ibd_1000_genomes

April 10, 2025

[ ]:
```
import os
import random
import json
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import seaborn as sns
from pathlib import Path
import networkx.algorithms.community as nx_comm
from matplotlib.colors import to_rgba
import matplotlib.patches as mpatches
import sys
import re
from collections import defaultdict
import matplotlib.patches as mpatches
from tqdm import tqdm

sys.path.append(os.path.dirname(os.getcwd()))
# Assuming utils.bonsaitree.bonsaitree.v3 is properly installed
try:
    from utils.bonsaitree.bonsaitree.v3 import bonsai
except ImportError:
    print("Warning: Unable to import bonsai module. Pedigree reconstruction␣
 ↪functions will not work.")

#######################
# 1. Data Preparation #
#######################
def load_genetic_data(seg_file, fam_file, dict_file=None):
    """
    Load and prepare genetic data from .seg, .fam, and optional dict files.
    Args:
        seg_file: Path to the .seg file
        fam_file: Path to the .fam file
        dict_file: Path to the ID mapping file (optional)
    Returns:
        seg_df: DataFrame with segment data
        individuals: Dictionary of individual metadata
        individual_to_bonsai: Mapping from original IDs to Bonsai IDs
    """
    # Load the ID mapping if provided
    individual_to_bonsai = {}
    if dict_file and os.path.exists(dict_file):
        print(f"Loading ID mapping from {dict_file}")
        try:
            with open(dict_file, 'r') as f:
                for line in f:
                    parts = line.strip().split('\t')
                    if len(parts) == 2:
                        individual_id, bonsai_id = parts
                        individual_to_bonsai[individual_id] = int(bonsai_id)
            print(f"Loaded {len(individual_to_bonsai)} ID mappings")
        except Exception as e:
            print(f"Error loading ID mapping: {e}")

    # Read the seg file
    seg_df = pd.read_csv(seg_file, sep="\t", header=None)
    if len(seg_df.columns) == 9:
        seg_df.columns = ["sample1", "sample2", "chrom", "phys_start",␣
↪"phys_end",
                          "ibd_type", "gen_start", "gen_end", "gen_seg_len"]
    else:
        print(f"Warning: Unexpected number of columns in seg file: {len(seg_df.
↪columns)}")
        print("Columns found:", seg_df.columns)
        return None, None, None

    # Extract unique individuals from seg file
    unique_individuals_from_seg = set(seg_df["sample1"]).
↪union(set(seg_df["sample2"]))
    print(f"Number of unique individuals in seg file:␣
↪{len(unique_individuals_from_seg)}")

    # Read the fam file to get individual metadata
    individuals = {}
    try:
        with open(fam_file, 'r') as file:
            fam_lines = file.readlines()

            # Process each line in the fam file
            for line in fam_lines:
                fields = line.strip().split()
                if len(fields) < 6:
                    continue

                family_id = fields[0]
                individual_id = fields[1]

                # Skip individuals not present in the ID mapping if using a␣
↪mapping
                if individual_to_bonsai and individual_id not in␣
↪individual_to_bonsai:
                    continue

                father_id = fields[2]
                mother_id = fields[3]
                sex = 'M' if fields[4] == '1' else 'F'

                # Extract generation using regex
                match = re.search(r'g(\d+)-', individual_id)
                generation = int(match.group(1)) if match else None

                # Store the individual information
                individuals[individual_id] = {
                    'family_id': family_id,
                    'father_id': father_id,
                    'mother_id': mother_id,
                    'sex': sex,
                    'generation': generation
                }

        print(f"Loaded metadata for {len(individuals)} individuals from FAM␣
↪file")

        # Print a sample of the individuals dictionary
        sample_keys = list(individuals.keys())[:3]
        print("\nSample of individuals data:")
        for key in sample_keys:
            print(f"{key}: {individuals[key]}")

        # Summary statistics
        generations = {}
        for ind_id, info in individuals.items():
            gen = info.get('generation')
            if gen:
                generations[gen] = generations.get(gen, 0) + 1

        print("\nIndividuals by generation:")
        for gen, count in sorted(generations.items()):
            print(f"Generation {gen}: {count} individuals")
```

```
    except Exception as e:
        print(f"Error loading FAM file: {e}")
        return None, None, None

    return seg_df, individuals, individual_to_bonsai

def create_bioinfo(individuals, individual_to_bonsai):
    """
    Create bioinfo list for Bonsai with ages assigned based on generation.
    Args:
        individuals: Dictionary of individual metadata
        individual_to_bonsai: Mapping of original IDs to Bonsai IDs
    Returns:
        bioinfo: List of dictionaries with individual metadata for Bonsai
    """
    # Check if we have generation information
    has_generation_info = any('generation' in info and info['generation'] is␣
 ↪not None
                              for info in individuals.values())

    if not has_generation_info:
        print("Warning: No generation information found in individuals data")
        return []

    # Get generation range
    generations = [info['generation'] for info in individuals.values()
                   if 'generation' in info and info['generation'] is not None]

    if not generations:
        print("Warning: No valid generation values found")
        return []

    latest_generation = max(generations)
    earliest_generation = min(generations)
    print(f"Generation range: {earliest_generation} to {latest_generation}")

    # Assign ages based on generation
    for individual_id, info in individuals.items():
        generation = info.get('generation')
        if generation is None:
            # Skip individuals without generation info
            continue

        if generation == latest_generation:
            # Latest generation: ages 18-40
            info['age'] = random.randint(18, 40)
        else:
            # Earlier generations: older based on generation gap
            gen_gap = latest_generation - generation
            min_age = 25 + (gen_gap * 20)
            max_age = 40 + (gen_gap * 20)
            info['age'] = random.randint(min_age, max_age)

    # Create bioinfo list for Bonsai
    bioinfo = []
    for individual_id, info in individuals.items():
        if 'generation' in info and info['generation'] is not None:
            if individual_id in individual_to_bonsai:
                bonsai_id = individual_to_bonsai[individual_id]
                age = info.get('age', 30)  # Default age if not calculated
                sex = info.get('sex', 'U')  # Default sex if not available
                bioinfo.append({'genotype_id': bonsai_id, 'age': age, 'sex':␣
 ↪sex})

    return bioinfo

def create_ibd_segment_list(seg_df):
    """Create an unphased IBD segment list for Bonsai from the segment␣
 ↪dataframe."""
    unphased_ibd_seg_list = []
    for _, row in seg_df.iterrows():
        try:
            id1 = int(row['sample1'])
            id2 = int(row['sample2'])
            chrom = str(row['chrom'])
            start_bp = float(row['phys_start'])
            end_bp = float(row['phys_end'])
            is_full = row['ibd_type'] == 2  # Assuming IBD2 indicates "full"␣
 ↪sharing
            len_cm = float(row['gen_seg_len'])

            unphased_ibd_seg_list.append([id1, id2, chrom, start_bp, end_bp,␣
 ↪is_full, len_cm])
        except ValueError as e:
            print(f"Error processing segment: {e}")

    return unphased_ibd_seg_list

def identify_reference_populations(sample_df):
    """
    Extract reference populations from sample dataframe
    """
    reference_pops = sorted(sample_df['Population'].unique())
    print(f"Found {len(reference_pops)} reference populations:")
    for pop in reference_pops:
        count = len(sample_df[sample_df['Population'] == pop])
        desc = sample_df[sample_df['Population'] == pop]['Population␣
↪Description'].iloc[0]
        print(f"  {pop}: {count} individuals - {desc}")
    return reference_pops

def identify_project_samples(segments, sample_df):
    """Identify project samples that aren't in reference panel"""
    # Get unique sample IDs from IBD data
    all_samples = pd.unique(segments[['sample1', 'sample2']].values.ravel())

    # Identify project samples (those not in reference panel metadata)
    project_samples = set(all_samples) - set(sample_df['Sample'])

    print(f"Total unique samples in IBD data: {len(all_samples)}")
    print(f"Samples in reference panel: {len(sample_df['Sample'])}")
    print(f"Project samples identified: {len(project_samples)}")

    return list(project_samples)

##########################
# 2. Community Detection #
##########################
def detect_communities(ibd_seg_list, bioinfo, resolution=1.0,␣
↪min_community_size=10):
    """
    Detect communities using Louvain algorithm to divide the dataset.
    Args:
        ibd_seg_list: List of IBD segments
        bioinfo: List of individual metadata
        resolution: Resolution parameter for Louvain (higher = smaller␣
↪communities)
        min_community_size: Minimum community size to keep
    Returns:
        communities: List of detected communities (sets of individual IDs)
    """
    # Create a graph from IBD segments
    G = nx.Graph()

    # Add nodes for all individuals in bioinfo
    genotype_ids = [info['genotype_id'] for info in bioinfo]
    G.add_nodes_from(genotype_ids)

    # Add edges weighted by IBD sharing
    edge_weights = defaultdict(float)
    for segment in ibd_seg_list:
        id1, id2 = segment[0], segment[1]
        cm_length = segment[6]  # Length in centiMorgans
        edge_weights[(id1, id2)] += cm_length

    # Add all edges to the graph
    for (id1, id2), weight in edge_weights.items():
        G.add_edge(id1, id2, weight=weight)

    # Find communities using Louvain
    try:
        communities = list(nx.community.louvain_communities(G,␣
↪resolution=resolution, weight='weight'))

        # Filter out communities that are too small
        communities = [comm for comm in communities if len(comm) >=␣
↪min_community_size]

        print(f"Detected {len(communities)} communities")
        for i, community in enumerate(communities):
            print(f"Community {i+1}: {len(community)} members")

        return communities

    except Exception as e:
        print(f"Error detecting communities: {e}")
        # If community detection fails, return a single community with all␣
↪individuals
        print("Falling back to using all individuals as one community")
        return [set(genotype_ids)]

def filter_for_community(community, bioinfo, ibd_seg_list):
    """Filter bioinfo and IBD segments for a specific community."""
    # Filter bioinfo
    community_bioinfo = [info for info in bioinfo if info['genotype_id'] in␣
↪community]

    # Filter IBD segments
    community_ibd = []
    for seg in ibd_seg_list:
        id1, id2 = seg[0], seg[1]
        if id1 in community and id2 in community:
            community_ibd.append(seg)

    return community_bioinfo, community_ibd

def visualize_communities(G, communities, output_file=None, figsize=(12, 12)):
    """Visualize communities in a graph."""
    plt.figure(figsize=figsize)

    # Create a colormap for communities
    colors = plt.cm.rainbow(np.linspace(0, 1, len(communities)))

    # Assign community colors to nodes
    node_colors = []
    for node in G.nodes():
        for i, community in enumerate(communities):
            if node in community:
                node_colors.append(colors[i])
                break
        else:
            # If node isn't in any community
            node_colors.append((0.7, 0.7, 0.7, 0.5))

    # Create color patches for legend
    patches = []
    for i, color in enumerate(colors):
        patches.append(mpatches.Patch(color=color, label=f'Community {i+1}'))

    # Apply layout - try different options depending on graph size
    if len(G.nodes()) > 500:
        print("Using sfdp layout for large graph...")
        try:
            pos = nx.nx_agraph.graphviz_layout(G, prog='sfdp')
        except:
            print("Graphviz sfdp layout failed, falling back to spring layout")
            pos = nx.spring_layout(G, k=0.3, iterations=50, seed=42)
    else:
        try:
            # For smaller graphs try neato first
            pos = nx.nx_agraph.graphviz_layout(G, prog='neato')
        except:
            print("Graphviz layout failed, falling back to spring layout")
            pos = nx.spring_layout(G, k=0.3, iterations=50, seed=42)

    # Draw the graph
    nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=50,␣
↪alpha=0.8)
    nx.draw_networkx_edges(G, pos, width=0.5, alpha=0.3)

    plt.title("IBD Network Communities", fontsize=16)
    plt.legend(handles=patches, loc='upper right')
    plt.axis('off')
    if output_file:
        plt.savefig(output_file, dpi=300, bbox_inches='tight')
        print(f"Network visualization saved to {output_file}")

    plt.show()

##########################
# 3. IBD-Based Ancestry  #
##########################
def calculate_time_stratified_ancestry(sample_id, sample_df, segments,␣
 ↪reference_populations, total_genome_length=3400):
    """
    Calculate ancestry components across different time periods based on IBD␣
 ↪segment length.

    Args:
        sample_id: ID of the sample to analyze
        sample_df: DataFrame with reference population metadata
        segments: DataFrame of IBD segments
        reference_populations: List of reference populations to analyze
        total_genome_length: Total genetic length of genome in cM (default 3400)

    Returns:
        ancestry_by_time: DataFrame with ancestry proportions by population and␣
 ↪time period
    """
    print(f"Calculating time-stratified ancestry for {sample_id}...")

    # Get IBD segments for this sample
    sample_segs = segments[(segments['sample1'] == sample_id) |␣
 ↪(segments['sample2'] == sample_id)].copy()

    if len(sample_segs) == 0:
        print(f"Warning: No IBD segments found for {sample_id}")
        return pd.DataFrame()

    # Extract other sample ID
    sample_segs['other_id'] = np.where(
        sample_segs['sample1'] == sample_id,
        sample_segs['sample2'],
        sample_segs['sample1']
    )

    # Merge with population information
    sharing_df = pd.merge(
        sample_segs,
        sample_df[['Sample', 'Population', 'Population Description']],
        left_on='other_id',
        right_on='Sample',
        how='left'
    )

    # Drop rows with no population information
    sharing_df = sharing_df.dropna(subset=['Population'])

    if len(sharing_df) == 0:
        print(f"Warning: No matches with reference populations for {sample_id}")
        return pd.DataFrame()

    # Define time periods - more granular, especially for recent periods
    time_periods = [
        (0, 50, 'Very Recent (0-50 years)'),
        (51, 100, 'Recent (51-100 years)'),
        (101, 150, 'Recent Historical (101-150 years)'),
        (151, 200, 'Historical (151-200 years)'),
        (201, 300, 'Early Modern (201-300 years)'),
        (301, 500, 'Colonial Era (301-500 years)'),
        (501, 1000, 'Medieval (501-1000 years)'),
        (1001, 2000, 'Ancient (1001-2000 years)'),
        (2001, float('inf'), 'Prehistoric (>2000 years)')
    ]

    # Calculate TMRCA (years) based on genetic length
    # Using the 50/genetic_length formula for TMRCA in generations
    # Then multiply by 25 years per generation
    sharing_df['tmrca_years'] = 25 * 50 / sharing_df['gen_seg_len']

    # Assign time periods to each segment
    def assign_time_period(years):
        for start, end, label in time_periods:
            if start <= years < end:
                return label
        return 'Unknown'

    sharing_df['time_period'] = sharing_df['tmrca_years'].
↪apply(assign_time_period)

    # Initialize results structure - for each population and time period
    ancestry_data = {}
    for pop in reference_populations:
        ancestry_data[pop] = {period[2]: 0 for period in time_periods}

    # Sum IBD sharing by population and time period
    for pop in reference_populations:
        pop_data = sharing_df[sharing_df['Population'] == pop]
        for period_start, period_end, period_name in time_periods:
            period_data = pop_data[(pop_data['tmrca_years'] >= period_start) &
                                  (pop_data['tmrca_years'] < period_end)]
            total_cm = period_data['gen_seg_len'].sum()
            ancestry_data[pop][period_name] = total_cm

    # Convert to dataframe
    ancestry_df = pd.DataFrame(ancestry_data)

    # Calculate percentages of genome (normalize by total genome length)
    ancestry_pct = ancestry_df / total_genome_length * 100

    # Add total row
    ancestry_pct.loc['Total'] = ancestry_pct.sum()

    # Add weighted time-based ancestry composition
    # Higher weight for more recent ancestry
    weights = {
        'Very Recent (0-50 years)': 1.0,
        'Recent (51-100 years)': 0.9,
        'Recent Historical (101-150 years)': 0.8,
        'Historical (151-200 years)': 0.7,
        'Early Modern (201-300 years)': 0.6,
        'Colonial Era (301-500 years)': 0.5,
        'Medieval (501-1000 years)': 0.4,
        'Ancient (1001-2000 years)': 0.3,
        'Prehistoric (>2000 years)': 0.2
    }

    # Calculate weighted ancestry
    weighted_ancestry = pd.Series(0.0, index=reference_populations)
    for period in weights:
        if period in ancestry_pct.index:
            weighted_ancestry += ancestry_pct.loc[period] * weights[period]

    # Normalize to sum to 100%
    if weighted_ancestry.sum() > 0:
        weighted_ancestry = weighted_ancestry / weighted_ancestry.sum() * 100

    ancestry_pct.loc['Weighted Total'] = weighted_ancestry

    return ancestry_pct

def create_consensus_ancestry_map(sample_id, sample_df, segments,␣
 ↪chromosome_lengths, window_size=1000000):
    """
    Create a position-specific ancestry map based on overlapping IBD segments.

    Args:
        sample_id: ID of the sample to analyze
        sample_df: DataFrame with reference population metadata
        segments: DataFrame of IBD segments
        chromosome_lengths: Dictionary of chromosome lengths {chrom: length}
        window_size: Size of genomic windows in bp

    Returns:
        ancestry_map: Dictionary {chrom: {position: {population: probability}}}
    """
    print(f"Creating consensus ancestry map for {sample_id}...")

    # Get IBD segments for this sample
    sample_segs = segments[(segments['sample1'] == sample_id) |␣
↪(segments['sample2'] == sample_id)].copy()

    if len(sample_segs) == 0:
        print(f"Warning: No IBD segments found for {sample_id}")
        return {}

    # Extract other sample ID and merge with population information
    sample_segs['other_id'] = np.where(
        sample_segs['sample1'] == sample_id,
        sample_segs['sample2'],
        sample_segs['sample1']
    )

    sharing_df = pd.merge(
        sample_segs,
        sample_df[['Sample', 'Population']],
        left_on='other_id',
        right_on='Sample',
        how='left'
    )

    # Drop rows with no population information
    sharing_df = sharing_df.dropna(subset=['Population'])

    if len(sharing_df) == 0:
        print(f"Warning: No matches with reference populations for {sample_id}")
        return {}

    # Calculate TMRCA in years for weighting
    sharing_df['tmrca_years'] = 25 * 50 / sharing_df['gen_seg_len']
    # Weight based on segment length and recency (shorter TMRCA = higher weight)
    sharing_df['weight'] = sharing_df['gen_seg_len'] * (1000 /␣
↪(sharing_df['tmrca_years'] + 100))

    # Get unique populations and chromosomes
    populations = sample_df['Population'].unique()
    chromosomes = sharing_df['chrom'].unique()

    # Initialize ancestry map
    ancestry_map = {}

    # Process each chromosome
    for chrom in chromosomes:
        chrom_segs = sharing_df[sharing_df['chrom'] == chrom]

        if str(chrom) not in chromosome_lengths:
            if chrom not in chromosome_lengths:
                print(f"Warning: No length data for chromosome {chrom}")
                continue

        # Get chromosome length
        chrom_length = chromosome_lengths[str(chrom)] if str(chrom) in␣
↪chromosome_lengths else chromosome_lengths[chrom]

        # Create windows for this chromosome
        windows = range(0, chrom_length + window_size, window_size)

        # Initialize ancestry probabilities for this chromosome
        chrom_ancestry = {window: {pop: 0.0 for pop in populations} for window␣
↪in windows}

        # Process each segment to contribute to window probabilities
        for _, segment in chrom_segs.iterrows():
            start_window = (int(segment['phys_start']) // window_size) *␣
↪window_size
            end_window = (int(segment['phys_end']) // window_size) * window_size

            # Apply segment contribution to each overlapping window
            for window in range(start_window, end_window + window_size,␣
↪window_size):
                if window in chrom_ancestry:
                    pop = segment['Population']
                    if pd.notna(pop):  # Skip if population is unknown
                        chrom_ancestry[window][pop] += segment['weight']

        # Normalize probabilities in each window
        for window in chrom_ancestry:
                total = sum(chrom_ancestry[window].values())
                if total > 0:
                    for pop in chrom_ancestry[window]:
                        chrom_ancestry[window][pop] /= total

            ancestry_map[str(chrom)] = chrom_ancestry

    return ancestry_map

def visualize_time_stratified_ancestry(sample_id, ancestry_data, output_dir):
    """
    Create visualizations for time-stratified ancestry.

    Args:
        sample_id: ID of the sample
        ancestry_data: DataFrame with ancestry by time period
        output_dir: Directory to save output files
    """
    # Skip if no data
    if ancestry_data.empty:
        print(f"No ancestry data to visualize for {sample_id}")
        return

    # Create output directory if needed
    os.makedirs(output_dir, exist_ok=True)

    # 1. Heatmap of ancestry by time period
    plt.figure(figsize=(15, 10))
    # Exclude the summary rows
    plot_data = ancestry_data.iloc[:-2] if len(ancestry_data) > 2 else␣
↪ancestry_data
    sns.heatmap(plot_data, annot=True, cmap='YlOrRd', fmt='.1f')
    plt.title(f"Time-Stratified Ancestry for {sample_id}")
    plt.ylabel("Time Period")
    plt.xlabel("Reference Population")
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f"{sample_id}_time_heatmap.png"),␣
↪dpi=300)
    plt.close()

    # 2. Stacked bar chart of population proportions over time
    plt.figure(figsize=(15, 8))
    # Transpose to have populations as rows and time periods as columns
    plot_data_t = plot_data.T
    # Normalize each column (time period) to sum to 100%
    for col in plot_data_t.columns:
        if plot_data_t[col].sum() > 0:
            plot_data_t[col] = plot_data_t[col] / plot_data_t[col].sum() * 100

    # Plot stacked bars
    plot_data_t.plot(kind='bar', stacked=True, colormap='tab20')
    plt.title(f"Ancestry Composition Over Time for {sample_id}")
    plt.xlabel("Reference Population")
    plt.ylabel("Percentage of Ancestry")
    plt.legend(title="Time Period")
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f"{sample_id}_time_stacked.png"),␣
↪dpi=300)
    plt.close()

    # 3. Area chart showing ancestry over time
    plt.figure(figsize=(15, 8))
    # Get time periods as x values (use numeric midpoints of ranges)
    time_midpoints = {
        'Very Recent (0-50 years)': 25,
        'Recent (51-100 years)': 75,
        'Recent Historical (101-150 years)': 125,
        'Historical (151-200 years)': 175,
        'Early Modern (201-300 years)': 250,
        'Colonial Era (301-500 years)': 400,
        'Medieval (501-1000 years)': 750,
        'Ancient (1001-2000 years)': 1500,
        'Prehistoric (>2000 years)': 2500
    }

    # Prepare data for area chart
    area_data = plot_data.copy()
    # Add time midpoints as index values
    area_data['midpoint'] = area_data.index.map(lambda x: time_midpoints.get(x,␣
↪0))
    area_data = area_data.sort_values('midpoint')

    # Normalize rows to get percentages
    for idx in area_data.index:
        if idx != 'midpoint' and area_data.loc[idx].sum() > 0:
            row_sum = area_data.loc[idx, [col for col in area_data.columns if␣
↪col != 'midpoint']].sum()
            if row_sum > 0:
                area_data.loc[idx, [col for col in area_data.columns if col !=␣
↪'midpoint']] = \
                    area_data.loc[idx, [col for col in area_data.columns if col␣
↪!= 'midpoint']] / row_sum * 100

    # Plot area chart
    x = area_data['midpoint']
    y_columns = [col for col in area_data.columns if col != 'midpoint']
    plt.stackplot(x, [area_data[col] for col in y_columns], labels=y_columns,␣
 ↪alpha=0.8)

    plt.title(f"Ancestry Composition Through Time for {sample_id}")
    plt.xlabel("Years Before Present")
    plt.ylabel("Percentage of Ancestry")
    plt.legend(title="Population", loc='upper right')
    plt.grid(alpha=0.3)
    plt.xlim(0, 2600)
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f"{sample_id}_time_area.png"), dpi=300)
    plt.close()

def visualize_chromosome_painting(sample_id, ancestry_map, output_dir):
    """
    Create chromosome painting visualizations from consensus ancestry map.

    Args:
        sample_id: ID of the sample
        ancestry_map: Dictionary {chrom: {position: {population: probability}}}
        output_dir: Directory to save output files
    """
    if not ancestry_map:
        print(f"No ancestry map data to visualize for {sample_id}")
        return

    # Create output directory if needed
    chrom_paint_dir = os.path.join(output_dir,␣
 ↪f"{sample_id}_chromosome_paintings")
    os.makedirs(chrom_paint_dir, exist_ok=True)

    # Create a colormap for populations
    populations = []
    for chrom in ancestry_map:
        for pos in ancestry_map[chrom]:
            populations.extend(ancestry_map[chrom][pos].keys())
    populations = sorted(set(populations))

    if not populations:
        print(f"No population data in ancestry map for {sample_id}")
        return

    # Create a dictionary mapping populations to colors
    pop_colors = {}
    cmap = plt.cm.get_cmap('tab20', len(populations))
    for i, pop in enumerate(populations):
        pop_colors[pop] = cmap(i)

    # Create a summary figure showing all chromosomes
    plt.figure(figsize=(15, 10))
    chrom_count = len(ancestry_map)
    fig, axes = plt.subplots(nrows=min(chrom_count, 6), ncols=max(1,␣
↪(chrom_count+5)//6),
                            figsize=(15, 2*min(chrom_count, 6)), squeeze=False)

    # Flatten axes array for easy iteration
    axes_flat = axes.flatten()

    for i, chrom in enumerate(sorted(ancestry_map.keys(), key=lambda x: int(x)␣
↪if x.isdigit() else x)):
        if i >= len(axes_flat):
            print(f"Warning: Not enough subplots for all chromosomes")
            break

        ax = axes_flat[i]
        chrom_data = ancestry_map[chrom]

        # Sort positions
        positions = sorted(chrom_data.keys())
        if not positions:
            continue

        # Create data arrays for visualization
        x_positions = np.array(positions) / 1_000_000  # Convert to Mb

        # For each position, stack colors proportionally
        for pos in positions:
            pos_data = chrom_data[pos]
            bottom = 0
            x_pos = pos / 1_000_000  # Convert to Mb

            # Sort populations by contribution (largest at bottom)
            sorted_pops = sorted(pos_data.items(), key=lambda x: x[1],␣
↪reverse=True)

            for pop, prob in sorted_pops:
                if prob > 0:
                    height = prob
                    ax.bar(x_pos, height, bottom=bottom, width=1,␣
↪color=pop_colors[pop],
                          edgecolor='none', align='center', alpha=0.7)
                    bottom += height
        ax.set_title(f"Chr {chrom}")
        ax.set_xlabel("Position (Mb)")
        ax.set_ylabel("Ancestry Proportion")
        ax.set_ylim(0, 1)
        ax.grid(alpha=0.3)

    # Hide any unused subplots
    for j in range(i+1, len(axes_flat)):
        axes_flat[j].axis('off')

    # Create legend with population colors
    legend_elements = [plt.Rectangle((0,0), 1, 1, color=pop_colors[pop],␣
↪label=pop) for pop in populations]
    fig.legend(handles=legend_elements, loc='upper center', bbox_to_anchor=(0.
↪5, 0.05),
               ncol=min(5, len(populations)))

    plt.tight_layout()
    plt.subplots_adjust(bottom=0.15)
    plt.savefig(os.path.join(output_dir, f"{sample_id}_all_chromosomes.png"),␣
↪dpi=300)
    plt.close()

    # Create individual chromosome paintings with higher resolution
    for chrom in sorted(ancestry_map.keys(), key=lambda x: int(x) if x.
↪isdigit() else x):
        chrom_data = ancestry_map[chrom]
        positions = sorted(chrom_data.keys())
        if not positions:
            continue

        plt.figure(figsize=(15, 5))

        # For each position, stack colors proportionally
        for pos in positions:
            pos_data = chrom_data[pos]
            bottom = 0
            x_pos = pos / 1_000_000  # Convert to Mb

            # Sort populations by contribution (largest at bottom)
            sorted_pops = sorted(pos_data.items(), key=lambda x: x[1],␣
↪reverse=True)

            for pop, prob in sorted_pops:
                if prob > 0:
                    height = prob
                    plt.bar(x_pos, height, bottom=bottom, width=1,␣
↪color=pop_colors[pop],
                          edgecolor='none', align='center', alpha=0.7)
                    bottom += height

        plt.title(f"Chromosome {chrom} Ancestry Painting for {sample_id}")
        plt.xlabel("Position (Mb)")
        plt.ylabel("Ancestry Proportion")
        plt.ylim(0, 1)
        plt.grid(alpha=0.3)

        # Add population legend
        legend_elements = [plt.Rectangle((0,0), 1, 1, color=pop_colors[pop],␣
↪label=pop) for pop in populations]
        plt.legend(handles=legend_elements, loc='upper right')

        plt.tight_layout()
        plt.savefig(os.path.join(chrom_paint_dir, f"chrom_{chrom}_painting.
↪png"), dpi=300)
        plt.close()

def compare_with_rfmix(sample_id, ancestry_map, rfmix_file, output_dir,␣
↪window_size=1000000):
    """
    Compare IBD-based ancestry inference with RFMix results.

    Args:
        sample_id: ID of the sample to analyze
        ancestry_map: Ancestry map from IBD-based method
        rfmix_file: Path to RFMix output file
        output_dir: Directory to save comparison results
        window_size: Size of genomic windows in bp (should match IBD map)

    Returns:
        comparison_df: DataFrame with comparison metrics
    """
    if not os.path.exists(rfmix_file):
        print(f"RFMix file not found: {rfmix_file}")
        return None

    print(f"Comparing IBD-based ancestry with RFMix for {sample_id}...")

    try:
        # Load RFMix results
        rfmix_df = pd.read_csv(rfmix_file, sep='\t')

        # Filter for the sample of interest
        rfmix_sample = rfmix_df[rfmix_df['sample'] == sample_id]

        if rfmix_sample.empty:
            print(f"Sample {sample_id} not found in RFMix results")
            return None

        # Initialize comparison results
        comparison = {
            'chrom': [],
            'position': [],
            'rfmix_ancestry': [],
            'ibd_ancestry': [],
            'agreement': [],
            'ibd_confidence': []
        }

        # For each chromosome in both datasets
        for chrom in ancestry_map:
            if chrom not in rfmix_sample['chrom'].astype(str).unique():
                continue

            rfmix_chrom = rfmix_sample[rfmix_sample['chrom'].astype(str) ==␣
↪chrom]

            # Iterate through IBD ancestry windows
            for window, pop_probs in ancestry_map[chrom].items():
                # Find corresponding RFMix windows
                rfmix_windows = rfmix_chrom[
                    (rfmix_chrom['start'] >= window) &
                    (rfmix_chrom['start'] < window + window_size)
                ]

                if rfmix_windows.empty:
                    continue

                # Get most likely ancestry from IBD method
                ibd_ancestry = max(pop_probs.items(), key=lambda x: x[1])
                ibd_pop = ibd_ancestry[0]
                ibd_confidence = ibd_ancestry[1]

                # Get most common RFMix ancestry in this window
                rfmix_counts = rfmix_windows['ancestry'].value_counts()
                if not rfmix_counts.empty:
                    rfmix_pop = rfmix_counts.index[0]

                    # Check if they agree
                    agreement = 1 if rfmix_pop == ibd_pop else 0
                    # Store comparison
                    comparison['chrom'].append(chrom)
                    comparison['position'].append(window)
                    comparison['rfmix_ancestry'].append(rfmix_pop)
                    comparison['ibd_ancestry'].append(ibd_pop)
                    comparison['agreement'].append(agreement)
                    comparison['ibd_confidence'].append(ibd_confidence)

        # Create comparison DataFrame
        comparison_df = pd.DataFrame(comparison)

        # Calculate overall agreement
        overall_agreement = comparison_df['agreement'].mean() if not␣
↪comparison_df.empty else 0
        print(f"Overall agreement with RFMix: {overall_agreement:.2%}")

        # Save comparison results
        if not comparison_df.empty:
            comparison_df.to_csv(os.path.join(output_dir,␣
↪f"{sample_id}_rfmix_comparison.csv"), index=False)

            # Create visualization of agreement
            plt.figure(figsize=(12, 8))

            # Plot agreement by chromosome as heatmap
            agreement_by_chrom = comparison_df.pivot_table(
                index='chrom',
                columns='ibd_ancestry',
                values='agreement',
                aggfunc='mean'
            )

            sns.heatmap(agreement_by_chrom, annot=True, cmap='YlGnBu', fmt='.
↪2f')
            plt.title(f"Agreement with RFMix by Chromosome and Ancestry␣
↪({sample_id})")
            plt.tight_layout()
            plt.savefig(os.path.join(output_dir, f"{sample_id}_rfmix_agreement.
↪png"), dpi=300)
            plt.close()

        return comparison_df

    except Exception as e:
        print(f"Error comparing with RFMix: {e}")
        return None
```

```
##############################
# 4. Main Analysis Function #
##############################
def run_ancestry_analysis(sample_id, segments, sample_df,␣
↪output_dir="ancestry_results",
                          rfmix_file=None, chromosome_lengths=None):
    """
    Run ancestry analysis for a single sample using IBD segments with a␣
↪reference panel.

    Args:
        sample_id: ID of the sample to analyze
        segments: DataFrame of IBD segments
        sample_df: DataFrame with reference population metadata
        output_dir: Directory to save results
        rfmix_file: Path to RFMix results for comparison (optional)
        chromosome_lengths: Dictionary with chromosome lengths

    Returns:
        results: Dictionary with analysis results
    """
    print(f"\nRunning ancestry analysis for sample: {sample_id}")

    # Create output directory
    os.makedirs(output_dir, exist_ok=True)
    sample_dir = os.path.join(output_dir, sample_id)
    os.makedirs(sample_dir, exist_ok=True)

    # Extract reference populations
    reference_pops = sorted(sample_df['Population'].unique())

    # Define default chromosome lengths if not provided
    if chromosome_lengths is None:
        # Example chromosome lengths (GRCh38) in base pairs
        chromosome_lengths = {
            '1': 248956422, '2': 242193529, '3': 198295559, '4': 190214555,
            '5': 181538259, '6': 170805979, '7': 159345973, '8': 145138636,
            '9': 138394717, '10': 133797422, '11': 135086622, '12': 133275309,
            '13': 114364328, '14': 107043718, '15': 101991189, '16': 90338345,
            '17': 83257441, '18': 80373285, '19': 58617616, '20': 64444167,
            '21': 46709983, '22': 50818468, 'X': 156040895, 'Y': 57227415
        }

    # Initialize results dictionary
    results = {}
    # 1. Time-stratified ancestry analysis
    print("Running time-stratified ancestry analysis...")
    ancestry_by_time = calculate_time_stratified_ancestry(
        sample_id, sample_df, segments, reference_pops
    )
    results['time_stratified'] = ancestry_by_time

    # Save time-stratified results
    if not ancestry_by_time.empty:
        ancestry_by_time.to_csv(
            os.path.join(sample_dir, f"time_stratified_ancestry.csv")
        )
        visualize_time_stratified_ancestry(sample_id, ancestry_by_time,␣
↪sample_dir)

    # 2. Consensus mapping approach
    print("Creating consensus ancestry map...")
    ancestry_map = create_consensus_ancestry_map(
        sample_id, sample_df, segments, chromosome_lengths
    )
    results['consensus_map'] = ancestry_map

    # Save consensus map to file
    if ancestry_map:
        with open(os.path.join(sample_dir, f"ancestry_map.json"), 'w') as f:
            # Convert ancestry map to serializable format
            serializable_map = {}
            for chrom, chrom_data in ancestry_map.items():
                serializable_map[chrom] = {
                    str(pos): {pop: float(prob) for pop, prob in pos_data.
↪items()}
                    for pos, pos_data in chrom_data.items()
                }
            json.dump(serializable_map, f, indent=2)

        # Create chromosome paintings
        visualize_chromosome_painting(sample_id, ancestry_map, sample_dir)

    # 3. Compare with RFMix if available
    if rfmix_file and os.path.exists(rfmix_file):
        print("Comparing with RFMix results...")
        comparison = compare_with_rfmix(
            sample_id, ancestry_map, rfmix_file, sample_dir
        )
        results['rfmix_comparison'] = comparison

    print(f"Analysis complete for {sample_id}")
    return results

def batch_ancestry_analysis(project_samples, segments, sample_df,␣
↪output_dir="ancestry_results",
                           rfmix_dir=None, chromosome_lengths=None):
    """
    Run ancestry analysis for multiple samples.

    Args:
        project_samples: List of sample IDs to analyze
        segments: DataFrame of IBD segments
        sample_df: DataFrame with reference population metadata
        output_dir: Directory to save results
        rfmix_dir: Directory containing RFMix results (optional)
        chromosome_lengths: Dictionary with chromosome lengths

    Returns:
        all_results: Dictionary with results for all samples
    """
    os.makedirs(output_dir, exist_ok=True)

    # Initialize results
    all_results = {}

    # Process each sample
    for i, sample_id in enumerate(project_samples):
        print(f"\nProcessing sample {i+1}/{len(project_samples)}: {sample_id}")

        # Define RFMix file path if available
        rfmix_file = None
        if rfmix_dir:
            potential_file = os.path.join(rfmix_dir, f"{sample_id}_rfmix.txt")
            if os.path.exists(potential_file):
                rfmix_file = potential_file

        # Run analysis for this sample
        results = run_ancestry_analysis(
            sample_id, segments, sample_df, output_dir, rfmix_file,␣
↪chromosome_lengths
        )

        all_results[sample_id] = results

    # Create summary across all samples
    print("\nCreating summary across all samples...")

    # Collect weighted ancestry totals across samples
    weighted_totals = {}
    for sample_id, results in all_results.items():
        if 'time_stratified' in results and not results['time_stratified'].
↪empty:
            weighted_totals[sample_id] = results['time_stratified'].
↪loc['Weighted Total']

    if weighted_totals:
        weighted_df = pd.DataFrame(weighted_totals).T

        # Save to file
        weighted_df.to_csv(os.path.join(output_dir,␣
↪"all_samples_weighted_ancestry.csv"))

        # Create heatmap visualization
        plt.figure(figsize=(15, 10))
        sns.heatmap(weighted_df, annot=True, cmap='YlOrRd', fmt='.1f')
        plt.title("Weighted Ancestry Composition Across All Samples")
        plt.xlabel("Reference Population")
        plt.ylabel("Sample ID")
        plt.tight_layout()
        plt.savefig(os.path.join(output_dir, "all_samples_heatmap.png"),␣
↪dpi=300)
        plt.close()

        # Create stacked bar chart
        plt.figure(figsize=(15, 10))
        weighted_df_pct = weighted_df.copy()
        # Normalize rows to 100%
        for idx in weighted_df_pct.index:
            row_sum = weighted_df_pct.loc[idx].sum()
            if row_sum > 0:
                weighted_df_pct.loc[idx] = weighted_df_pct.loc[idx] / row_sum *␣
↪100

        weighted_df_pct.plot(kind='barh', stacked=True, colormap='tab20')
        plt.title("Ancestry Composition Across All Samples")
        plt.xlabel("Percentage")
        plt.ylabel("Sample ID")
        plt.legend(title="Population", bbox_to_anchor=(1.05, 1), loc='upper␣
↪left')
        plt.tight_layout()
        plt.savefig(os.path.join(output_dir, "all_samples_stacked.png"),␣
↪dpi=300)
        plt.close()
    print("\nBatch analysis complete!")
    return all_results

############################
# 5. Main Entry Point      #
############################
def main(seg_file, fam_file, dict_file=None,
         output_dir="ancestry_results",
         rfmix_dir=None,
         sample_list=None):
    """
    Main function to run the analysis pipeline.

    Args:
        seg_file: Path to the IBD segment file
        fam_file: Path to the sample metadata file
        dict_file: Path to ID mapping file (optional)
        output_dir: Directory to save results
        rfmix_dir: Directory containing RFMix results (optional)
        sample_list: List of sample IDs to analyze (optional)
    """
    os.makedirs(output_dir, exist_ok=True)

    print("1. Loading genetic data...")
    segments, individuals, individual_to_bonsai = load_genetic_data(seg_file,␣
 ↪fam_file, dict_file)

    if segments is None:
        print("Error loading data. Exiting.")
        return

    # Load 1000 Genomes population information
    reference_file = os.path.join(os.path.dirname(fam_file),␣
 ↪"20140502_complete_sample_summary.txt")
    if os.path.exists(reference_file):
        print(f"Loading reference population information from {reference_file}")
        sample_df = pd.read_csv(reference_file, sep='\t')
    else:
        print("Reference population file not found. Creating empty DataFrame.")
        sample_df = pd.DataFrame(columns=['Sample', 'Population', 'Population␣
 ↪Description'])

    # Identify reference populations
    reference_pops = identify_reference_populations(sample_df)

    # Identify project samples (if not provided)
    if sample_list is None:
```

```python
        project_samples = identify_project_samples(segments, sample_df)
    else:
        project_samples = sample_list
        print(f"Using provided list of {len(project_samples)} samples")

    # Limit to a reasonable number of samples for testing
    if len(project_samples) > 10:
        print(f"Limiting analysis to first 10 samples for demonstration")
        project_samples = project_samples[:10]

    # Run batch analysis
    results = batch_ancestry_analysis(
        project_samples, segments, sample_df, output_dir, rfmix_dir
    )

    print("\nAnalysis pipeline completed!")

if __name__ == "__main__":
    # Example usage
    if len(sys.argv) > 2:
        seg_file = sys.argv[1]
        fam_file = sys.argv[2]
        dict_file = sys.argv[3] if len(sys.argv) > 3 else None
        main(seg_file, fam_file, dict_file)
    else:
        # Default example files
        seg_file = "../data/class_data/ped_sim_run2.seg"
        fam_file = "../data/class_data/ped_sim_run2-everyone.fam"
        dict_file = "../data/class_data/ped_sim_run2.seg_dict.txt"
        main(seg_file, fam_file, dict_file)
```

[ ]:
```
!poetry run jupyter nbconvert --to pdf ibd_1000_genomes.ipynb
```