict compatibility outputs for downstream analysis notebooks.

1.1 1) Runtime contract and imports

- No runtime package installation.
- Fail fast if required dependencies are missing.
- Keep environment deterministic and explicit.

```
# Purpose: verify required packages, import shared dependencies, and
 ↪  initialize NLP runtime objects.
# Why this matters: later cells assume these imports/models exist, so we fail
 ↪  early if the environment is incomplete.

import importlib

REQUIRED_IMPORTS = {
    "numpy": "numpy",
    "pandas": "pandas",
    "torch": "torch",
    "ftfy": "ftfy",
    "spacy": "spacy",
    "symspellpy": "symspellpy",
    "sentence_transformers": "sentence_transformers",
    "openpyxl": "openpyxl",
    "dotenv": "python-dotenv",
}

missing = []
for import_name, package_name in REQUIRED_IMPORTS.items():
    if importlib.util.find_spec(import_name) is None:
        missing.append((import_name, package_name))

if missing:
    message = "Missing required dependencies (install via uv sync): " + ",
 ↪  ".join(
        f"{import_name} ({package_name})" for import_name, package_name in
         ↪  missing
    )
    raise ImportError(message)

import hashlib
import json
import logging
import math
import os
import re
import sys
import warnings
from collections import Counter, defaultdict
from dataclasses import dataclass
from datetime import datetime
```

```
from pathlib import Path
from typing import Any, Mapping, Sequence

import ftfy
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import spacy
import torch
from dotenv import load_dotenv
from symspellpy.symspellpy import SymSpell, Verbosity

# Silence widget-only tqdm warning in environments without Jupyter progress
 ↪ widgets.
try:
    from tqdm import TqdmWarning
except Exception:  # pragma: no cover - defensive for uncommon tqdm import
 ↪ edge cases
    TqdmWarning = None

if TqdmWarning is not None:
    warnings.filterwarnings(
        "ignore",
        message="IProgress not found.*",
        category=TqdmWarning,
    )

# Reduce HF Hub warning noise in notebook output. If HF_TOKEN is set,
 ↪ authenticated requests still work.
try:
    from huggingface_hub import logging as hf_logging

    hf_logging.set_verbosity_error()
except Exception:  # pragma: no cover - keeps notebook robust if hub
 ↪ internals change
    pass

logging.getLogger("huggingface_hub").setLevel(logging.ERROR)
logging.getLogger("hf_xet").setLevel(logging.ERROR)

from sentence_transformers import SentenceTransformer, util
```

```
# spaCy model loading is config-driven in the setup cell so behavior is
 ↪ explicit.
NLP: Any | None = None
SPACY_MODEL_LABEL = "UNINITIALIZED (loaded in setup cell)"
print("spaCy pipeline: pending config-driven load")
```

```
spaCy pipeline: pending config-driven load
```

1.2 2) Config and constants

1.2.1 Core API (single-cell contract)

This cell defines the notebook's public in-notebook interfaces: - `ClassifierConfig` - `get_cc_column` - `normalize_cc` - `build_prototype_resources` - `build_segment_cache` - `classify_segments_cached` - `assign_rvc_per_segment_and_visit_cached` - `validate_output_schema` - `save_with_suffix`

```
# Purpose: centralize the notebook's core API (config, constants, data
 ↪ structures, and reusable functions).
# Why this matters: keeping definitions in one place makes behavior
 ↪ predictable and easier for new contributors to trace.

@dataclass(frozen=True)
class ClassifierConfig:
    """Configuration for notebook-contained NLP classification."""

    seed: int = 1234
    work_dir_env_var: str = "WORK_DIR"
    data_dirname: str = "MIMIC tabular data"
    input_filename: str = "MIMICIV all with CC.xlsx"
    input_filename_env_var: str = "CLASSIFIER_INPUT_FILENAME"
    output_suffix: str = "_with_NLP"
    output_sheet_name: str = "with_NLP"
    model_name: str = "NeuML/bioclinical-modernbert-base-embeddings"
    model_trust_remote_code: bool = True
    model_batch_size: int = 256
    segment_batch_size: int = 512
    scoring_method: str = "max"  # {"max", "lse"}
    alpha: float = 12.0
    abstain_threshold: float = 0.40
    max_rfv: int = 5
    max_segments_per_visit: int = 5
```

```
    prototype_dedup_sim_thresh: float = 0.92
    summary_weight_strategy: str = "uniform"  # {"uniform", "by_rows"}
    run_optional_plots: bool = False
    require_spacy_model: bool = True
    allow_uncodable_primary: bool = True
    empty_primary_policy: str = "blank"  # {"blank", "uncodable"}
    appendix_relpath: str = "Annotation/nhamcs_rvc_2022_appendixII_codes.csv"
    summary_relpath: str =
↪   "Annotation/nhamcs_rvc_2022_summary_by_top_level_17.csv"
    hf_token_env_var: str = "HF_TOKEN"


# --- Ontology labels and precedence used by final visit-level outputs. ---
RVC_NAME: 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",
}

PRECEDENCE_RVC: list[str] = [
    "RVC-INJ",
    "RVC-SYM-RESP",
    "RVC-SYM-CIRC",
    "RVC-SYM-NERV",
    "RVC-SYM-DIG",
    "RVC-SYM-GU",
    "RVC-SYM-MSK",
    "RVC-SYM-SKIN",
    "RVC-SYM-EYE",
```

```
    "RVC-SYM-GEN",
    "RVC-SYM-PSY",
    "RVC-DIS",
    "RVC-TEST",
    "RVC-DIAG",
    "RVC-TREAT",
    "RVC-ADMIN",
    "RVC-UNCL",
]

RVC_SHORT: dict[str, str] = {
    "RVC-INJ": "injury",
    "RVC-SYM-RESP": "resp",
    "RVC-SYM-CIRC": "circ",
    "RVC-SYM-NERV": "neuro",
    "RVC-SYM-DIG": "gi",
    "RVC-SYM-GU": "gu",
    "RVC-SYM-MSK": "msk",
    "RVC-SYM-SKIN": "skin",
    "RVC-SYM-EYE": "eye_ear",
    "RVC-SYM-GEN": "general",
    "RVC-SYM-PSY": "psych",
    "RVC-DIS": "disease",
    "RVC-TEST": "abn_test",
    "RVC-DIAG": "screen",
    "RVC-TREAT": "treatment",
    "RVC-ADMIN": "admin",
    "RVC-UNCL": "uncodable",
}

# --- Text normalization vocabulary and complaint parsing constants. ---
CC_CANDIDATES: tuple[str, ...] = (
    "ed_triage_cc",
    "chief_complaint",
    "ed_chief_complaint",
    "chiefcomplaint",
    "chief_complaint_text",
    "ed_cc",
    "cc",
)

PROTECT = ("n/v", "n/v/d", "s/p", "w/", "w/o", "h/o", "c/o", "h/a", "f/c",
 ↪  "c/p")
```

```
MISSING_RE = re.compile(
    r"^(?:n/?a|na|none|unknown|unk|no chief complaint|no complaint|not
     ↪  applicable|not available)$",
    re.I,
)
NEG_RE = re.compile(r"^(?:denies?|no|not|without|w/o)\b", re.I)

# Preserve a small set of clinically meaningful "absence" complaints instead
 ↪  of dropping as generic negation.
ABSENCE_SYMPTOM_RULES: tuple[tuple[re.Pattern[str], str], ...] = (
    (
        re.compile(r"^(?:no|without|w/o)\s+(?:urine|urinary)\s+(?:output|
 ↪  void(?:ing)?)\b", re.I),
        "urinary retention",
    ),
    (
        re.compile(r"^(?:no|without|w/o)\s+(?:bowel movement|bm|stool(?:
 ↪  output)?)\b", re.I),
        "constipation",
    ),
)

SEG_OVR: dict[str, str] = {
    "n/v": "nausea vomiting",
    "n/v/d": "nausea vomiting diarrhea",
    "nvd": "nausea vomiting diarrhea",
    "w/": "with",
    "w/o": "without",
    "s/p": "status post",
    "h/o": "history of",
    "c/o": "complains of",
    "h/a": "headache",
    "f/c": "fever chills",
    "c/p": "chest pain",
    "mva": "motor vehicle accident",
    "mvc": "motor vehicle collision",
    "ped vs auto": "pedestrian struck by vehicle",
    "gsw": "gunshot wound",
    "sa": "sexual assault",
}

TOK_OVR: dict[str, str] = {
    "sob": "shortness of breath",
```

```
"doe": "dyspnea on exertion",
"cp": "chest pain",
"palps": "palpitations",
"ha": "headache",
"h/a": "headache",
"f/c": "fever chills",
"c/p": "chest pain",
"loc": "loss of consciousness",
"ams": "altered mental status",
"sz": "seizure",
"szs": "seizure",
"cva": "stroke",
"tia": "transient ischemic attack",
"abd": "abdominal",
"abdo": "abdominal",
"rlq": "right lower quadrant",
"ruq": "right upper quadrant",
"llq": "left lower quadrant",
"luq": "left upper quadrant",
"lbp": "low back pain",
"brbpr": "rectal bleeding",
"uti": "urinary tract infection",
"dysuria": "painful urination",
"hematuria": "blood in urine",
"nv": "nausea vomiting",
"nvd": "nausea vomiting diarrhea",
"sorethroat": "sore throat",
"uri": "upper respiratory infection",
"pna": "pneumonia",
# Conservative high-frequency shorthand expansions from uncodable output
 ↪  audit.
"ich": "intracranial hemorrhage",
"sah": "subarachnoid hemorrhage",
"stemi": "st elevation myocardial infarction",
"nstemi": "non st elevation myocardial infarction",
"chf": "congestive heart failure",
"pe": "pulmonary embolism",
"sbo": "small bowel obstruction",
"gib": "gastrointestinal bleeding",
"hypoglycemia": "hypoglycemia",
"hypokalemia": "hypokalemia",
"arf": "acute renal failure",
"ili": "influenza like illness",
```

```
    "covid": "covid",
    "flu": "influenza",
    "si": "suicidal ideation",
    "hi": "homicidal ideation",
    "avh": "auditory visual hallucinations",
    "etoh": "alcohol",
    "od": "overdose",
    "fb": "foreign body",
    "vag": "vaginal",
    "vb": "vaginal bleeding",
    "hg": "hyperemesis gravidarum",
    "s/p": "status post",
    "w/": "with",
    "w/o": "without",
    "h/o": "history of",
    "c/o": "complains of",
    "n/v": "nausea vomiting",
    "n/v/d": "nausea vomiting diarrhea",
}

CTX_OVR: dict[str, tuple[Any, str, str | None]] = {
    "od": (
        lambda seg: not
         ↪  re.search(r"\b(eye|vision|ophth|ophthalm|cornea|ocular)\b", seg),
        "overdose",
        "right eye",
    ),
    "os": (
        lambda seg:
         ↪  re.search(r"\b(eye|vision|ophth|ophthalm|cornea|ocular)\b", seg),
        "left eye",
        None,
    ),
    "ou": (
        lambda seg:
         ↪  re.search(r"\b(eye|vision|ophth|ophthalm|cornea|ocular)\b", seg),
        "both eyes",
        None,
    ),
}

ABBR_MAP: dict[str, str] = {}
```

```
SAFE_SHORT_TOKENS = {"cp", "ha", "nv", "gi", "gu", "ms", "sx", "bp", "hr",
 ↪  "doe", "sob", "loc"}

_PUNCT_EDGE = re.compile(r"(^[^\w/]+)|([^\w/]+$)")
SEP_SPLIT_RE = re.compile(r"\s*(?:[;:,]|[+&])\s*")

CLINICAL_KEYS = {
    "pain",
    "fever",
    "chill",
    "cough",
    "wheez",
    "dyspnea",
    "shortness",
    "hemopty",
    "nausea",
    "vomit",
    "diarrhea",
    "bleeding",
    "rash",
    "injury",
    "wound",
    "laceration",
    "fracture",
    "sprain",
    "assault",
    "fall",
    "seizure",
    "syncope",
    "weakness",
    "numbness",
    "tingling",
    "headache",
    "dizziness",
    "palpitation",
    "chest",
    "abdominal",
    "flank",
    "pelvic",
    "back",
    "neck",
    "urinary",
    "dysuria",
```

```
    "hematuria",
    "pregnan",
    "vaginal",
    "overdose",
    "intoxication",
    "anxiety",
    "depression",
    "psychosis",
    "sore",
    "throat",
    "ear",
    "eye",
    "dental",
    "asthma",
    "copd",
    "flu",
    "covid",
    "pneumonia",
}

_DUR_RE = re.compile(
    r"\b([a-z]{2,})\s*x\s*(\d{1,3})\s*(d|day|h|hr|hour|w|wk|week|m|mo|mon|
    ↪  month|y|yr|year)s?\b",
    re.I,
)
_UNIT_MAP = {
    "d": "days",
    "day": "days",
    "h": "hours",
    "hr": "hours",
    "hour": "hours",
    "w": "weeks",
    "wk": "weeks",
    "week": "weeks",
    "m": "months",
    "mo": "months",
    "mon": "months",
    "month": "months",
    "y": "years",
    "yr": "years",
    "year": "years",
}
```

```
# --- Rule-gate regex definitions for deterministic high-confidence
 ↪  overrides. ---
RX_FOCAL_WEAK = re.compile(
    r"\b(focal|left|right|arm|leg|hand|face|hemiparesis|hemiplegia).*\b
     ↪  weakness\b|\bweakness\b.*\b(focal|left|right|arm|leg|hand|face)\b",
    re.I,
)

RULE_RX: dict[str, re.Pattern[str]] = {
    "RVC-INJ": re.compile(
        r"\b(mvc|mva|collision|ped(\s|)vs(\s|)auto|assault|fell|fall|gsw|
         ↪  gunshot|stab|laceration|fracture|burn|foreign
         ↪  body|poison(ing)?|overdose|od\b|adverse (drug|medication)
         ↪  reaction)\b",
        re.I,
    ),
    "RVC-SYM-RESP": re.compile(
        r"\b(shortness of breath|dyspnea|doe\b|wheeze|respiratory
         ↪  distress|cough(ing)?|hemoptysis|sputum)\b",
        re.I,
    ),
    "RVC-SYM-CIRC": re.compile(
        r"\b(chest pain|palpitation(s)?|hypotension|low
         ↪  bp|bradycardia|tachycardia|arrhythmia|irregular
         ↪  heartbeat|edema|leg swelling|ankle swelling|peripheral
         ↪  edema|claudication)\b",
        re.I,
    ),
    "RVC-SYM-NERV": re.compile(
        r"\b(alter(ed)? mental status|ams\b|confusion|letharg(y|ic)|
         ↪  unresponsive|syncope|faint(ing)?|seizure(s)?|slurred
         ↪  speech|dysarthria|aphasia|facial
         ↪  droop|headache|migraine|dizziness|vertigo|numbness|tingling)\b",
        re.I,
    ),
    "RVC-SYM-DIG": re.compile(
        r"\b(abd(ominal)? pain|abdo pain|nausea|vomit(ting)?|diarrhea|rectal
         ↪  bleeding|hematemesis|dysphagia|jaundice|gastrointestinal
         ↪  bleeding|gi bleeding)\b",
        re.I,
    ),
    "RVC-SYM-GU": re.compile(
```

```
    r"\b(dysuria|hematuria|urinary (frequency|urgency)|flank pain|pelvic
     ↪  pain|vaginal (bleeding|discharge)|penile discharge)\b",
    re.I,
),
"RVC-SYM-MSK": re.compile(
    r"\b(back pain|neck pain|shoulder pain|knee pain|hip pain|joint
     ↪  pain|gait problem)\b",
    re.I,
),
"RVC-SYM-SKIN":
 ↪  re.compile(r"\b(rash|pruritus|itch(ing)?|cellulitis|wound(?!
 ↪  check))\b", re.I),
"RVC-SYM-EYE": re.compile(
    r"\b(eye (pain|redness|irritation)|red eye|pink
     ↪  ?eye|conjunctivitis|blurry vision|vision (loss|change)|ear
     ↪  pain|hearing (loss|change)|tinnitus|ear discharge|vertigo)\b",
    re.I,
),
"RVC-SYM-GEN":
 ↪  re.compile(r"\b(fever|chills|fatigue|malaise|weakness(?!.*focal))\b",
 ↪  re.I),
"RVC-SYM-PSY": re.compile(
    r"\b(anxiety|depression|insomnia|agitation|suicidal
     ↪  ideation|homicidal ideation|si\b|hi\b|intoxication)\b",
    re.I,
),
"RVC-DIS": re.compile(
    r"\b(asthma attack|copd
     ↪  flare|pneumonia|diabetes(?!.*test)|hypertensive crisis|stroke
     ↪  diagnosed|stroke|intracranial hemorrhage|subarachnoid
     ↪  hemorrhage|st elevation myocardial infarction|non st elevation
     ↪  myocardial infarction|congestive heart failure|pulmonary
     ↪  embolism|small bowel obstruction|acute renal failure|influenza
     ↪  like illness|hypoglycemia|hypokalemia)\b",
    re.I,
),
"RVC-TEST": re.compile(
    r"\b(abnormal (lab|labs|imaging|ct|mri|x[- ]?ray|ekg|ecg)|told (to
     ↪  come|results?)|positive (test|culture|blood culture)|blood
     ↪  culture (positive|grew|growth))\b",
    re.I,
),
"RVC-DIAG": re.compile(
```

```
        r"\b((needs|for) (covid|flu|strep) test|screen(ing)?|bp
        ↪  (check|recheck)|workup|routine testing|prenatal)\b",
        re.I,
    ),
    "RVC-TREAT": re.compile(
        r"\b(refill|prescription refill|rx refill|dressing change|suture
        ↪  removal|injection|detox clearance|wound check)\b",
        re.I,
    ),
    "RVC-ADMIN":
    ↪  re.compile(r"\b(form|paperwork|note|letter|clearance|insurance|work
    ↪  (note|letter)|transfer(red)?|intubated transfer)\b", re.I),
}

GROUP_MAP: list[tuple[re.Pattern[str], str]] = [
    (re.compile(r"\binjur|poison|overdose|adverse effect", re.I), "RVC-INJ"),
    (re.compile(r"\brespir", re.I), "RVC-SYM-RESP"),
    (re.compile(r"\bcircul|cardio|lymph", re.I), "RVC-SYM-CIRC"),
    (re.compile(r"\bnerv|neuro", re.I), "RVC-SYM-NERV"),
    (re.compile(r"\bdigest|gastro|abd", re.I), "RVC-SYM-DIG"),
    (re.compile(r"\bgenitour|urinar|renal|pelvic|vagin|penil|breast", re.I),
↪   "RVC-SYM-GU"),
    (re.compile(r"\bmusculoskelet|msk|joint|back|neck", re.I),
↪   "RVC-SYM-MSK"),
    (re.compile(r"\bskin|hair|nail|derma", re.I), "RVC-SYM-SKIN"),
    (re.compile(r"\beye|ear|vision|hearing|vertigo", re.I), "RVC-SYM-EYE"),
    (re.compile(r"\bgeneral|fever|malaise|weakness", re.I), "RVC-SYM-GEN"),
    (re.compile(r"\bpsych|mental", re.I), "RVC-SYM-PSY"),
    (re.compile(r"\bdisease|diagnos|known|asthma|copd|diabetes|hypertens|
↪   pneumonia|stroke", re.I), "RVC-DIS"),
    (re.compile(r"\babnormal .*test|abnormal (lab|imaging|ekg|ecg)|positive
↪   (test|culture|result)", re.I), "RVC-TEST"),
    (re.compile(r"\bscreen|diagnostic|prenatal|immuniz|vaccine|bp
↪   check|test\b", re.I), "RVC-DIAG"),
    (re.compile(r"\btreat|medicat|refill|suture|dressing|injection|therapy|
↪   wound check", re.I), "RVC-TREAT"),
    (re.compile(r"\badmin|paperwork|form|clearance|insurance|work
↪   (note|letter)", re.I), "RVC-ADMIN"),
    (re.compile(r"\buncodable|unknown|illegible|none", re.I), "RVC-UNCL"),
]

RE_DROP = re.compile(r"\b(nec|nos|unspecified|other|other and
↪   unspecified)\b", re.I)
```

```
RE_SPACE = re.compile(r"\s+")
BODY_PART_ONLY_RE = re.compile(r"(arm|leg|back|knee|hip|wrist|elbow|ankle|
 ↪  hand|foot|toe|finger|eye|ear)")

CANON_REPLACERS: list[tuple[re.Pattern[str], str]] = [
    (re.compile(r"\bshortness of breath\b", flags=re.I), "dyspnea"),
    (re.compile(r"\bdyspneic?\b", flags=re.I), "dyspnea"),
    (re.compile(r"\bdyspnea on exertion\b", flags=re.I), "dyspnea_exertion"),
    (re.compile(r"\bstatus post\b", flags=re.I), "status_post"),
]




RVC_PROTOS_CURATED: dict[str, list[str]] = {
    "RVC-INJ": [
        "injury: laceration cut",
        "injury: fracture",
        "injury: burn",
        "injury: contusion bruise",
        "injury: foreign body",
        "injury: motor vehicle collision",
        "injury: fall",
        "adverse effect: medication reaction",
        "poisoning overdose",
        "injury: assault",
        "gunshot wound",
    ],
    "RVC-SYM-RESP": [
        "respiratory symptom: shortness of breath dyspnea",
        "respiratory symptom: dyspnea on exertion",
        "respiratory symptom: wheeze",
        "respiratory symptom: cough",
        "respiratory symptom: hemoptysis",
        "respiratory symptom: sputum production",
        "respiratory symptom: respiratory distress",
    ],
    "RVC-SYM-CIRC": [
        "circulatory symptom: chest pain suspected cardiac",
        "circulatory symptom: palpitations",
        "circulatory symptom: hypotension low blood pressure",
        "circulatory symptom: bradycardia",
        "circulatory symptom: tachycardia",
```

```
    "circulatory symptom: peripheral edema leg swelling ankle swelling",
    "circulatory symptom: claudication",
],
"RVC-SYM-NERV": [
    "neurologic symptom: altered mental status confusion",
    "neurologic symptom: lethargy decreased responsiveness",
    "neurologic symptom: slurred speech dysarthria",
    "neurologic symptom: aphasia word finding difficulty",
    "neurologic symptom: facial droop",
    "neurologic symptom: syncope fainting",
    "neurologic symptom: seizure",
    "neurologic symptom: headache",
    "neurologic symptom: dizziness vertigo",
    "neurologic symptom: numbness tingling focal weakness",
],
"RVC-SYM-DIG": [
    "digestive symptom: abdominal pain",
    "digestive symptom: nausea vomiting",
    "digestive symptom: diarrhea",
    "digestive symptom: rectal bleeding",
    "digestive symptom: dysphagia",
    "digestive symptom: jaundice",
],
"RVC-SYM-GU": [
    "genitourinary symptom: dysuria painful urination",
    "genitourinary symptom: urinary frequency urgency",
    "genitourinary symptom: hematuria blood in urine",
    "genitourinary symptom: flank pain",
    "genitourinary symptom: pelvic pain",
    "genitourinary symptom: vaginal discharge",
],
"RVC-SYM-MSK": [
    "musculoskeletal symptom: back pain",
    "musculoskeletal symptom: neck pain",
    "musculoskeletal symptom: joint pain",
    "musculoskeletal symptom: hip knee shoulder pain",
    "musculoskeletal symptom: gait problem",
],
"RVC-SYM-SKIN": [
    "skin symptom: rash",
    "skin symptom: pruritus itching",
    "skin symptom: cellulitis redness warmth",
    "skin symptom: nontraumatic wound",
```

```
],
"RVC-SYM-EYE": [
    "eye symptom: eye pain",
    "eye symptom: red eye eye redness",
    "eye symptom: blurry vision vision loss",
    "ear symptom: ear pain discharge",
    "vestibular symptom: vertigo spinning",
],
"RVC-SYM-GEN": [
    "general symptom: fever chills",
    "general symptom: fatigue malaise",
    "general symptom: weakness generalized",
    "general symptom: weight change",
    "general symptom: edema swelling",
],
"RVC-SYM-PSY": [
    "psychological symptom: anxiety",
    "psychological symptom: depression",
    "psychological symptom: agitation",
    "psychological symptom: insomnia",
    "psychological symptom: suicidal ideation",
    "psychological symptom: homicidal ideation",
    "psychological symptom: substance intoxication without overdose",
],
"RVC-DIS": [
    "patient stated diagnosis: asthma attack",
    "patient stated diagnosis: copd flare",
    "patient stated diagnosis: pneumonia",
    "patient stated diagnosis: diabetes problem",
    "patient stated diagnosis: hypertensive crisis",
    "patient stated diagnosis: stroke diagnosed",
],
"RVC-TEST": [
    "abnormal test: abnormal laboratory result",
    "abnormal test: abnormal imaging result",
    "abnormal test: positive culture",
    "abnormal test: abnormal ecg ekg",
    "abnormal test: positive blood culture",
],
"RVC-DIAG": [
    "diagnostic screening: needs covid test",
    "diagnostic screening: blood pressure check",
    "diagnostic screening: routine testing screening",
```

```
        "diagnostic screening: prenatal check",
    ],
    "RVC-TREAT": [
        "treatment medication: prescription refill",
        "treatment medication: dressing change",
        "treatment medication: injection therapy",
        "treatment medication: suture removal",
        "treatment: wound check",
    ],
    "RVC-ADMIN": [
        "administrative reason: work form school form insurance paperwork",
        "administrative reason: clearance note",
    ],
    "RVC-UNCL": ["uncodable or unknown reason"],
}

for _group in RVC_NAME:
    RVC_PROTOS_CURATED.setdefault(_group, [])

LABEL2CODE = {
    re.sub(r"\s+", " ", value.strip().lower().replace("-", "-").replace("-",
 ↪  "-")): code
    for code, value in RVC_NAME.items()
}

sym = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)


# --- Lightweight data containers passed between pipeline stages. ---
@dataclass
class SegmentEmbedCache:
    """Embeddings and similarity cache for unique complaint segments."""

    uniq_segments: list[str]
    seg2idx: dict[str, int]
    emb: torch.Tensor
    sim_proto: torch.Tensor


@dataclass
class SegPred:
    """Per-segment prediction record."""
```

```
    seg_idx: int
    segment: str
    code: str
    name: str
    sim: float
    rule_code: str | None
    rule_used: bool


def _add_lexicon_from_phrases(phrases: list[str]) -> None:
    for phrase in phrases:
        for token in str(phrase).split():
            if token.isalpha():
                sym.create_dictionary_entry(token, 1)


_add_lexicon_from_phrases(list(SEG_OVR.values()) + list(TOK_OVR.values()) +
 ↪  list(ABBR_MAP.values()))


# --- Core text preprocessing helpers. ---
def get_cc_column(df: pd.DataFrame) -> str:
    """Return the first recognized chief-complaint column."""
    for column in CC_CANDIDATES:
        if column in df.columns:
            return column
    raise KeyError(f"No chief-complaint column found in candidates:
     ↪  {CC_CANDIDATES}")


def _clean_tok(token: str) -> str:
    return _PUNCT_EDGE.sub("", token)


def _spell(token: str) -> str:
    if len(token) <= 3 and token not in SAFE_SHORT_TOKENS:
        return token
    if not token.isalpha():
        return token
    candidates = sym.lookup(token, Verbosity.CLOSEST, max_edit_distance=2)
    return candidates[0].term if candidates else token
```

```
def _expand_tokens(segment: str) -> str:
    """Apply contextual and curated abbreviation expansions in one
     ↪ segment."""
    output: list[str] = []
    segment_context = segment
    for token in segment.split():
        cleaned = _clean_tok(token)
        if not cleaned:
            continue
        if cleaned in CTX_OVR:
            predicate, yes_value, no_value = CTX_OVR[cleaned]
            if predicate(segment_context):
                output.extend(yes_value.split())
                continue
            if no_value:
                output.extend(no_value.split())
                continue
        if cleaned in TOK_OVR:
            output.extend(TOK_OVR[cleaned].split())
            continue
        if cleaned in ABBR_MAP:
            output.extend(ABBR_MAP[cleaned].split())
            continue
        output.append(cleaned)
    return " ".join(output)


def _looks_clinical(text: str) -> bool:
    lowered = text.lower()
    return any(key in lowered for key in CLINICAL_KEYS) and
     ↪ bool(re.search(r"[a-z]", lowered))


def _split_on_and_if_clinical(segment: str) -> list[str]:
    parts = [part.strip() for part in re.split(r"\band\b", segment,
 ↪ flags=re.I)]
    if len(parts) == 1:
        return parts
    output: list[str] = []
    buffer = parts[0]
    for next_part in parts[1:]:
        if _looks_clinical(buffer) and _looks_clinical(next_part):
            output.append(buffer.strip())
```

```
            buffer = next_part
        else:
            buffer = f"{buffer} and {next_part}"
    output.append(buffer.strip())
    return output


def _normalize_absence_symptom(segment: str) -> str | None:
    """Convert selected absence-style phrases into symptom phrases we can
    ↪ classify."""
    cleaned = segment.strip()
    if not cleaned:
        return None

    for regex, replacement in ABSENCE_SYMPTOM_RULES:
        if regex.search(cleaned):
            return regex.sub(replacement, cleaned).strip()

    return None


def _expand_duration(segment: str) -> str:
    def repl(match: re.Match[str]) -> str:
        base = match.group(1)
        count = match.group(2)
        unit = match.group(3).lower()
        resolved_unit = _UNIT_MAP.get(unit, unit)
        return f"{base} for {count} {resolved_unit}".strip()

    return _DUR_RE.sub(repl, segment)


def _lemmatize_segment(segment: str) -> str:
    doc = NLP(segment)
    lemmas: list[str] = []
    for token in doc:
        lemma = token.lemma_.strip() if token.lemma_ else ""
        if not lemma or lemma == "-PRON-":
            lemma = token.text
        lemmas.append(lemma)
    return " ".join(lemmas).strip()
```

```
def segment_cc(raw: str, keep_negated: bool = False) -> list[str]:
    """Split a raw chief complaint into normalized text segments."""
    if not isinstance(raw, str) or not raw.strip():
        return []

    text = ftfy.fix_text(raw).lower().strip()

    for protected in PROTECT:
        text = text.replace(protected, protected.replace("/", "§"))

    text = re.sub(r"\bn\s*[,/&+]\s*v\s*[,/&+]\s*d\b", "n§v§d", text)
    text = re.sub(r"\bn\s*[,/&+]\s*v\b", "n§v", text)
    text = re.sub(r"\bn\s*/\s*v\s*/\s*d\b", "n§v§d", text)
    text = re.sub(r"\bn\s*/\s*v\b", "n§v", text)
    text = re.sub(r"\bc\s*/\s*p\b", "c§p", text)
    text = re.sub(r"\bs\s*/\s*p\b", "s§p", text)
    text = re.sub(r"\bw\s*/\s*o\b", "w§o", text)
    text = re.sub(r"\bh\s*/\s*o\b", "h§o", text)
    text = re.sub(r"\bc\s*/\s*o\b", "c§o", text)

    text = text.replace("/", "; ")

    segments: list[str] = []
    for block in SEP_SPLIT_RE.split(text):
        if not block:
            continue
        block = block.replace("§", "/").strip()
        block = " ".join(block.split())
        if MISSING_RE.fullmatch(block):
            continue
        for segment in _split_on_and_if_clinical(block):
            if not segment:
                continue

            normalized_absence = _normalize_absence_symptom(segment)
            if normalized_absence is not None:
                segment = normalized_absence
            elif not keep_negated and NEG_RE.match(segment):
                continue

            if re.fullmatch(r"[a-z]$", segment):
                continue
            segments.append(segment)
```

```
    return segments


def normalize_cc(raw: str, keep_negated: bool = False) -> list[str]:
    """Normalize a raw chief complaint to deduplicated, lemmatized
     ↪ segments."""
    output: list[str] = []
    seen: set[str] = set()

    for segment in segment_cc(raw, keep_negated=keep_negated):
        if segment in SEG_OVR:
            segment = SEG_OVR[segment]
        else:
            tokens = segment.split()
            if len(tokens) == 1:
                token = _clean_tok(tokens[0])
                if token in TOK_OVR:
                    segment = TOK_OVR[token]
                else:
                    segment = ABBR_MAP.get(segment, segment)

        segment = _expand_duration(segment)
        segment = _expand_tokens(segment)
        corrected_tokens = [_spell(token) for token in segment.split()]
        segment = _lemmatize_segment(" ".join(corrected_tokens))

        if segment and segment not in seen:
            output.append(segment)
            seen.add(segment)

    return output


def canonize_cc_segments(segments: list[str]) -> list[str]:
    """Apply light concept-level canonicalization for
     ↪ diagnostics/visualization."""
    output: list[str] = []
    for segment in segments:
        transformed = segment
        for regex, replacement in CANON_REPLACERS:
            transformed = regex.sub(replacement, transformed)
        output.append(transformed)
    return output
```

```
def rule_gate(segment: str) -> tuple[str | None, list[str]]:
    """Apply deterministic rule overrides for high-confidence pattern
    ↪ matches."""
    text = f" {segment.lower()} "
    if "weakness" in text and RX_FOCAL_WEAK.search(text):
        return "RVC-SYM-NERV", ["focal weakness"]

    hits: list[str] = []
    for code, regex in RULE_RX.items():
        if regex.search(text):
            hits.append(code)

    if not hits:
        return None, []

    for preferred_code in PRECEDENCE_RVC:
        if preferred_code in hits:
            return preferred_code, [preferred_code]

    return hits[0], [hits[0]]


# --- Ontology file parsing helpers (for Appendix and summary resources). ---
def _guess_col(columns: list[str], candidates: list[str]) -> str | None:
    lowered = [column.lower().strip() for column in columns]
    for candidate in candidates:
        if candidate in lowered:
            return columns[lowered.index(candidate)]
    return None


def _norm_label(label: str) -> str:
    if label is None:
        return ""
    cleaned = str(label).strip().lower().replace("-", "-").replace("-", "-")
    return re.sub(r"\s+", " ", cleaned)


def map_group(label: str) -> str | None:
```

```
    for regex, code in GROUP_MAP:
        if regex.search(str(label)):
            return code
    return None


def _label_to_code(label: str) -> str | None:
    normalized = _norm_label(label)
    if normalized in LABEL2CODE:
        return LABEL2CODE[normalized]
    return map_group(label)


def canon_phrase(text: str) -> str:
    cleaned = str(text).strip().replace("’", "'").replace("–", "-")
    cleaned = RE_SPACE.sub(" ", cleaned)
    return cleaned.strip(" .;:,")


def keep_phrase(text: str, group_code: str) -> bool:
    lowered = text.lower()
    if len(lowered) < 3:
        return False
    if RE_DROP.search(lowered):
        return False
    if group_code not in {"RVC-INJ", "RVC-SYM-MSK", "RVC-SYM-EYE",
     ↪  "RVC-SYM-SKIN"}:
        if BODY_PART_ONLY_RE.fullmatch(lowered):
            return False
    return True


def load_appendix_phrases(path: Path) -> pd.DataFrame:
    """Load and normalize Appendix II prototype phrases by RVC code."""
    if not path.exists():
        return pd.DataFrame(columns=["group_code", "phrase"])

    appendix_df = pd.read_csv(path, dtype=str)

    if {"top_level_group17", "text"}.issubset(appendix_df.columns):
        group_col = "top_level_group17"
        phrase_col = "text"
    else:
```

```
        group_col = _guess_col(
            list(appendix_df.columns),
            ["top_level_group17", "group", "top_level_group", "rvc_group",
↪   "module"],
        )
        phrase_col = _guess_col(
            list(appendix_df.columns),
            ["text", "phrase", "subentry", "entry", "label"],
        )
        if group_col is None or phrase_col is None:
            raise ValueError(f"Unable to identify expected columns in
             ↪  appendix file: {path}")

    normalized = appendix_df[[group_col, phrase_col]].rename(
        columns={group_col: "group_raw", phrase_col: "phrase_raw"}
    )
    normalized["group_code"] = normalized["group_raw"].map(_label_to_code)
    normalized = normalized[normalized["group_code"].notna()].copy()
    normalized["phrase"] = normalized["phrase_raw"].map(canon_phrase)
    normalized = normalized[
        normalized.apply(lambda row: keep_phrase(row["phrase"],
↪   row["group_code"]), axis=1)
    ]
    return normalized[["group_code", "phrase"]].drop_duplicates()


def load_summary_weights(path: Path, strategy: str = "uniform") -> dict[str,
↪   float]:
    """Load optional group weights and map them to canonical RVC codes."""
    if not path.exists():
        return {}

    summary_df = pd.read_csv(path, dtype=str)
    group_col = _guess_col(
        list(summary_df.columns),
        ["top_level_group17", "group", "top_level_group", "rvc_group",
↪   "module"],
    )
    if group_col is None:
        group_col = summary_df.columns[0]

    mapped_codes = summary_df[group_col].map(_label_to_code).dropna()
    if mapped_codes.empty:
```

```
        return {}

    counts = mapped_codes.value_counts().astype(float)
    if strategy == "uniform":
        counts[:] = 1.0
    elif strategy != "by_rows":
        raise ValueError("strategy must be one of {'uniform', 'by_rows'}")

    scaled = np.log1p(counts) / np.log1p(counts).max()
    return {code: float(max(1e-3, scaled.get(code, 1.0))) for code in
     ↪  PRECEDENCE_RVC}


def dedup_prototypes(
    rvc_prototypes: dict[str, list[str]],
    encoder: SentenceTransformer,
    sim_thresh: float = 0.92,
    batch_size: int = 256,
) -> dict[str, list[str]]:
    """Deduplicate near-identical prototype phrases within each group."""
    output: dict[str, list[str]] = {}

    for group_code, phrases in rvc_prototypes.items():
        normalized = sorted(dict.fromkeys(canon_phrase(phrase) for phrase in
 ↪  phrases))
        if not normalized:
            output[group_code] = []
            continue

        with torch.inference_mode():
            embedding = encoder.encode(
                normalized,
                convert_to_tensor=True,
                normalize_embeddings=True,
                batch_size=batch_size,
            )

        keep: list[str] = []
        taken = torch.zeros(len(normalized), dtype=torch.bool)
        for idx in range(len(normalized)):
            if taken[idx]:
                continue
            keep.append(normalized[idx])
```

```
            if len(normalized) == 1:
                break
            similarities = util.cos_sim(embedding[idx : idx + 1],
↪   embedding).squeeze(0).cpu().numpy()
            duplicate_indices = np.where(similarities >= sim_thresh)[0]
            taken[duplicate_indices] = True

        output[group_code] = keep

    return output


# --- Embedding/prototype resource construction. ---
def build_prototype_resources(cfg: ClassifierConfig) -> dict[str, Any]:
    """Build model, prototype embeddings, scoring indices, and metadata."""
    device = "cuda" if torch.cuda.is_available() else "cpu"
    hf_token = os.getenv(cfg.hf_token_env_var, "").strip() or None
    encoder = SentenceTransformer(
        cfg.model_name,
        device=device,
        trust_remote_code=cfg.model_trust_remote_code,
        token=hf_token,
    )

    work_dir = Path(os.getenv(cfg.work_dir_env_var,
↪   Path.cwd())).expanduser().resolve()
    appendix_path = work_dir / cfg.appendix_relpath
    summary_path = work_dir / cfg.summary_relpath

    appendix_df = load_appendix_phrases(appendix_path)
    group_weights = load_summary_weights(summary_path,
↪   strategy=cfg.summary_weight_strategy)

    prototypes_augmented = {code: list(values) for code, values in
↪   RVC_PROTOS_CURATED.items()}
    for group_code, group_df in appendix_df.groupby("group_code"):
        prototypes_augmented[group_code].extend(group_df["phrase"].tolist())

    prototypes_final = dedup_prototypes(
        prototypes_augmented,
        encoder=encoder,
        sim_thresh=cfg.prototype_dedup_sim_thresh,
        batch_size=cfg.model_batch_size,
```

```
    )

    all_prototype_texts: list[str] = []
    proto_to_code: list[str] = []
    for code, prototype_list in prototypes_final.items():
        prefix = RVC_NAME[code].split("-")[0].strip().lower()
        for phrase in prototype_list:
            text = phrase if ":" in phrase else f"{prefix}: {phrase}"
            all_prototype_texts.append(text)
            proto_to_code.append(code)

    with torch.inference_mode():
        proto_emb = encoder.encode(
            all_prototype_texts,
            convert_to_tensor=True,
            normalize_embeddings=True,
            batch_size=cfg.model_batch_size,
        )

    code_to_proto_indices: dict[str, list[int]] = defaultdict(list)
    for idx, code in enumerate(proto_to_code):
        code_to_proto_indices[code].append(idx)

    proto_weight = np.ones(len(proto_to_code), dtype="float32")
    if group_weights:
        for idx, code in enumerate(proto_to_code):
            proto_weight[idx] = max(1e-3, float(group_weights.get(code,
↪   1.0)))

    metadata = {
        "model_name": cfg.model_name,
        "device": device,
        "prototype_total": len(all_prototype_texts),
        "prototype_count_by_group": {code:
        ↪   len(code_to_proto_indices.get(code, [])) for code in
        ↪   PRECEDENCE_RVC},
        "appendix_rows_used": int(len(appendix_df)),
        "spacy_model": SPACY_MODEL_LABEL,
        "hf_token_present": bool(hf_token),
    }

    return {
        "encoder": encoder,
```

```
        "device": device,
        "proto_emb": proto_emb,
        "proto_to_code": proto_to_code,
        "code_to_proto_indices": code_to_proto_indices,
        "proto_weight": proto_weight,
        "metadata": metadata,
    }


# --- Segment caching and scoring helpers. ---
def build_segment_cache(
    df: pd.DataFrame,
    seg_col: str,
    resources: dict[str, Any],
    batch_size: int = 512,
) -> SegmentEmbedCache:
    """Encode unique complaint segments once and cache similarity to all
     ↪ prototypes."""
    unique_segments: list[str] = []
    seen: set[str] = set()

    for value in df[seg_col]:
        if isinstance(value, list):
            iterable = value
        elif isinstance(value, str):
            iterable = [value]
        else:
            iterable = []

        for segment in iterable:
            cleaned = str(segment).strip()
            if cleaned and cleaned not in seen:
                unique_segments.append(cleaned)
                seen.add(cleaned)

    unique_segments.sort()

    if not unique_segments:
        return SegmentEmbedCache(
            uniq_segments=[],
            seg2idx={},
            emb=torch.empty(0, 0),
            sim_proto=torch.empty(0, 0),
```

```
        )

    encoder: SentenceTransformer = resources["encoder"]
    proto_emb: torch.Tensor = resources["proto_emb"]

    with torch.inference_mode():
        segment_emb = encoder.encode(
            unique_segments,
            convert_to_tensor=True,
            normalize_embeddings=True,
            batch_size=batch_size,
            device=resources["device"],
        )
        segment_emb = torch.nn.functional.normalize(segment_emb, p=2, dim=1)
        segment_to_proto = segment_emb @ proto_emb.T

    seg2idx = {segment: idx for idx, segment in enumerate(unique_segments)}
    return SegmentEmbedCache(
        uniq_segments=unique_segments,
        seg2idx=seg2idx,
        emb=segment_emb,
        sim_proto=segment_to_proto,
    )


def group_scores_from_proto_row(
    proto_row: torch.Tensor | np.ndarray,
    resources: dict[str, Any],
    method: str = "max",
    alpha: float = 12.0,
) -> dict[str, float]:
    """Aggregate prototype-level similarities to group-level scores."""
    vector = proto_row.detach().cpu().numpy() if isinstance(proto_row,
    ↪  torch.Tensor) else np.asarray(proto_row)

    scores: dict[str, float] = {}
    code_to_proto = resources["code_to_proto_indices"]
    proto_weight = resources["proto_weight"]

    for group_code, indices in code_to_proto.items():
        values = vector[indices]
        if method == "max":
            score = float(values.max(initial=-1.0))
```

```
        elif method == "lse":
            weights = proto_weight[indices]
            logits = alpha * values + np.log(weights)
            max_logit = logits.max()
            score = float(max_logit + np.log(np.exp(logits -
↪   max_logit).sum()))
        else:
            raise ValueError("method must be one of {'max', 'lse'}")
        scores[group_code] = score

    if method == "lse" and scores:
        max_score = max(scores.values())
        min_score = min(scores.values())
        denom = (max_score - min_score) if max_score > min_score else 1.0
        scores = {group: (score - min_score) / denom for group, score in
↪   scores.items()}

    return scores


def score_one_segment_soft(
    segment: str,
    resources: dict[str, Any],
    method: str = "max",
    alpha: float = 12.0,
) -> dict[str, float]:
    """Score a single segment directly against prototype embeddings."""
    if not isinstance(segment, str) or not segment.strip():
        return {group: -1.0 for group in RVC_NAME}

    encoder: SentenceTransformer = resources["encoder"]
    proto_emb: torch.Tensor = resources["proto_emb"]

    with torch.inference_mode():
        segment_emb = encoder.encode(
            [segment],
            convert_to_tensor=True,
            normalize_embeddings=True,
            device=resources["device"],
        )
        similarities = util.cos_sim(segment_emb, proto_emb).squeeze(0)
```

```
    return group_scores_from_proto_row(similarities, resources=resources,
     ↪  method=method, alpha=alpha)


def classify_segments_cached(
    segments: list[str],
    cache: SegmentEmbedCache,
    resources: dict[str, Any],
    abstain: float = 0.40,
    method: str = "max",
    alpha: float = 12.0,
) -> list[SegPred]:
    """Classify all segments for one visit using cache + rule-gate
     ↪  overrides."""
    predictions: list[SegPred] = []

    for seg_idx, segment in enumerate(segments):
        cleaned = str(segment).strip()
        if not cleaned:
            continue

        override_code, _ = rule_gate(cleaned)

        if cleaned in cache.seg2idx:
            row_idx = cache.seg2idx[cleaned]
            score_map = group_scores_from_proto_row(
                cache.sim_proto[row_idx], resources=resources, method=method,
 ↪  alpha=alpha
            )
        else:
            score_map = score_one_segment_soft(
                cleaned, resources=resources, method=method, alpha=alpha
            )

        best_code, best_score = max(score_map.items(), key=lambda kv: kv[1])

        if override_code is not None:
            best_code = override_code

        if best_score < abstain and override_code is None:
            best_code = "RVC-UNCL"

        predictions.append(
```

```
            SegPred(
                seg_idx=seg_idx,
                segment=cleaned,
                code=best_code,
                name=RVC_NAME.get(best_code, best_code),
                sim=float(best_score),
                rule_code=override_code,
                rule_used=override_code is not None,
            )
        )

    return predictions


def rfv_from_segment_preds(segpreds: list[SegPred], max_rfv: int = 5) ->
 ↪  list[dict[str, Any]]:
    """Aggregate segment-level predictions to visit-level RFV slots."""
    first_position: dict[str, int] = {}
    best_similarity: dict[str, float] = {}
    support_segment: dict[str, str] = {}

    for prediction in segpreds:
        group_code = prediction.code
        if group_code == "RVC-UNCL":
            continue

        if group_code not in first_position or prediction.seg_idx <
         ↪  first_position[group_code]:
            first_position[group_code] = prediction.seg_idx
            support_segment[group_code] = prediction.segment

        best_similarity[group_code] = max(best_similarity.get(group_code,
 ↪  -1.0), prediction.sim)

    ordered = sorted(
        first_position.keys(),
        key=lambda code: (PRECEDENCE_RVC.index(code), first_position[code]),
    )
    ordered = ordered[:max_rfv]

    return [
        {
            "code": code,
```

```
            "name": RVC_NAME[code],
            "support": support_segment[code],
            "sim": best_similarity[code],
        }
        for code in ordered
    ]


# --- Visit-level RFV assignment and output shaping. ---
def assign_rvc_per_segment_and_visit_cached(
    df: pd.DataFrame,
    cache: SegmentEmbedCache,
    resources: dict[str, Any],
    seg_col: str = "cc_cleaned",
    cfg: ClassifierConfig | None = None,
) -> pd.DataFrame:
    """Attach standardized RFV outputs to each visit using cached segment
     ↪ embeddings."""
    active_cfg = cfg or ClassifierConfig()
    max_rfv = active_cfg.max_rfv
    if active_cfg.empty_primary_policy not in {"blank", "uncodable"}:
        raise ValueError("empty_primary_policy must be one of {'blank',
         ↪ 'uncodable'}")

    rows: list[dict[str, Any]] = []
    for value in df[seg_col].tolist():
        if isinstance(value, list):
            segments = [str(segment).strip() for segment in value if
 ↪ str(segment).strip()]
        elif isinstance(value, str) and value.strip():
            segments = [value.strip()]
        else:
            segments = []

        segments = segments[: active_cfg.max_segments_per_visit]
        segment_predictions = classify_segments_cached(
            segments,
            cache=cache,
            resources=resources,
            abstain=active_cfg.abstain_threshold,
            method=active_cfg.scoring_method,
            alpha=active_cfg.alpha,
        )
```

```
        visit_rfv = rfv_from_segment_preds(segment_predictions,
↪  max_rfv=max_rfv)

        # If all non-empty segment predictions are uncodable, explicitly
         ↪  carry that into RFV1.
        if (
            not visit_rfv
            and active_cfg.allow_uncodable_primary
            and segments
            and segment_predictions
        ):
            uncodable_predictions = [pred for pred in segment_predictions if
↪  pred.code == "RVC-UNCL"]
            if uncodable_predictions:
                chosen = max(uncodable_predictions, key=lambda pred:
↪  (pred.sim, -pred.seg_idx))
                visit_rfv = [
                    {
                        "code": "RVC-UNCL",
                        "name": RVC_NAME["RVC-UNCL"],
                        "support": chosen.segment,
                        "sim": float(chosen.sim),
                    }
                ]

        # Optional policy for truly empty complaints.
        if (
            not visit_rfv
            and not segments
            and active_cfg.allow_uncodable_primary
            and active_cfg.empty_primary_policy == "uncodable"
        ):
            visit_rfv = [
                {
                    "code": "RVC-UNCL",
                    "name": RVC_NAME["RVC-UNCL"],
                    "support": "",
                    "sim": math.nan,
                }
            ]

        row: dict[str, Any] = {
```

```
            "segment_preds": json.dumps([prediction.__dict__ for prediction
             ↪  in segment_predictions], ensure_ascii=False)
        }

        for idx in range(max_rfv):
            if idx < len(visit_rfv):
                code = visit_rfv[idx]["code"]
                name = visit_rfv[idx]["name"]
                support = visit_rfv[idx]["support"]
                sim = float(visit_rfv[idx]["sim"])
            else:
                code = ""
                name = ""
                support = ""
                sim = math.nan

            slot = idx + 1
            row[f"RFV{slot}"] = RVC_SHORT.get(code, "")
            row[f"RFV{slot}_name"] = name
            row[f"RFV{slot}_support"] = support
            row[f"RFV{slot}_sim"] = sim

        rows.append(row)

    drop_columns = [column for column in df.columns if
 ↪  str(column).startswith("RFV") or column == "segment_preds"]
    output_df = df.drop(columns=drop_columns, errors="ignore").copy()
    output_df = pd.concat([output_df, pd.DataFrame(rows,
 ↪  index=output_df.index)], axis=1)
    return output_df


# --- Output QA and diagnostic helpers used later in the notebook. ---
def validate_output_schema(
    df: pd.DataFrame,
    max_rfv: int = 5,
    require_primary_for_nonempty: bool = True,
) -> None:
    """Validate required output columns, uniqueness, and JSON shape for
     ↪  segment predictions."""
    required = ["segment_preds"]
    for slot in range(1, max_rfv + 1):
        required.extend(
```

```
        [
            f"RFV{slot}",
            f"RFV{slot}_name",
            f"RFV{slot}_support",
            f"RFV{slot}_sim",
        ]
    )

duplicates = df.columns[df.columns.duplicated()].tolist()
if duplicates:
    raise ValueError(f"Duplicate columns detected: {duplicates}")

missing = [column for column in required if column not in df.columns]
if missing:
    raise ValueError(f"Missing required output columns: {missing}")

bad_json_rows = 0
parsed_payloads: list[list[dict[str, Any]] | None] = []
for value in df["segment_preds"].fillna("[]"):
    if not isinstance(value, str):
        bad_json_rows += 1
        parsed_payloads.append(None)
        continue
    try:
        parsed = json.loads(value)
    except json.JSONDecodeError:
        bad_json_rows += 1
        parsed_payloads.append(None)
        continue
    if not isinstance(parsed, list):
        bad_json_rows += 1
        parsed_payloads.append(None)
        continue
    parsed_payloads.append(parsed)

if bad_json_rows:
    raise ValueError(f"Found {bad_json_rows} malformed segment_preds
    ↪ rows")

if require_primary_for_nonempty:
    blank_primary_violations = 0
    for primary_name, payload in zip(df["RFV1_name"], parsed_payloads):
```

```
            primary_blank = pd.isna(primary_name) or not
↪   str(primary_name).strip()
            has_segments = isinstance(payload, list) and len(payload) > 0
            if primary_blank and has_segments:
                blank_primary_violations += 1

        if blank_primary_violations:
            raise ValueError(
                "Found blank RFV1_name for rows with non-empty segment
                ↪  predictions: "
                f"{blank_primary_violations}"
            )

    print(f"Schema validation passed for {len(df):,} rows.")


def save_with_suffix(
    df: pd.DataFrame,
    working_path: str | Path,
    suffix: str = "_with_NLP",
    sheet_name: str = "with_NLP",
) -> Path:
    """Write workbook with suffix and return the new path."""
    path = Path(working_path)
    output_path = path.with_name(path.stem + suffix + path.suffix)
    df.to_excel(output_path, index=False, sheet_name=sheet_name)
    print(f"Wrote: {output_path}")
    return output_path


def summarize_overrides(df: pd.DataFrame, column: str = "segment_preds") ->
↪   dict[str, Any]:
    """Summarize rule-override usage from serialized segment prediction
    ↪  payloads."""
    if column not in df.columns:
        raise KeyError(f"Missing column: {column}")

    segment_total = 0
    segment_overridden = 0
    by_rule: Counter = Counter()
    by_final: Counter = Counter()
    rows_total = 0
    rows_with_override = 0
```

```
for value in df[column].dropna():
    rows_total += 1
    payload: Any
    if isinstance(value, str):
        try:
            payload = json.loads(value)
        except Exception:
            continue
    elif isinstance(value, list):
        payload = value
    else:
        continue

    if not isinstance(payload, list):
        continue

    row_has_override = False
    for segment_prediction in payload:
        if not isinstance(segment_prediction, dict):
            continue
        segment_total += 1
        if segment_prediction.get("rule_used"):
            segment_overridden += 1
            row_has_override = True
            by_rule[segment_prediction.get("rule_code", "UNKNOWN")] += 1
            by_final[segment_prediction.get("code", "UNKNOWN")] += 1

    if row_has_override:
        rows_with_override += 1

return {
    "segments_total": segment_total,
    "segments_overridden": segment_overridden,
    "segment_override_rate": (segment_overridden / segment_total) if
     ↪  segment_total else float("nan"),
    "rows_total": rows_total,
    "rows_with_override": rows_with_override,
    "row_override_rate": (rows_with_override / rows_total) if rows_total
     ↪  else float("nan"),
    "by_rule": by_rule,
    "by_final": by_final,
}
```

```
def validate_pipeline(
    df: pd.DataFrame,
    cache: SegmentEmbedCache,
    resources: dict[str, Any],
    seg_col: str = "cc_cleaned",
) -> dict[str, Any]:
    """Run smoke checks for prototype health, segment coverage, and
     ↪  rule/classifier sanity."""
    metadata = resources["metadata"]
    prototype_counts = metadata["prototype_count_by_group"]
    missing_groups = [group for group, count in prototype_counts.items() if
 ↪  count == 0]
    if missing_groups:
        raise ValueError(f"Groups missing prototypes: {missing_groups}")

    row_count = len(df)
    nonempty_segment_rows = int(df[seg_col].apply(lambda value:
 ↪  isinstance(value, list) and len(value) > 0).sum())
    unique_segments = len(cache.uniq_segments)

    probes = [
        "shortness of breath",
        "bradycardia",
        "positive blood culture",
        "alter mental status",
        "abdominal pain / shortness of breath",
        "right eye redness / pain",
    ]
    probe_results: dict[str, list[tuple[str, str, float, bool]]] = {}
    for probe in probes:
        segments = segment_cc(probe)
        predictions = classify_segments_cached(
            segments,
            cache=cache,
            resources=resources,
            method="max",
        )
        probe_results[probe] = [
            (prediction.segment, prediction.code, prediction.sim,
 ↪  prediction.rule_used)
            for prediction in predictions
```

```
        ]

    print("Prototype integrity:")
    print(prototype_counts)
    print(f"Total prototypes: {metadata['prototype_total']}")
    print()
    print("Segment column health:")
    print(f"rows={row_count:,} | with>=1 segment={nonempty_segment_rows:,} |
     ↪  unique segments={unique_segments:,}")
    print()
    print("Probe classifications:")
    for probe, result in probe_results.items():
        print(f"{probe} -> {result}")

    return {
        "row_count": row_count,
        "rows_with_segments": nonempty_segment_rows,
        "unique_segments": unique_segments,
        "prototype_counts": prototype_counts,
        "probe_results": probe_results,
    }


def top_cc_plot(
    df: pd.DataFrame,
    candidates: tuple[str, ...] = CC_CANDIDATES,
    top_n: int = 30,
    normalize_case: str | None = "upper",
) -> pd.DataFrame:
    """Optional EDA: plot top raw chief complaints."""
    cc_column = get_cc_column(df)
    series = (
        df[cc_column]
        .astype(str)
        .str.strip()
        .str.replace(r"\s+", " ", regex=True)
    )

    if normalize_case == "upper":
        series = series.str.upper()
    elif normalize_case == "title":
        series = series.str.title()
```

```
    counts = series.value_counts().head(top_n).sort_values(ascending=True)
    axis = counts.plot(kind="barh", figsize=(8, 10))
    axis.set_xlabel("Count")
    axis.set_ylabel("Chief complaint")
    axis.set_title(f"Top {top_n} Chief Complaints ({cc_column})")
    plt.tight_layout()
    plt.show()

    return counts.sort_values(ascending=False).rename_axis("chief_complaint"
     ↪  ).reset_index(name="count")
```

1.3 2B) Local helper functions (embedded core notebook logic)

```
# Core stage helpers are embedded in this notebook by project policy.

DATA_DIRNAME = "MIMIC tabular data"
PRIOR_RUNS_DIRNAME = "prior runs"

CANONICAL_COHORT_FILENAME = "MIMICIV all with CC.xlsx"
CANONICAL_NLP_FILENAME = "MIMICIV all with CC_with_NLP.xlsx"

CLASSIFIER_TRANSITIONAL_ALIASES: dict[str, tuple[str, ...]] = {
    "age": ("age_at_admit",),
    "hr": ("ed_first_hr_model", "ed_first_hr", "ed_triage_hr_model",
     ↪  "ed_triage_hr"),
    "rr": ("ed_first_rr_model", "ed_first_rr", "ed_triage_rr_model",
     ↪  "ed_triage_rr"),
    "sbp": ("ed_first_sbp_model", "ed_first_sbp", "ed_triage_sbp_model",
     ↪  "ed_triage_sbp"),
    "dbp": ("ed_first_dbp_model", "ed_first_dbp", "ed_triage_dbp_model",
     ↪  "ed_triage_dbp"),
    "temp": ("ed_first_temp_model", "ed_first_temp", "ed_triage_temp_model",
     ↪  "ed_triage_temp"),
    "spo2": (
        "ed_first_o2sat_model",
        "ed_first_o2sat",
        "ed_triage_o2sat_model",
        "ed_triage_o2sat",
    ),
    "race": ("race_ed_raw",),
}
```

```
def data_dir(work_dir: Path) -> Path:
    """Return canonical data directory rooted at ``work_dir``."""
    return (work_dir / DATA_DIRNAME).expanduser().resolve()


def resolve_classifier_input_path(
    work_dir: Path, input_filename: str | None = None
) -> Path:
    """Resolve classifier input workbook under canonical data directory."""
    filename = input_filename or CANONICAL_COHORT_FILENAME
    input_path = data_dir(work_dir) / filename
    if not input_path.exists():
        raise FileNotFoundError(
            "Expected classifier input workbook was not found at "
            f"{input_path}. Run the cohort notebook first or set "
            "CLASSIFIER_INPUT_FILENAME."
        )
    return input_path


def resolve_analysis_input_path(work_dir: Path, input_filename: str | None =
 ↪  None) -> Path:
    """Resolve analysis input workbook under canonical data directory."""
    filename = input_filename or CANONICAL_NLP_FILENAME
    input_path = data_dir(work_dir) / filename
    if not input_path.exists():
        raise FileNotFoundError(
            "Expected analysis input workbook was not found at "
            f"{input_path}. Run the classifier notebook first or set "
            "ANALYSIS_INPUT_FILENAME."
        )
    return input_path


def resolve_rater_nlp_input_path(work_dir: Path, input_filename: str | None =
 ↪  None) -> Path:
    """Resolve rater notebook NLP input workbook under canonical data
     ↪  directory."""
    filename = input_filename or CANONICAL_NLP_FILENAME
    input_path = data_dir(work_dir) / filename
    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


def resolve_classifier_output_paths(
    work_dir: Path,
    *,
    run_dt: datetime | None = None,
) -> tuple[Path, Path]:
    """Return canonical NLP output path and dated archive path."""
    base_data_dir = data_dir(work_dir)
    canonical_path = base_data_dir / CANONICAL_NLP_FILENAME

    run_date = (run_dt or datetime.now()).strftime("%Y-%m-%d")
    archive_dir = base_data_dir / PRIOR_RUNS_DIRNAME
    archive_path = archive_dir / f"{run_date} {CANONICAL_NLP_FILENAME}"

    return canonical_path, archive_path


def normalize_classifier_input_schema(
    df: pd.DataFrame,
    alias_map: Mapping[str, str | Sequence[str]] | None = None,
) -> pd.DataFrame:
    """Add transitional alias columns for classifier compatibility.

    Existing destination columns are preserved. Source columns are never
 ↪  removed.
    """
    resolved_alias_map = alias_map or CLASSIFIER_TRANSITIONAL_ALIASES
    normalized = df.copy()
    for destination, source_spec in resolved_alias_map.items():
        if destination in normalized.columns:
            continue
        if isinstance(source_spec, str):
            candidate_sources = (source_spec,)
        else:
            candidate_sources = tuple(source_spec)
        for source_name in candidate_sources:
            if source_name in normalized.columns:
```

```
                normalized[destination] = normalized[source_name]
                break
    return normalized


def ensure_required_columns(
    df: pd.DataFrame,
    required: list[str],
    *,
    context: str,
) -> None:
    """Validate required columns with context-aware error message."""
    missing = sorted(set(required).difference(df.columns))
    if missing:
        raise KeyError(f"{context}: missing required columns: {missing}")

PSEUDO_MISSING_EXACT_TOKENS = {
    "-",
    "--",
    "_",
    "n",
    "na",
    "n/a",
    "none",
    "unknown",
    "refused",
}
PSEUDO_MISSING_REGEX = re.compile(r"^_+$")
PSEUDO_MISSING_PUNCT_REGEX = re.compile(r"^[^a-z0-9]+$")


def _canonicalize_cc_text(value: object) -> str:
    text = "" if pd.isna(value) else str(value)
    text = text.strip().lower()
    text = text.replace("-", "-").replace("-", "-")
    return " ".join(text.split())


def _to_numeric_series(df: pd.DataFrame, column_name: str) -> pd.Series:
    if column_name in df.columns:
        return pd.to_numeric(df[column_name], errors="coerce")
    return pd.Series(math.nan, index=df.index)
```

```python
def _to_text_series(df: pd.DataFrame, column_name: str) -> pd.Series:
    if column_name in df.columns:
        return df[column_name].astype("string")
    return pd.Series(pd.NA, index=df.index, dtype="string")


def _to_binary_indicator(df: pd.DataFrame, column_name: str) -> pd.Series:
    if column_name not in df.columns:
        return pd.Series(pd.NA, index=df.index, dtype="Float64")
    numeric = pd.to_numeric(df[column_name], errors="coerce")
    return (numeric > 0).astype("Float64").where(numeric.notna(), pd.NA)


def add_hypercapnia_flags(
    df: pd.DataFrame,
    *,
    art_col: str = "first_abg_pco2",
    art_uom_col: str = "first_abg_pco2_uom",
    vbg_col: str = "first_vbg_pco2",
    vbg_uom_col: str = "first_vbg_pco2_uom",
    fallback_art_col: str = "poc_abg_paco2",
    fallback_art_uom_col: str = "poc_abg_paco2_uom",
    fallback_vbg_col: str = "poc_vbg_paco2",
    fallback_vbg_uom_col: str = "poc_vbg_paco2_uom",
    authoritative_abg_col: str = "flag_abg_hypercapnia",
    authoritative_vbg_col: str = "flag_vbg_hypercapnia",
    out_abg: str = "hypercap_by_abg",
    out_vbg: str = "hypercap_by_vbg",
    out_any: str = "hypercap_by_bg",
) -> pd.DataFrame:
    """Add ABG/VBG hypercapnia flags using cohort-schema defaults.

    Precedence is applied per-row:
    1) authoritative route flags from cohort (`flag_*_hypercapnia`) when
 ↪  present,
    2) first-route pCO2 values (`first_abg_pco2` / `first_vbg_pco2`),
    3) fallback POC pCO2 values.
    """

    def to_mmhg(values: pd.Series, uoms: pd.Series) -> pd.Series:
        numeric = pd.to_numeric(values, errors="coerce")
        unit_text = uoms.astype("string").str.strip().str.lower()
```

```
    is_kpa = unit_text.str.contains("kpa", na=False)
    converted = numeric.copy()
    converted.loc[is_kpa] = converted.loc[is_kpa] * 7.50062
    return converted

def derive_from_value(
    *,
    value_col: str,
    uom_col: str,
    threshold: float,
) -> pd.Series:
    values = _to_numeric_series(out, value_col)
    units = _to_text_series(out, uom_col)
    mmhg = to_mmhg(values, units)
    derived = (mmhg >= threshold).astype("Float64")
    return derived.where(mmhg.notna(), pd.NA)

out = df.copy()
abg_flag = _to_binary_indicator(out, authoritative_abg_col)
if abg_flag.isna().any():
    primary_abg = derive_from_value(
        value_col=art_col,
        uom_col=art_uom_col,
        threshold=45.0,
    )
    abg_flag = abg_flag.where(abg_flag.notna(), primary_abg)
if abg_flag.isna().any():
    fallback_abg = derive_from_value(
        value_col=fallback_art_col,
        uom_col=fallback_art_uom_col,
        threshold=45.0,
    )
    abg_flag = abg_flag.where(abg_flag.notna(), fallback_abg)

vbg_flag = _to_binary_indicator(out, authoritative_vbg_col)
if vbg_flag.isna().any():
    primary_vbg = derive_from_value(
        value_col=vbg_col,
        uom_col=vbg_uom_col,
        threshold=50.0,
    )
    vbg_flag = vbg_flag.where(vbg_flag.notna(), primary_vbg)
if vbg_flag.isna().any():
```

```
        fallback_vbg = derive_from_value(
            value_col=fallback_vbg_col,
            uom_col=fallback_vbg_uom_col,
            threshold=50.0,
        )
        vbg_flag = vbg_flag.where(vbg_flag.notna(), fallback_vbg)

    out[out_abg] = abg_flag.fillna(0).astype("int8")
    out[out_vbg] = vbg_flag.fillna(0).astype("int8")
    out[out_any] = ((out[out_abg] == 1) | (out[out_vbg] == 1)).astype("int8")
    return out


def annotate_cc_missingness(
    df: pd.DataFrame,
    *,
    cc_column: str,
    nlp_input_column: str = "cc_text_for_nlp",
    missing_flag_column: str = "cc_missing_flag",
    pseudo_flag_column: str = "cc_pseudomissing_flag",
    missing_reason_column: str = "cc_missing_reason",
) -> pd.DataFrame:
    """Annotate true-vs-pseudo missing CCs and derive deterministic NLP input
    ↪  text."""
    if cc_column not in df.columns:
        raise KeyError(f"Chief complaint column not found: {cc_column}")

    out = df.copy()
    canonical = out[cc_column].map(_canonicalize_cc_text)
    is_true_missing = canonical.eq("")
    is_pseudo_missing = (
        canonical.isin(PSEUDO_MISSING_EXACT_TOKENS)
        | canonical.str.match(PSEUDO_MISSING_REGEX, na=False)
        | canonical.str.match(PSEUDO_MISSING_PUNCT_REGEX, na=False)
    )

    reason = pd.Series("valid", index=out.index, dtype="string")
    reason.loc[is_true_missing] = "true_missing"
    reason.loc[is_pseudo_missing] = "pseudo_missing_token"

    out[missing_flag_column] = (is_true_missing |
↪   is_pseudo_missing).astype("boolean")
    out[pseudo_flag_column] = is_pseudo_missing.astype("boolean")
```

```
    out[missing_reason_column] = reason
    out[nlp_input_column] = out[cc_column].astype("string")
    out.loc[is_true_missing | is_pseudo_missing, nlp_input_column] = pd.NA
    return out


def apply_pseudomissing_uncodable_policy(
    df: pd.DataFrame,
    *,
    max_rfv: int = 5,
    pseudo_flag_column: str = "cc_pseudomissing_flag",
) -> pd.DataFrame:
    """Force pseudo-missing CC rows to a deterministic uncodable RFV
     ↪  payload."""
    if pseudo_flag_column not in df.columns:
        return df

    out = df.copy()
    pseudo_mask = out[pseudo_flag_column].fillna(False).astype(bool)
    if not pseudo_mask.any():
        return out

    payload = json.dumps(
        [
            {
                "seg_idx": 0,
                "segment": "",
                "code": "RVC-UNCL",
                "name": "Uncodable/Unknown",
                "sim": None,
                "rule_code": "pseudo_missing_cc",
                "rule_used": True,
            }
        ]
    )
    _apply_uncodable_payload(
        out,
        mask=pseudo_mask,
        max_rfv=max_rfv,
        payload=payload,
    )
    return out
```

```python
def apply_blank_primary_uncodable_policy(
    df: pd.DataFrame,
    *,
    max_rfv: int = 5,
    missing_reason_column: str = "cc_missing_reason",
) -> pd.DataFrame:
    """Force non-true-missing blank RFV primaries to deterministic uncodable
     ↪  output."""
    if "RFV1_name" not in df.columns:
        return df

    out = df.copy()
    blank_primary =
 ↪  out["RFV1_name"].fillna("").astype(str).str.strip().eq("")
    true_missing = (
        out[missing_reason_column].astype(str).eq("true_missing")
        if missing_reason_column in out.columns
        else pd.Series(False, index=out.index)
    )
    fallback_mask = blank_primary & ~true_missing
    if not fallback_mask.any():
        return out

    payload = json.dumps(
        [
            {
                "seg_idx": 0,
                "segment": "",
                "code": "RVC-UNCL",
                "name": "Uncodable/Unknown",
                "sim": None,
                "rule_code": "blank_primary_fallback",
                "rule_used": True,
            }
        ]
    )
    _apply_uncodable_payload(
        out,
        mask=fallback_mask,
        max_rfv=max_rfv,
        payload=payload,
    )
```

```
    return out


def _apply_uncodable_payload(
    out: pd.DataFrame,
    *,
    mask: pd.Series,
    max_rfv: int,
    payload: str,
) -> None:
    """Apply a deterministic uncodable RFV payload in-place."""
    out.loc[mask, "RFV1"] = "uncodable"
    out.loc[mask, "RFV1_name"] = "Uncodable/Unknown"
    out.loc[mask, "RFV1_support"] = ""
    out.loc[mask, "RFV1_sim"] = math.nan

    for slot in range(2, max_rfv + 1):
        out.loc[mask, f"RFV{slot}"] = ""
        out.loc[mask, f"RFV{slot}_name"] = ""
        out.loc[mask, f"RFV{slot}_support"] = ""
        out.loc[mask, f"RFV{slot}_sim"] = math.nan

    out.loc[mask, "segment_preds"] = payload


def build_cc_missing_audit(
    df: pd.DataFrame,
    *,
    cc_column: str,
    missing_reason_column: str = "cc_missing_reason",
    max_examples: int = 5,
) -> pd.DataFrame:
    """Summarize CC missingness categories with example raw values."""
    if cc_column not in df.columns:
        raise KeyError(f"Chief complaint column not found: {cc_column}")
    if missing_reason_column not in df.columns:
        raise KeyError(f"Missing reason column not found:
        ↪ {missing_reason_column}")

    frame = df[[cc_column, missing_reason_column]].copy()
    frame[cc_column] = frame[cc_column].astype("string")
    rows: list[dict[str, Any]] = []
    for reason, group in frame.groupby(missing_reason_column, dropna=False):
```

```
        examples = (
            group[cc_column]
            .fillna("<NA>")
            .astype(str)
            .value_counts(dropna=False)
            .head(max_examples)
            .index.tolist()
        )
        rows.append(
            {
                "cc_missing_reason": str(reason),
                "row_count": int(len(group)),
                "examples": "; ".join(examples),
            }
        )
    return pd.DataFrame(rows).sort_values("cc_missing_reason"
    ↪ ).reset_index(drop=True)


def classifier_resource_paths(
    work_dir: Path,
    *,
    appendix_relpath: str,
    summary_relpath: str,
) -> dict[str, Path]:
    """Return required classifier resource paths under ``work_dir``."""
    return {
        "appendix_csv": (work_dir / appendix_relpath).expanduser().resolve(),
        "summary_weights_csv": (work_dir /
        ↪ summary_relpath).expanduser().resolve(),
    }


def _sha256(path: Path) -> str:
    hasher = hashlib.sha256()
    with path.open("rb") as handle:
        for chunk in iter(lambda: handle.read(8192), b""):
            hasher.update(chunk)
    return hasher.hexdigest()


def verify_classifier_resources(
    work_dir: Path,
```

```
    *,
    appendix_relpath: str,
    summary_relpath: str,
    manifest_path: str | None = "Annotation/resource_manifest.json",
    strict_hash: bool = True,
) -> dict[str, Any]:
    """Validate required classifier resources and optional hash manifest."""
    checks: list[dict[str, Any]] = []
    missing_paths: list[str] = []
    for name, path in classifier_resource_paths(
        work_dir,
        appendix_relpath=appendix_relpath,
        summary_relpath=summary_relpath,
    ).items():
        exists = path.exists()
        checks.append({"resource": name, "path": str(path), "exists":
↪   exists})
        if not exists:
            missing_paths.append(str(path))

    hash_findings: list[dict[str, Any]] = []
    manifest_resolved: Path | None = None
    if manifest_path:
        manifest_resolved = (work_dir / manifest_path).expanduser().resolve()
        if manifest_resolved.exists():
            manifest = json.loads(manifest_resolved.read_text())
            resources = manifest.get("resources", [])
            for entry in resources:
                rel_path = entry.get("path")
                expected = str(entry.get("sha256", "")).strip().lower()
                if not rel_path or not expected:
                    continue
                path = (work_dir / rel_path).expanduser().resolve()
                if not path.exists():
                    hash_findings.append(
                        {
                            "path": str(path),
                            "status": "missing",
                            "expected_sha256": expected,
                            "actual_sha256": None,
                        }
                    )
                    continue
```

```
            actual = _sha256(path)
            if actual != expected:
                hash_findings.append(
                    {
                        "path": str(path),
                        "status": "hash_mismatch",
                        "expected_sha256": expected,
                        "actual_sha256": actual,
                    }
                )

errors: list[str] = []
warnings: list[str] = []

if missing_paths:
    errors.append(
        "Missing required classifier resources: " + ",
        ↪  ".join(sorted(missing_paths))
    )

for finding in hash_findings:
    message = (
        f"{finding['status']} for {finding['path']} "
        f"(expected={finding['expected_sha256']},
        ↪  actual={finding['actual_sha256']})"
    )
    if strict_hash:
        errors.append(message)
    else:
        warnings.append(message)

status = "pass"
if errors:
    status = "fail"
elif warnings:
    status = "warning"

return {
    "status": status,
    "strict_hash": bool(strict_hash),
    "manifest_path": str(manifest_resolved) if manifest_resolved else
    ↪  None,
    "checks": checks,
```

```
        "hash_findings": hash_findings,
        "errors": errors,
        "warnings": warnings,
    }


def validate_classifier_contract(
    df: pd.DataFrame,
    *,
    max_rfv: int = 5,
    pseudo_flag_column: str = "cc_pseudomissing_flag",
    missing_reason_column: str = "cc_missing_reason",
    missing_flag_column: str = "cc_missing_flag",
    authoritative_disagreement_tolerance: float = 0.0,
) -> dict[str, Any]:
    """Run deterministic post-export classifier contract checks."""
    findings: list[dict[str, str]] = []

    if "hadm_id" in df.columns:
        duplicates = int(df.duplicated(subset=["hadm_id"]).sum())
        if duplicates:
            findings.append(
                {
                    "severity": "error",
                    "code": "hadm_id_not_unique",
                    "message": f"Found {duplicates} duplicate hadm_id rows.",
                }
            )

    if "segment_preds" not in df.columns:
        findings.append(
            {
                "severity": "error",
                "code": "missing_segment_preds",
                "message": "segment_preds column missing from classifier
                 ↪  output.",
            }
        )
    else:
        bad_json_rows = 0
        for value in df["segment_preds"].fillna("[]"):
            try:
```

```
            parsed = json.loads(value) if isinstance(value, str) else
↪ value
        except json.JSONDecodeError:
            bad_json_rows += 1
            continue
        if not isinstance(parsed, list):
            bad_json_rows += 1
    if bad_json_rows:
        findings.append(
            {
                "severity": "error",
                "code": "segment_preds_malformed",
                "message": f"Found {bad_json_rows} malformed
                 ↪  segment_preds rows.",
            }
        )

if {"hypercap_by_abg", "hypercap_by_vbg",
 ↪  "hypercap_by_bg"}.issubset(df.columns):
    abg = pd.to_numeric(df["hypercap_by_abg"],
↪ errors="coerce").fillna(0).astype(int)
    vbg = pd.to_numeric(df["hypercap_by_vbg"],
↪ errors="coerce").fillna(0).astype(int)
    bg = pd.to_numeric(df["hypercap_by_bg"],
↪ errors="coerce").fillna(0).astype(int)
    mismatch = int(((abg | vbg) != bg).sum())
    if mismatch:
        findings.append(
            {
                "severity": "error",
                "code": "bg_union_mismatch",
                "message": f"hypercap_by_bg mismatch in {mismatch}
                 ↪  rows.",
            }
        )
    if int(abg.sum()) == 0:
        findings.append(
            {
                "severity": "warning",
                "code": "abg_all_zero",
                "message": "hypercap_by_abg has zero positives.",
            }
        )
```

```
        if "flag_abg_hypercapnia" in df.columns:
            abg_authoritative = pd.to_numeric(
                df["flag_abg_hypercapnia"], errors="coerce"
            )
            valid_abg = abg_authoritative.notna()
            if bool(valid_abg.any()):
                authoritative_binary = (
                    abg_authoritative.loc[valid_abg].fillna(0).astype(int) >
↪ 0
                ).astype(int)
                disagreement = int((abg.loc[valid_abg] !=
↪ authoritative_binary).sum())
                rate = float(disagreement / int(valid_abg.sum()))
                if rate > authoritative_disagreement_tolerance:
                    findings.append(
                        {
                            "severity": "error",
                            "code": "abg_authoritative_disagreement",
                            "message": (
                                "hypercap_by_abg disagrees with
                                ↪  flag_abg_hypercapnia in "
                                f"{disagreement} rows ({rate:.4%}),
                                ↪  tolerance="
                                f"{authoritative_disagreement_tolerance:.4%}
                                ↪  ."
                            ),
                        }
                    )
        if "flag_vbg_hypercapnia" in df.columns:
            vbg_authoritative = pd.to_numeric(
                df["flag_vbg_hypercapnia"], errors="coerce"
            )
            valid_vbg = vbg_authoritative.notna()
            if bool(valid_vbg.any()):
                authoritative_binary = (
                    vbg_authoritative.loc[valid_vbg].fillna(0).astype(int) >
↪ 0
                ).astype(int)
                disagreement = int((vbg.loc[valid_vbg] !=
↪ authoritative_binary).sum())
                rate = float(disagreement / int(valid_vbg.sum()))
                if rate > authoritative_disagreement_tolerance:
                    findings.append(
```

```
                    {
                        "severity": "error",
                        "code": "vbg_authoritative_disagreement",
                        "message": (
                            "hypercap_by_vbg disagrees with
                             ↪  flag_vbg_hypercapnia in "
                            f"{disagreement} rows ({rate:.4%}),
                             ↪  tolerance="
                            f"{authoritative_disagreement_tolerance:.4%}
                             ↪  ."
                        ),
                    }
                )

    required_cc_columns = {missing_reason_column, pseudo_flag_column,
↪   missing_flag_column}
    missing_cc_columns = sorted(required_cc_columns.difference(df.columns))
    if missing_cc_columns:
        findings.append(
            {
                "severity": "error",
                "code": "missing_cc_missingness_columns",
                "message": f"Missing CC missingness columns:
                 ↪  {missing_cc_columns}",
            }
        )

    if "RFV1_name" in df.columns:
        blank_rfv1 =
↪   df["RFV1_name"].fillna("").astype(str).str.strip().eq("")
        if missing_reason_column in df.columns:
            allowed_blank =
↪   df[missing_reason_column].astype(str).eq("true_missing")
            unexpected_blank = int((blank_rfv1 & ~allowed_blank).sum())
        else:
            unexpected_blank = int(blank_rfv1.sum())
        if unexpected_blank:
            findings.append(
                {
                    "severity": "error",
                    "code": "unexpected_blank_rfv1",
                    "message": (
```

```
                        "Found blank RFV1_name rows outside allowed
                         ↪  true-missing policy: "
                        f"{unexpected_blank}"
                    ),
                }
            )

    pseudo_count = 0
    if pseudo_flag_column in df.columns:
        pseudo_count = int(df[pseudo_flag_column].fillna(False).sum())

    error_count = sum(1 for finding in findings if finding["severity"] ==
↪   "error")
    warning_count = sum(1 for finding in findings if finding["severity"] ==
↪   "warning")
    status = "pass"
    if error_count:
        status = "fail"
    elif warning_count:
        status = "warning"

    return {
        "status": status,
        "error_count": error_count,
        "warning_count": warning_count,
        "pseudo_missing_rows": pseudo_count,
        "findings": findings,
        "row_count": int(len(df)),
        "column_count": int(df.shape[1]),
        "max_rfv": int(max_rfv),
    }

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
```

1.4 3) Input load and schema checks

```
# Purpose: instantiate config, set deterministic seeds, load the input
↪   workbook, and validate required identifiers.
# Why this matters: stable setup reduces run-to-run variability and catches
↪   input issues before expensive model steps.

cfg = ClassifierConfig()

try:
    NLP = spacy.load("en_core_web_sm", disable=["ner", "parser", "textcat",
↪   "senter"])
    SPACY_MODEL_LABEL = "en_core_web_sm"
except OSError as exc:
    if cfg.require_spacy_model:
        raise ImportError(
            "spaCy model 'en_core_web_sm' is required but not installed in
            ↪   this environment. "
            "Install it once with: python -m spacy download en_core_web_sm"
        ) from exc
    NLP = spacy.blank("en")
    SPACY_MODEL_LABEL = "spacy.blank('en')"

print(f"spaCy pipeline: {SPACY_MODEL_LABEL}")

np.random.seed(cfg.seed)
torch.manual_seed(cfg.seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(cfg.seed)

load_dotenv()
WORK_DIR = Path(os.getenv(cfg.work_dir_env_var,
↪   Path.cwd())).expanduser().resolve()
```

```
SRC_DIR = WORK_DIR / "src"
if str(SRC_DIR) not in sys.path:
    sys.path.insert(0, str(SRC_DIR))

from hypercap_cc_nlp.pipeline_contracts import write_contract_report
from hypercap_cc_nlp.pipeline_audit import collect_run_manifest
ensure_required_workflow_columns = ensure_required_columns

classifier_input_filename = os.getenv(cfg.input_filename_env_var,
 ↪  cfg.input_filename)
file_path = resolve_classifier_input_path(WORK_DIR,
 ↪  classifier_input_filename)

strict_resource_hash = os.getenv("CLASSIFIER_STRICT_RESOURCE_HASH",
 ↪  "1").strip() == "1"
resource_report = verify_classifier_resources(
    WORK_DIR,
    appendix_relpath=cfg.appendix_relpath,
    summary_relpath=cfg.summary_relpath,
    strict_hash=strict_resource_hash,
)
if resource_report["status"] == "fail":
    raise FileNotFoundError(
        "Classifier resource verification failed: "
        + "; ".join(resource_report["errors"])
    )
if resource_report["warnings"]:
    print("Resource verification warnings:")
    for warning_message in resource_report["warnings"]:
        print("-", warning_message)

df = pd.read_excel(file_path, sheet_name=0, engine="openpyxl")
df.columns = df.columns.str.strip()
pre_alias_columns = set(df.columns)
df = normalize_classifier_input_schema(df)
added_aliases = sorted(set(df.columns).difference(pre_alias_columns))

ensure_required_workflow_columns(
    df,
    ["hadm_id", "subject_id"],
    context="classifier input",
)
```

```
print(f"Loaded input rows={len(df):,}, cols={df.shape[1]:,}")
print(f"Input path: {file_path}")
print(f"Added transitional aliases: {added_aliases if added_aliases else
 ↪  'None'}")
cc_column = get_cc_column(df)
print(f"Detected chief complaint column: {cc_column}")
```

```
spaCy pipeline: en_core_web_sm
Loaded input rows=41,322, cols=234
Input path: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Research Pro
CC-NLP/MIMIC tabular data/MIMICIV all with CC.xlsx
Added transitional aliases: ['age', 'dbp', 'hr', 'race', 'rr', 'sbp', 'spo2', 'temp']
Detected chief complaint column: ed_triage_cc
```

1.5 4) Text cleaning and segmentation functions

```
# Purpose: define data-enrichment helpers and generate normalized
 ↪  chief-complaint text used by the classifier.
# Why this matters: high-quality text normalization is foundational for both
 ↪  rule matching and embedding similarity.

df = add_hypercapnia_flags(df)
df = annotate_cc_missingness(df, cc_column=cc_column)
df["cc_cleaned"] = df["cc_text_for_nlp"].apply(normalize_cc)
df["cc_cleaned_str"] = df["cc_cleaned"].str.join("; ")
df["cc_cleaned_canon"] = df["cc_cleaned"].apply(canonize_cc_segments)
df["cc_cleaned_canon_str"] = df["cc_cleaned_canon"].str.join("; ")

cc_missing_audit = build_cc_missing_audit(df, cc_column=cc_column)
print("CC missingness audit:")
print(cc_missing_audit.to_string(index=False))
print("Pseudo-missing rows:",
 ↪  int(df["cc_pseudomissing_flag"].fillna(False).sum()))

print("Text normalization complete.")
print(
    df[
        [
            cc_column,
            "cc_missing_flag",
            "cc_missing_reason",
            "cc_pseudomissing_flag",
```

```
            "cc_cleaned",
            "cc_cleaned_str",
        ]
    ]
    .head(5)
    .to_string()
)
```

```
CC missingness audit:
   cc_missing_reason  row_count                                                                  example
pseudo_missing_token         18                                                  ___; -
; N; ___,___; --
               valid      41304 Dyspnea; Altered mental status; Chest pain; Abd pain; s/p Fal
Pseudo-missing rows: 18
Text normalization complete.
         ed_triage_cc  cc_missing_flag cc_missing_reason  cc_pseudomissing_flag
0  Abd pain, Vomiting           False             valid                  False  [abdominal p
1             DYSPNEA           False             valid                  False
2         BRADYCARDIA           False             valid                  False            [
3      FEVER,HEADACHE           False             valid                  False        [fever
4             DYSPNEA           False             valid                  False
```

1.6 5) Rule system and ontology mappings

```
# Purpose: run a quick rule-coverage spot check on representative phrases.
# Why this matters: confirms deterministic override logic is wired correctly
 ↪  before full inference.

print("Rule coverage preview:")
probe_texts = [
    "shortness of breath",
    "bradycardia",
    "positive blood culture",
    "alter mental status",
    "abdominal pain",
    "right eye redness / pain",
    "paperwork clearance",
]
for text in probe_texts:
    override, _ = rule_gate(text)
    print(f"{text!r} -> {override}")
```

```
Rule coverage preview:
'shortness of breath' -> RVC-SYM-RESP
'bradycardia' -> RVC-SYM-CIRC
'positive blood culture' -> RVC-TEST
'alter mental status' -> RVC-SYM-NERV
'abdominal pain' -> RVC-SYM-DIG
'right eye redness / pain' -> RVC-SYM-EYE
'paperwork clearance' -> RVC-ADMIN
```

1.7 6) Prototype loading, deduplication, and embedding preparation

```
# Purpose: build prototype embeddings/resources once and print metadata for
 ↪ reproducibility auditing.
# Why this matters: model/device/prototype counts help explain classification
 ↪ behavior during review.

resources = build_prototype_resources(cfg)
metadata = resources["metadata"]

print("Model metadata:")
print(f"- model_name: {metadata['model_name']}")
print(f"- device: {metadata['device']}")
print(f"- spacy_model: {metadata['spacy_model']}")
print(f"- hf_token_present: {metadata['hf_token_present']}")
print(f"- appendix_rows_used: {metadata['appendix_rows_used']}")
print(f"- total_prototypes: {metadata['prototype_total']}")
print("- prototype_count_by_group:")
print(metadata["prototype_count_by_group"])
```

```
Model metadata:
- model_name: NeuML/bioclinical-modernbert-base-embeddings
- device: cpu
- spacy_model: en_core_web_sm
- hf_token_present: True
- appendix_rows_used: 606
- total_prototypes: 693
- prototype_count_by_group:
{'RVC-INJ': 75, 'RVC-SYM-RESP': 43, 'RVC-SYM-CIRC': 18, 'RVC-SYM-NERV': 25, 'RVC-
SYM-DIG': 52, 'RVC-SYM-GU': 76, 'RVC-SYM-MSK': 22, 'RVC-SYM-SKIN': 29, 'RVC-
SYM-EYE': 50, 'RVC-SYM-GEN': 38, 'RVC-SYM-PSY': 40, 'RVC-DIS': 92, 'RVC-
TEST': 12, 'RVC-DIAG': 47, 'RVC-TREAT': 54, 'RVC-ADMIN': 14, 'RVC-UNCL': 6}
```

1.8 7) Segment cache build and classification

```
# Purpose: build the segment cache and run visit-level RFV assignment across
 ↪  the full dataset.
# Why this matters: caching unique segments improves runtime while preserving
 ↪  deterministic outputs.

cache = build_segment_cache(
    df=df,
    seg_col="cc_cleaned",
    resources=resources,
    batch_size=cfg.segment_batch_size,
)

df_out = assign_rvc_per_segment_and_visit_cached(
    df=df,
    cache=cache,
    resources=resources,
    seg_col="cc_cleaned",
    cfg=cfg,
)
df_out = apply_pseudomissing_uncodable_policy(
    df_out,
    max_rfv=cfg.max_rfv,
    pseudo_flag_column="cc_pseudomissing_flag",
)
df_out = apply_blank_primary_uncodable_policy(
    df_out,
    max_rfv=cfg.max_rfv,
    missing_reason_column="cc_missing_reason",
)

print(f"Unique segments cached: {len(cache.uniq_segments):,}")
print(f"Cache emb shape: {tuple(cache.emb.shape)}")
print(f"Cache sim shape: {tuple(cache.sim_proto.shape)}")


Unique segments cached: 3,154
Cache emb shape: (3154, 768)
Cache sim shape: (3154, 693)
```

1.9 8) Output shaping and compatibility mapping

Standardized output columns: - RFV1..RFV5 (legacy short-code compatibility) - RFV1_name..RFV5_name - RFV1_support..RFV5_support - RFV1_sim..RFV5_sim - segment_preds

```
# Purpose: validate the output schema contract and preview standardized RFV
 ↪ columns.
# Why this matters: downstream notebooks rely on these exact column names and
 ↪ structures.

validate_output_schema(
    df_out,
    max_rfv=cfg.max_rfv,
    require_primary_for_nonempty=cfg.allow_uncodable_primary,
)

display_cols = ["cc_cleaned"]
for slot in range(1, cfg.max_rfv + 1):
    display_cols.extend(
        [
            f"RFV{slot}",
            f"RFV{slot}_name",
            f"RFV{slot}_support",
            f"RFV{slot}_sim",
        ]
    )

print(df_out[display_cols].head(10).to_string())
```

```
Schema validation passed for 41,322 rows.
                                cc_cleaned     RFV1              RFV1_name              RFV1
0                  [abdominal pain, vomit]       gi    Symptom - Digestive            abdomi
1                                [dyspnea]     resp  Symptom - Respiratory
2                            [bradycardia]     circ  Symptom - Circulatory              brad
3                       [fever, headache]    neuro      Symptom - Nervous                 h
4                                [dyspnea]     resp  Symptom - Respiratory
5                    [shortness of breath]     resp  Symptom - Respiratory       shortness o
6                           [nausea vomit]       gi    Symptom - Digestive             nause
7                [right eye redness / pain]  eye_ear      Symptom - Eye/Ear  right eye redness
8  [abdominal pain, nausea vomit diarrhea]       gi    Symptom - Digestive            abdomi
9                                [dyspnea]     resp  Symptom - Respiratory
```

1.10 9) Diagnostics and QA checks

```
# Purpose: run QA smoke checks and summarize regression metrics after
 ↪  classification.
# Why this matters: immediate diagnostics make it easier to detect unintended
 ↪  behavioral drift.

qa_summary = validate_pipeline(df_out, cache=cache, resources=resources,
 ↪  seg_col="cc_cleaned")

override_summary = summarize_overrides(df_out, column="segment_preds")

print("\nRegression summary")
print("------------------")
print(f"Input rows: {len(df):,}")
print(f"Output rows: {len(df_out):,}")
print(f"Row count preserved: {len(df) == len(df_out)}")

print("\nTop RFV1_name counts:")
print(df_out["RFV1_name"].value_counts(dropna=True).head(15).to_string())

print("\nOverride usage:")
print(f"Segments total: {override_summary['segments_total']:,}")
print(f"Segments overridden: {override_summary['segments_overridden']:,}")
print(f"Segment override rate:
 ↪  {override_summary['segment_override_rate']:.4f}")
print(f"Rows total: {override_summary['rows_total']:,}")
print(f"Rows with >=1 override: {override_summary['rows_with_override']:,}")
print(f"Row override rate: {override_summary['row_override_rate']:.4f}")

parsed_segment_preds = df_out["segment_preds"].fillna("[]").map(
    lambda value: json.loads(value) if isinstance(value, str) else []
)
blank_rfv1 = df_out["RFV1_name"].fillna("").astype(str).str.strip().eq("")
nonempty_segment_rows = parsed_segment_preds.map(lambda payload:
 ↪  isinstance(payload, list) and len(payload) > 0)
all_uncodable_rows = parsed_segment_preds.map(
    lambda payload: isinstance(payload, list)
    and len(payload) > 0
    and all(isinstance(item, dict) and item.get("code") == "RVC-UNCL" for
     ↪  item in payload)
)
```

```
blank_rfv1_total = int(blank_rfv1.sum())
blank_rfv1_with_nonempty_segments = int((blank_rfv1 &
 ↪  nonempty_segment_rows).sum())
all_uncodable_total = int(all_uncodable_rows.sum())

print("\nPrimary-label completeness:")
print(f"blank_rfv1_total: {blank_rfv1_total:,}")
print(f"blank_rfv1_with_nonempty_segments:
 ↪  {blank_rfv1_with_nonempty_segments:,}")
print(f"all_uncodable_rows: {all_uncodable_total:,}")

uncodable_segment_counter: Counter = Counter()
for payload in parsed_segment_preds[all_uncodable_rows]:
    for item in payload:
        if isinstance(item, dict):
            uncodable_segment_counter[item.get("segment", "")] += 1

print("\nTop uncodable segments (post-fix):")
for segment, count in uncodable_segment_counter.most_common(15):
    print(f"{count:4d} | {segment}")

if cfg.allow_uncodable_primary and blank_rfv1_with_nonempty_segments != 0:
    raise AssertionError(
        "Expected zero blank RFV1_name values for rows with non-empty segment
         ↪  predictions, "
        f"found {blank_rfv1_with_nonempty_segments}."
    )

if cfg.run_optional_plots:
    top_table = top_cc_plot(df, top_n=30, normalize_case="upper")
    print(top_table.head(10).to_string(index=False))
else:
    print("\nOptional plots skipped (cfg.run_optional_plots=False).")
```

```
Prototype integrity:
{'RVC-INJ': 75, 'RVC-SYM-RESP': 43, 'RVC-SYM-CIRC': 18, 'RVC-SYM-NERV': 25, 'RVC-
SYM-DIG': 52, 'RVC-SYM-GU': 76, 'RVC-SYM-MSK': 22, 'RVC-SYM-SKIN': 29, 'RVC-
SYM-EYE': 50, 'RVC-SYM-GEN': 38, 'RVC-SYM-PSY': 40, 'RVC-DIS': 92, 'RVC-
TEST': 12, 'RVC-DIAG': 47, 'RVC-TREAT': 54, 'RVC-ADMIN': 14, 'RVC-UNCL': 6}
Total prototypes: 693

Segment column health:
rows=41,322 | with>=1 segment=41,302 | unique segments=3,154
```

```
Probe classifications:
shortness of breath -> [('shortness of breath', 'RVC-SYM-RESP', 0.8401217460632324, True)]
bradycardia -> [('bradycardia', 'RVC-SYM-CIRC', 0.6315359473228455, True)]
positive blood culture -> [('positive blood culture', 'RVC-TEST', 0.8695230484008789, True)]
alter mental status -> [('alter mental status', 'RVC-SYM-NERV', 0.7240737080574036, True)]
abdominal pain / shortness of breath -> [('abdominal pain', 'RVC-SYM-DIG', 0.830388069152832
SYM-RESP', 0.8401217460632324, True)]
right eye redness / pain -> [('right eye redness', 'RVC-SYM-EYE', 0.9090079069137573, True),
SYM-DIG', 0.7539336085319519, False)]

Regression summary
------------------
Input rows: 41,322
Output rows: 41,322
Row count preserved: True

Top RFV1_name counts:
RFV1_name
Symptom - Respiratory         8094
Symptom - Nervous             6579
Symptom - Digestive           5782
Symptom - Circulatory         4186
Diseases (patient-stated)     3324
Injuries & adverse effects    2927
Symptom - Genitourinary       2596
Symptom - General             2588
Symptom - Eye/Ear             1027
Symptom - Skin/Hair/Nails      988
Uncodable/Unknown              928
Symptom - Musculoskeletal      714
Administrative                 650
Abnormal test result           399
Treatment/Medication           284

Override usage:
Segments total: 63,744
Segments overridden: 46,500
Segment override rate: 0.7295
Rows total: 41,322
Rows with >=1 override: 32,954
Row override rate: 0.7975
```

```
Primary-label completeness:
blank_rfv1_total: 0
blank_rfv1_with_nonempty_segments: 0
all_uncodable_rows: 928

Top uncodable segments (post-fix):
  54 | unknown - cc
  45 | alter of
  41 | mcc
  31 | gtube renal
  24 | unable to ambulate
  21 | iph
  20 |
  18 | clot fistula
  16 | mca
  16 | ftt
  15 | unwitnessed small
  15 | icd renal
  14 | aortic dissection
  11 | atrial fibrillation
   9 | intubated heart blood

Optional plots skipped (cfg.run_optional_plots=False).
```

```
cc_missing_render = cc_missing_audit.copy()
if "examples" in cc_missing_render.columns:
    cc_missing_render["examples"] =
 ↪  cc_missing_render["examples"].astype(str).str.slice(0, 180)
print(
    render_latex_longtable(
        cc_missing_render,
        caption="Chief complaint missingness audit (pseudo-missing and
 ↪  true-missing policy).",
        label="tab:classifier_cc_missing_audit",
        index=False,
    )
)
```

```
\begin{longtable}{lrl}
\caption{Chief complaint missingness audit (pseudo-missing and true-missing policy).} \label
\toprule
cc_missing_reason & row_count & examples \\
```

```
\midrule
\endfirsthead
\caption[]{Chief complaint missingness audit (pseudo-missing and true-missing policy).} \\
\toprule
cc_missing_reason & row_count & examples \\
\midrule
\endhead
\midrule
\multicolumn{3}{r}{Continued on next page} \\
\midrule
\endfoot
\bottomrule
\endlastfoot
pseudo_missing_token & 18 & ___; -; N; ___,___; -- \\
valid & 41304 & Dyspnea; Altered mental status; Chest pain; Abd pain; s/p Fall \\
\end{longtable}
```

1.11 10) Export

```
# Purpose: export the NLP-augmented workbook using canonical handoff names
 ↪  and a dated archive copy.
# Why this matters: downstream analysis reads a stable canonical path while
 ↪  preserving run-history snapshots.

canonical_output_path, archive_output_path =
 ↪  resolve_classifier_output_paths(WORK_DIR)
archive_output_path.parent.mkdir(parents=True, exist_ok=True)

df_out.to_excel(canonical_output_path, index=False,
 ↪  sheet_name=cfg.output_sheet_name)
df_out.to_excel(archive_output_path, index=False,
 ↪  sheet_name=cfg.output_sheet_name)

run_date = pd.Timestamp.now().strftime("%Y-%m-%d")
cc_missing_audit_path = archive_output_path.parent / f"{run_date}
 ↪  classifier_cc_missing_audit.csv"
cc_missing_audit.to_csv(cc_missing_audit_path, index=False)

classifier_contract = validate_classifier_contract(df_out,
 ↪  max_rfv=cfg.max_rfv)
classifier_contract["resource_report"] = resource_report
```

```
contract_report_path = archive_output_path.parent / f"{run_date}
 ↪  classifier_contract_report.json"
contract_report_path.write_text(json.dumps(classifier_contract, indent=2))
contract_artifact_payload = {
    "generated_utc": pd.Timestamp.utcnow().isoformat(),
    "status": classifier_contract["status"],
    "contracts": {"classifier": classifier_contract},
    "findings": classifier_contract["findings"],
}
contract_artifact_paths = write_contract_report(contract_artifact_payload,
 ↪  work_dir=WORK_DIR)

stage_manifest = collect_run_manifest(
    WORK_DIR,
    run_id=f"classifier_{pd.Timestamp.now().strftime('%Y%m%d_%H%M%S')}",
)
stage_manifest["stage"] = "classifier"
stage_manifest["model_metadata"] = {
    "model_name": metadata.get("model_name"),
    "spacy_model": metadata.get("spacy_model"),
    "device": metadata.get("device"),
    "appendix_rows_used": metadata.get("appendix_rows_used"),
    "prototype_total": metadata.get("prototype_total"),
}
stage_manifest["resource_verification"] = resource_report
stage_manifest["outputs"] = {
    "canonical_output_path": str(canonical_output_path),
    "archive_output_path": str(archive_output_path),
    "cc_missing_audit_path": str(cc_missing_audit_path),
    "classifier_contract_report_path": str(contract_report_path),
}
stage_manifest_path = archive_output_path.parent / f"{run_date}
 ↪  classifier_run_manifest.json"
stage_manifest_path.write_text(json.dumps(stage_manifest, indent=2))

contract_mode = os.getenv("PIPELINE_CONTRACT_MODE", "fail").strip().lower()
if contract_mode not in {"fail", "warn"}:
    raise ValueError("PIPELINE_CONTRACT_MODE must be one of {'fail',
     ↪  'warn'}")
if classifier_contract["status"] == "fail" and contract_mode == "fail":
    raise ValueError(
        "Classifier contract failed. See report at "
        f"{contract_report_path} for details."
```

```
    )

print(f"Wrote canonical output: {canonical_output_path}")
print(f"Wrote archive output: {archive_output_path}")
print(f"Wrote CC missingness audit: {cc_missing_audit_path}")
print(f"Wrote classifier contract report: {contract_report_path}")
print(f"Wrote classifier run manifest: {stage_manifest_path}")
print(f"Wrote pipeline contract artifact:
 ↪  {contract_artifact_paths['contract_report_path']}")
if contract_artifact_paths["failed_contract_path"].exists():
    print(f"Wrote failed contract artifact:
     ↪  {contract_artifact_paths['failed_contract_path']}")
print(f"Classifier contract status: {classifier_contract['status']}")
print(
    "Schema summary: "
    f"rows={len(df_out):,}, cols={df_out.shape[1]:,}, "
    f"rfv_slots={cfg.max_rfv}"
)
print(f"Export complete: {canonical_output_path}")
```

```
Wrote canonical output: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke
CC-NLP/MIMIC tabular data/MIMICIV all with CC_with_NLP.xlsx
Wrote archive output: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Re
CC-NLP/MIMIC tabular data/prior runs/2026-02-17 MIMICIV all with CC_with_NLP.xlsx
Wrote CC missingness audit: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Lo
CC-NLP/MIMIC tabular data/prior runs/2026-02-17 classifier_cc_missing_audit.csv
Wrote classifier contract report: /Users/blocke/Box Sync/Residency Personal Files/Scholarly W
CC-NLP/MIMIC tabular data/prior runs/2026-02-17 classifier_contract_report.json
Wrote classifier run manifest: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work
CC-NLP/MIMIC tabular data/prior runs/2026-02-17 classifier_run_manifest.json
Wrote pipeline contract artifact: /Users/blocke/Box Sync/Residency Personal Files/Scholarly W
CC-NLP/debug/contracts/20260217_084517/contract_report.json
Classifier contract status: pass
Schema summary: rows=41,322, cols=266, rfv_slots=5
Export complete: /Users/blocke/Box Sync/Residency Personal Files/Scholarly Work/Locke Researc
CC-NLP/MIMIC tabular data/MIMICIV all with CC_with_NLP.xlsx
```