```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
OHCA survivorship screening (titles + abstracts) with:
- Locked inclusion regex gate
- Hard exclude for BLS/simulation/manikin training
- Minimal negative semantic prototypes (PCI/cath lab, vasopressors, ECMO)
as down-weights only
- Uses pre-fetched abstracts from an ABSTRACT column (no PubMed calls)
- SBERT (MiniLM) if available; TF-IDF fallback otherwise
Outputs:
  screen_minneg_full.csv      # all rows, annotated
  screen_minneg_filtered.csv  # rows kept after rules
"""

import os
import time
import re
import html
import unicodedata
import json
from typing import List, Dict, Any

import pandas as pd
from tqdm import tqdm

# -------------------- CONFIG --------------------
INPUT_CSV = "abstract_with_doi.csv"     # your input file with TITLE /
ABSTRACT / etc
OUTPUT_PREFIX = "screen_minneg"         # file prefix for outputs
CONTACT_EMAIL = "your.email@example.com"  # optional; only used if PubMed
helpers are used
NCBI_API_KEY = os.getenv("NCBI_API_KEY", "")  # optional, speeds up API
rate limits
TOOL_NAME = "ohca_survivorship_screen_minneg"

# Decision thresholds
SEMANTIC_KEEP_THRESHOLD = 0.41        # raise/lower after calibration
ALLOW_REVIEW_BYPASS = False           # if False, items below threshold
are dropped

REQUIRE_INCLUSION_LEXICAL = True      # keep only if inclusion regex hits
title+abstract
HARD_EXCLUDE_BLS_TRAINING = True      # drop simulation/manikin/CPR
training outright
USE_TITLE_ABSTRACT_FOR_LEX = True     # lexical checks use title+abstract
(recommended)

# Rate limiting & retries (NCBI polite usage, unused now)
DELAY = 0.34
RETRIES = 3

# -------------------- TEXT NORMALISATION --------------------
def norm_text(s: str) -> str:
    if not isinstance(s, str):
        s = "" if s is None else str(s)
    s = unicodedata.normalize("NFKC", s)
```

```
    s = s.replace("\n", " ").replace("\r", " ")
    s = re.sub(r"\s+", " ", s).strip()
    return s

# -------------------- INCLUSION TERMS / REGEX --------------------
INCLUSION_TERMS = [
    "survivor", "survivors", "survivorship",
    "follow-up", "follow up", "post-discharge", "post discharge",
    "long-term", "long term", "chronic phase", "late outcome",
    "quality of life", "hrqol", "health-related quality of life",
    "life satisfaction", "well-being", "wellbeing",
    "neuropsychological", "neuropsychology",
    "cognitive", "cognition", "cognitive impairment", "memory",
"attention",
    "executive function", "executive dysfunction", "planning",
"concentration",
    "fatigue", "mental fatigue", "sleep", "insomnia", "sleep quality",
    "psychological", "anxiety", "depression", "ptsd", "post-traumatic
stress",
    "distress", "mental health",
    "participation", "community reintegration", "social reintegration",
    "return to work", "employment", "work resumption",
    "activities of daily living", "adl",
    "instrumental activities of daily living", "iadl",
    "functional outcome", "functional status",
    "rehabilitation", "rehab", "aftercare",
    "outpatient clinic", "follow-up clinic",
    "caregiver", "carer", "family member", "partner", "spouse",
    "relative", "significant other",
    "care burden", "caregiver burden", "carer burden",
    "support group", "peer support",
    "psychosocial", "emotional", "coping", "adjustment",
    "daily life", "everyday life", "lifestyle",
]

def inc_phrase_to_pattern_exact_first(t: str) -> str:
    """
    Turn an inclusion phrase into a regex:
    - For multiword / hyphenated phrases, match as written (case-
insensitive),
      allowing some flexibility with spaces/hyphens.
    - For single words, allow a few plural-ish / variant forms.
    """
    t = t.strip()
    if " " in t or "-" in t:
        pat = re.escape(t)
        pat = pat.replace(r"\-", r"[-\s]?").replace(r"\ ", r"\s+")
        return pat
    pluralish = {
        "survivor": r"survivor(s)?",
        "survivorship": r"survivorship",
        "follow-up": r"follow[-\s]?up",
        "post-discharge": r"post[-\s]?discharge",
        "long-term": r"long[-\s]?term",
        "caregiver": r"care[-\s]?giver(s)?|carer(s)?",
        "aftercare": r"after[-\s]?care",
        "rehab": r"rehab",
        "outcome": r"outcome(s)?",
```

```
        "resumption": r"resumption|resume|resuming",
    }
    return pluralish.get(t, re.escape(t))

INCLUSION_REGEX = re.compile(
    r"(?i)(" + "|".join(inc_phrase_to_pattern_exact_first(t) for t in
INCLUSION_TERMS) + r")"
)

# -------------------- HARD EXCLUDE: BLS/simulation/manikin training ----
----------------
HARD_TRAINING_EXCLUDE = re.compile(
    r"(?i)\b("
    r"manikin|mannequin|simulation training|skills training|BLS
course|BLS training|"
    r"CPR training|dispatcher[-\s]?assisted CPR|AED training|skills
lab|simulation-based"
    r")\b"
)

# -------------------- MINIMAL NEGATIVE PROTOTYPES (down-weights only) --
------------------
POSITIVE_QUERIES = [
    "Long-term outcomes after cardiac arrest; survivorship; follow-up",
    "Quality of life; HRQoL; health-related quality of life",
    "Neuropsychological and cognitive outcomes; memory; attention;
executive function",
    "Psychological outcomes; anxiety; depression; mental health",
    "Return to work; participation; community reintegration; activities
of daily living",
    "Rehabilitation; outpatient rehab; aftercare; post-discharge
sequelae",
]

NEGATIVE_QUERIES = [
    "PCI; percutaneous coronary intervention; cath lab; coronary
angiography",
    "Vasopressors; norepinephrine; hemodynamic support; ventilator
settings",
    "ECMO; ECLS; cannulation; oxygenator; refractory arrest; transport on
ECMO",
]

# -------------------- OPTIONAL: PubMed helper functions (now unused) ---
----------------
import requests

def entrez_get(url: str, params: Dict[str, Any]) -> str:
    params = dict(params)
    if CONTACT_EMAIL:
        params["email"] = CONTACT_EMAIL
    if NCBI_API_KEY:
        params["api_key"] = NCBI_API_KEY
    params["tool"] = TOOL_NAME
    for attempt in range(1, RETRIES + 1):
        try:
            resp = requests.get(url, params=params, timeout=15)
            if resp.status_code == 200:
```

```
                return resp.text
        except Exception:
            pass
        time.sleep(DELAY * attempt)
    return ""

def esearch_title_to_pmid(title: str, year: str = None) -> str:
    if not title.strip():
        return ""
    q = f'"{title}"[Title]'
    if year and str(year).isdigit():
        q += f" AND {year}[DP]"
    xml = entrez_get(
        "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi",
        {"db": "pubmed", "term": q, "retmode": "xml", "retmax": 1},
    )
    if not xml:
        return ""
    m = re.search(r"<Id>(\d+)</Id>", xml)
    return m.group(1) if m else ""

def efetch_pmids(pmids: List[str]) -> Dict[str, Dict[str, Any]]:
    out: Dict[str, Dict[str, Any]] = {}
    if not pmids:
        return out
    chunk = 100
    for i in range(0, len(pmids), chunk):
        sub = pmids[i : i + chunk]
        xml = entrez_get(
            "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi",
            {"db": "pubmed", "id": ",".join(sub), "retmode": "xml"},
        )
        if not xml:
            continue
        blocks = xml.split("<PubmedArticle>")
        for blk in blocks:
            if "<PMID" not in blk:
                continue
            mpm = re.search(r"<PMID[^>]*>(\d+)</PMID>", blk)
            if not mpm:
                continue
            pmid = mpm.group(1)
            title = re.search(r"<ArticleTitle>(.*?)</ArticleTitle>", blk,
flags=re.S | re.I)
            title = re.sub("<.*?>", "", title.group(1)).strip() if title
else ""
            abstr = " ".join(
                re.findall(r"<AbstractText[^>]*>(.*?)</AbstractText>",
blk, flags=re.S | re.I)
            )
            abstr = re.sub("<.*?>", "", abstr).strip()
            journal = re.search(r"<Title>(.*?)</Title>", blk, flags=re.S
| re.I)
            journal = re.sub("<.*?>", "", journal.group(1)).strip() if
journal else ""
            ym = re.search(
```

```
r"<JournalIssue>.*?<PubDate>.*?(?:<Year>(\d{4})</Year>|<MedlineDate>(\d{4
}))",
                blk,
                flags=re.S | re.I,
            )
            year = ""
            if ym:
                for g in ym.groups():
                    if g:
                        year = g
                        break
            mesh = "; ".join(

re.findall(r"<DescriptorName[^>]*>(.*?)</DescriptorName>", blk,
flags=re.S | re.I)
            )
            out[pmid] = {
                "pmid": pmid,
                "title": html.unescape(title),
                "abstract": html.unescape(abstr),
                "mesh": html.unescape(mesh),
                "journal": html.unescape(journal),
                "year": year,
            }
    return out

# -------------------- SEMANTIC SCORER (multi-prototype; SBERT -> TF-IDF
fallback) --------------------
def get_semantic_scorer_multi(positives: List[str], negatives:
List[str]):
    try:
        from sentence_transformers import SentenceTransformer
        import numpy as np

        model_name = "all-MiniLM-L6-v2"
        model = SentenceTransformer(model_name)

        # Encode all prototypes (positive + negative)
        all_protos = positives + negatives
        proto_embs = model.encode(
            all_protos,
            show_progress_bar=False,
            convert_to_numpy=True,
        )
        pos_embs = proto_embs[: len(positives)]
        neg_embs = proto_embs[len(positives) :]

        # Centroids
        pos_centroid = pos_embs.mean(axis=0)
        neg_centroid = neg_embs.mean(axis=0)

        def score_fn(texts: List[str]) -> List[float]:
            # Accept either a single string or a list of strings
            if isinstance(texts, str):
                txts = [texts]
            else:
                txts = list(texts)
```

```
            emb = model.encode(
                txts,
                show_progress_bar=False,
                convert_to_numpy=True,
            )

            def cos_sim(a, b):
                # a: batch of embeddings, b: single vector
                denom = (np.linalg.norm(a, axis=-1) * np.linalg.norm(b) +
1e-9)
                return (a @ b) / denom

            pos = cos_sim(emb, pos_centroid)
            neg = cos_sim(emb, neg_centroid)

            # Positive minus a weighted negative similarity
            return list(pos - 0.7 * neg)

        return score_fn, f"SBERT::{model_name}"

    except Exception:
        # TF-IDF fallback if sentence-transformers or numpy not available
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.metrics.pairwise import cosine_similarity

        all_protos = positives + negatives
        vec = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
        vec.fit(all_protos)

        pos_vec = vec.transform(positives).mean(axis=0)
        neg_vec = vec.transform(negatives).mean(axis=0)

        def score_fn(texts: List[str]) -> List[float]:
            if isinstance(texts, str):
                txts = [texts]
            else:
                txts = list(texts)

            X = vec.transform(txts)
            pos_sim = cosine_similarity(X, pos_vec)
            neg_sim = cosine_similarity(X, neg_vec)

            return list((pos_sim - 0.7 * neg_sim).ravel())

        return score_fn, "TFIDF_fallback"

# -------------------- MAIN PIPELINE --------------------
def main():
    print(f"Loading {INPUT_CSV} ...")
    df = pd.read_csv(INPUT_CSV)

    # Identify title / year columns
    title_col = next((c for c in df.columns if c.lower() in ["title",
"ti", "article_title"]), None)
    if not title_col:
        title_col = next((c for c in df.columns if "title" in c.lower()),
None)
```

```
    if not title_col:
        raise RuntimeError("Could not identify a title column. Please
rename to 'Title' or similar.")

    year_col = next((c for c in df.columns if c.lower() in ["year",
"pub_year", "py"]), None)
    if not year_col:
        year_col = next((c for c in df.columns if "year" in c.lower()),
None)

    pmid_col = next((c for c in df.columns if c.lower() in ["pmid",
"pubmedid", "pubmed_id"]), None)

    titles = df[title_col].astype(str).map(norm_text)
    years = df[year_col].astype(str) if year_col else pd.Series([None] *
len(df))

    # Use locally available abstracts instead of fetching from PubMed
    abs_cands = ["ABSTRACT", "Abstract", "abstract", "ABSTRACT_TEXT",
"AbstractText"]
    abstract_col = next((c for c in df.columns if c in abs_cands), None)
    if abstract_col is None:
        abstract_col = next((c for c in df.columns if "abstract" in
c.lower()), None)
    if abstract_col is None:
        raise RuntimeError("Could not find an ABSTRACT column in the
input CSV.")

    abs_series = df[abstract_col].fillna("").astype(str)
    df["_abstract"] = abs_series.map(norm_text)
    # Placeholder columns for compatibility with earlier PubMed-based
version
    df["_mesh"] = ""
    df["_journal"] = ""
    df["_pmid_year"] = years

    # Lexical checks
    if USE_TITLE_ABSTRACT_FOR_LEX:
        TA = (titles.fillna("") + " " +
df["_abstract"].fillna("")).map(norm_text)
    else:
        TA = titles.map(norm_text)

    def matched_terms(text: str, pat: re.Pattern) -> str:
        return "; ".join(sorted({m.group(0).lower() for m in
pat.finditer(text)}))

    df["_incl_lex_match"] = TA.apply(lambda s:
bool(INCLUSION_REGEX.search(s)))
    df["_incl_lex_terms"] = TA.apply(lambda s: matched_terms(s,
INCLUSION_REGEX))

    if HARD_EXCLUDE_BLS_TRAINING:
        df["_hard_blstrain_exclude"] = TA.apply(lambda s:
bool(HARD_TRAINING_EXCLUDE.search(s)))
    else:
        df["_hard_blstrain_exclude"] = False
```

```
    # Semantic scorer
    score_fn, model_name = get_semantic_scorer_multi(POSITIVE_QUERIES,
NEGATIVE_QUERIES)
    print(f"Using semantic model: {model_name}")
    df["_semantic_score"] = score_fn(df["_abstract"].fillna("").tolist())

    # Final keep / drop rules
    keep_mask = pd.Series([True] * len(df))

    if REQUIRE_INCLUSION_LEXICAL:
        keep_mask &= df["_incl_lex_match"]

    if HARD_EXCLUDE_BLS_TRAINING:
        keep_mask &= ~df["_hard_blstrain_exclude"]

    above_thresh = df["_semantic_score"] >= SEMANTIC_KEEP_THRESHOLD
    if not ALLOW_REVIEW_BYPASS:
        keep_mask &= above_thresh

    df["_final_semantic_keep"] = keep_mask

    # Outputs
    full_out = f"{OUTPUT_PREFIX}_full.csv"
    filt_out = f"{OUTPUT_PREFIX}_filtered.csv"

    print(f"Writing full annotated output to {full_out}")
    df.to_csv(full_out, index=False)

    kept_df = df[df["_final_semantic_keep"]].copy()
    print(f"Writing filtered (kept) records to {filt_out}
(n={len(kept_df)})")
    kept_df.to_csv(filt_out, index=False)

    # Summary
    summary = {
        "records": len(df),
        "pmids_resolved": int(df[pmid_col].notna().sum()) if pmid_col
else 0,
        "incl_lex_matches": int(df["_incl_lex_match"].sum()),
        "hard_excluded_blstrain": int(df["_hard_blstrain_exclude"].sum())
if HARD_EXCLUDE_BLS_TRAINING else 0,
        "kept_after_semantic": int(df["_final_semantic_keep"].sum()),
        "semantic_model": model_name,
        "threshold": SEMANTIC_KEEP_THRESHOLD,
        "outputs": {
            "full": full_out,
            "filtered": filt_out,
        },
    }
    print(json.dumps(summary, indent=2))

if __name__ == "__main__":
    main()
```