Tanmay Bhatkar-9
Mazin Bangi-8
Sahil Bangera-7
Shannen Anthony-4

DS-Experiment 10

Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import spotipy
from spotipy.oauth2 import SpotifyOAuth
from fuzzywuzzy import fuzz
import os
from dotenv import load_dotenv
from collections import Counter
import time
import requests
import webbrowser
from urllib.parse import urlparse, parse_qs
import http.server
import socketserver
import threading
from sklearn.decomposition import PCA

# Load environment variables for Spotify API
load_dotenv()
feature_weights = {
    'valence': 1.8,       # Increase impact of mood/positivity
    'energy': 1.5,        # Keep energy as important
    'danceability': 1.3,  # Slightly increase danceability
    'acousticness': 1.7,  # Increase impact of acoustic vs electronic
    'instrumentalness': 1.8,  # Significantly increase instrumentalness weight
    'explicit': 1.6       # Increase explicit content weight
}
# Step 1: Load and preprocess the Spotify dataset
def load_and_preprocess_data(file_path):
    """Load the Spotify dataset and preprocess it for clustering."""
    print("Loading dataset...")
    df = pd.read_csv(file_path)

    # Select relevant features for clustering

features = ['valence', 'acousticness', 'danceability', 'energy', 'instrumentalness']

    # Handle explicit column (convert to numeric if it's not)
    if 'explicit' in df.columns:
        if df['explicit'].dtype == 'object':
            df['explicit'] = df['explicit'].map({'True': 1, 'False': 0})
        features.append('explicit')

    # Drop rows with missing values
    df_clean = df[features].dropna()

    # applying feature weights
    for feature, weight in feature_weights.items():
        if feature in df_clean.columns:
            df_clean[feature] = df_clean[feature] * weight

    # Normalize the features
    scaler = StandardScaler()
    df_scaled = pd.DataFrame(
        scaler.fit_transform(df_clean),
        columns=features
    )

    return df_clean, df_scaled, scaler, features

# Step 2: Create clusters using KMeans
def create_clusters(df_scaled, n_clusters=10):
    """Create clusters using KMeans algorithm."""
    print(f"Creating {n_clusters} clusters...")
    # Suppress the warning about number of cores
    import os

os.environ["LOKY_MAX_CPU_COUNT"] = "4"  # Set to a reasonable number for your system

    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)

clusters = kmeans.fit_predict(df_scaled)

    # Add cluster labels to the dataframe
    df_with_clusters = df_scaled.copy()
    df_with_clusters['cluster'] = clusters

    return kmeans, df_with_clusters

# Step 3: Analyze clusters to create archetypes
def analyze_clusters(df_original, clusters, features):
```

```
    """Analyze each cluster to determine its characteristics."""
    print("Analyzing clusters...")
    df_analysis = df_original.copy()
    df_analysis['cluster'] = clusters

    cluster_profiles = {}

    for cluster in range(max(clusters) + 1):
        cluster_data = df_analysis[df_analysis['cluster'] == cluster]
        profile = {}

        for feature in features:
            mean_val = cluster_data[feature].mean()
            profile[feature] = mean_val

        cluster_profiles[cluster] = profile

    return cluster_profiles

# Step 4: Map clusters to archetypes
def map_clusters_to_archetypes(cluster_profiles):
    """Map numerical cluster profiles to meaningful archetypes."""
    archetypes = {
        "The Energetic Socializer": {
            "traits": {
                "danceability": "high",
                "energy": "high",
                "valence": "high",
                "acousticness": "low"
            }
        },
        "The Contemplative Introvert": {
            "traits": {
                "acousticness": "high",
                "energy": "low",
                "valence": "medium",
                "danceability": "low"
            }
        },
        "The Rebellious Free Spirit": {
            "traits": {
                "energy": "high",
                "valence": "medium",
                "explicit": "high",
                "instrumentalness": "low"
            }
        },
        "The Nostalgic Romantic": {
            "traits": {
                "valence": "medium",
                "danceability": "medium",
                "explicit": "low",
                "acousticness": "medium"
            }
        },
        "The Focused Intellectual": {
            "traits": {
                "instrumentalness": "high",
                "danceability": "low"
            }
        },
        "The Global Explorer": {
            "traits": {
                "danceability": "high",
                "energy": "medium-high",
                "acousticness": "medium",
                "valence": "high"
            }
        },
        "The High-Octane Athlete": {
            "traits": {
                "energy": "very high",
                "explicit": "high",
                "valence": "medium-high"
            }
        },
        "The Laid-Back Stoner": {
            "traits": {
                "energy": "low",
                "valence": "medium-high"
            }
        },
        "The Heartbroken Healer": {
            "traits": {
                "valence": "low",
                "danceability": "low",
                "acousticness": "high"
            }
        },
        "The Trendy Mainstreamer": {
            "traits": {
                "valence": "high",
                "danceability": "medium"
            }
        }
    }

    # Find closest archetype for each cluster
    cluster_to_archetype = {}
    already_assigned = set()

    for cluster, profile in sorted(cluster_profiles.items(),key=lambda x: sum(x[1].values())):  # Sort clusters by feature sum
        best_match = None
        best_score = -float('inf')
        second_best = None
        second_score = -float('inf')

        for archetype, arch_profile in archetypes.items():
            score = 0

            # Calculate match score based on profile traits
            for trait, value in arch_profile["traits"].items():
                if trait in profile:
                    feature_val = profile[trait]
                    if value == "low" and feature_val < 0.4:
                        score += (1 - feature_val) * 2
                    elif value == "medium" and 0.3 <= feature_val <= 0.7:
                        score += 1 - abs(feature_val - 0.5) * 2
                    elif value == "medium-high" and 0.4 <= feature_val <= 0.8:
                        score += 1 - abs(feature_val - 0.6) * 2
                    elif value == "high" and feature_val > 0.6:
                        score += feature_val * 2
                    elif value == "very high" and feature_val > 0.8:
```

```
                score += feature_val *
2.5

        if score > best_score:
            second_best = best_match
            second_score = best_score
            best_score = score
            best_match = archetype
        elif score > second_score:
            second_score = score
            second_best = archetype
    if best_match in already_assigned
and second_score > best_score * 0.8:
        cluster_to_archetype[cluster] =
second_best

already_assigned.add(second_best)
    else:
        cluster_to_archetype[cluster] =
best_match

already_assigned.add(best_match)

  missing_archetypes =
set(archetypes.keys()) -
already_assigned

  # If some archetypes weren't
assigned, try to assign them to
next-best matches
  if missing_archetypes and
len(cluster_profiles) >=
len(archetypes):
    # Find the best clusters to reassign
    for archetype in
missing_archetypes:
        best_cluster = None
        best_fit_score = -float('inf')

        for cluster, profile in
cluster_profiles.items():
            # Calculate fit score for this
archetype
            score = 0
            for trait, value in
archetypes[archetype]["traits"].items():
                if trait in profile:
                    feature_val =
profile[trait]
                    # Similar scoring as
above
                    if value == "low" and
feature_val < 0.4:
                        score += 1
                    elif value == "medium"
and 0.3 <= feature_val <= 0.7:
                        score += 1
                    elif value == "high" and
feature_val > 0.6:
                        score += 1

            if score > best_fit_score:
                best_fit_score = score
                best_cluster = cluster

        if best_cluster is not None:

cluster_to_archetype[best_cluster] =
archetype

  return cluster_to_archetype

# Updated Spotify OAuth setup
def setup_spotify_oauth():
  """Setup OAuth 2.0 for Spotify API
with user authorization flow."""
  client_id =
os.getenv("SPOTIFY_CLIENT_ID")
  client_secret =
os.getenv("SPOTIFY_CLIENT_SECR
ET")
  # Use loopback IP address instead of
localhost
  redirect_uri =
os.getenv("SPOTIFY_REDIRECT_UR
I", "http://127.0.0.1:8888/callback")

  if not client_id or not client_secret:
    raise ValueError("Spotify API
credentials not found. Set
SPOTIFY_CLIENT_ID and
SPOTIFY_CLIENT_SECRET in your
environment.")

  # Define the scope needed for access
  scope = "user-library-read
playlist-read-private user-read-email"

  # Create OAuth manager
  sp_oauth = SpotifyOAuth(
    client_id=client_id,
    client_secret=client_secret,
    redirect_uri=redirect_uri,
    scope=scope,
    cache_path=".spotifycache"
  )

  # Check if we have a valid token
cached
  token_info =
sp_oauth.get_cached_token()

  if not token_info or
sp_oauth.is_token_expired(token_info)
:
    # We need to get a new token
    auth_url =
sp_oauth.get_authorize_url()
    print(f"Please navigate to this
URL to authorize the app: {auth_url}")

    # Open web browser for
authorization
    webbrowser.open(auth_url)

    # Set up a simple HTTP server to
catch the callback
    auth_code = None

    class
AuthHandler(http.server.SimpleHTTP
RequestHandler):
      def do_GET(self):
        nonlocal auth_code
        query =
urlparse(self.path).query
        if query:
          params = parse_qs(query)
          if 'code' in params:
            auth_code =
params['code'][0]
            self.send_response(200)

self.send_header('Content-type',
'text/html')
            self.end_headers()
```

```
self.wfile.write(b"Authentication successful! You can close this window.")
                return

            self.send_response(404)
            self.end_headers()

    # Start HTTP server with the loopback IP, not "localhost"
    httpd = socketserver.TCPServer(("127.0.0.1", 8888), AuthHandler)

    # Run server in a separate thread
    server_thread = threading.Thread(target=httpd.handle_request)
    server_thread.daemon = True
    server_thread.start()

    # Wait for auth code
    print("Waiting for authorization...")
    while auth_code is None:
        time.sleep(1)

    # Exchange code for token
    token_info = sp_oauth.get_access_token(auth_code)

  # Create Spotipy client with token
  sp = spotipy.Spotify(auth=token_info['access_token'])

  return sp, token_info['access_token']

# Updated function to get playlist tracks using current API
def get_playlist_tracks(sp, playlist_id):
  """Get all tracks from a playlist using the current Spotify API."""
  print(f"Fetching tracks from playlist: {playlist_id}")

  try:
    # First, verify the playlist exists by fetching its metadata
    playlist_data = sp.playlist(playlist_id)
    print(f"Found playlist: {playlist_data['name']} by {playlist_data['owner']['display_name']}")

    # Get all tracks using pagination
    results = sp.playlist_items(playlist_id)
    tracks = results['items']

    while results['next']:
      results = sp.next(results)
      tracks.extend(results['items'])

    print(f"Found {len(tracks)} tracks in playlist")

    # Extract track IDs and info
    track_data = []
    track_ids = []

    for item in tracks:
      # Skip non-track items (like podcasts)
      if 'track' not in item or not item['track']:
        continue

      track = item['track']

      # Skip local tracks or None tracks
      if track is None or track.get('id') is None:
        continue

      track_data.append({
        'id': track['id'],
        'track_name': track['name'],
        'artist': track['artists'][0]['name'] if track['artists'] else "Unknown",
        'explicit': 1 if track.get('explicit', False) else 0
      })
      track_ids.append(track['id'])

    return track_ids, track_data

  except Exception as e:
    print(f"Error fetching playlist: {e}")
    return None, None

# Search for tracks by query
def search_tracks(sp, queries):
  """Search for tracks by name/artist and return their IDs."""
  print("Searching for tracks...")

  track_ids = []
  track_data = []

  for query in queries:
    try:
      results = sp.search(q=query, type='track', limit=1)

      if not results['tracks']['items']:
        print(f"No tracks found for query: '{query}'")
        continue

      track = results['tracks']['items'][0]
      track_ids.append(track['id'])

      track_data.append({
        'id': track['id'],
        'track_name': track['name'],
        'artist': track['artists'][0]['name'] if track['artists'] else "Unknown",
        'explicit': 1 if track.get('explicit', False) else 0
      })

      print(f"Found: {track['name']} by {track['artists'][0]['name'] if track['artists'] else 'Unknown'}")

    except Exception as e:
      print(f"Error searching for '{query}': {e}")
```

```
            continue

    return track_ids, track_data

# Get audio features for tracks
"""def get_audio_features(sp,
track_ids):
    print("Fetching audio features...")

    all_features = []

    # Process in batches of 100 (Spotify
API limit)
    for i in range(0, len(track_ids), 100):
        batch = track_ids[i:i+100]

        try:
            features =
sp.audio_features(batch)
            # Filter out None values
            features = [f for f in features if
f]
            all_features.extend(features)

            # Brief pause to avoid rate
limiting
            time.sleep(0.5)

        except Exception as e:
            print(f"Error fetching audio
features for batch: {e}")

    return all_features"""

# Modified function to replace
get_audio_features
def
match_tracks_with_dataset(track_data,
dataset_path):
    """Match user tracks with tracks in
the dataset."""
    print("Matching tracks with
dataset...")

    # Load dataset
    try:
        dataset =
pd.read_csv(dataset_path)
    except Exception as e:
        print(f"Error loading dataset:
{e}")
        return None

    # Check if 'id' column exists in
dataset
    if 'track_id' not in dataset.columns:
        print("Dataset doesn't contain
track IDs. Using track name and artist
for matching.")
        # Create a column for matching
based on name and artist
        if 'track_name' in dataset.columns
and 'artist_name' in dataset.columns:
            dataset['match_key'] =
dataset['track_name'].str.lower() + " - "
+ dataset['artist_name'].str.lower()
        else:
            print("Dataset doesn't contain
necessary columns for matching.")
            return None

    # Create a list to store matched track
data
    matched_tracks = []

    for track in track_data:
        track_id = track['id']
        track_name =
track['track_name'].lower()
        artist = track['artist'].lower()

        # Try to match by ID first if
available
        if 'track_id' in dataset.columns:
            match =
dataset[dataset['track_id'] == track_id]

            if not match.empty:
                # Found a match by ID
                track_row =
match.iloc[0].to_dict()
                track_row.update({
                    'name':
track['track_name'],
                    'artist': track['artist'],
                    'explicit': track['explicit']
                })
matched_tracks.append(track_row)
                continue

        # If no ID match or ID not
available, try name and artist
        match_key = f"{track_name} -
{artist}"

        if 'match_key' in dataset.columns:
            # Find closest match
            dataset['similarity'] =
dataset['match_key'].apply(
                lambda x: fuzz.ratio(x,
match_key) if isinstance(x, str) else 0
            )

            # Get best match above
threshold
            best_match =
dataset.loc[dataset['similarity'].idxmax(
)]

            if best_match['similarity'] > 70:
# 70% similarity threshold
                track_row =
best_match.to_dict()
                track_row.update({
                    'name':
track['track_name'],
                    'artist': track['artist'],
                    'explicit': track['explicit']
                })

matched_tracks.append(track_row)
            else:
                print(f"No good match found
for: {track['track_name']} by
{track['artist']}")
        else:
            print(f"Skipping track without
match: {track['track_name']} by
{track['artist']}")

    if matched_tracks:
        # Create DataFrame from matched
tracks
        matched_df =
pd.DataFrame(matched_tracks)
```

```
        # Ensure all required features exist
        required_features = ['valence',
'acousticness', 'danceability', 'energy',
'instrumentalness', 'explicit']
        for feature in required_features:
            if feature not in
matched_df.columns:
                if feature == 'explicit' and
'explicit' in track_data[0]:
                    # We have this from
track_data
                    continue
                print(f"Warning: Required
feature '{feature}' missing from
matched data")
                # Add default middle value
                matched_df[feature] = 0.5

        return matched_df
    else:
        print("No tracks could be matched
with the dataset.")
        return None

# Process tracks and their audio
features
def process_audio_data(track_data,
audio_features):
    """Combine track data with audio
features and create DataFrame."""
    if not audio_features:
        print("No audio features found.")
        return None

    # Create a dictionary mapping track
IDs to their audio features
    features_dict = {feature['id']: feature
for feature in audio_features if feature}

    # Combine track data with audio
features
    processed_data = []

    for track in track_data:
        track_id = track['id']

        if track_id in features_dict:
            features =
features_dict[track_id]

            track_info = {
                'name': track['track_name'],
                'artist': track['artist'],
                'valence': features['valence'],
                'acousticness':
features['acousticness'],
                'danceability':
features['danceability'],
                'energy': features['energy'],
                'instrumentalness':
features['instrumentalness'],
                'explicit': track['explicit']
            }


processed_data.append(track_info)

    if processed_data:
        return
pd.DataFrame(processed_data)
    else:
        print("Could not match any tracks
with their audio features.")
        return None

# Classify user based on their tracks
def classify_user(music_df, kmeans,
scaler, features, cluster_to_archetype):
    """Classify a user based on their
music into archetypes."""
    print("Classifying user based on
music...")

    # Extract feature columns from
music dataframe
    music_features =
music_df[features].copy()

    # Scale the features using the same
scaler used for the dataset
    music_scaled = pd.DataFrame(
        scaler.transform(music_features),
        columns=features
    )

    # Predict clusters for each track
    track_clusters =
kmeans.predict(music_scaled)

    # Map clusters to archetypes
    track_archetypes =
[cluster_to_archetype[cluster] for
cluster in track_clusters]

    # Create DataFrame with track info
and archetypes
    result_df = music_df[['name',
'artist']].copy()
    result_df['archetype'] =
track_archetypes
    result_df['cluster'] = track_clusters

    # Count frequency of each archetype
    archetype_counts =
Counter(track_archetypes)
    top_archetypes =
archetype_counts.most_common()

    # Calculate percentages
    total_tracks = len(track_archetypes)
    archetype_percentages = {archetype:
count/total_tracks*100 for archetype,
count in archetype_counts.items()}

    return result_df, top_archetypes,
archetype_percentages

# Visualize user's archetype distribution
def
visualize_user_archetypes(archetype_p
ercentages):
    """Create a visual representation of
user's archetype distribution."""
    plt.figure(figsize=(12, 8))

    # Create bar chart
    archetypes =
list(archetype_percentages.keys())
    percentages =
list(archetype_percentages.values())

    # Sort by percentage
    sorted_indices =
np.argsort(percentages)[::-1]
```

```
    archetypes = [archetypes[i] for i in sorted_indices]
    percentages = [percentages[i] for i in sorted_indices]

    colors = plt.cm.viridis(np.linspace(0, 0.8, len(archetypes)))

    # Create horizontal bar chart
    plt.barh(archetypes, percentages, color=colors)
    plt.xlabel('Percentage of Music (%)')
    plt.title('Your Music Personality Profile')
    plt.tight_layout()


plt.savefig('music_personality_profile.png')
    print("Visualization saved as 'music_personality_profile.png'")
    plt.show()

# NEW FUNCTION: Visualize clusters
def visualize_clusters(df_scaled, clusters, cluster_to_archetype):
    """Create a visual representation of clusters using PCA."""
    print("Visualizing clusters...")

    # Use PCA to reduce dimensionality to 2D for visualization
    pca = PCA(n_components=2)
    df_pca = pca.fit_transform(df_scaled.drop('cluster', axis=1, errors='ignore'))

    # Create a DataFrame with PCA components and cluster labels
    df_plot = pd.DataFrame({
        'PCA1': df_pca[:, 0],
        'PCA2': df_pca[:, 1],
        'Cluster': clusters,
        'Archetype': [cluster_to_archetype.get(c, "Unknown") for c in clusters]
    })

    # Create a colorful scatter plot
    plt.figure(figsize=(12, 10))

    # Get unique archetypes for color mapping
    unique_archetypes = list(set(cluster_to_archetype.values()))
    # Create a color map
    colors = plt.cm.tab10(np.linspace(0, 1, len(unique_archetypes)))
    color_map = {archetype: color for archetype, color in zip(unique_archetypes, colors)}

    # Plot each cluster with its own color
    for archetype in unique_archetypes:
        subset = df_plot[df_plot['Archetype'] == archetype]
        plt.scatter(
            subset['PCA1'],
            subset['PCA2'],
            alpha=0.7,
            s=50,
            label=archetype,
            color=color_map[archetype]
        )

    plt.title('Music Personality Clusters')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()

    plt.savefig('music_clusters.png')
    print("Cluster visualization saved as 'music_clusters.png'")
    plt.show()

# Generate personality report
def generate_user_report(top_archetypes, archetype_percentages):
    """Generate a detailed personality report based on the user's top archetypes."""
    print("\n🎵 YOUR MUSIC PERSONALITY REPORT 🎵\n")

    # Archetype descriptions
    archetype_descriptions = {
        "The Energetic Socializer":
            "You're likely extroverted, energetic, and thrive in social settings. Your music "
            "suggests you use it to boost your mood and connect with others. "
            "Your musical choices reflect a love for upbeat, danceable tracks that get people moving.",

        "The Contemplative Introvert":
            "Your music suggests you have a thoughtful, introspective nature. You likely appreciate "
            "solitude and use music for reflection and relaxation. Your playlist indicates a preference "
            "for acoustic tracks with emotional depth and lyrical authenticity.",

        "The Rebellious Free Spirit":
            "Your music reveals a bold, nonconformist streak. You're likely drawn to music that "
            "pushes boundaries and expresses raw emotion. You might use music as a form of rebellion "
            "or self-expression, gravitating toward high-energy tracks with unfiltered lyrics.",

        "The Nostalgic Romantic":
            "Your music choices suggest a sentimental, emotionally attuned personality. You likely "
            "value deep connections and emotional storytelling in your music. Your selections indicate "
            "a soft spot for love songs and music that evokes fond memories.",

        "The Focused Intellectual":
            "Your music reveals an analytical, detail-oriented mind. You likely appreciate complexity "
```

```
        "and depth in your music choices. Your preference for instrumental tracks suggests you may "
        "use music to aid concentration or immerse yourself in sophisticated sound structures.",

    "The Global Explorer":
        "Your musical choices indicate curiosity about different cultures and sounds. You're likely "
        "open-minded and adventurous, eager to discover new rhythms and musical traditions from around "
        "the world. Your music suggests you value diversity and cross-cultural connections.",

    "The High-Octane Athlete":
        "Your music reveals a highly energetic, possibly competitive personality. You likely use "
        "music to motivate yourself during physical activities. Your preference for driving beats and "
        "high-energy tracks suggests determination and a desire to push your limits.",

    "The Laid-Back Stoner":
        "Your music suggests a relaxed, possibly philosophical outlook on life. You may appreciate "
        "music for its immersive, mind-expanding qualities. Your selections indicate a preference for "
        "laid-back vibes and tracks that create a specific atmosphere.",

    "The Heartbroken Healer":
        "Your music reveals emotional sensitivity and depth. You likely connect deeply with music "
        "that explores themes of loss, longing, or emotional processing. Your musical choices suggest "
        "you may use songs as a form of emotional catharsis or healing.",

    "The Trendy Mainstreamer":
        "Your music choices indicate you stay connected to current popular culture. You likely enjoy "
        "being 'in the know' about trending songs and artists. Your selections suggest you value music "
        "that's familiar and shareable within your social circles."
  }

  # Display primary archetype
  primary_archetype, primary_percentage = top_archetypes[0]
  print(f"Primary Archetype: {primary_archetype} ({primary_percentage:.1f}%)")
  print("-" * 50)

print(archetype_descriptions[primary_archetype])
  print("\n")

  # Display secondary archetype if it has a significant presence
  if len(top_archetypes) > 1 and top_archetypes[1][1] > 15:  # If second archetype is >15%
      secondary_archetype, secondary_percentage = top_archetypes[1]
      print(f"Secondary Archetype: {secondary_archetype} ({secondary_percentage:.1f}%)")
      print("-" * 50)

print(archetype_descriptions[secondary_archetype])
      print("\n")

  # Display full breakdown
  print("FULL PERSONALITY BREAKDOWN:")
  print("-" * 50)
  for archetype, percentage in sorted(archetype_percentages.items(), key=lambda x: x[1], reverse=True):
      print(f"{archetype}: {percentage:.1f}%")

# Get user input
def get_user_input():
  """Get user input for either a playlist or individual tracks."""
  print("\nHow would you like to analyze your music taste?")
  print("1. Analyze a Spotify playlist")
  print("2. Enter up to 10 specific songs")

  choice = input("Enter your choice (1 or 2): ")

  if choice == "1":
      playlist_url = input("Enter your Spotify playlist URL or ID: ")

      # Extract playlist ID from URL if needed
      if "spotify.com/playlist/" in playlist_url:
          playlist_id = playlist_url.split("spotify.com/playlist/")[1].split("?")[0]
      else:
          playlist_id = playlist_url  # Assume they provided the ID directly

      return {"type": "playlist", "id": playlist_id}
  elif choice == "2":
      tracks = []
      print("\nEnter up to 10 songs (format: Artist - Song Title)")
      print("Enter 'done' when finished")

      for i in range(10):
          track = input(f"Song {i+1}: ")
          if track.lower() == 'done':
              break
          tracks.append(track)

      return {"type": "tracks", "list": tracks}
  else:
```

```
        print("Invalid choice. Defaulting to playlist option.")
        playlist_url = input("Enter your Spotify playlist URL or ID: ")

        # Extract playlist ID from URL if needed
        if "spotify.com/playlist/" in playlist_url:
            playlist_id = playlist_url.split("spotify.com/playlist/")[1].split("?")[0]
        else:
            playlist_id = playlist_url  # Assume they provided the ID directly

        return {"type": "playlist", "id": playlist_id}

# Main function
def main():
    try:
        # Step 1: Load the dataset for clustering
        dataset_path = "combined_unique_audio_features.csv"  # Update with your dataset path
        df_original, df_scaled, scaler, features = load_and_preprocess_data(dataset_path)

        # Step 2: Create clusters
        kmeans, df_with_clusters = create_clusters(df_scaled, n_clusters=10)

        # Step 3: Analyze clusters
        cluster_profiles = analyze_clusters(df_original, df_with_clusters['cluster'], features)

        # Step 4: Map clusters to archetypes
        cluster_to_archetype = map_clusters_to_archetypes(cluster_profiles)

        # NEW: Visualize clusters
        visualize_clusters(df_with_clusters, df_with_clusters['cluster'], cluster_to_archetype)

        # Step 5: Setup Spotify OAuth
        try:
            sp, token = setup_spotify_oauth()
            print("Successfully authenticated with Spotify API using OAuth")
        except Exception as e:
            print(f"Error authenticating with Spotify API: {e}")
            return

        # Step 6-7: Get user input and process music data
        user_input = get_user_input()

        if user_input["type"] == "playlist":
            # Get playlist tracks using updated method
            track_ids, track_data = get_playlist_tracks(sp, user_input["id"])
            if not track_ids:
                print("Could not retrieve tracks from playlist. Please try another playlist or enter tracks manually.")
                return
        else:
            # Search for individual tracks
            track_ids, track_data = search_tracks(sp, user_input["list"])
            if not track_ids:
                print("Could not find any valid tracks. Please try again with different search terms.")
                return

        # Get audio features for the tracks
        #audio_features = get_audio_features(sp, track_ids)

        # Process the audio data
        music_df = match_tracks_with_dataset(track_data, dataset_path)

        if music_df is None or music_df.empty:
            print("Could not process music data. Please try again.")
            return

        # Step 8: Classify user
        result_df, top_archetypes, archetype_percentages = classify_user(
            music_df, kmeans, scaler, features, cluster_to_archetype
        )

        # Step 9: Visualize results
        visualize_user_archetypes(archetype_percentages)

        # Step 10: Generate report
        generate_user_report(top_archetypes, archetype_percentages)

        # Show track-by-track breakdown
        print("\nTRACK-BY-TRACK BREAKDOWN:")
        print("-" * 50)
        print(result_df[['name', 'artist', 'archetype']])

    except Exception as e:
        print(f"An error occurred: {e}")
        import traceback
        traceback.print_exc()

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

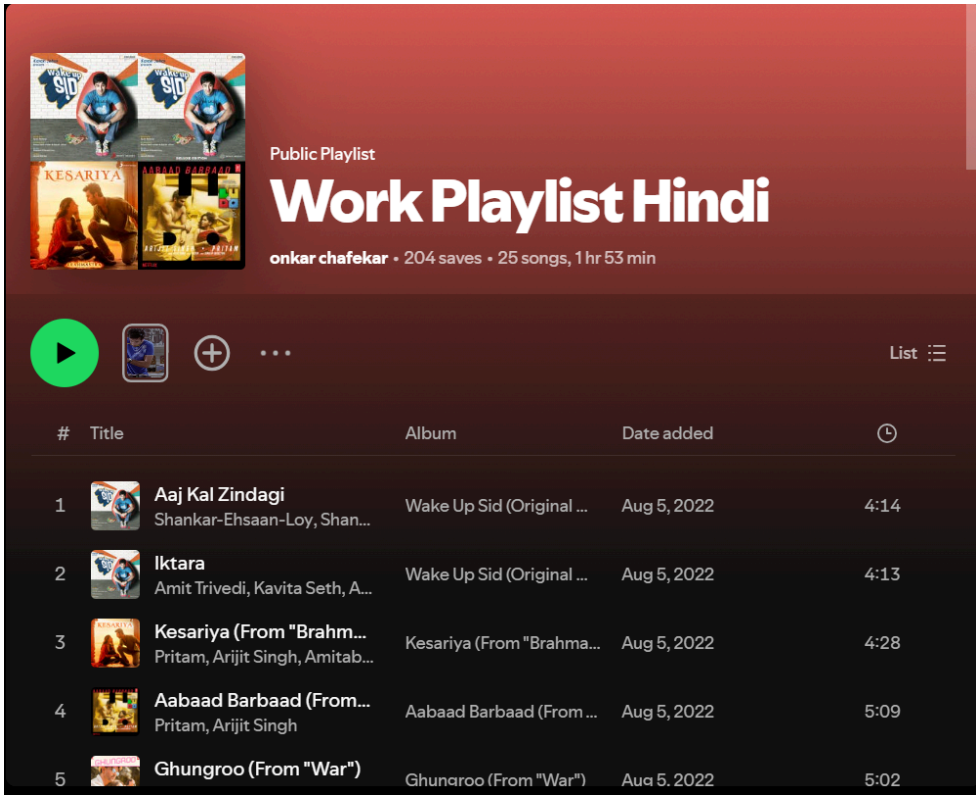
Dataset

music_personality.py | combined_unique_audio_features.csv

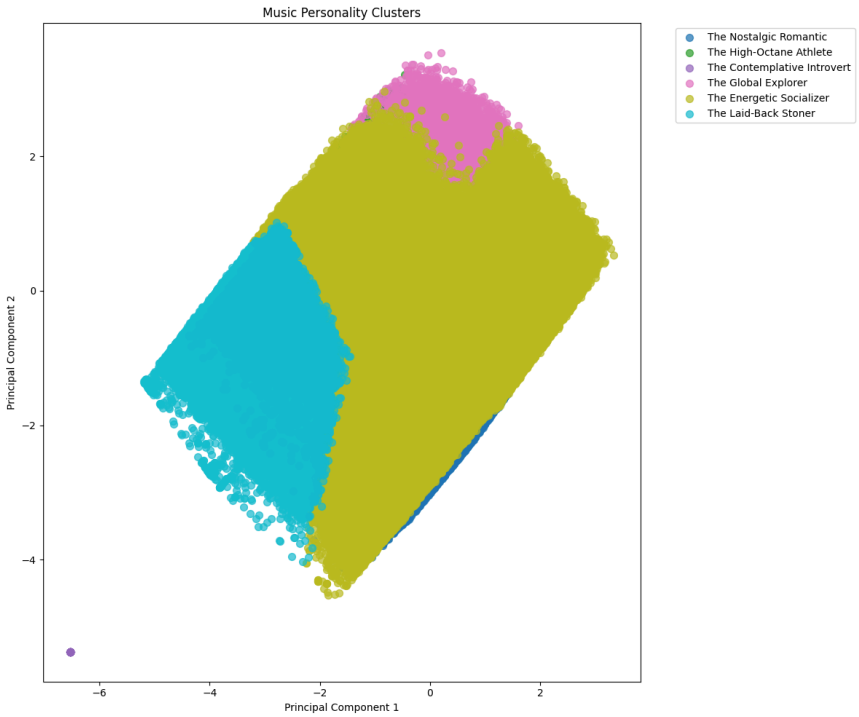
combined_unique_audio_features.csv

```
track_id,track_name,valence,acousticness,danceability,energy,instrumentalness
2RM4jf1Xa9zPgMGRDiht8O,"Big Bank feat. 2 Chainz, Big Sean, Nicki Minaj",0.118,0.00582,0.743,0.3389999999999999,0.0
1tHDG53xJNGsItRA3vfVgs,BAND DRUM (feat. A$AP Rocky),0.371,0.0244,0.846,0.557,0.0
6Wosx2euFPMT14UXiWudMy,Radio Silence,0.382,0.025,0.603,0.723,0.0
3J2Jpw61sO7l6Hc7qdYV91,Lactose,0.6409999999999999,0.0294,0.8,0.579,0.912
2jbYvQCyPgX3CdmAzeVeuS,Same - Original mix,0.928,3.52e-05,0.7829999999999999,0.792,0.878
26Y1lX7ZOpw9Ql3gGAlqLK,Debauchery - Original mix,0.8370000000000001,0.00115,0.81,0.417,0.919
5eIyK73BrxHLnly4F9PWqg,Grandma - Original mix,0.934,0.000539,0.8190000000000001,0.72,0.863
13Mf2ZBpfNkgWJowvM5hXh,Bon appétit,0.18,0.115,0.885,0.348,0.0
7BQaRTHk44DkMhIVNcXy2D,Among - Original mix,0.622,5.84e-05,0.74,0.472,0.847
049RxG2laEl9U1PGYeIqLV,Hazard - Original mix,0.944,8.109999999999999e-05,0.813,0.731,0.91
118GQ70Sp6pMqn6w1oKuki,Strummer - Original mix,0.8959999999999999,0.000784,0.787,0.83,0.784
6S7cr72a7a8RVAXzDCRj6m,Big Racks - Original mix,0.695,0.000106,0.777,0.865,0.812
7h2qWpMJzIVtiP30E8VDW4,Rulet - Original mix,0.964,0.000223,0.7959999999999999,0.698,0.919
3KVQFxJ5CWOcbxdpPYdi4o,Head - Original mix,0.489,3.74e-05,0.815,0.7090000000000001,0.938
0JjNrI1xmsTfhaiU1R6OVc,Defeding - Original mix,0.7659999999999999,1.35e-05,0.799,0.573,0.915
3HjTcZt29JUHg5m60QhlMw,Jupiter - Original mix,0.503,7.42e-05,0.812,0.581,0.914
42LWRdkWxM9aWmDImWvH6C,Koonra - Original mix,0.653,0.000851,0.8109999999999999,0.7090000000000001,0.929
32dMH9MvlTJaABrPHY52Yb,Spouse - Original mix,0.0359,0.0315,0.746,0.645,0.966
5RCPsfzmEpTXMCTNk7wEfQ,Prop - Original mix,0.623,0.000256,0.813,0.598,0.8909999999999999
0y0mwXrdEzjUK5Nq8GDPnY,Particular - Original mix,0.585,9.1e-06,0.812,0.504,0.944
3RSMqu36JZnmMkrnNmnqyd,Tremor - Original mix,0.591,0.000428,0.8140000000000001,0.8240000000000001,0.738
1o0fkWCltFHVeFIRHqvR5b,Cent - Original mix,0.503,0.000133,0.813,0.703,0.927
2iGShSeV6WcDbez5SLJ2bJ,Pantheon - Original mix,0.523,7.01e-05,0.81,0.537,0.932
2rNTo0tGUMW6rn0uHzV5er,Samplerup - Original mix,0.616,6.66e-06,0.8059999999999999,0.5870000000000001,0.895999999999
```

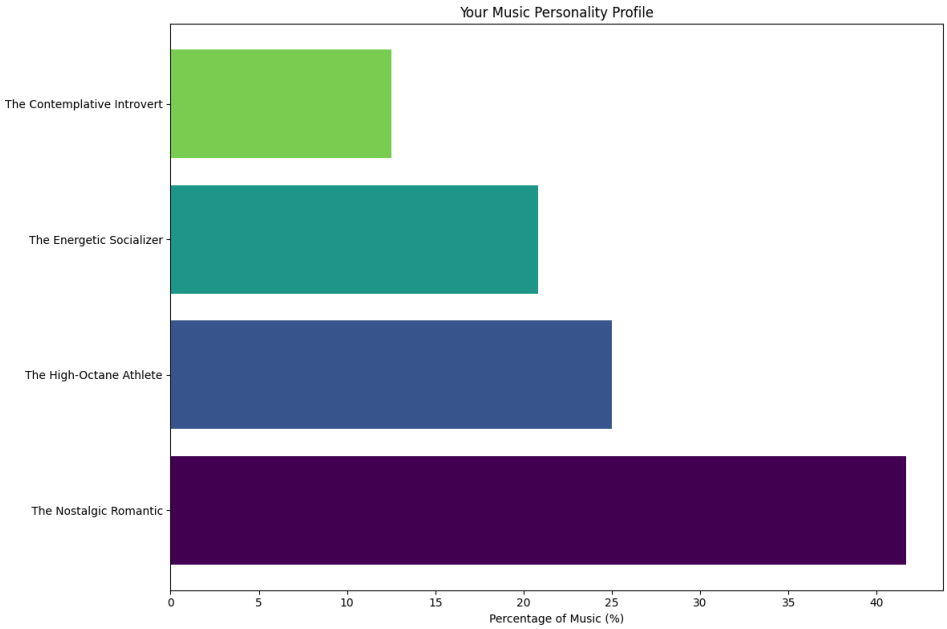
Dataset of 800,000 songs and their song feature parameters(Valence, Energy, Acousticness, etc)



Chosen Playlist



Music Clusters



Music Personality Profile

```
(venv) C:\Users\tanma\Desktop\MusicCluster>python music_personality.py
Loading dataset...
Creating 10 clusters...
Analyzing clusters...
Visualizing clusters...
Cluster visualization saved as 'music_clusters.png'
Error authenticating with Spotify API: HTTPSConnectionPool(host='accounts.spotify.com', port=443): Max retries exceeded with url: /api/token (Caused by Name
ResolutionError("<urllib3.connection.HTTPSConnection object at 0x000001AFDC8BD760>: Failed to resolve 'accounts.spotify.com' ([Errno 11001] getaddrinfo fail
ed)"))

(venv) C:\Users\tanma\Desktop\MusicCluster>python music_personality.py
Loading dataset...
Creating 10 clusters...
Analyzing clusters...
Visualizing clusters...
Cluster visualization saved as 'music_clusters.png'
Successfully authenticated with Spotify API using OAuth

How would you like to analyze your music taste?
1. Analyze a Spotify playlist
2. Enter up to 10 specific songs
Enter your choice (1 or 2): 1
Enter your Spotify playlist URL or ID: https://open.spotify.com/playlist/74H6Zs2aXSeNu7xBizIN4q
Fetching tracks from playlist: 74H6Zs2aXSeNu7xBizIN4q
Found playlist: Work Playlist Hindi by onkar chafekar
Found 25 tracks in playlist
Matching tracks with dataset...
Skipping track without match: Jai Shri Ram (From "Adipurush") by Ajay-Atul
Classifying user based on music...
Visualization saved as 'music_personality_profile.png'

🎵 YOUR MUSIC PERSONALITY REPORT 🎵

Primary Archetype: The Nostalgic Romantic (10.0%)
--------------------------------------------------
Your music choices suggest a sentimental, emotionally attuned personality. You likely value deep connections and emotional storytelling in your music. Your
selections indicate a soft spot for love songs and music that evokes fond memories.


FULL PERSONALITY BREAKDOWN:
--------------------------------------------------
The Nostalgic Romantic: 41.7%
The High-Octane Athlete: 25.0%
The Energetic Socializer: 20.8%
The Contemplative Introvert: 12.5%
```

```
TRACK-BY-TRACK BREAKDOWN:
--------------------------------------------------
                           name              artist                    archetype
0                 Aaj Kal Zindagi  Shankar-Ehsaan-Loy      The Nostalgic Romantic
1                          Iktara        Amit Trivedi      The Nostalgic Romantic
2     Kesariya (From "Brahmastra")              Pritam      The Nostalgic Romantic
3     Aabaad Barbaad (From "Ludo")              Pritam     The High-Octane Athlete
4            Ghungroo (From "War")      Vishal-Shekhar    The Energetic Socializer
5                Agar Tum Saath Ho         Alka Yagnik  The Contemplative Introvert
6                Jashn-E-Bahaaraa         A.R. Rahman     The High-Octane Athlete
7   Raanjhanaa (From "Raanjhanaa")         A.R. Rahman    The Energetic Socializer
8                        Tere Bina         A.R. Rahman    The Energetic Socializer
9                       Tere Naina   Shankar Mahadevan     The High-Octane Athlete
10                 Dil Chahta Hai   Shankar Mahadevan    The Energetic Socializer
11                        Lakshya  Shankar-Ehsaan-Loy      The Nostalgic Romantic
12                   Taake Jhanke        Arijit Singh     The High-Octane Athlete
13                    Besabriyaan        Armaan Malik  The Contemplative Introvert
14                     Phir Kabhi        Arijit Singh      The Nostalgic Romantic
15                          Manja        Amit Trivedi     The High-Octane Athlete
16   Deva Deva (From "Brahmastra")              Pritam      The Nostalgic Romantic
17                 Buddhu Sa Mann         Amaal Mallik      The Nostalgic Romantic
18                 Aas Paas Khuda      Vishal-Shekhar      The Nostalgic Romantic
19              Raataan Lambiyan      Tanishk Bagchi    The Energetic Socializer
20                         Ranjha       Jasleen Royal      The Nostalgic Romantic
21                  Kabhii Tumhhe         Javed-Mohsin  The Contemplative Introvert
22                         Hairat      Vishal-Shekhar      The Nostalgic Romantic
23              Tujhe Bhula Diya      Vishal-Shekhar     The High-Octane Athlete

(venv) C:\Users\tanma\Desktop\MusicCluster>
```

Output on CMD