1/26/25

Music Exclusive Matching Dating App and through Spotify API integration create an algorithm for compatibility between users

-> even tho Tastebuds.fm (web) but app looks scuffed

-> apps of similar that exist: Vinylly, makromusic, Turn Up, Raya, Discz

-> use AI and data for matching algorithm

->n

Step 1: Setting Up Flask for Your Music-Matching App

Before we integrate Spotify's API, let's start by setting up the **Flask backend** as the foundation for your dating app. Flask is a lightweight Python web framework that will handle authentication, API requests, and matchmaking logic.

---

1. Install Dependencies

First, ensure you have Python installed. Then, install Flask and required packages:

bash
CopyEdit
```
pip install flask requests spotipy flask-cors flask_sqlalchemy
```

- **`flask`** → Main web framework
- **`requests`** → To handle API calls
- **`spotipy`** → Spotify API client
- **`flask-cors`** → Handle cross-origin resource sharing (useful for frontend)
- **`flask_sqlalchemy`** → ORM to manage user data

---

2. Set Up the Flask Project

Create a project folder, e.g., **`music-match-app`**, and inside it, create:

- `app.py` (Main Flask app)
- `.env` (Store API credentials)
- `config.py` (Configuration settings)
- `models.py` (Database models)
- `routes.py` (Define API endpoints)

---

3. Configure Flask & Spotify API

Create a `.env` file to store sensitive credentials:

ini
CopyEdit
```
SPOTIFY_CLIENT_ID=your_client_id_here
SPOTIFY_CLIENT_SECRET=your_client_secret_here
SPOTIFY_REDIRECT_URI=http://127.0.0.1:5000/callback
SECRET_KEY=supersecretkey
DATABASE_URL=sqlite:///music_match.db
```

Next, set up `config.py`:

python
CopyEdit
```
import os
from dotenv import load_dotenv

load_dotenv()

class Config:
    SECRET_KEY = os.getenv("SECRET_KEY")
    SPOTIFY_CLIENT_ID = os.getenv("SPOTIFY_CLIENT_ID")
    SPOTIFY_CLIENT_SECRET = os.getenv("SPOTIFY_CLIENT_SECRET")
```

```
    SPOTIFY_REDIRECT_URI = os.getenv("SPOTIFY_REDIRECT_URI")
    SQLALCHEMY_DATABASE_URI = os.getenv("DATABASE_URL")
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

---

4. Set Up the Flask App (`app.py`)

Now, let's set up Flask, the database, and register routes.

python
CopyEdit

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_cors import CORS
from config import Config

# Initialize Flask
app = Flask(__name__)
app.config.from_object(Config)

# Enable CORS (for frontend interaction)
CORS(app)

# Initialize database
db = SQLAlchemy(app)

# Import routes after initializing app (to avoid circular
imports)
from routes import *

if __name__ == "__main__":
    app.run(debug=True)
```

---

5. Define Database Models (`models.py`)

Create models for storing user data, preferences, and matches.

python
CopyEdit

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True,
nullable=False)
    spotify_id = db.Column(db.String(100), unique=True,
nullable=False)
    access_token = db.Column(db.String(255), nullable=False)
    refresh_token = db.Column(db.String(255), nullable=False)
    top_artists = db.Column(db.Text)  # Store as JSON string
    top_genres = db.Column(db.Text)  # Store as JSON string

    def __repr__(self):
        return f"<User {self.username}>"

class Match(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user1_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    user2_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    compatibility_score = db.Column(db.Float, nullable=False)

    def __repr__(self):
        return f"<Match {self.user1_id} - {self.user2_id}
(Score: {self.compatibility_score})>"
```

After defining models, **initialize the database** by running:

bash
CopyEdit

```
python
>>> from app import db
>>> db.create_all()
```

---

6. Set Up Spotify Authentication (`routes.py`)

We'll implement Spotify login using OAuth.

python
CopyEdit

```
from flask import request, redirect, session, jsonify
import requests
import json
import spotipy
from spotipy.oauth2 import SpotifyOAuth
from app import app, db
from models import User
from config import Config

# Spotify Auth Flow
sp_oauth = SpotifyOAuth(
    client_id=Config.SPOTIFY_CLIENT_ID,
    client_secret=Config.SPOTIFY_CLIENT_SECRET,
    redirect_uri=Config.SPOTIFY_REDIRECT_URI,
    scope="user-top-read"
)

@app.route("/login")
def login():
    auth_url = sp_oauth.get_authorize_url()
    return redirect(auth_url)
```

```
@app.route("/callback")
def callback():
    session.clear()
    code = request.args.get("code")
    token_info = sp_oauth.get_access_token(code)

    if not token_info:
        return jsonify({"error": "Failed to get access token"}),
400

    access_token = token_info['access_token']
    refresh_token = token_info['refresh_token']
    sp = spotipy.Spotify(auth=access_token)

    # Get user profile info
    user_info = sp.current_user()
    spotify_id = user_info["id"]
    username = user_info["display_name"]
    email = user_info["email"]

    # Get top artists and genres
    top_artists = sp.current_user_top_artists(limit=10)
    artist_names = [artist["name"] for artist in
top_artists["items"]]
    genres = list(set([genre for artist in top_artists["items"]
for genre in artist["genres"]]))

    # Save user data
    user = User.query.filter_by(spotify_id=spotify_id).first()
    if not user:
        user = User(username=username, email=email,
spotify_id=spotify_id, access_token=access_token,
```

```
                    refresh_token=refresh_token,
top_artists=json.dumps(artist_names),
top_genres=json.dumps(genres))
        db.session.add(user)
        db.session.commit()

    session["user_id"] = user.id
    return jsonify({"message": "Login successful", "username":
username, "top_artists": artist_names, "top_genres": genres})

@app.route("/logout")
def logout():
    session.clear()
    return jsonify({"message": "Logged out successfully"}), 200
```

---

7. Running the App

Run the Flask app:

bash
CopyEdit
```
python app.py
```

- Visit `http://127.0.0.1:5000/login` to authenticate with Spotify.
- After logging in, it fetches user's top artists and stores them in the database.

Next Steps

1. **Build the Compatibility Algorithm**:
    - Compare top artists and genres between users.
    - Calculate similarity scores.
2. **Create a Matching System**:
    - Return potential matches for users.

3. **Develop a Frontend**:
    ○ Use **React or Vue.js** to display user profiles and matches.

Step 2: Implement Matching Algorithm

Want to proceed with building the **compatibility scoring algorithm** next? 🚀

1. Validate and Refine Your Concept

1. **Idea Check**: Talk to potential users—friends, online communities, or music lovers—to see if a "shared music taste" dating concept resonates.
2. **Scope**: Decide on key features from the start:
    - User profiles with their top artists or genres.
    - The compatibility algorithm using Spotify data (e.g., top tracks, artists, genres).
    - Matching suggestions and chat functionality.

You want to ensure you're not building something overly complicated right away. Keep it **MVP-friendly**—focus on the essential features to test your idea.

---

2. Set Up the Basics

1. **Choose a Platform**: Are you building a web app, mobile app, or both? For example:
    - **Mobile**: React Native, Flutter, or native Swift/Java/Kotlin.
    - **Web**: React, Vue, or Angular for the frontend; Node.js, Django, or Ruby on Rails for the backend.
2. **Design Your Data Model**:
    - **User Table**: Basic info (username, email), plus references to Spotify-related data (Spotify user ID, tokens).
    - **Preference Table**: Store top artists, genres, or track IDs to reference for matches.
    - **Matches Table**: Store matched user pairs, match scores, etc.

---

3. Get Started with Spotify API Integration

1. **Developer Account**: Sign up for a Spotify Developer account and create a new app to get a `client_id` and `client_secret`.
2. **OAuth Flow**:
   - You'll typically use **Authorization Code Flow** with Spotify.
   - When a user logs in through Spotify, you get an authorization code.
   - Exchange it for an **access token** and **refresh token** on your server to make API calls on their behalf.
3. **Scopes**: Request permission for the data you need (e.g., `user-top-read`, `user-read-recently-played`).
4. **Test Requests**:
   - Try hitting an endpoint like **`GET /v1/me/top/tracks`** or **`GET /v1/me/top/artists`** to confirm you're pulling the correct user data.

---

4. Design Your Matching Algorithm

Core Idea: Use the user's Spotify data to find overlap or similarity. A simple approach might look like:

1. **Collect Data**: For each user, gather:
   - Their **top artists** (e.g., top 5–10).
   - Their **top genres** or top tracks.
2. **Score Similarities**:
   - For each pair of users, compute overlap in top artists (like a Jaccard similarity): Score=∣Shared Artists∣∣Total Unique Artists in Both Sets∣\text{Score} = \frac{|\text{Shared Artists}|}{|\text{Total Unique Artists in Both Sets}|}Score=∣Total Unique Artists in Both Sets∣∣Shared Artists∣
   - Do the same for top genres or even track "audio features" from Spotify (danceability, energy, tempo, etc.).
3. **Combine Results**: Average or weight these scores to get a final "compatibility score."
4. **Future Enhancements**:
   - Consider machine learning approaches using users' listening histories, creating embeddings, or recommending matches based on a more sophisticated model.
   - But to start, keep it straightforward for your MVP.

---

5. Build an MVP

1. **User Sign-Up and Spotify Connect**:
   - Let new users sign up using email or phone number.
   - Prompt them to **connect Spotify** or sign in with Spotify directly.
   - Immediately pull and store their music data once permissions are granted.
2. **Profile Setup**:
   - Display each user's top artists or recently played tracks.
   - Optionally allow them to edit or highlight certain favorite artists/tracks on their profile.
3. **Basic Matchmaking**:
   - On a user's dashboard/home screen, show them a handful of potential matches with a short snippet of why they match (e.g., "You both love Radiohead!").
4. **Chat or Connection Feature**:
   - Once users "match" (or like each other), let them message each other.
   - Focus on making sure they can talk about music right away (maybe even share Spotify tracks).

---

6. Refine and Expand

1. **Iterate on the Algorithm**:
   - See which approach yields matches that users enjoy.
   - Try weighting certain genres or artists more if they're important to a user.
   - Add or remove data points (like mood or instrumentation) to see how it affects matches.
2. **User Feedback**:
   - Build a quick feedback loop. Let users upvote or downvote potential matches, so you can gather data on which matches are "good" or "bad."
   - Use this feedback to refine the scoring algorithm.
3. **UI/UX Improvements**:
   - Make matching visually engaging. Incorporate album cover art, track previews, or "shared playlist" features that let matched users listen together.

---

7. Handle Security & Privacy

1. **Token Management**:
   - Store **access tokens** securely. Remember that Spotify tokens expire, so you'll need to refresh them periodically.

2. **Personal Data**:
   - Explicitly state how you use personal (and music) data. Follow data protection regulations (GDPR if in Europe, etc.).
3. **Server-Side Protections**:
   - Keep your `client_secret` hidden and never expose it on the frontend.

---

8. Deploy Your App

1. **Hosting**:
   - **Backend**: Services like AWS, Heroku, DigitalOcean, or Google Cloud.
   - **Frontend**: Netlify, Vercel, or integrated with your backend.
2. **Environment Variables**:
   - Keep your Spotify credentials as environment variables.
3. **Monitoring & Analytics**:
   - Implement basic error logging, performance monitoring, and track user metrics to measure engagement.

---

9. Gather Real-World Feedback and Iterate

1. **Invite Beta Users**: Friends, music fans, or small communities to test and give feedback.
2. **Measure Engagement**:
   - Are users finding matches they like?
   - Do they stay in the app or churn quickly?
3. **Adjust and Improve**:
   - Tweak the matching algorithm, refine the UI, or experiment with new features (e.g., shared playlists, virtual listening parties).

Further Tips & Suggestions

- **Focus on the Music Aspect**: That's your unique selling point. Keep the experience strongly tied to users discovering each other's taste.
- **Start Simple**: Don't overcomplicate with too many features at once. It's better to have a stable foundation you can build on.
- **Check Spotify's Rate Limits**: If you're doing a lot of matching and data pulls, watch out for rate limits and optimize requests where possible.

Good luck building your music-matchmaking vision! With a solid plan and an iterative approach, you'll have an engaging app where people bond over shared playlists and favorite artists.

NEURAL NETWORK

Here's a **step-by-step** look at how you could incorporate a **neural network** into your music dating app's matching system, focusing on **music taste compatibility**. We'll assume you've already **integrated Spotify's API** to gather user data (top tracks, artists, genres, audio features, etc.), and now you want a more sophisticated model to predict compatibility.

---

1. Plan Your Data Inputs

1. **User Music Data**:
   - **Top Artists**: A list of artist IDs (e.g., 5–10 top artists).
   - **Top Tracks**: A list of track IDs or track features (danceability, energy, etc.).
   - **Genres**: Either from Spotify's `artist.genres` or user's top genres.
   - **Audio Features**: Spotify provides numerical features like energy, tempo, valence. They're useful for training a network.
2. **Additional Data** (Optional):
   - Demographics (e.g., age, location) if relevant.
   - User interactions (e.g., likes or swipes from earlier versions of the app).
   - Self-reported preferences (like "I love jazz more than anything").

You'll want to **numerically encode** these inputs for your neural network. For text-based or categorical data (e.g., genres), you can use **one-hot encoding** or **embeddings**. For audio features, keep them as normalized floats.

---

2. Select a Modeling Approach

2.1 Embedding-Based Approach (User Embeddings)

1. **Concept**: Train an embedding vector for each user that summarizes their taste in music. Then measure compatibility between users based on a **similarity function** (like cosine similarity) of their embeddings.
2. **How**:
    - Create an embedding layer that takes input about user's music data (artist IDs, track features).
    - Use a **neural network** that outputs a single embedding vector per user, e.g., `dim=32` or `dim=64`.
    - **Compatibility** can then be computed via: Compatibility(u,v)=cos(Eu,Ev)\text{Compatibility}(u, v) = \cos(\mathbf{E}_u, \mathbf{E}_v)Compatibility(u,v)=cos(Eu,Ev) where Eu\mathbf{E}_uEu and Ev\mathbf{E}_vEv are the embeddings for users uuu and vvv.

2.2 Pairwise (Siamese) Network

1. **Concept**: You feed the network **two users' data** at once (User A's features and User B's features) to get a **match score**.
2. **How**:
    - A common architecture is a **siamese network**: the same sub-network processes each user's input and outputs an embedding, then you combine embeddings (e.g., by concatenation or difference) and pass them through a final dense layer to get a **score** between **0** and **1**.
    - During training, you might have pairs labeled as "matched" or "not matched" (or a rating from 1 to 5, etc.).

2.3 Neural Collaborative Filtering

1. **Concept**: Similar to collaborative filtering used in recommender systems (like suggesting songs), but here it's about "user-to-user" compatibility.
2. **How**:

    - Use a **matrix factorization** approach as a baseline, then extend it with neural layers (popular approach known from the paper "Neural Collaborative Filtering" by He et al.).
    - Instead of user–item interactions, you have **user–user** interactions (like "User A matched with User B").
    - Model learns latent factors for users, then a multi-layer perceptron (MLP) refines these embeddings for predicting if two users are compatible.

---

3. Data Preparation and Labeling

1. **Gather Training Data**:
    - You need examples of pairs of users and whether they're "compatible" or not.
    - If you don't have match data yet, you might build synthetic data from "overlap" in music taste. For instance, if two users share many artists, label them as high compatibility. This is a weak proxy, but it can be a starting point.
    - Once your app is live, you can gather real feedback from user likes/matches.
2. **Feature Transformation**:
    - **Normalize** numerical features (like tempo, energy) to a `[0,1]` range or standard normal distribution.
    - **Embed** discrete features (like artist IDs, genre IDs). You can train an embedding layer for artists or create a one-hot vector for each artist.
    - **Padding or Truncation** for variable-length data (if a user has 10 top artists, you might fix a maximum of 10 and truncate or pad with a special "no artist" token if they have fewer).
3. **Train/Validation/Test Split**:
    - Randomly split your user-user pairs or user data into **train/val/test** sets to properly evaluate the model.

---

4. Neural Network Architectures

4.1 Example Siamese Network Architecture

- **Input Layer**:
    - For each user:
        - A list of artists → Embedding layer → Flatten
        - Numerical audio features → Dense layer

- Concatenate the outputs from above for each user, or produce an embedding vector for each user separately.
- **Combine User Embeddings**:
  - If you produce Eu\mathbf{E}_uEu and Ev\mathbf{E}_vEv for each user: h=[∣Eu−Ev∣,Eu⊙Ev](concatenate absolute difference and element-wise product)\mathbf{h} = [|\mathbf{E}_u - \mathbf{E}_v|, \mathbf{E}_u \odot \mathbf{E}_v] \quad (\text{concatenate absolute difference and element-wise product})h=[∣Eu−Ev∣,Eu⊙Ev](concatenate absolute difference and element-wise product)
  - **Dense Layers**:
    - One or more fully connected layers to reduce h\mathbf{h}h to a single compatibility score (0 to 1).
- **Output Layer**:
  - A single neuron (e.g., sigmoid activation) representing the **probability** of match.

4.2 Example User Embedding Approach

- **User Encoder**:
  - Input: user's top 10 artists (IDs), each artist ID is turned into an embedding vector. Summarize via an RNN, CNN, or even a simple average pooling of embeddings.
  - Append user's average audio feature vector, pass everything through a few dense layers.
  - Output: **User Embedding** Eu\mathbf{E}_uEu in a smaller dimensional space, e.g. R32\mathbb{R}^{32}R32.
- **Match Scoring**:
  - For any two users uuu and vvv, compute: Compatibility(u,v)=σ(Eu⊤Ev)\text{Compatibility}(u, v) = \sigma(\mathbf{E}_u^\top \mathbf{E}_v)Compatibility(u,v)=σ(Eu⊤Ev)
  - Or use **cosine similarity** directly.
  - If you want a trainable approach, you can incorporate a small MLP on top of Eu⊤Ev\mathbf{E}_u^\top \mathbf{E}_vEu⊤Ev.

---

5. Training the Model

1. **Loss Function**:
   - For a binary classification approach (match vs. no match), use **binary cross-entropy**: L=−1N∑i=1N[yilog(y^i)+(1−yi)log(1−y^i)]\mathcal{L} = -

\frac{1}{N} \sum_{i=1}^N \big[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\big]L=−N1i=1∑N[yilog(y^i)+(1−yi)log(1−y^i)]
   - yiy_iyi is the label (1 = match, 0 = no match), y^i\hat{y}_iy^i is the predicted score.
2. **Optimizer**:
   - **Adam** or **RMSProp** are common choices.
   - Tune the learning rate (start with something like 1e-3 or 1e-4).
3. **Batching**:
   - Collect pairs in mini-batches, making sure you have a balanced set of matched and non-matched examples.
4. **Regularization**:
   - **Dropout** or **L2 regularization** in dense layers can help prevent overfitting.
   - **Batch normalization** can also help with training stability.

---

6. Evaluating Performance

1. **Metrics**:
   - **Accuracy**: For binary classification, but might not tell the full story if data is imbalanced.
   - **Precision & Recall** (or **F1 Score**): Good for measuring how well the model distinguishes matches.
   - **ROC AUC**: Common for binary classification problems.
2. **User-Centric Metrics**:
   - **Average rating** given by users about how well the recommended matches align with their taste.
   - **Match acceptance rate**: Percentage of suggestions that lead to a mutual match.
3. **A/B Testing**:
   - If you have enough users, test your neural matching approach against a simpler approach (like overlap in top artists) to see which yields better user engagement.

---

7. Serving the Model in Production

1. **Model Inference**:
   - When a new user signs up, generate their user embedding. You can do this in **real-time** or batch once they connect Spotify.
   - To find potential matches, compare user embeddings to others.

        - One approach is a **nearest neighbors search** in the embedding space (e.g., **Faiss** library by Facebook).
    - If you're using a siamese approach that takes pairs, you can compute the compatibility score on demand or cache results for faster retrieval.
2. **Scaling Considerations**:
    - As you grow in user base, calculating pairwise matches for large numbers of users becomes expensive.
    - That's why an **embedding + nearest neighbors** approach can be more efficient than pairwise calls to a siamese network.
3. **Monitoring**:
    - Track how often your system calls the model, how often embeddings are updated, and the latency for generating new matches.

---

8. Iterate and Improve

1. **Refine Inputs**:
    - Maybe you want to incorporate the user's listening sessions, time-of-day preferences, or mood-based data (valence, energy).
2. **Fine-Tune Embeddings**:
    - For example, if you find that certain genre embeddings cluster too broadly, you could refine the granularity.
3. **Collect User Feedback**:
    - Let users rate if the suggested matches "fit" their music taste. Use that feedback as **labels** in your dataset for further supervised training.
4. **Explore Other Architectures**:
    - Graph Neural Networks (GNNs) if you want to consider a user–song–user network.
    - Transformer-based approaches that consider sequences of tracks listened to.

---

9. Practical Tools & Libraries

- **Frameworks**:
    - **PyTorch** or **TensorFlow/Keras** for building and training neural networks.
    - **Scikit-learn** for preprocessing, metric calculations, and basic baseline models.
- **Embedding Lookup**:

    - **Faiss** (by Meta) or **Annoy** (Approximate Nearest Neighbors Oh Yeah) to do large-scale similarity searches on user embeddings.
- **Data Handling**:
    - **Pandas** for data wrangling.
    - **Airflow** or **Dagster** for building data pipelines if you scale up.

---

10. Summary and Key Takeaways

- You can use **neural networks** to learn complex patterns in user's music data and produce a single score indicating how compatible two users might be.
- Start with a **simple approach** (embedding or siamese) and gather real feedback. Then **iterate** to more complex methods if it proves valuable.
- Make sure you handle data collection, labeling, and privacy in a responsible manner (Spotify tokens, user data, etc.).
- Ultimately, the **UX** (how users experience these match suggestions) will be as important as the raw accuracy of your neural network.

Good Luck!

By integrating a neural network approach, your music-exclusive dating app can become more **dynamic and personalized**—delivering matches that reflect deeper musical compatibility than just overlapping artists or genres. The **key** is gathering the right data, picking the right modeling technique, and continually **iterating** based on user feedback.

—---

Spotifriend doesn’t work due to Spotify’s TOS of having “spot” in any other brand name using their API

Especially on Mobile

- True Shuffle (unless there is a reason that it doesn’t exist yet)
- Multiq/Add songs to playlist

Soo

new name is

Taste Mate (subject to change lmao)

Future developments to import and integrate other music platforms such as Apple Music and onto platforms like Instagram or have as a feature for any user-to-user interaction-based service/app/platform

Description

Initially from,, Discord rich presence for a Spotify status displaying a cool cartoon compatibility percentage and art as a fun way to show people when listening to music and share your general tastes to others and maybe meet and make some friends that way.

Maybe more in general doesn’t necessarily have to be through Discord but more as an external application/feature of Spotify or Discord but more Spotify to make music/edm social media platform or some type of community

Basically a public way blended percentage but without the playlist which can be…

Implementation

Combine Discord and Spotify API to access user’s profile and all their tracks from their listening history and playlist to categorize the genres of each song and calcuate compatibility through an algorithm.

GPT-4o

“Creating a software implementation for a "Discord rich presence for a Spotify status" with additional features like compatibility percentage and artwork involves several steps. This project will require familiarity with APIs (Application Programming Interfaces), particularly Spotify's and Discord's, as well as knowledge in programming (likely using Python or JavaScript). Here's a step-by-step guide to help you get started:

Step 1: Setting Up the Environment

1. **Programming Language**: Choose a programming language you are comfortable with. Python is a good choice due to its simplicity and the availability of libraries.
2. **Development Environment**: Set up a development environment. Install necessary tools like code editors (VS Code, PyCharm), and version control (Git).

Step 2: Integrate Spotify API

1. **Spotify Developer Account**: Create a Spotify Developer account and register your application to get client credentials (Client ID and Client Secret).
2. **Authorization**: Implement the OAuth2 authorization flow to get access tokens required for making API requests.

Fetching User Data: Use Spotify's Web API to fetch the current playing track and user profile information.
python
Copy code
```
import requests

SPOTIFY_CLIENT_ID = 'your_client_id'
SPOTIFY_CLIENT_SECRET = 'your_client_secret'
REDIRECT_URI = 'your_redirect_uri'
SCOPE = 'user-read-playback-state user-read-currently-playing'
```

```
def get_spotify_token():
    # Implement OAuth2 flow here
    pass

def get_current_playing(token):
    headers = {
        'Authorization': f'Bearer {token}'
    }
    response =
requests.get('https://api.spotify.com/v1/me/player/currently-pla
ying', headers=headers)
    return response.json()
```

3.

Step 3: Integrate Discord Rich Presence

1. **Discord Developer Account**: Create a Discord Developer account and register your application to get a Client ID.

Discord RPC: Use a library like `pypresence` (for Python) to integrate Discord Rich Presence.
python
Copy code

```
from pypresence import Presence
import time

DISCORD_CLIENT_ID = 'your_discord_client_id'
discord_rpc = Presence(DISCORD_CLIENT_ID)
discord_rpc.connect()

def update_discord_presence(track_name, artist_name,
album_art_url):
    discord_rpc.update(
        state=f"Listening to {track_name}",
        details=f"by {artist_name}",
        large_image=album_art_url,
        large_text=track_name
```

```
)
```

2.

Step 4: Calculate Compatibility Percentage

1. **User Preferences**: Fetch user preferences and listening history using Spotify API.

Algorithm: Develop an algorithm to calculate compatibility based on shared artists, genres, and track history.
python
Copy code

```
def calculate_compatibility(user1_data, user2_data):
    shared_artists = set(user1_data['artists']) &
set(user2_data['artists'])
    compatibility = len(shared_artists) /
max(len(user1_data['artists']), len(user2_data['artists'])) *
100
    return compatibility
```

2.

Step 5: User Interface

1. **Desktop Application**: Use a GUI framework like Tkinter (Python) or Electron (JavaScript) to create a simple desktop application.

Displaying Information: Display current track information, compatibility percentage, and album art.
python
Copy code

```
import tkinter as tk
from PIL import Image, ImageTk
import requests
from io import BytesIO

class SpotifyApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Spotify Discord Rich Presence")
```

```
        self.label = tk.Label(root, text="Current Track")
        self.label.pack()
        self.canvas = tk.Canvas(root, width=300, height=300)
        self.canvas.pack()

    def update_ui(self, track_name, artist_name, album_art_url):
        self.label.config(text=f"{track_name} by {artist_name}")
        response = requests.get(album_art_url)
        img_data = response.content
        img = Image.open(BytesIO(img_data))
        img = img.resize((300, 300), Image.ANTIALIAS)
        photo = ImageTk.PhotoImage(img)
        self.canvas.create_image(0, 0, anchor=tk.NW,
image=photo)
        self.root.mainloop()

root = tk.Tk()
app = SpotifyApp(root)
```

2.

Step 6: Deployment and Sharing

1. **Packaging**: Package your application for distribution using tools like PyInstaller for Python or Electron Packager for JavaScript.
2. **Sharing**: Share your application with friends or community through GitHub or other distribution platforms.

Step 7: Enhancements

1. **Community Features**: Add features like friend lists, shared playlists, and user profiles.
2. **Feedback and Iteration**: Collect feedback from users and continuously improve the application.

This guide provides a high-level overview of the steps involved. Each step will require more detailed implementation and testing. Start small, and gradually add features as you become more comfortable with the APIs and tools.”