Dense Mineral Points Generation (Polygon-Based)

```
Dense Mineral Points Generation (Polygon-based)

Creates dense point samples inside target polygon feature classes (e.g., mineral polygons).
The script scans polygon FCs with a given prefix (default: 'o'), then generates a dense point grid
inside each polygon using one of three strategies depending on polygon area:
  1) Very small polygons: centroid only
  2) Small polygons: simple grid (spacing-based)
  3) Medium polygons: fishnet-based sampling

Requirements:
- ArcGIS Pro Python environment with arcpy available.
- Polygon feature classes stored in the workspace GDB.

from __future__ import annotations

import os
import math
import logging
from dataclasses import dataclass
from typing import List, Optional

import arcpy


# ================================
# Section 1: Logging
# ================================

def configure_logging(level: int = logging.INFO) -> None:
    """Configure console logging."""
    logging.basicConfig(
        level=level,
        format="%(asctime)s | %(levelname)s | %(message)s",
    )


# ================================
# Section 2: Configuration
# ================================

@dataclass(frozen=True)
class Config:
    """Runtime configuration."""
    workspace_gdb: str
    polygon_prefix: str = "o"
    overwrite_output: bool = True

    # Strategy thresholds (area in square meters)
    centroid_area_threshold: float = 10.0
    grid_area_threshold: float = 100.0

    # Grid strategy spacing (meters) for small polygons
    simple_grid_spacing_m: float = 0.25
    # Fishnet constraints (meters)
    fishnet_cell_min_m: float = 0.5
    fishnet_cell_max_m: float = 5.0


# ================================
# Section 3: Utilities
# ================================

def safe_delete(dataset: str) -> None:
    """Delete a dataset if it exists."""
    try:
        if arcpy.Exists(dataset):
            arcpy.Delete_management(dataset)
    except Exception:
        # Best-effort cleanup; do not raise.
        pass


def ensure_overwrite(overwrite: bool) -> None:
    """Set ArcPy overwrite behavior."""
    arcpy.env.overwriteOutput = bool(overwrite)


def set_workspace(workspace_gdb: str) -> None:
    """Set ArcPy workspace."""
    arcpy.env.workspace = workspace_gdb


def list_target_polygons(prefix: str) -> List[str]:
    """
    List polygon feature classes starting with a prefix.

    Returns:
        List of feature class names (not full paths) in the current workspace.
    """
    targets: List[str] = []
    for fc in (arcpy.ListFeatureClasses() or []):
        if not fc.startswith(prefix):
            continue
        try:
            desc = arcpy.Describe(fc)
            if getattr(desc, "shapeType", "").lower() == "polygon":
                targets.append(fc)
        except Exception:
            continue
    return targets


def polygon_metrics(polygon_fc: str) -> tuple[float, float, float]:
    """
    Approximate polygon metrics from extent.

    Returns:
        (width_m, height_m, area_m2) computed from the feature extent.
    """
    desc = arcpy.Describe(polygon_fc)
    extent = desc.extent
    width = float(extent.XMax - extent.XMin)
    height = float(extent.YMax - extent.YMin)
    area = width * height
    return width, height, area


# ================================
# Section 4: Point Generation Methods
# ================================

def create_dense_points_in_small_polygon(
    polygon_fc: str,
    output_fc: str,
    points_per_meter: float = 0.5,
) -> int:
    """
    Create a dense regular grid of points for a small polygon, then keep only points within polygon.

    Args:
        polygon_fc: Input polygon feature class.
        output_fc: Output point feature class.
        points_per_meter: Point density per meter (default 0.5 => spacing 2m).

    Returns:
        Number of points created in the output.
    """
    desc = arcpy.Describe(polygon_fc)
    extent = desc.extent

    width = float(extent.XMax - extent.XMin)
    height = float(extent.YMax - extent.YMin)

    logging.info("Polygon extent size: %.2f x %.2f meters", width, height)
    logging.info("Extent area (approx): %.2f m^2", width * height)

    if width < 1.0 or height < 1.0:
        logging.info("Very small polygon: creating centroid only.")
        safe_delete(output_fc)
        arcpy.FeatureToPoint_management(polygon_fc, output_fc, "CENTROID")
        return 1

    if points_per_meter <= 0:
        raise ValueError("points_per_meter must be > 0")

    point_spacing = 1.0 / points_per_meter
    cols = max(2, int(math.ceil(width / point_spacing)))
    rows = max(2, int(math.ceil(height / point_spacing)))
    logging.info("Grid: %d x %d = %d points (spacing=%.2f m)", rows, cols, rows * cols, point_spacing)

    # Generate coordinates
    x_coords: List[float] = []
    y_coords: List[float] = []
    for col in range(cols):
        x = float(extent.XMin + col * point_spacing)
        if x <= extent.XMax:
            x_coords.append(x)

    for row in range(rows):
        y = float(extent.YMin + row * point_spacing)
        if y <= extent.YMax:
            y_coords.append(y)

    points = [arcpy.Point(x, y) for x in x_coords for y in y_coords]

    # Create temp point FC in memory
    temp_fc = arcpy.CreateFeatureclass_management(
        "in_memory",
        "grid_temp_points",
        "POINT",
        spatial_reference=desc.spatialReference,
    )[0]

    with arcpy.da.InsertCursor(temp_fc, ["SHAPE@"]) as cur:
        for p in points:
            cur.insertRow([p])

    # Select points within polygon
    points_layer = "in_memory/points_layer"
    arcpy.MakeFeatureLayer_management(temp_fc, points_layer)
    arcpy.SelectLayerByLocation_management(points_layer, "WITHIN", polygon_fc)

    count = int(arcpy.GetCount_management(points_layer)[0])
    safe_delete(output_fc)

    if count > 0:
        arcpy.CopyFeatures_management(points_layer, output_fc)
    else:
        logging.info("No points within polygon after selection: creating centroid.")
        arcpy.FeatureToPoint_management(polygon_fc, output_fc, "CENTROID")
        count = 1

    safe_delete(points_layer)
    safe_delete(temp_fc)
    return count


def create_points_using_fishnet(
    polygon_fc: str,
    output_fc: str,
    cell_size: float = 2.0,
) -> int:
    """
    Use ArcPy CreateFishnet and keep points within polygon.

    Args:
        polygon_fc: Input polygon feature class.
        output_fc: Output point feature class.
        cell_size: Fishnet cell size in meters.
    Returns:
        Number of output points.
    """
    desc = arcpy.Describe(polygon_fc)
    extent = desc.extent

    width = float(extent.XMax - extent.XMin)
    height = float(extent.YMax - extent.YMin)

    # Adjust for very small polygons
    if width < cell_size or height < cell_size:
        cell_size = max(0.1, min(width, height) / 2.0)

    rows = max(2, int(math.ceil(height / cell_size)))
    cols = max(2, int(math.ceil(width / cell_size)))

    logging.info("Creating fishnet: %dx%d (cell_size=%.3f m)", rows, cols, cell_size)

    fishnet_poly = arcpy.CreateFishnet_management(
        out_feature_class="in_memory/fishnet_poly",
        origin_coord=f"{extent.XMin} {extent.YMin}",
        y_axis_coord=f"{extent.XMin} {extent.YMin + cell_size}",
        cell_width=cell_size,
        cell_height=cell_size,
        number_rows=rows,
        number_columns=cols,
        labels="LABELS",
        geometry_type="POLYGON",
    )[0]

    # Convert fishnet polygons to label points (centroids)
    fishnet_labels = "in_memory/fishnet_labels"
    arcpy.FeatureToPoint_management(fishnet_poly, fishnet_labels, "CENTROID")

    # Select points within polygon
    points_inside = arcpy.SelectLayerByLocation_management(fishnet_labels, "WITHIN", polygon_fc)

    count = int(arcpy.GetCount_management(points_inside)[0])
    safe_delete(output_fc)

    if count > 0:
        arcpy.CopyFeatures_management(points_inside, output_fc)
    else:
        logging.info("No fishnet points inside polygon: creating centroid.")
        arcpy.FeatureToPoint_management(polygon_fc, output_fc, "CENTROID")
        count = 1

    safe_delete(points_inside)
    safe_delete(fishnet_labels)
    safe_delete(fishnet_poly)
    return count


def simple_grid_points(
    polygon_fc: str,
    output_fc: str,
    spacing: float = 0.5,
) -> int:
    """
    Simple spacing-based grid inside extent; then keep only points within polygon.

    Args:
        polygon_fc: Input polygon feature class.
        output_fc: Output point feature class.
        spacing: Grid spacing in meters.

    Returns:
        Number of output points.
    """
    if spacing <= 0:
        raise ValueError("spacing must be > 0")

    desc = arcpy.Describe(polygon_fc)
    extent = desc.extent

    x_min, y_min = float(extent.XMin), float(extent.YMin)
    x_max, y_max = float(extent.XMax), float(extent.YMax)

    # Create a temp FC for all points (will be filtered by spatial selection)
    temp_fc = arcpy.CreateFeatureclass_management(
        arcpy.env.workspace,
        "temp_grid_points__to_delete",
        "POINT",
        spatial_reference=desc.spatialReference,
    )[0]

    points: List[arcpy.Point] = []
    x = x_min
    while x <= x_max:
        y = y_min
        while y <= y_max:
            points.append(arcpy.Point(x, y))
            y += spacing
        x += spacing

    with arcpy.da.InsertCursor(temp_fc, ["SHAPE@"]) as cur:
        for p in points:
            cur.insertRow([p])

    layer = "temp_points_layer"
    arcpy.MakeFeatureLayer_management(temp_fc, layer)
    arcpy.SelectLayerByLocation_management(layer, "WITHIN", polygon_fc)

    safe_delete(output_fc)
    arcpy.CopyFeatures_management(layer, output_fc)

    count = int(arcpy.GetCount_management(output_fc)[0])

    safe_delete(layer)
    safe_delete(temp_fc)
    return count
```

```
# ================================
# Section 5: Main Runner
# ================================

def choose_method_and_generate(
    polygon_fc: str,
    cfg: Config,
) -> tuple[str, int]:
    """
    Choose a generation method based on polygon extent area and generate points.

    Returns:
        (output_fc_name, points_count)
    """
    width, height, area = polygon_metrics(polygon_fc)
    logging.info("Processing %s | size=%.1f x %.1f m | area=%.1f m^2", polygon_fc, width, height, area)

    output_fc = f"{polygon_fc}_DensePoints"
    safe_delete(output_fc)

    if area <= cfg.centroid_area_threshold:
        logging.info("Area <= %.1f: centroid-only.", cfg.centroid_area_threshold)
        arcpy.FeatureToPoint_management(polygon_fc, output_fc, "CENTROID")
        return output_fc, 1

    if area <= cfg.grid_area_threshold:
        logging.info("Area <= %.1f: simple grid (spacing=%.3f m).", cfg.grid_area_threshold,
cfg.simple_grid_spacing_m)
        count = simple_grid_points(polygon_fc, output_fc, spacing=cfg.simple_grid_spacing_m)
        return output_fc, count

    # Fishnet cell size heuristic (similar to the original logic)
    cell_size = math.sqrt(area) / 5.0
    cell_size = max(cfg.fishnet_cell_min_m, min(cell_size, cfg.fishnet_cell_max_m))
    logging.info("Fishnet method (cell_size=%.3f m).", cell_size)

    count = create_points_using_fishnet(polygon_fc, output_fc, cell_size=cell_size)
    return output_fc, count


def main(cfg: Config) -> int:
    """Main entrypoint."""
    configure_logging()
    set_workspace(cfg.workspace_gdb)
    ensure_overwrite(cfg.overwrite_output)

    logging.info("Workspace: %s", cfg.workspace_gdb)
    logging.info("Polygon prefix: %s", cfg.polygon_prefix)

    targets = list_target_polygons(cfg.polygon_prefix)
    logging.info("Found %d target polygon FC(s).", len(targets))

    if not targets:
        logging.error("No target polygons found. Nothing to do.")
        return 2

    successful = 0
    for idx, poly_fc in enumerate(targets, 1):
        logging.info("(%d/%d) %s", idx, len(targets), poly_fc)
        try:
            out_fc, count = choose_method_and_generate(poly_fc, cfg)
            logging.info("Created %s (%d points).", out_fc, count)
            successful += 1
        except Exception as ex:
            logging.exception("Failed processing %s: %s", poly_fc, str(ex))
            # Fallback: centroid-only
            try:
                out_fc = f"{poly_fc}_DensePoints"
                safe_delete(out_fc)
                arcpy.FeatureToPoint_management(poly_fc, out_fc, "CENTROID")
                logging.info("Fallback centroid created: %s", out_fc)
                successful += 1
            except Exception:
                logging.exception("Fallback centroid also failed for %s.", poly_fc)

    dense_fcs = arcpy.ListFeatureClasses("*_DensePoints") or []
    logging.info("Summary: success=%d | fail=%d", successful, max(0, len(targets) - successful))

    if dense_fcs:
        logging.info("Generated DensePoints feature classes:")
        for fc in dense_fcs:
            try:
                cnt = int(arcpy.GetCount_management(fc)[0])
                logging.info(" - %s: %d points", fc, cnt)
            except Exception:
                logging.info(" - %s: count unavailable", fc)

    return 0 if successful > 0 else 1



if __name__ == "__main__":
    # Update this path to your GDB workspace
    CONFIG = Config(
        workspace_gdb=r"M:\Reza\Survey\WGIS\P24_GOSAL\Gosal\Gosal.gdb",
        polygon_prefix="o",
        overwrite_output=True,
    )
    raise SystemExit(main(CONFIG))
```

Non-Mineral Points Generation (Distance-Based Filtering Logic)

```
Non-mineral points generation (Fixed logic)

Logic (distance to mineral points):
  0-10 m   : delete 100%
  10-20 m  : delete 80%
  20-35 m  : delete 60%
  35-50 m  : delete 40%
  50-70 m  : delete 20%
  >=70 m   : delete 0% (keep all)

Grid spacing: 5 meters (configurable)
"""

from __future__ import annotations

import os
import re
import math
import time
import random
import logging
from dataclasses import dataclass
from typing import List, Tuple, Optional

import arcpy


# ----------------------------
# Config
# ----------------------------

@dataclass(frozen=True)
class NMConfig:
    base_workspace: str                # folder workspace (not necessarily a gdb)
    input_raster_gdb: str              # lithology rasters gdb
    sample_points_gdb: str             # mineral points gdb (o*)
    output_gdb: str                    # where to write n* points (can be same as input)

    raster_filter_enabled: bool = True
    raster_filter_text: str = "ColorRaster"  # or "Reclassify", etc.

    grid_spacing_m: float = 5.0

    # (min_dist, max_dist, delete_ratio)
    distance_zones: Tuple[Tuple[float, float, float], ...] = (
        (0.0, 10.0, 1.00),
        (10.0, 20.0, 0.80),
        (20.0, 35.0, 0.60),
        (35.0, 50.0, 0.40),
        (50.0, 70.0, 0.20),
    )

    # systematic removal: every k-th point approx
    removal_method: str = "systematic"  # "systematic" or "random"
    seed: int = 42


def configure_logging(level=logging.INFO) -> None:
    logging.basicConfig(level=level, format="%(asctime)s | %(levelname)s | %(message)s")


def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def ensure_gdb(gdb_path: str) -> None:
    if arcpy.Exists(gdb_path):
        return
    folder = os.path.dirname(gdb_path)
    name = os.path.basename(gdb_path)
    arcpy.CreateFileGDB_management(folder, name)


def extract_height_from_name(name: str) -> Optional[str]:
    """
    Extract height token like 12_5 or 15 from names.
    Returns normalized string with '.' instead of '_' (e.g., '12.5').
    """
    m = re.search(r"(?:f|b)?(\d+(?:_\d+)?)", name)
    if not m:
        return None
    return m.group(1).replace("_", ".")


def list_rasters(cfg: NMConfig) -> List[dict]:
    arcpy.env.workspace = cfg.input_raster_gdb
    rasters = arcpy.ListRasters() or []
    out = []
    for r in rasters:
        if cfg.raster_filter_enabled:
            if cfg.raster_filter_text and (cfg.raster_filter_text not in r):
                continue
        h = extract_height_from_name(r) or r
        out.append({"name": r, "path": os.path.join(cfg.input_raster_gdb, r), "height": h})
    return out


def list_mineral_point_fcs(cfg: NMConfig) -> List[dict]:
    arcpy.env.workspace = cfg.sample_points_gdb
    fcs = arcpy.ListFeatureClasses("o*", "Point") or []
    out = []
    for fc in fcs:
        # height often appears as o12_5 etc
        m = re.search(r"o(\d+(?:_\d+)?)", fc)
        if m:
            h = m.group(1).replace("_", ".")
        else:
            h = extract_height_from_name(fc) or fc
        out.append({"name": fc, "path": os.path.join(cfg.sample_points_gdb, fc), "height": h})
    return out


def match_pairs(rasters: List[dict], points: List[dict]) -> List[dict]:
    pairs = []
    for r in rasters:
        for p in points:
            if r["height"] == p["height"]:
                pairs.append({"height": r["height"], "raster": r, "points": p})
    return pairs


def _systematic_keep_mask(n: int, delete_ratio: float) -> List[bool]:
    """
    Return list[bool] of length n indicating keep(True)/delete(False)
    for systematic deletion. Approximate delete_ratio.
    """
    if n <= 0:
        return []
    if delete_ratio <= 0:
        return [True] * n
    if delete_ratio >= 1:
        return [False] * n

    delete_count = int(round(n * delete_ratio))
    keep = [True] * n
    if delete_count <= 0:
        return keep

    step = n / delete_count
    # delete indices: 0, step, 2*step, ...
    idx = 0.0
    deleted = 0
    while deleted < delete_count:
        i = int(idx)
        if i >= n:
            break
        keep[i] = False
        deleted += 1
        idx += step
    return keep


def create_fixed_nonmineral_points(
    raster_path: str,
    mineral_points_fc: str,
    output_fc: str,
    cfg: NMConfig,
) -> tuple[str, int, int]:
    """
    Returns (output_fc, final_count, initial_count)
    """
    random.seed(cfg.seed)

    raster = arcpy.Raster(raster_path)
    desc = arcpy.Describe(raster)
    extent = desc.extent
    sr = desc.spatialReference

    safe_delete(output_fc)
    arcpy.CreateFeatureclass_management(os.path.dirname(output_fc),      os.path.basename(output_fc),      "POINT",
spatial_reference=sr)

    width = float(extent.XMax - extent.XMin)
    height = float(extent.YMax - extent.YMin)

    cols = int(width / cfg.grid_spacing_m) + 1
    rows = int(height / cfg.grid_spacing_m) + 1

    # insert grid points
    points = []
    for r in range(rows):
        y = float(extent.YMin + r * cfg.grid_spacing_m)
        for c in range(cols):
            x = float(extent.XMin + c * cfg.grid_spacing_m)
            if x <= extent.XMax and y <= extent.YMax:
                points.append(arcpy.Point(x, y))

    with arcpy.da.InsertCursor(output_fc, ["SHAPE@"]) as cur:
        for pt in points:
            cur.insertRow([pt])

    initial_count = len(points)
    if initial_count == 0:
        return output_fc, 0, 0

    # if mineral points exists, compute near distance
    if arcpy.Exists(mineral_points_fc) and int(arcpy.GetCount_management(mineral_points_fc)[0]) > 0:
        arcpy.Near_analysis(output_fc, mineral_points_fc)

        # keep NEAR_DIST but remove other near fields after copying
        if "DISTANCE" not in [f.name for f in arcpy.ListFields(output_fc)]:
            arcpy.AddField_management(output_fc, "DISTANCE", "DOUBLE")
        arcpy.CalculateField_management(output_fc, "DISTANCE", "!NEAR_DIST!", "PYTHON3")

        # cleanup near fields
        for f in ["NEAR_FID", "NEAR_DIST"]:
            if f in [ff.name for ff in arcpy.ListFields(output_fc)]:
                arcpy.DeleteField_management(output_fc, [f])

        # apply deletions by zones
        oid = arcpy.Describe(output_fc).OIDFieldName

        for (min_d, max_d, delete_ratio) in cfg.distance_zones:
            where = f"DISTANCE >= {min_d} AND DISTANCE < {max_d}"
            layer = "zone_lyr"
            arcpy.MakeFeatureLayer_management(output_fc, layer, where)
            zone_count = int(arcpy.GetCount_management(layer)[0])
            if zone_count <= 0:
                safe_delete(layer)
                continue

            if delete_ratio >= 1.0:
                arcpy.DeleteFeatures_management(layer)
                safe_delete(layer)
                continue
            # fetch OIDs in zone (deterministic order by OID)
            oids = [row[0] for row in arcpy.da.SearchCursor(layer, [oid], sql_clause=(None, f"ORDER BY {oid}"))]

            if cfg.removal_method == "random":
                delete_count = int(round(zone_count * delete_ratio))
                if delete_count > 0:
                    to_delete = set(random.sample(oids, k=min(delete_count, len(oids))))
                else:
                    to_delete = set()
            else:
                mask = _systematic_keep_mask(len(oids), delete_ratio)
                to_delete = {o for o, keep in zip(oids, mask) if not keep}

            if to_delete:
                # chunk deletes to avoid SQL length issues
                to_delete_list = list(to_delete)
                chunk = 900
                for i in range(0, len(to_delete_list), chunk):
                    sub = to_delete_list[i : i + chunk]
                    oids_csv = ",".join(map(str, sub))
                    del_layer = "del_lyr"
                    arcpy.MakeFeatureLayer_management(output_fc, del_layer, f"{oid} IN ({oids_csv})")
                    arcpy.DeleteFeatures_management(del_layer)
                    safe_delete(del_layer)

            safe_delete(layer)

        # remove DISTANCE field (optional)
        if "DISTANCE" in [f.name for f in arcpy.ListFields(output_fc)]:
            arcpy.DeleteField_management(output_fc, ["DISTANCE"])

    final_count = int(arcpy.GetCount_management(output_fc)[0])
    return output_fc, final_count, initial_count


def run(cfg: NMConfig) -> int:
    configure_logging()
    arcpy.env.overwriteOutput = True
    arcpy.env.workspace = cfg.base_workspace

    ensure_gdb(cfg.output_gdb)

    rasters = list_rasters(cfg)
    pts = list_mineral_point_fcs(cfg)
    pairs = match_pairs(rasters, pts)

    logging.info("Rasters: %d | Mineral point FCs: %d | Matched pairs: %d", len(rasters), len(pts), len(pairs))
    if not pairs:
        logging.error("No matched pairs (height match).")
        return 2

    ok = 0
    for i, pair in enumerate(pairs, 1):
        height = str(pair["height"])
        height_str = height.replace(".", "_")
        out_name = f"n{height_str}"
        out_fc = os.path.join(cfg.output_gdb, out_name)

        logging.info("(%d/%d) Height=%s -> %s", i, len(pairs), height, out_name)
        try:
            out_fc, final_n, init_n = create_fixed_nonmineral_points(
                raster_path=pair["raster"]["path"],
                mineral_points_fc=pair["points"]["path"],
                output_fc=out_fc,
                cfg=cfg,
            )
            logging.info("Created %s | initial=%d | final=%d", out_name, init_n, final_n)
            ok += 1
        except Exception as e:
            logging.exception("Failed %s: %s", out_name, str(e))

    return 0 if ok > 0 else 1



if __name__ == "__main__":
    # ---- EDIT THESE PATHS ----
    cfg = NMConfig(
        base_workspace=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal",
        input_raster_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Litho_Rasters.gdb",
        sample_points_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Gosal.gdb",
        output_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Litho_Rasters.gdb",
        raster_filter_enabled=True,
        raster_filter_text="ColorRaster",
        grid_spacing_m=5.0,
        removal_method="systematic",
    )
    raise SystemExit(run(cfg))
```

Bayesian Mineral Potential Modeling

```
Bayesian (class-conditional) model from Mineral/NonMineral points + categorical lithology raster.

Requires:
- ArcGIS Pro (arcpy)
- Spatial Analyst (ExtractValuesToPoints)

Outputs:
- One CSV per matched height token with per-class:
  counts (mineral/nonmineral), prior, likelihoods, posterior.
"""

from __future__ import annotations

import os
import re
import csv
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple
```

```
import arcpy


@dataclass(frozen=True)
class BayesConfig:
    input_raster_gdb: str
    sample_points_gdb: str
    out_folder: str
    raster_filter_enabled: bool = True
    raster_filter_text: str = "ColorRaster"
    mineral_prefix: str = "o"
    nonmineral_prefix: str = "n"
    token_regex: str = r"(\d+(?:_\d+)?)"
    smoothing_eps: float = 1e-9
    overwrite_output: bool = True


def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def extract_token(name: str, token_regex: str) -> Optional[str]:
    m = re.search(token_regex, name)
    return m.group(1) if m else None


def normalize_token(token: str) -> str:
    return token.replace("_", ".")


def list_rasters(cfg: BayesConfig) -> List[Dict[str, str]]:
    arcpy.env.workspace = cfg.input_raster_gdb
    rasters = arcpy.ListRasters() or []
    out: List[Dict[str, str]] = []
    for r in rasters:
        if cfg.raster_filter_enabled and cfg.raster_filter_text:
            if cfg.raster_filter_text not in r:
                continue
        tok_raw = extract_token(r, cfg.token_regex)
        if not tok_raw:
            continue
        out.append(
            {
                "name": r,
                "path": os.path.join(cfg.input_raster_gdb, r),
                "tok_raw": tok_raw,
                "tok_norm": normalize_token(tok_raw),
            }
        )
    return out
```

```
def list_point_fcs(cfg: BayesConfig, prefix: str) -> Dict[str, str]:
    arcpy.env.workspace = cfg.sample_points_gdb
    fcs = arcpy.ListFeatureClasses(f"{prefix}*", "Point") or []
    mp: Dict[str, str] = {}
    for fc in fcs:
        tok_raw = extract_token(fc, cfg.token_regex)
        if not tok_raw:
            continue
        mp[normalize_token(tok_raw)] = os.path.join(cfg.sample_points_gdb, fc)
    return mp


def count_fc(path: str) -> int:
    return int(arcpy.GetCount_management(path)[0])


def extract_values_to_points(points_fc: str, raster_path: str, out_mem_name: str) -> str:
    out_fc = os.path.join("in_memory", out_mem_name)
    safe_delete(out_fc)
    arcpy.sa.ExtractValuesToPoints(points_fc, raster_path, out_fc, "NONE", "VALUE_ONLY")
    return out_fc


def counts_by_class(points_with_vals_fc: str, field: str = "RASTERVALU") -> Dict[int, int]:
    d: Dict[int, int] = {}
    with arcpy.da.SearchCursor(points_with_vals_fc, [field]) as cur:
        for (v,) in cur:
            if v is None:
                continue
            try:
                vv = int(v)
            except Exception:
                continue
            d[vv] = d.get(vv, 0) + 1
    return d


def ensure_out_folder(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def write_csv(out_csv: str, rows: List[List[object]]) -> None:
    with open(out_csv, "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(
            [
                "HeightToken",
                "RasterName",
                "ClassValue",
                "MineralCount",
                "NonMineralCount",
                "Prior_P(M)",
                "Likelihood_P(Class|M)",
                "Likelihood_P(Class|NM)",
                "Posterior_P(M|Class)",
            ]
        )
        w.writerows(rows)


def run_bayes(cfg: BayesConfig) -> int:
    arcpy.env.overwriteOutput = bool(cfg.overwrite_output)
    ensure_out_folder(cfg.out_folder)

    rasters = list_rasters(cfg)
    mineral_map = list_point_fcs(cfg, cfg.mineral_prefix)
    nonmineral_map = list_point_fcs(cfg, cfg.nonmineral_prefix)

    if not rasters:
        return 2

    ok = 0
    for ras in rasters:
        tok = ras["tok_norm"]
        o_fc = mineral_map.get(tok)
        n_fc = nonmineral_map.get(tok)
        if not o_fc or not n_fc:
            continue

        oN = count_fc(o_fc)
        nN = count_fc(n_fc)
        total = oN + nN
        if total <= 0 or oN <= 0 or nN <= 0:
            continue

        prior = oN / total
        eps = float(cfg.smoothing_eps)

        o_vals_fc = extract_values_to_points(o_fc, ras["path"], f"o_val_{tok}".replace(".", "_"))
        n_vals_fc = extract_values_to_points(n_fc, ras["path"], f"n_val_{tok}".replace(".", "_"))

        o_counts = counts_by_class(o_vals_fc)
        n_counts = counts_by_class(n_vals_fc)

        classes = sorted(set(o_counts.keys()) | set(n_counts.keys()))
        if not classes:
            continue

        k = len(classes)
        rows: List[List[object]] = []

        for c in classes:
            o_c = int(o_counts.get(c, 0))
            n_c = int(n_counts.get(c, 0))

            p_c_given_M = (o_c + eps) / (oN + eps * k)
            p_c_given_NM = (n_c + eps) / (nN + eps * k)

            denom = (p_c_given_M * prior) + (p_c_given_NM * (1.0 - prior))
            post = (p_c_given_M * prior) / denom if denom > 0 else 0.0

            rows.append(
                [
                    tok,
                    ras["name"],
                    c,
                    o_c,
                    n_c,
                    prior,
                    p_c_given_M,
                    p_c_given_NM,
                    post,
                ]
            )

        out_csv = os.path.join(cfg.out_folder, f"Bayes_{tok.replace('.', '_')}.csv")
        write_csv(out_csv, rows)
        ok += 1

        safe_delete(o_vals_fc)
        safe_delete(n_vals_fc)

    return 0 if ok > 0 else 1


if __name__ == "__main__":
    cfg = BayesConfig(
        input_raster_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Litho_Rasters.gdb",
        sample_points_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Gosal.gdb",
        out_folder=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Bayes_Reports",
        raster_filter_enabled=True,
        raster_filter_text="ColorRaster",
        smoo
```

Support Vector Machine (SVM) Mineral Potential Modeling

```
Support Vector Machine (SVM) mineral potential modeling from Mineral/NonMineral points + raster predictors.

Inputs:
- Predictor rasters in a GDB (matched by height token)
- Point FCs in a GDB:
    mineral points: o*
    non-mineral points: n*

Outputs (per height token):
- Trained model (joblib)
- Metrics CSV
- Optional probability raster (GeoTIFF)

Requires:
- ArcGIS Pro arcpy + Spatial Analyst
- scikit-learn, numpy, joblib
"""

from __future__ import annotations
```

```
import os
import re
import csv
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

import numpy as np
import arcpy

from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
    confusion_matrix,
)
import joblib


@dataclass(frozen=True)
class SVMConfig:
    predictor_raster_gdb: str
    sample_points_gdb: str
    out_folder: str

    mineral_prefix: str = "o"
    nonmineral_prefix: str = "n"
    raster_filter_enabled: bool = True
    raster_filter_text: str = "ColorRaster"  # set "" to disable filtering

    token_regex: str = r"(\d+(?:_\d+)?)"
    overwrite_output: bool = True

    test_size: float = 0.25
    random_state: int = 42

    # SVM hyperparameters
    kernel: str = "rbf"
    C: float = 10.0
    gamma: str = "scale"
    class_weight: str = "balanced"
    probability: bool = True

    # Data cleaning
    drop_nodata: bool = True

    # Raster prediction
    write_probability_raster: bool = True
    prob_raster_format: str = "TIFF"  # "TIFF" recommended
    prob_raster_nodata: float = -9999.0
```

```
def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def extract_token(name: str, token_regex: str) -> Optional[str]:
    m = re.search(token_regex, name)
    return m.group(1) if m else None


def norm_token(tok: str) -> str:
    return tok.replace("_", ".")


def list_rasters(cfg: SVMConfig) -> List[Dict[str, str]]:
    arcpy.env.workspace = cfg.predictor_raster_gdb
    rasters = arcpy.ListRasters() or []
    out: List[Dict[str, str]] = []
    for r in rasters:
        if cfg.raster_filter_enabled and cfg.raster_filter_text:
            if cfg.raster_filter_text not in r:
                continue
        tok_raw = extract_token(r, cfg.token_regex)
        if not tok_raw:
            continue
        out.append(
            {
                "name": r,
                "path": os.path.join(cfg.predictor_raster_gdb, r),
                "tok_raw": tok_raw,
                "tok_norm": norm_token(tok_raw),
            }
        )
    return out


def list_point_fcs(cfg: SVMConfig, prefix: str) -> Dict[str, str]:
    arcpy.env.workspace = cfg.sample_points_gdb
    fcs = arcpy.ListFeatureClasses(f"{prefix}*", "Point") or []
    mp: Dict[str, str] = {}
    for fc in fcs:
        tok_raw = extract_token(fc, cfg.token_regex)
        if not tok_raw:
            continue
        mp[norm_token(tok_raw)] = os.path.join(cfg.sample_points_gdb, fc)
    return mp
```

```
def count_fc(path: str) -> int:
    return int(arcpy.GetCount_management(path)[0])


def extract_values_to_points(points_fc: str, raster_path: str, out_mem_name: str) -> str:
    out_fc = os.path.join("in_memory", out_mem_name)
    safe_delete(out_fc)
    arcpy.sa.ExtractValuesToPoints(points_fc, raster_path, out_fc, "NONE", "VALUE_ONLY")
    return out_fc


def points_to_xyv(points_fc_with_vals: str, val_field: str = "RASTERVALU") -> Tuple[np.ndarray, np.ndarray]:
    x_list: List[float] = []
    y_list: List[float] = []
    v_list: List[float] = []
    with arcpy.da.SearchCursor(points_fc_with_vals, ["SHAPE@XY", val_field]) as cur:
        for (xy, v) in cur:
            if v is None:
                v_list.append(np.nan)
            else:
                try:
                    v_list.append(float(v))
                except Exception:
                    v_list.append(np.nan)
            x_list.append(float(xy[0]))
            y_list.append(float(xy[1]))
    xy = np.column_stack([np.array(x_list, dtype="float64"), np.array(y_list, dtype="float64")])
    vv = np.array(v_list, dtype="float64")
    return xy, vv


def build_training_set(
    raster_path: str,
    mineral_points_fc: str,
    nonmineral_points_fc: str,
    cfg: SVMConfig,
) -> Tuple[np.ndarray, np.ndarray]:
    oN = count_fc(mineral_points_fc)
    nN = count_fc(nonmineral_points_fc)
    if oN <= 0 or nN <= 0:
        raise RuntimeError("Empty mineral/nonmineral point set.")

    o_vals_fc = extract_values_to_points(mineral_points_fc, raster_path, "svm_o_vals")
    n_vals_fc = extract_values_to_points(nonmineral_points_fc, raster_path, "svm_n_vals")

    _, o_vals = points_to_xyv(o_vals_fc)
    _, n_vals = points_to_xyv(n_vals_fc)

    safe_delete(o_vals_fc)
    safe_delete(n_vals_fc)

    X = np.concatenate([o_vals, n_vals], axis=0).reshape(-1, 1)
    y = np.concatenate([np.ones(o_vals.shape[0], dtype="int32"), np.zeros(n_vals.shape[0], dtype="int32")], axis=0)

    if cfg.drop_nodata:
        mask = ~np.isnan(X[:, 0])
        X = X[mask]
        y = y[mask]

    return X, y


def train_svm(X: np.ndarray, y: np.ndarray, cfg: SVMConfig) -> Tuple[Pipeline, Dict[str, float]]:
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=cfg.test_size, random_state=cfg.random_state, stratify=y
    )

    model = Pipeline(
        steps=[
            ("scaler", StandardScaler(with_mean=True, with_std=True)),
            (
                "svm",
                SVC(
                    kernel=cfg.kernel,
                    C=cfg.C,
                    gamma=cfg.gamma,
                    class_weight=cfg.class_weight,
                    probability=cfg.probability,
                    random_state=cfg.random_state,
                ),
            ),
        ]
    )

    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    if cfg.probability:
        y_prob = model.predict_proba(X_test)[:, 1]
        auc = float(roc_auc_score(y_test, y_prob)) if len(np.unique(y_test)) > 1 else float("nan")
    else:
        y_prob = None
        auc = float("nan")

    cm = confusion_matrix(y_test, y_pred)
    tn, fp, fn, tp = (int(cm[0, 0]), int(cm[0, 1]), int(cm[1, 0]), int(cm[1, 1])) if cm.shape == (2, 2) else (0, 0, 0, 0)

    metrics = {
        "accuracy": float(accuracy_score(y_test, y_pred)),
        "precision": float(precision_score(y_test, y_pred, zero_division=0)),
        "recall": float(recall_score(y_test, y_pred, zero_division=0)),
        "f1": float(f1_score(y_test, y_pred, zero_divisio
```

Deep Self-Attention Mineral Potential Modeling

Deep Self-Attention (Transformer encoder) for mineral potential modeling from Mineral/NonMineral points + raster predictors.

```
Inputs:
- Predictor rasters in a GDB (matched by height token)
- Point FCs in a GDB:
    mineral points: o*
    non-mineral points: n*

This implementation:
- Extracts raster values at point locations for a set of predictor rasters (features)
- Trains a small Transformer encoder (self-attention over feature tokens)
- Saves per-token metrics + trained model
- Optionally produces a probability raster using a reference raster grid

Requires:
- ArcGIS Pro arcpy + Spatial Analyst
- numpy
- torch (PyTorch)

Notes:
- Each sample is a sequence of length F (number of predictors), with 1 value per token.
- The transformer attends across predictors to learn interactions.
"""

from __future__ import annotations

import os
import re
import csv
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

import numpy as np
import arcpy

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader


@dataclass(frozen=True)
class AttnConfig:
    predictor_raster_gdb: str
    sample_points_gdb: str
    out_folder: str

    mineral_prefix: str = "o"
    nonmineral_prefix: str = "n"

    # If rasters in the GDB include multiple types, use filter_text to select predictors
    raster_filter_enabled: bool = True
    raster_filter_text: str = ""  # set to e.g. "ColorRaster" to restrict; empty disables

    token_regex: str = r"(\d+(?:_\d+)?)"
    overwrite_output: bool = True

    # Training
    batch_size: int = 256
    epochs: int = 25
    lr: float = 1e-3
    weight_decay: float = 1e-4
    random_state: int = 42
    test_size: float = 0.25

    # Model
    d_model: int = 64
    n_heads: int = 4
    n_layers: int = 2
    dropout: float = 0.15

    # Data handling
    drop_nodata: bool = True
    nodata_sentinel: float = -9999.0
    standardize: bool = True

    # Raster prediction
    write_probability_raster: bool = True
    reference_raster_name: str = ""  # if empty, first matched raster is used as reference
    prob_raster_nodata: float = -9999.0


def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def extract_token(name: str, token_regex: str) -> Optional[str]:
    m = re.search(token_regex, name)
    return m.group(1) if m else None


def norm_token(tok: str) -> str:
    return tok.replace("_", ".")


def list_rasters(cfg: AttnConfig) -> List[Dict[str, str]]:
    arcpy.env.workspace = cfg.predictor_raster_gdb
    rasters = arcpy.ListRasters() or []
    out: List[Dict[str, str]] = []
    for r in rasters:
        if cfg.raster_filter_enabled and cfg.raster_filter_text:
            if cfg.raster_filter_text not in r:
                continue
        tok_raw = extract_token(r, cfg.token_regex)
        if not tok_raw:
            continue
        out.append(
            {
                "name": r,
                "path": os.path.join(cfg.predictor_raster_gdb, r),
                "tok_raw": tok_raw,
                "tok_norm": norm_token(tok_raw),
            }
        )
    return out


def list_point_fcs(cfg: AttnConfig, prefix: str) -> Dict[str, str]:
    arcpy.env.workspace = cfg.sample_points_gdb
    fcs = arcpy.ListFeatureClasses(f"{prefix}*", "Point") or []
    mp: Dict[str, str] = {}
    for fc in fcs:
        tok_raw = extract_token(fc, cfg.token_regex)
        if not tok_raw:
            continue
        mp[norm_token(tok_raw)] = os.path.join(cfg.sample_points_gdb, fc)
    return mp


def count_fc(path: str) -> int:
    return int(arcpy.GetCount_management(path)[0])


def extract_values_to_points(points_fc: str, raster_paths: List[str], out_mem_name: str) -> str:
    """
    Extract multiple raster values to points. Output includes one field per raster.
    """
    out_fc = os.path.join("in_memory", out_mem_name)
    safe_delete(out_fc)

    # Spatial Analyst tool supports multiple rasters as a semicolon-separated list.
    in_rasters = ";".join(raster_paths)
    arcpy.sa.ExtractMultiValuesToPoints(points_fc, in_rasters, "NONE")

    # Copy to in_memory FC to isolate modifications
    arcpy.CopyFeatures_management(points_fc, out_fc)
    return out_fc


def get_value_fields_from_points(points_fc: str) -> List[str]:
    """
    ExtractMultiValuesToPoints creates fields based on raster names, truncated by GDB rules.
    This function returns numeric fields excluding geometry/OID.
    """
    fields = []
    for f in arcpy.ListFields(points_fc):
        if f.type in ("OID", "Geometry"):
            continue
        if f.name.upper() in ("SHAPE", "SHAPE_LENGTH", "SHAPE_AREA"):
            continue
        if f.type in ("Integer", "SmallInteger", "Single", "Double"):
            fields.append(f.name)
    return fields


def points_to_matrix(points_fc: str, value_fields: List[str], nodata_sentinel: float) -> np.ndarray:
    """
    Returns X with shape (N, F). Missing values are set to np.nan.
    """
    rows = []
    with arcpy.da.SearchCursor(points_fc, value_fields) as cur:
        for r in cur:
            vals = []
            for v in r:
                if v is None:
                    vals.append(np.nan)
                else:
                    try:
                        vals.append(float(v))
                    except Exception:
                        vals.append(np.nan)
            rows.append(vals)
    X = np.array(rows, dtype="float64")
    return X


def standardize_fit(X: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    mu = np.nanmean(X, axis=0)
    sd = np.nanstd(X, axis=0)
    sd = np.where(sd == 0, 1.0, sd)
    return mu, sd


def standardize_apply(X: np.ndarray, mu: np.ndarray, sd: np.ndarray) -> np.ndarray:
    return (X - mu) / sd


def train_test_split_np(X: np.ndarray, y: np.ndarray, test_size: float, seed: int) -> Tuple[np.ndarray, ...]:
    rng = np.random.default_rng(seed)
    idx = np.arange(X.shape[0])
    rng.shuffle(idx)
    n_test = int(round(X.shape[0] * test_size))
    test_idx = idx[:n_test]
    train_idx = idx[n_test:]
    return X[train_idx], X[test_idx], y[train_idx], y[test_idx]


class TabTokenDataset(Dataset):
    def __init__(self, X: np.ndarray, y: np.ndarray):
        self.X = torch.from_numpy(X.astype("float32"))
        self.y = torch.from_numpy(y.astype("int64"))

    def __len__(self) -> int:
        return int(self.X.shape[0])

    def __getitem__(self, i: int):
        return self.X[i], self.y[i]
```

```
class FeatureTokenizer(nn.Module):
    """
    Converts (B, F) into (B, F, d_model) with:
    - value projection from scalar -> d_model
    - learned feature embeddings (like token embeddings)
    """
    def __init__(self, n_features: int, d_model: int, dropout: float):
        super().__init__()
        self.n_features = n_features
        self.d_model = d_model
        self.value_proj = nn.Linear(1, d_model)
        self.feature_emb = nn.Embedding(n_features, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: (B, F)
        b, f = x.shape
        v = x.unsqueeze(-1)  # (B, F, 1)
        v = self.value_proj(v)  # (B, F, d_model)

        feat_ids = torch.arange(f, device=x.device).unsqueeze(0).expand(b, f)  # (B, F)
        e = self.feature_emb(feat_ids)  # (B, F, d_model)
        return self.dropout(v + e)


class AttnClassifier(nn.Module):
    def __init__(self, n_features: int, d_model: int, n_heads: int, n_layers: int, dropout: float):
        super().__init__()
        self.tok = FeatureTokenizer(n_features, d_model, dropout)

        enc_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=n_heads,
            dim_feedforward=4 * d_model,
            dropout=dropout,
            activation="gelu",
            batch_first=True,
            norm_first=True,
        )
        self.enc = nn.TransformerEncoder(enc_layer, num_layers=n_layers)
        self.head = nn.Sequential(
            nn.LayerNorm(d_model),
            nn.Linear(d_model, d_model),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_model, 2),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: (B, F)
        z = self.tok(x)         # (B, F, d_model)
        z = self.enc(z)         # (B, F, d_model)
        z = z.mean(dim=1)       # (B, d_model)
        return self.head(z)     # (B, 2)
```

```
def batch_metrics(y_true: np.ndarray, y_prob: np.ndarray, thr: float = 0.5) -> Dict[str, float]:
    y_pred = (y_prob >= thr).astype("int32")
    tp = int(np.sum((y_true == 1) & (y_pred == 1)))
    tn = int(np.sum((y_true == 0) & (y_pred == 0)))
    fp = int(np.sum((y_true == 0) & (y_pred == 1)))
    fn = int(np.sum((y_true == 1) & (y_pred == 0)))

    acc = float((tp + tn) / max(1, (tp + tn + fp + fn)))
    prec = float(tp / max(1, (tp + fp)))
    rec = float(tp / max(1, (tp + fn)))
    f1 = float((2 * prec * rec) / max(1e-12, (prec + rec)))

    return {"accuracy": acc, "precision": prec, "recall": rec, "f1": f1, "tp": tp, "tn": tn, "fp": fp, "fn": fn}


def write_metrics_csv(out_csv: str, header: List[str], row: List[object]) -> None:
    new = not os.path.exists(out_csv)
    with open(out_csv, "a", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        if new:
            w.writerow(header)
        w.writerow(row)


@torch.no_grad()
def predict_proba(model: nn.Module, X: np.ndarray, batch_size: int, device: str) -> np.ndarray:
    model.eval()
    out = []
    dl = DataLoader(TabTokenDataset(X, np.zeros((X.shape[0],), dtype="int32")), batch_size=batch_size,
shuffle=False)
    for xb, _ in dl:
        xb = xb.to(device)
        logits = model(xb)
        prob = torch.softmax(logits, dim=1)[:, 1]
        out.append(prob.detach().cpu().numpy())
    return np.concatenate(out, axis=0)


def train_model(model: nn.Module, X_tr: np.ndarray, y_tr: np.ndarray, X_te: np.ndarray, y_te: np.ndarray, cfg:
AttnConfig):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model.to(device)

    tr_dl = DataLoader(TabTokenDataset(X_tr, y_tr), batch_size=cfg.batch_size, shuffle=True, drop_last=False)
    te_dl = DataLoader(TabTokenDataset(X_te, y_te), batch_size=cfg.batch_size, shuffle=False, drop_last=False)

    opt = torch.optim.AdamW(model.parameters(), lr=cfg.lr, weight_decay=cfg.weight_decay)
    loss_fn = nn.CrossEntropyLoss()

    best = {"f1": -1.0, "state": None}

    for _ in range(cfg.epochs):
        model.train()
        for xb, yb in tr_dl:
            xb = xb.to(device)
            yb = yb.to(device)
            opt.zero_grad(set_to_none=True)
            logits = model(xb)
            loss = loss_fn(logits, yb)
            loss.backward()
            opt.step()

        y_prob = predict_proba(model, X_te, cfg.batch_size, device=device)
        m = batch_metrics(y_te, y_prob, thr=0.5)
        if m["f1"] > best["f1"]:
            best["f1"] = m["f1"]
            best["state"] = {k: v.detach().cpu().clone() for k, v in model.state_dict().items()}

    if best["state"] is not None:
        model.load_state_dict(best["state"])

    y_prob = predict_proba(model, X_te, cfg.batch_size, device=device)
    m = batch_metrics(y_te, y_prob, thr=0.5)
    return model, m


def probability_raster_from_model(
    model: nn.Module,
    ref_raster_path: str,
    predictor_raster_paths: List[str],
    mu: Optional[np.ndarray],
    sd: Optional[np.ndarray],
    cfg: AttnConfig,
    out_tif: str,
) -> None:
    ras0 = arcpy.Raster(ref_raster_path)
    desc = arcpy.Describe(ras0)
    extent = desc.extent
    cell_w = float(ras0.meanCellWidth)
    cell_h = float(ras0.meanCellHeight)

    arrays = []
    for rp in predictor_raster_paths:
        a = arcpy.RasterToNumPyArray(arcpy.Raster(rp), nodata_to_value=np.nan).astype("float64")
        arrays.append(a)
    stack = np.stack(arrays, axis=-1)  # (H, W, F)

    H, W, F = stack.shape
    flat = stack.reshape(-1, F)

    valid = ~np.any(np.isnan(flat), axis=1)
    prob = np.full((flat.shape[0],), cfg.prob_raster_nodata, dtype="float32")

    if np.any(valid):
        Xv = flat[valid].astype("float64")
        if cfg.standardize and (mu is not None) and (sd is not None):
            Xv = standardize_apply(Xv, mu, sd)
        ypv = predict_proba(model, Xv.astype("float32"), cfg.batch_size, device=("cuda" if torch.cuda.is_available()
else "cpu"))
        prob[valid] = ypv.astype("float32")
    prob_2d = prob.reshape(H, W).astype("float32")
    lower_left = arcpy.Point(float(extent.XMin), float(extent.YMin))

    out_ras = arcpy.NumPyArrayToRaster(
        prob_2d,
        lower_left,
        cell_w,
        cell_h,
        value_to_nodata=cfg.prob_raster_nodata,
    )
    out_ras.spatialReference = desc.spatialReference

    safe_delete(out_tif)
    out_ras.save(out_tif)


def run(cfg: AttnConfig) -> int:
    arcpy.env.overwriteOutput = bool(cfg.overwrite_output)
    ensure_dir(cfg.out_folder)

    rasters = list_rasters(cfg)
    mineral_map = list_point_fcs(cfg, cfg.mineral_prefix)
    nonmineral_map = list_point_fcs(cfg, cfg.nonmineral_prefix)

    metrics_csv = os.path.join(cfg.out_folder, "DeepSelfAttention_metrics.csv")
    header = [
        "HeightToken",
        "n_features",
        "epochs",
        "batch_size",
        "d_model",
        "n_heads",
        "n_layers",
        "dropout",
        "accuracy",
        "precision",
        "recall",
        "f1",
        "tp",
        "tn",
        "fp",
        "fn",
        "n_train",
        "n_test",
    ]

    if not rasters:
        return 2

    # group rasters by token (each token -> multiple predictors)
    by_tok: Dict[str, List[Dict[str, str]]] = {}
    for r in rasters:
        by_tok.setdefault(r["tok_norm"], []).append(r)

    ok = 0
    for tok, rlist in by_tok.items():
```

```
o_fc = mineral_map.get(tok)
n_fc = nonmineral_map.get(tok)
if not o_fc or not n_fc:
    continue

predictor_paths = [r["path"] for r in sorted(rlist, key=lambda x: x["name"])]
if len(predictor_paths) < 2:
    continue

# Extract multi raster values to points (mineral/nonmineral)
o_vals_fc = extract_values_to_points(o_fc, predictor_paths, f"attn_o_{tok}".replace(".", "_"))
n_vals_fc = extract_values_to_points(n_fc, predictor_paths, f"attn_n_{tok}".replace(".", "_"))

value_fields = get_value_fields_from_points(o_vals_fc)
value_fields = [f for f in value_fields if f in {ff.name for ff in arcpy.ListFields(n_vals_fc)}]
value_fields = sorted(value_fields)

if len(value_fields) < 2:
    safe_delete(o_vals_fc)
    safe_delete(n_vals_fc)
    continue

Xo = points_to_matrix(o_vals_fc, value_fields, cfg.nodata_sentinel)
Xn = points_to_matrix(n_vals_fc, value_fields, cfg.nodata_sentinel)

safe_delete(o_vals_fc)
safe_delete(n_vals_fc)

y = np.concatenate([np.ones((Xo.shape[0],), dtype="int32"), np.zeros((Xn.shape[0],), dtype="int32")], axis=0)
X = np.concatenate([Xo, Xn], axis=0)

if cfg.drop_nodata:
    mask = ~np.any(np.isnan(X), axis=1)
    X = X[mask]
    y = y[mask]

if X.shape[0] < 50:
    continue

mu, sd = (None, None)
if cfg.standardize:
    mu, sd = standardize_fit(X)
    X = standardize_apply(X, mu, sd)

X_tr, X_te, y_tr, y_te = train_test_split_np(X, y, cfg.test_size, cfg.random_state)

model = AttnClassifier(
    n_features=X.shape[1],
    d_model=cfg.d_model,
    n_heads=cfg.n_heads,
    n_layers=cfg.n_layers,
    dropout=cfg.dropout,
)

model, m = train_model(model, X_tr, y_tr, X_te, y_te, cfg)
```

```
        row = [
            tok,
            X.shape[1],
            cfg.epochs,
            cfg.batch_size,
            cfg.d_model,
            cfg.n_heads,
            cfg.n_layers,
            cfg.dropout,
            m["accuracy"],
            m["precision"],
            m["recall"],
            m["f1"],
            m["tp"],
            m["tn"],
            m["fp"],
            m["fn"],
            int(X_tr.shape[0]),
            int(X_te.shape[0]),
        ]
        write_metrics_csv(metrics_csv, header, row)

        model_path = os.path.join(cfg.out_folder, f"DeepSelfAttention_{tok.replace('.', '_')}.pt")
        torch.save(
            {
                "state_dict": model.state_dict(),
                "value_fields": value_fields,
                "mu": mu,
                "sd": sd,
                "cfg": cfg.__dict__,
            },
            model_path,
        )

        if cfg.write_probability_raster:
            ref_name = cfg.reference_raster_name.strip()
            ref_path = os.path.join(cfg.predictor_raster_gdb, ref_name) if ref_name else predictor_paths[0]
            out_tif = os.path.join(cfg.out_folder, f"DeepSelfAttention_Prob_{tok.replace('.', '_')}.tif")
            probability_raster_from_model(
                model=model,
                ref_raster_path=ref_path,
                predictor_raster_paths=predictor_paths,
                mu=mu,
                sd=sd,
                cfg=cfg,
                out_tif=out_tif,
            )

        ok += 1

    return 0 if ok > 0 else 1


if __name__ == "__main__":
    cfg = AttnConfig(
        predictor_raster_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Litho_Rasters.gdb",
        sample_points_gdb=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\Gosal.gdb",
        out_folder=r"M:\ Reza\Survey\WGIS\P24_GOSAL\Gosal\DeepSelfAttention_Results",
        raster_filter_enabled=False,
        raster_filter_text="",
        epochs=25,
        batch_size=256,
        d_model=64,
        n_heads=4,
        n_layers=2,
        dropout=0.15,
        lr=1e-3,
        weight_decay=1e-4,
        write_probability_raster=True,
    )
    raise SystemExit(run(cfg))
```

Dempster–Shafer Belief Function Combination

```
Dempster-Shafer belief combination (Dempster's rule) for two-hypothesis frame {M, NM}.

Given multiple evidences i=1..K as rasters of mass functions:
  m_i(M), m_i(NM), m_i(Theta)
combine them pixel-wise into a fused mass:
  m(M), m(NM), m(Theta)

Outputs:
- Fused mass rasters (GeoTIFF or GDB raster)
- Optional Belief(M)=m(M), Plausibility(M)=m(M)+m(Theta)

Requires:
- ArcGIS Pro arcpy
- numpy
"""

from __future__ import annotations

import os
from dataclasses import dataclass
from typing import List, Tuple, Optional

import numpy as np
import arcpy


@dataclass(frozen=True)
class DSConfig:
    workspace_gdb: str
    out_folder: str

    # Evidence rasters must be aligned (same grid/extent/cellsize)
    evidence_triplets: List[Tuple[str, str, str]]  # [(mM, mNM, mTheta), ...]

    overwrite_output: bool = True
    # Output
    out_prefix: str = "DS"
    write_geotiff: bool = True
    nodata_value: float = -9999.0

    # Numerical safety
    eps: float = 1e-12
    clip_min: float = 0.0
    clip_max: float = 1.0


def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def raster_to_array(path: str) -> Tuple[np.ndarray, arcpy.Raster, arcpy.Describe]:
    ras = arcpy.Raster(path)
    desc = arcpy.Describe(ras)
    arr = arcpy.RasterToNumPyArray(ras, nodata_to_value=np.nan).astype("float64")
    return arr, ras, desc


def array_to_raster(
    arr: np.ndarray,
    ref_desc: arcpy.Describe,
    ref_raster: arcpy.Raster,
    out_path: str,
    nodata_value: float,
) -> None:
    extent = ref_desc.extent
    cell_w = float(ref_raster.meanCellWidth)
    cell_h = float(ref_raster.meanCellHeight)
    lower_left = arcpy.Point(float(extent.XMin), float(extent.YMin))

    out_ras = arcpy.NumPyArrayToRaster(
        arr.astype("float32"),
        lower_left,
        cell_w,
        cell_h,
        value_to_nodata=nodata_value,
    )
    out_ras.spatialReference = ref_desc.spatialReference
    safe_delete(out_path)
    out_ras.save(out_path)


def normalize_masses(mM: np.ndarray, mNM: np.ndarray, mT: np.ndarray, eps: float) -> Tuple[np.ndarray,
np.ndarray, np.ndarray]:
    s = mM + mNM + mT
    s = np.where(np.isfinite(s) & (s > eps), s, np.nan)
    return mM / s, mNM / s, mT / s


def combine_two(
    m1M: np.ndarray,
    m1NM: np.ndarray,
    m1T: np.ndarray,
    m2M: np.ndarray,
    m2NM: np.ndarray,
    m2T: np.ndarray,
    eps: float,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Dempster's rule for frame {M, NM} with Theta.

    Conflict:
      K = m1(M)m2(NM) + m1(NM)m2(M)

    Combined:
      m(M)  = [m1(M)m2(M) + m1(M)m2(]()
```

Dempster–Shafer Uncertainty Function Combination

```
Dempster-Shafer uncertainty combination for frame {M, NM}.

This script:
- Combines multiple evidences (m(M), m(NM), m(Theta)) using Dempster's rule
- Computes uncertainty measures from the fused mass:
    1) m(Theta) as raw uncertainty
    2) Discord/Conflict K (pairwise and cumulative)
    3) Shannon entropy of the pignistic probability BetP
    4) Total uncertainty = entropy(BetP) + m(Theta) (optional, configurable)

Outputs:
- Fused masses: mM, mNM, mTheta
- Uncertainty rasters: U_theta, U_entropy, U_total, K_conflict

Requires:
- ArcGIS Pro arcpy
- numpy
"""

from __future__ import annotations

import os
from dataclasses import dataclass
from typing import List, Tuple

import numpy as np
import arcpy
```

```
@dataclass(frozen=True)
class DSUConfig:
    out_folder: str
    evidence_triplets: List[Tuple[str, str, str]]  # [(mM, mNM, mTheta), ...]

    overwrite_output: bool = True
    write_geotiff: bool = True
    out_prefix: str = "DSU"

    nodata_value: float = -9999.0
    eps: float = 1e-12
    clip_min: float = 0.0
    clip_max: float = 1.0

    # Uncertainty options
    compute_entropy_betp: bool = True
    compute_total_uncertainty: bool = True


def safe_delete(path: str) -> None:
    try:
        if arcpy.Exists(path):
            arcpy.Delete_management(path)
    except Exception:
        pass


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def raster_to_array(path: str):
    ras = arcpy.Raster(path)
    desc = arcpy.Describe(ras)
    arr = arcpy.RasterToNumPyArray(ras, nodata_to_value=np.nan).astype("float64")
    return arr, ras, desc


def array_to_raster(arr: np.ndarray, ref_ras: arcpy.Raster, ref_desc, out_path: str, nodata_value: float) -> None:
    extent = ref_desc.extent
    cell_w = float(ref_ras.meanCellWidth)
    cell_h = float(ref_ras.meanCellHeight)
    ll = arcpy.Point(float(extent.XMin), float(extent.YMin))

    out = arcpy.NumPyArrayToRaster(arr.astype("float32"), ll, cell_w, cell_h, value_to_nodata=nodata_value)
    out.spatialReference = ref_desc.spatialReference
    safe_delete(out_path)
    out.save(out_path)


def normalize_masses(mM: np.ndarray, mNM: np.ndarray, mT: np.ndarray, eps: float):
    s = mM + mNM + mT
    s = np.where(np.isfinite(s) & (s > eps), s, np.nan)
    return mM / s, mNM / s, mT / s
```

```
def ds_combine_two(
    m1M: np.ndarray,
    m1NM: np.ndarray,
    m1T: np.ndarray,
    m2M: np.ndarray,
    m2NM: np.ndarray,
    m2T: np.ndarray,
    eps: float,
):
    K = (m1M * m2NM) + (m1NM * m2M)
    denom = 1.0 - K
    denom = np.where(np.isfinite(denom) & (np.abs(denom) > eps), denom, np.nan)

    mM = (m1M * m2M) + (m1M * m2T) + (m1T * m2M)
    mNM = (m1NM * m2NM) + (m1NM * m2T) + (m1T * m2NM)
    mT = (m1T * m2T)

    mM = mM / denom
    mNM = mNM / denom
    mT = mT / denom

    return mM, mNM, mT, K


def betp(mM: np.ndarray, mNM: np.ndarray, mT: np.ndarray) -> np.ndarray:
    # BetP(M) = m(M) + 0.5*m(Theta)
    return mM + 0.5 * mT


def shannon_entropy(p: np.ndarray, eps: float) -> np.ndarray:
    p = np.clip(p, eps, 1.0 - eps)
    return -(p * np.log(p) + (1.0 - p) * np.log(1.0 - p))


def run(cfg: DSUConfig) -> int:
    arcpy.env.overwriteOutput = bool(cfg.overwrite_output)
    ensure_dir(cfg.out_folder)

    if not cfg.evidence_triplets:
        return 2

    mM, ref_ras, ref_desc = raster_to_array(cfg.evidence_triplets[0][0])
    mNM, _, _ = raster_to_array(cfg.evidence_triplets[0][1])
    mT, _, _ = raster_to_array(cfg.evidence_triplets[0][2])

    base_mask = np.isnan(mM) | np.isnan(mNM) | np.isnan(mT)

    mM, mNM, mT = normalize_masses(mM, mNM, mT, cfg.eps)
    mM = np.clip(mM, cfg.clip_min, cfg.clip_max)
    mNM = np.clip(mNM, cfg.clip_min, cfg.clip_max)
    mT = np.clip(mT, cfg.clip_min, cfg.clip_max)

    mM[base_mask] = np.nan
    mNM[base_mask] = np.nan
    mT[base_mask] = np.nan

    # Cumulative conflict (mean of pairwise Ks applied sequentially)
    K_cum = np.zeros_like(mM, dtype="float64")
```

3D Model Preparation

```
3D model preparation from rasters (stack) and export to 3D-friendly formats.

This script:
- Reads multiple rasters (aligned grid)
- Builds a 3D mesh where Z comes from a chosen raster (elevation or probability)
- Optionally drapes additional rasters as vertex attributes (saved to NPZ)
- Exports:
    1) OBJ mesh (with vertices + faces)
    2) NPZ (vertices, faces, attributes) for downstream 3D workflows

Requires:
- ArcGIS Pro arcpy
- numpy

Notes:
- OBJ is geometry-only here (no MTL/texture). Vertex colors are not written.
- All rasters must have identical shape (rows/cols), extent, and cell size.
"""

from __future__ import annotations

import os
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

import numpy as np
import arcpy


@dataclass(frozen=True)
class Model3DConfig:
    raster_paths: List[str]                # aligned rasters
    z_raster_path: str                     # raster used as Z
    out_folder: str

    overwrite_output: bool = True
    nodata_value: float = -9999.0
    scale_xy: float = 1.0                  # scale for XY spacing
    scale_z: float = 1.0                   # scale for Z values
    decimate_step: int = 1                 # 1 = full resolution, 2 = every other cell, ...
    clip_to_extent: bool = False           # reserved (not used)

    out_basename: str = "Model3D"
    export_obj: bool = True
    export_npz: bool = True
```

```
def safe_delete(path: str) -> None:
    try:
        if os.path.exists(path):
            os.remove(path)
    except Exception:
        pass


def ensure_dir(path: str) -> None:
    os.makedirs(path, exist_ok=True)


def raster_to_array(path: str) -> Tuple[np.ndarray, arcpy.Raster, object]:
    ras = arcpy.Raster(path)
    desc = arcpy.Describe(ras)
    arr = arcpy.RasterToNumPyArray(ras, nodata_to_value=np.nan).astype("float64")
    return arr, ras, desc


def assert_aligned(a: np.ndarray, b: np.ndarray, name_a: str, name_b: str) -> None:
    if a.shape != b.shape:
        raise ValueError(f"Raster arrays not aligned: {name_a} shape={a.shape} vs {name_b} shape={b.shape}")


def build_vertices_faces(
    z_arr: np.ndarray,
    ref_ras: arcpy.Raster,
    ref_desc: object,
    nodata_value: float,
    scale_xy: float,
    scale_z: float,
    decimate_step: int,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Returns:
    - vertices: (N, 3) float32
    - faces: (M, 3) int32 (0-based)
    - valid_mask_grid: (H, W) bool after decimation, True means vertex exists
    """
    if decimate_step < 1:
        raise ValueError("decimate_step must be >= 1")

    z = z_arr[::decimate_step, ::decimate_step].copy()
    H, W = z.shape

    extent = ref_desc.extent
    cell_w = float(ref_ras.meanCellWidth) * scale_xy * decimate_step
    cell_h = float(ref_ras.meanCellHeight) * scale_xy * decimate_step

    x0 = float(extent.XMin)
    y0 = float(extent.YMin)

    # Pixel-to-world: NumPyArrayToRaster uses lower-left origin, RasterToNumPyArray returns row 0 at top.
    # We'll compute Y by flipping rows so that vertex grid aligns to world coordinates.
    valid = np.isfinite(z)
```

```
# Create vertex index grid (-1 for invalid)
vid = -np.ones((H, W), dtype="int32")
idx = np.flatnonzero(valid.ravel())
vid.ravel()[idx] = np.arange(idx.size, dtype="int32")

# Build vertices
rr, cc = np.nonzero(valid)
# Convert row/col to world: X increases with col, Y increases with row from bottom
# Since rr=0 is top, convert to bottom-based row:
rr_bottom = (H - 1) - rr

xs = x0 + (cc.astype("float64") + 0.5) * cell_w
ys = y0 + (rr_bottom.astype("float64") + 0.5) * cell_h
zs = z[rr, cc] * scale_z

vertices = np.stack([xs, ys, zs], axis=1).astype("float32")

# Build fac
```