ChatGPT

# Emotion-Aware Music Recommendation System

## Project Overview

The **Emotion-Aware Music Recommendation System** is an AI-driven application that identifies a user's emotional context from free-form text input and suggests music tailored to that emotion [1]. By analyzing how the user *feels* (in their own words), the system maps those feelings to an appropriate mood and music genre, then fetches relevant songs via the Spotify API. This creates a convenient and intuitive listening experience where playlists match the user's current emotional state [1]. Whether the user is *heartbroken*, *joyful*, *anxious*, or *excited*, the system will attempt to provide a playlist that resonates with their mood.

**How it works:** The system uses natural language processing (NLP) models to detect emotions in text, then translates those emotions into a **musical mood and genre**. It leverages Spotify's vast music catalog to find tracks (with preview clips and links) matching that genre. Importantly, it has multiple fallback strategies – including public playlist search, Spotify's own recommendation engine, and even an AI-driven keyword search – to ensure it returns some music recommendations even if the initial approach fails. The result is an emotion-aware music recommender that can be used by developers (via an API or code) or by non-technical users through a simple interface.

## Features

- **Free-Form Emotion Input:** Accepts natural language input describing feelings (e.g. *"I just got dumped and everything tastes like dust"* or *"My friend surprised me with coffee this morning!"*). Users are not limited to choosing from a fixed set of moods; they can express emotions in their own words.

- **Emotion Detection via NLP:** Utilizes two NLP models to accurately gauge the user's emotional state:

- *BERT-Emotion Model*: A lightweight BERT-based classifier fine-tuned for 13 distinct emotions (e.g. joy, sadness, anger, love) [2].

- *Zero-Shot Classifier*: A DistilBART model (fine-tuned on MNLI) that can classify text into a broader set of ~20+ emotion labels without explicit training for those labels (zero-shot learning).

- **Emotion-to-Music Mapping:** Maps the detected emotion(s) to a corresponding **mood descriptor** and **Spotify music genre**. For example, *anger* might map to a *"furious"* mood and *punk-rock* genre, whereas *disappointment* maps to a *"pensive"* mood and *indie* genre [3] [4]. (These mappings are fully customizable by the user.)

- **Spotify Integration:** Connects to the Spotify Web API to retrieve song recommendations. It performs a multi-step retrieval:

- **Direct Track Search** – Searches for tracks in the target genre (using Spotify's genre filter in the search query [5]) and picks songs with available preview audio.
- **Public Playlist Mining** – If direct track search yields insufficient results, it searches for public playlists matching the genre and scrapes tracks from them (looking for songs with previews).
- **Spotify Recommendations** – If playlist scraping fails, it uses Spotify's recommendation endpoint to get tracks based on the genre seed (Spotify's own algorithmic suggestions).
- **AI Keyword Search** – As a last resort, it takes the top emotion keywords and performs a general Spotify search for tracks matching those terms (e.g. searching the term "loneliness" if the user felt lonely), effectively letting the **AI guide the music search**.

- **Fallback Mechanisms:** The tiered approach above ensures that even if one method fails (due to network issues, lack of genre data, etc.), another method can still provide songs. For instance, if a genre search returns no playable tracks, the system automatically *"falls back to public playlists"*, and if that fails, it tries *"Spotify's built-in recommendations"*, and so on [6].

- **Song Previews and Links:** The recommended songs come with 30-second preview clips (MP3 snippets) and direct Spotify links. This allows users to sample the music immediately. In the console interface, the recommendations are displayed in a pandas DataFrame format with columns for *Track*, *Artist*, *Preview URL*, and *Spotify Link*. In the web app interface (see below), these can be presented as clickable audio players or buttons.

- **Interactive Web App (Streamlit):** Besides a console/terminal mode, the project includes a **Streamlit** web application for a user-friendly experience. Non-technical users can run the app and get a simple interface: they enter how they feel in a text box, and the app will display the detected emotions and a curated playlist of song recommendations.

- **Local and Docker Deployment:** The system can run locally on any machine with Python, or be deployed in a Docker container for ease of setup. Docker support means you can spin up the whole service with one command, without manually installing dependencies, and even host it on a server or cloud service.

- **Customization Options:** Advanced users can tweak various aspects: add new emotion mappings or genres, adjust the list of emotion labels used by the zero-shot model, or reorder/disable certain fallback steps. The aim is to make the system flexible for different contexts or preferences (for example, emphasizing some genres more than others, or integrating new mood definitions).

- **Error Handling & Caching:** The system is designed to handle API errors gracefully. It includes logging for troubleshooting (e.g., printing messages when a search fails or when a socket permission error occurs) and uses Spotify's token caching to avoid re-authentication whenever possible.

## System Architecture and Pipeline

**Overall Workflow:** The following diagram illustrates the architecture and data flow of the Emotion-Aware Music Recommendation System, from user input to final playlist output:

```
flowchart TD
    A[User Input\n(emotional text)] --> B{{Emotion Detection}};
    B --> B1[bert-emotion model\n(primary emotion)];
    B --> B2[distilbart-mnli model\n(top 3 emotions)];
    B1 & B2 --> C[Emotion & Confidence Scores];
    C --> D[Emotion-to-Mood & Genre Mapping];
    D --> E{{Music Retrieval Pipeline}};
    E --> F1[Spotify Track Search\n(by genre)];
    F1 -- if no tracks --> F2[Playlist Search Fallback\n(public playlists)];
    F2 -- if no tracks --> F3[Spotify Recommendations Fallback\n(seed genre)];
    F3 -- if no tracks --> F4[AI Keyword Search Fallback\n(by emotion terms)];
    F1 -- tracks found --> G[Playlist of tracks];
    F2 -- tracks found --> G[Playlist of tracks];
    F3 -- tracks found --> G[Playlist of tracks];
    F4 -- tracks found --> G[Playlist of tracks];
    F4 -- none found --> H[No result (failure message)];
```

*(Diagram: The system first uses NLP models to classify the emotion, then maps to a mood/genre, and finally tries several methods to fetch songs. Green arrows indicate a successful path yielding a playlist, whereas red arrows (not shown in text) indicate a fallback to the next step.)*

**Step 1: Emotion Detection.** When the user provides an input describing their feelings, the system uses two NLP models in tandem: - A **BERT-based emotion classifier** (`boltuix/bert-emotion`) quickly predicts a primary emotion label for the text. This model is efficient (~20MB) yet highly accurate on short texts [2]. For example, given *"I feel a bit hopeful about the future"*, it might predict "Joy" or "Optimism". - A **Zero-Shot classifier** (`valhalla/distilbart-mnli-12-3`) is then applied to the text with a predefined set of possible emotion labels (like *anger, fear, hope, gratitude, loneliness*, etc.). This model can assign scores to each label indicating how well the input fits that emotion category, even if it wasn't explicitly trained on those labels [7]. We take the **top 3 emotions** by score as the refined emotional context. For instance, *"I just got dumped and everything tastes like dust"* might yield top labels like *"disappointment" (39%)*, *"disgust" (14%)*, and *"embarrassment" (9%)* [8].

The combination of these models ensures both **breadth** (capturing nuanced or uncommon emotions via zero-shot) and **accuracy** (leveraging a model specialized for emotion nuances in text). The detected emotions and their confidence scores are then used for the next step.

**Step 2: Emotion-to-Genre Mapping.** The system translates the detected emotion(s) into a target **mood** and **music genre** using a mapping dictionary. This `emotion_map` is essentially a lookup table that links dozens of emotion keywords to a mood descriptor and a fitting Spotify genre. For example, some mappings include [3]:

- *anger* → ("furious" mood, **punk-rock** genre)
- *sadness* → ("heartbroken" mood, **blues** genre)
- *joy* → ("euphoric" mood, **electro-house** genre)
- *surprise* → ("curious" mood, **experimental** genre)
- *neutral* → ("balanced" mood, **chill** genre)

- *love* → ("romantic" mood, **indie-pop** genre)
- *disappointment* → ("pensive" mood, **indie** genre)
- *(etc. – see Advanced Usage for customizing these)*

The mapping is designed to pick a genre that reflects or complements the emotion. If multiple emotions were detected, the system will use the first one that appears in the mapping (which is typically the highest-confidence emotion). For example, if the emotions were *disappointment* and *disgust*, the mapping finds **indie** for disappointment (with mood "pensive") and uses that, rather than disgust's genre **goth** [4]. (If an emotion isn't found in the map, a default genre like *"experimental"* is used as a fallback.)

**Step 3: Music Retrieval Pipeline.** Once we have the target **genre**, the system attempts to fetch songs in that genre through Spotify. It follows a pipeline of methods, each with a fallback to the next if it doesn't produce results:

1. **Spotify Track Search:** The system first performs a direct search for tracks of the given genre using the Spotify API. It uses a query of the form `genre:"<genre_name>"` which is an official filter to find songs by genre [5]. For example, if the genre is *indie*, the query becomes `genre:"indie"`. Spotify returns up to 50 tracks; the system filters those to only include tracks that have a playable preview URL. If at least `N` tracks (configurable, by default 10) with previews are found, it randomly selects 10 and returns those as the recommended playlist. This is the ideal path when it works, as it yields a **personalized playlist** directly related to the emotion's genre.

2. **Public Playlist Fallback:** If the direct track search returns too few results (or none), the system prints a message like *"No luck with direct track search. Trying public playlists..."* and proceeds to search for **playlists** matching the genre [9]. It queries Spotify for playlists whose name or description contain the genre term (e.g. "indie"). It then inspects up to a few of these playlists (by retrieving their tracks via the API) to find songs that have preview URLs. If it finds any, it will take up to 10 tracks from the first suitable playlist and return those. This method leverages the curation of other Spotify users – for instance, if searching for *"indie"*, it might find a public playlist titled "Indie Classics" and use some songs from there. The system logs each step, e.g., *"Inspecting playlist: Indie Classics (ID: 3MgX...)* and either finds tracks or moves on [10] [11]. If no playable tracks are found in any of the sampled playlists, it prints *"No usable tracks found in any fallback playlists."* [12] and proceeds to the next fallback.

3. **Spotify Recommendations:** If both direct search and playlist scraping fail to yield a result, the system tries Spotify's recommendation engine. It calls the `recommendations` endpoint with the emotion's genre as a seed (along with a fixed number of results, e.g. 50). Spotify then returns tracks that are often associated with that genre. As before, the system filters for tracks with preview URLs, then randomly selects up to 10 of them. If this succeeds, it prints something like *"Spotify's algorithm came through after all!"* and outputs the playlist [13]. If it fails (e.g., the genre seed was invalid or the API returns nothing useful), the system moves to the final fallback.

4. **AI-Based Keyword Search:** This is the last resort when everything else has *"failed"*. In this step, the system uses the **detected emotion keywords directly as search queries** on Spotify [14]. It essentially says, "if we can't get songs by genre, let's search the whole Spotify library for songs related to these emotions." For each emotion (from the top 3 list) it performs a general search (e.g. searching for the term *"disappointment"* in Spotify's track search), gathering any tracks that have

previews. It aggregates results from each emotion query (avoiding duplicates) until it has enough tracks. If this yields any songs, it prints *"Here's your AI-curated playlist:"* – meaning the songs were found by letting the AI's understanding of emotion guide the search [6]. If even this yields nothing, the system will ultimately print an apology like *"All systems failed. AI and Spotify have nothing left to give you. Good luck out there."* [15]. This signals that no recommendation could be made (which would be rare unless there are severe connectivity issues or the input was extremely unusual).

All of these steps happen within a single function call (the `mood_bot()` in the code) so the user experience is just: input emotion text, then receive output. The fallback transitions are logged to the console or debug log for transparency. For example, you might see logs such as [16]:

- *Searching for genre:'alternative' tracks* (then a failure message if none found)
- *No previewable tracks found via standard search.*
- *Falling back to public playlists for genre: 'indie'* (then perhaps a failure message)
- *Trying Spotify's built-in recommendations...*
- *All Spotify logic failed. Using AI to find tracks based on emotion keywords...* [6]

Each step only triggers if the previous ones did not succeed in finding songs, ensuring the system doesn't do unnecessary work.

**System Components:** Internally, the project is organized into modules that handle different aspects: - *NLP Models:* Loaded at startup (using Hugging Face's Transformers pipeline for text classification and zero-shot classification). - *Emotion Mapper:* A dictionary (`emotion_map`) and some helper logic to get the corresponding mood/genre and validate that genre against Spotify's allowed seed list. (The project includes a list of valid Spotify genre seeds to ensure the recommendations API won't error out [17] [18].) - *Spotify Client:* Utilizes the **Spotipy** library to interact with Spotify's API. An authentication manager is set up (either via Client Credentials or OAuth, see Setup) and the `spotipy.Spotify` client is used for all API calls (search, playlist fetch, recommendations, etc.). - *Main Orchestration:* The core function (or class method) that ties everything together reads the user input, calls the models, does the mapping and retrieval, and then formats the results for output.

In summary, the architecture cleanly separates **emotion analysis** from **music retrieval**, with a mapping bridge between them, and employs a resilient multi-step pipeline to fetch songs. This ensures that even if one approach fails (e.g., no songs found for a niche genre), the system can fall back gracefully to other methods and still serve the user.

## Models Used and Why

The project employs two key NLP models for emotion detection, each chosen for its strengths, and uses the Spotify API for music data:

- **BERT-Emotion (boltuix/bert-emotion)** – *Emotion Classification Model:* This is a fine-tuned BERT-based model optimized for detecting emotions from short text. It can classify input text into **13 distinct emotion categories** (e.g. Sadness, Anger, Love, Surprise, Fear, Happiness, Neutral, Disgust, Shame, Guilt, Confusion, Desire, Sarcasm) [19] [2]. We use this model to get a quick primary read on the user's emotion. It's very fast and lightweight (~20MB) compared to full BERT models, which makes it ideal for real-time use or deployment on limited hardware [2]. Despite its small size (~6

million parameters), BERT-Emotion delivers high accuracy (the model card reports ~1.0 accuracy/F1 on its dataset) [20] [21]. In our context, this model helps ground the prediction in common emotion categories (like the "big six" emotions and a few others). For example, if a user says *"I'm absolutely ecstatic about my new job!"*, BERT-Emotion would likely output **"Happiness"** with high confidence.

- **DistilBART Zero-Shot (valhalla/distilbart-mnli-12-3)** – *Zero-Shot Classification Model:* This model is a distilled version of BART trained on the MultiNLI (MNLI) dataset, which is commonly used for zero-shot classification tasks. We leverage it with Hugging Face's `pipeline("zero-shot-classification", model="valhalla/distilbart-mnli-12-3")` to classify text into a **broader set of emotion labels** beyond the 13 fixed ones from BERT-Emotion. The zero-shot model works by treating the classification task as an *entailment problem*: it checks how likely the user's input implies each candidate emotion label (e.g. "This text is about *anger*") [7]. This allows us to include fine-grained or unconventional emotions without having to train a new model. In our implementation, we defined a list of 23 labels (anger, disgust, fear, sadness, joy, surprise, neutral, love, embarrassment, confusion, curiosity, excitement, gratitude, grief, hope, pride, relief, romance, anxiety, loneliness, disappointment, shame, guilt, trust). The model assigns a score to each, and we take the top 3 as the detected emotions [22] [23]. The reason for using this model is **coverage**: users might express feelings like *"anxiety"* or *"hopefulness"* which the first model might not explicitly recognize if they weren't in its 13 classes. The zero-shot classifier can catch those nuances (e.g. detecting *"hope"* or *"anxiety"* as labels). DistilBART is also faster and smaller than the original BART, making it feasible to run alongside the BERT model. By combining it with BERT-Emotion, we get the best of both worlds: the reliability of a model trained specifically on emotion data, and the flexibility of a model that can adapt to arbitrary emotion labels on the fly.

*Why both?* In practice, we primarily rely on the zero-shot model's output for mapping to music (since our `emotion_map` contains many of those 23 labels). The BERT-Emotion model can be seen as a sanity check or an additional signal if needed – for example, one could imagine weighting the mapping if BERT's top prediction disagrees with the zero-shot's top. However, currently the system uses the refined list from the zero-shot classifier to do the mapping. We included BERT-Emotion because it's highly optimized for emotion detection and could be useful for extension (like providing a quick baseline emotion or in cases where a simpler set of emotions is desired). It's also pre-trained on emoji-rich data, which might capture nuances of tone. In short, **BERT-Emotion** was chosen for its speed and specialization in emotion, and **DistilBART-MNLI** for its generality and breadth.

- **Spotify API (Spotipy):** Rather than a single model, the music recommendation part uses Spotify's service. The project uses the open-source **Spotipy** library as a Python client to the Spotify Web API. This handles the HTTP requests and auth under the hood. No machine learning model is used for recommending music; instead, we harness Spotify's search and recommendation algorithms (which are quite powerful globally) and apply our own logic on top (genre filtering, random sampling, etc.). Using Spotify ensures we have a huge database of songs and metadata. The system does not attempt to *learn* or predict user musical taste (no collaborative filtering or content-based ML on the songs), which keeps the scope focused: **emotion in → relevant genre out**. All heavy lifting of finding actual songs is delegated to Spotify's well-tuned infrastructure.

In summary, **NLP Models** provide the "emotion understanding," and **Spotify's API** provides the "music data." This design is modular – you could swap in a different emotion model (say a fine-tuned RoBERTa) or even a different music database, without changing the overall flow.

## Dependencies and Installation

**Programming Language:** Python 3.7+ is required (the code should run on Python 3.7, 3.8, or above). We recommend using a Python 3.10 environment for best compatibility.

**Key Dependencies:**

- **spotipy** – The Spotify API client for Python. (Used for authenticating and making requests to Spotify.)
- **transformers** – Hugging Face Transformers library (used to load and run the BERT-Emotion and DistilBART models).
- **torch** – PyTorch, required by Transformers to run the models.
- **pandas** – Used for creating and displaying the playlist DataFrame (for console output).
- **numpy** – Used internally (e.g. for any numeric computations or vectorization, possibly by transformers or other code).
- **scikit-learn** – Used for utility functions; the code imports `TfidfVectorizer` and `cosine_similarity` (though these might not be heavily used in the current pipeline, they were included, possibly for future content-based enhancements).
- **sentence_splitter** – Utility to split text into sentences (not heavily used in basic emotion detection, but could be used to preprocess complex inputs).
- **textblob** – (Optional) A text processing library that was imported in the code. It might not be actively used for core logic, but was likely included for sentiment analysis or text cleaning experiments.
- **gliclass** – (Optional) Stands for "General Language Inference Classification." This is a lesser-known library possibly used as an alternative approach to zero-shot classification. The code did install it, but our main pipeline uses HuggingFace directly. You typically do *not* need this library unless you want to experiment with it. All necessary functionality is already handled by Transformers.
- **streamlit** – Only needed if you plan to run the web app. Streamlit is the framework for the interactive UI. (Not required for core CLI usage.)
- **requests**, **regex**, etc. – These are indirect dependencies via Transformers or Spotipy and should be installed automatically.

All needed packages (and specific version pins if any) are listed in `requirements.txt` in the repository. To install everything, you can use pip:

```
# Create and activate a virtual environment (optional but recommended)
python3 -m venv venv
source venv/bin/activate   # on Windows: venv\Scripts\activate

# Install the project dependencies
pip install -r requirements.txt
```

This will fetch and install all libraries. Make sure you also have **ffmpeg** installed if you plan to play audio previews (Streamlit's audio player might require it, and some platforms need it for playing MP3 streams). On most systems, this isn't necessary for basic usage, but for full functionality, ensure multimedia support.

**Model Downloads:** The first time you run the system, Hugging Face will automatically download the `boltuix/bert-emotion` and `valhalla/distilbart-mnli-12-3` model weights to your machine

(likely under `~/.cache/huggingface`). These total a few hundred MB. Ensure you have an internet connection for that initial download. Subsequent runs will use the cached models.

**Spotify Credentials:** *This is important.* You must have Spotify API credentials set up (Client ID and Client Secret) before using the music recommendation features. See the **Setup** section below for detailed instructions.

## Setup and Configuration

Before running the project, a one-time setup is needed to configure Spotify API access:

1. **Obtain Spotify API Credentials:** Go to the Spotify Developer Dashboard and log in with your Spotify account. Create a new application (it's free). Once created, you will find a **Client ID** and **Client Secret** for your app. You'll also need to set a **Redirect URI** for OAuth (for example, `http://127.0.0.1:8888/callback` – this is used for the auth flow). If you plan to use only the client credentials (non-user) approach, the redirect URI may not be needed, but it doesn't hurt to set one.

2. **Configure Credentials:** The project will look for your Spotify credentials to authenticate. There are a few ways to provide them:

3. **Environment Variables:** The Spotipy library can pick up credentials from environment variables. You can export the following in your shell:

   ```
   export SPOTIPY_CLIENT_ID='<your_client_id>'
   export SPOTIPY_CLIENT_SECRET='<your_client_secret>'
   export SPOTIPY_REDIRECT_URI='http://127.0.0.1:8888/callback'
   ```

   Replace the values with your actual Client ID/Secret, and ensure the redirect URI matches what you set on the dashboard. If you use this method, the code can automatically use `SpotifyOAuth()` or `SpotifyClientCredentials()` without explicitly passing the values.
4. **Hardcode or Config File (not recommended):** In the code (for example, in a Jupyter notebook or a config module), you could assign `SPOTIPY_CLIENT_ID = '...'` and `SPOTIPY_CLIENT_SECRET = '...'` [24]. The provided example in the code uses a direct assignment for demonstration, but **for security reasons, do NOT commit your secrets**. It's better to use environment variables or a separate config file not in version control.

5. **Use SpotifyOAuth Prompt:** If using the Streamlit app or running `spotipy.Spotify(auth_manager=SpotifyOAuth(...))` without setting environment vars, the library may open a browser for you to log into Spotify. On success, it will redirect to your provided redirect URI with a token. This flow will create a cache file (like `.cache`) to store your access token for reuse. You may see console messages about *"Couldn't read cache at .cache"* on first run – that's normal (it means no cached token yet).

6. **Choose Auth Flow:** This project can work with either the **Client Credentials flow** (no Spotify user login, limited to public data) or the **Authorization Code flow** (user login required, but not strictly necessary for our use-case since we only need public tracks).

7. *By default*, the code is set up with `SpotifyOAuth` (user auth) [25]. This will require you to log in once in your browser to grant the app access (scope `user-read-private` was used in code as an example [26], though for our needs, we aren't accessing private data, so even a basic scope or no special scope would work).
8. If you prefer to avoid the login step, you can modify the setup to use `SpotifyClientCredentials` (app-only auth). For example:

   ```
   import spotipy
   from spotipy.oauth2 import SpotifyClientCredentials
   sp = spotipy.Spotify(auth_manager=SpotifyClientCredentials())
   ```

   This will use your Client ID/Secret to get a token without user context. It should be sufficient for searching tracks and genres. The trade-off is that certain endpoints that require a user (like personalized recommendations) wouldn't be available – but in our pipeline, we only use genre-based recommendations which work with app tokens.

9. In either case, ensure the credentials are correctly supplied. On successful auth, the system will be ready to hit the API. The Spotipy client object (`sp`) is created like:

   ```
   auth_manager = SpotifyOAuth(client_id=..., client_secret=...,
   redirect_uri=..., scope="user-read-private")
   sp = spotipy.Spotify(auth_manager=auth_manager)
   ```

   (or using client credentials as shown above).

10. **Test the Connection:** A quick test after setup: run a simple search outside the main app, for example:

    ```
    sp.search(q="genre:rock", type="track", limit=1)
    ```

    This should return a JSON with one rock track if the auth is working. If you get an error about authentication, double-check your keys or scopes.

With the above configured, you are ready to run the application either in console or via Streamlit.

## Local Usage Instructions (Console/CLI)

If you want to use the system in a script or interactive console (without the web interface), you can do so by calling the main function that runs the pipeline. Assuming you have obtained credentials and installed everything as described:

```
from emotion_recommender import mood_bot  # hypothetical module & function name
```

```
# Start the interactive mood-based music recommendation
mood_bot()
```

When you call `mood_bot()`, it will prompt you to enter how you're feeling:

```
Tell me how you're feeling:
```

You can then type any phrase or sentence describing your emotion. The system will output the detected emotions and the recommended songs. For example:

```
Tell me how you're feeling: My friend surprised me with coffee this morning!

Top Detected Emotions:
  surprise: 60.07%
  joy: 13.34%
  excitement: 8.76%

Mapped Mood: curious
Suggested Genre: experimental

Here's your personalized playlist:

           Track                          Artist
Preview                              Spotify Link
0  Electric Feel                    MGMT                    https://p.scdn.co/
mp3-preview/...   https://open.spotify.com/track/3r3Sw...
1  People                          Sylvan Esso              https://p.scdn.co/mp3-
preview/...   https://open.spotify.com/track/7of9A...
2  Surprise Yourself               Jack Garratt            https://p.scdn.co/mp3-
preview/...   https://open.spotify.com/track/5GmJR...
... and so on for 10 tracks ...
```

*(The above is an illustrative example; actual results will vary based on Spotify's data and what tracks have previews.)*

A breakdown of the console output: - First, it echos your input. - Then "Top Detected Emotions" are listed with percentages (the zero-shot model's top 3 predictions, as shown in the example where *surprise* was dominant [27]). - Next, it prints the chosen mood and genre (in this case, *curious* mood, *experimental* genre, derived from *surprise*). - Finally, it prints the playlist. In a Jupyter notebook, this appears as a nicely formatted table if you use `display(df)`. In a plain terminal, it will print a text representation of the pandas DataFrame. Each row includes the track name, artist, a preview URL (which you could copy to a browser to listen to a 30s snippet), and a Spotify link (to open the full song in Spotify).

If the system had to fallback through multiple methods, you would see messages in between. For instance, if no tracks were found in the first search, you'd see the "No luck with direct search..." message, etc. These are there to help you understand what the system is doing under the hood, and to aid in debugging if something goes wrong.

**Running the code**: If the code is packaged as a module or script, you might run it like:

```
python emotion_recommender.py
```

and it would execute similar to the above, asking for input. In the repository's context, if there is a Jupyter notebook (`Emotion_aware_music_system.ipynb` was provided), you can step through it to see the process with the example inputs given. For production or command-line usage, wrapping `mood_bot()` in an `if __name__ == "__main__":` block or creating a small CLI interface would be ideal.

## Advanced Usage and Customization

One of the strengths of this project is that you can **tune it to your needs**. Here are some ways to customize or extend the system:

- **Customizing Emotion→Genre Mappings:** You can edit the `emotion_map` dictionary to change which mood and genre correspond to a given emotion. For example, you might decide that *"anger"* should map to a *heavy metal* genre instead of punk-rock, or that *"loneliness"* should use *blues* instead of acoustic. Simply update the dictionary entries in the code. Ensure that the genre you map to is recognized by Spotify (check Spotify's list of available genre seeds – the project already includes a comprehensive list `VALID_SEEDS` to validate against). After changing mappings, you might also adjust the descriptive mood text accordingly. This is useful if you want to experiment with different music associations or adapt the system to a different culture/audience where, say, *"sad"* might mean a different genre preference.

- **Refining Zero-Shot Labels:** The list of candidate emotion labels used by the zero-shot model (`zero_shot_labels`) is configurable [28]. You can add new emotion words or remove ones that are not relevant to your use case. For instance, you might add "nostalgia" or "contentment" to the list if those are emotions you expect your users to express. If you do add labels, remember to also add them to the `emotion_map` with an appropriate genre. You could also reduce the list if you want to restrict the system to a core set (e.g., maybe you only care about the six basic emotions). Keep in mind that zero-shot classification is more reliable with concise, distinct labels. Also, by default the system takes the **top 3** emotions; you could choose to take only the top 1 (if you want a single-emotion interpretation) or top 5 (to capture more nuance), depending on your application. That would simply involve changing the slice in `classify_emotion_refined` where it currently does `[:3]` [23].

- **Tuning Search and Fallback Priorities:** The order and conditions of the fallback pipeline can be modified. If, for example, you prefer to use Spotify's built-in recommendations before scraping playlists, you can rearrange those calls. Or if you find the playlist scraping to be too slow or unreliable, you might disable it. Each step in the pipeline is encapsulated as a function

(`fetch_spotify_tracks`, `_fallback_to_playlist`, etc.), so you can invoke them in a different order or add new logic. Advanced users could also integrate **audio feature filtering** – for instance, once you have tracks, you could filter by energy or valence using Spotify's audio features API to better match mood (this is not implemented by default, but is a logical extension).

- **Adjusting the Number of Results:** By default, the system aims to provide 10 tracks in the playlist. You can change this by passing a different `limit` parameter to the functions or modifying the default in code. For example, to get 5 tracks instead (perhaps for a quick sample), set `limit=5` when calling `fetch_spotify_tracks` and likewise sample 5 in the fallback sections. With Streamlit, you could even expose this as a slider for the user to choose how many songs they want.

- **Performance Considerations:** The first time the models run, there might be some initialization delay (loading the model into memory). For subsequent requests, it should be quite fast (a few hundred milliseconds for classification). The slower part is usually the Spotify API calls (network latency). If you plan to handle many requests, you might want to implement **caching** for certain calls. For example, caching the result of `map_emotion_to_genre` for a given input text, or caching Spotify search results for a genre for some minutes, to avoid hitting the rate limits. The Spotipy client by default does not cache responses (except the OAuth token), so this would be something you'd add. Python's `functools.lru_cache` or an external cache (Redis etc.) could be used if scaling up.

- **Using Different Models:** If you have a custom emotion detection model (say you trained your own on a special dataset), you can replace the model in the pipeline. As long as you output an emotion label that exists in the mapping, the rest of the system will work. For example, you might use a multilingual model to support inputs in other languages. Or use a **speech-to-text** front-end to allow spoken input to be converted to text for analysis (and perhaps even integrate facial emotion detection from a webcam feed as another input – the original repository seems to have had a vision component, which could be combined with this text approach for a multi-modal system).

- **Zero-Shot Classification Settings:** The Hugging Face zero-shot pipeline has a parameter `multi_label`. By default, we did *not* enable multi_label, meaning it treats it as a single-label problem (the scores sum to 1 and it picks the most likely single class – then we took top3 by raw scores anyway). If you believe multiple emotions can be true at once, you could set `multi_label=True` when calling the pipeline. This would score each label independently and could potentially allow the system to consider combinations of emotions with less mutual exclusivity. You would then interpret the scores differently (they won't sum to 1). This is more advanced and might not change much for our use, but it's something to be aware of.

In essence, the system is quite adaptable. The defaults have been chosen based on general assumptions (e.g., what genres fit which emotions) and practical limits (e.g., top 3 emotions, 10 tracks) but you are encouraged to play with these settings. The code is commented and structured to facilitate changes. Always test after any major change to ensure the pipeline still runs end-to-end and returns music as expected.

# Deployment Instructions

You can deploy the Emotion-Aware Music Recommendation System in two main ways: as a **Streamlit web application** or within a **Docker container** (these are not mutually exclusive – the Docker container could run the Streamlit app, for example). Below are instructions for each:

## Streamlit App (Interactive Web Interface)

The project includes a Streamlit app script (commonly named `App.py` or similar). Streamlit allows you to run the app locally and interact with it through your web browser.

**Steps to run the Streamlit app:**

1. **Install Streamlit:** If you haven't already, install Streamlit (`pip install streamlit`). Also ensure you have completed the **Setup** step for Spotify credentials – the Streamlit app will need access to the same environment variables or credentials.

2. **Run the App:** In your terminal, navigate to the project directory and run:

   ```
   streamlit run App.py
   ```

   (Replace `App.py` with the actual name/path of the Streamlit script if it's different.)

3. **Allow Network/Auth if prompted:** The first time, the app may need to open a browser to authenticate with Spotify (if using OAuth). Since you're already running a browser for Streamlit (usually at `http://localhost:8501`), it might reuse that session or pop up another tab to log in to Spotify. Complete the login if prompted – afterwards, a token will be cached and you shouldn't need to log in again for a while. *(If using client credentials flow, this step won't occur.)*

4. **Using the Web Interface:** You should see the Streamlit interface load in your browser. The UI typically will contain:

5. A title or header.
6. A text input box (prompt like "Tell me how you're feeling") where you can type your emotion description.
7. Possibly a button to submit (Streamlit also can submit on pressing Enter).
8. After submission, the app will display the detected emotions (e.g., as a list or bar chart) and the recommended tracks. Each track could be displayed with its name, artist, and possibly a way to play the preview. Streamlit supports audio widgets, so the app might use `st.audio(preview_url)` to provide a play button for each preview. Or it could display the DataFrame in a table form.

For example, you might enter *"I am extremely stressed about my exams."* The app would then show maybe: - *Detected emotions:* fear (45%), anxiety (30%), sadness (10%) – perhaps as text or a plot. - *Mapped mood & genre:* tense mood, downtempo genre. - *Playlist:* A list/table of songs in the Downtempo genre suitable for

calming stress (with play buttons or Spotify links). You could click the play icon to listen to a short snippet of each song, or click the Spotify icon/link to open the full track in Spotify.

The web app makes it easy for non-technical users to use the system without dealing with code or JSON outputs. It's essentially a friendly wrapper around the pipeline.

1. **Adjusting Streamlit Settings:** Streamlit usually opens your default browser automatically. If it doesn't, check the console for a network URL (something like `Network URL: http://192.168.x.x:8501`). You can manually open that. If you want to run the app on a different port (say 8502) or make it accessible on your network, you can add options:

   ```
   streamlit run App.py --server.port 8502 --server.headless true --
   server.enableCORS false
   ```

   The `headless` and `enableCORS` flags can help when deploying on a remote server with no display.

2. **Stopping the App:** In your terminal where Streamlit is running, press `Ctrl+C` to stop the server.

**Note:** The Streamlit app provided in this project may also include additional features such as capturing emotion via webcam (the presence of a `fer` dependency suggests there was an option to use facial emotion recognition). If that is the case, the app might have a toggle for "Text Mode" vs "Camera Mode." For the scope of this documentation, we focus on the text-based usage. Using the webcam would typically require granting camera access and would detect your facial expression to recommend music – an interesting extension, but ensure to follow any instructions in the app UI if you explore that feature (like "click Start Camera" etc., if implemented).

## Docker Container

Deploying via Docker is a convenient way to package all dependencies and run the app in an isolated environment. A Docker setup is especially useful if you want to deploy this on a cloud server or simply avoid installing everything locally.

**Building the Docker Image:**

1. Ensure you have **Docker installed** on your system.
2. In the project directory, there may be a `Dockerfile` provided. (If not, you can easily create one – see below.)
3. Build the Docker image by running:

   ```
   docker build -t emotion-music:latest .
   ```

   This will package the application into an image named `emotion-music`. The Dockerfile would typically:
4. Start from a base Python image (e.g. `python:3.9-slim`).

5. Install system dependencies (if any, e.g., for `ffmpeg` or others).
6. Copy the project files into the image.
7. Install Python dependencies (`pip install -r requirements.txt`).
8. Set environment variables for Spotify credentials (if you want to bake them in, though it's often better to supply them at runtime for security).
9. Expose the Streamlit port (8501).
10. Set the entrypoint/command to run the Streamlit app.

For example, a simple Dockerfile (if you're writing one) might look like:

```
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
ENV SPOTIPY_CLIENT_ID=<your_id> \
    SPOTIPY_CLIENT_SECRET=<your_secret> \
    SPOTIPY_REDIRECT_URI=http://127.0.0.1:8888/callback
EXPOSE 8501
CMD ["streamlit", "run", "App.py", "--server.headless=true", "--
server.port=8501", "--server.enableCORS=false"]
```

*(In practice, consider not hardcoding secrets in the Dockerfile – you can pass them via command line instead, as shown below.)*

1. **Running the Container:** Use Docker run with appropriate port mapping and environment variables:

   ```
   docker run -p 8501:8501 \
     -e SPOTIPY_CLIENT_ID='<your_id>' \
     -e SPOTIPY_CLIENT_SECRET='<your_secret>' \
     -e SPOTIPY_REDIRECT_URI='http://127.0.0.1:8888/callback' \
     emotion-music:latest
   ```

   This will start the container, and Streamlit should start up inside it on port 8501. The `-p 8501:8501` maps that port to your host, so you can open your browser on `http://localhost:8501` to see the app. (If you changed the Streamlit port, adjust accordingly.)

2. **Using the App via Docker:** It will be the same Streamlit interface described earlier. The only difference is that it's running inside a container. You might deploy this container on a server and access it remotely. Make sure to configure any firewall or cloud security group to allow the port (8501) if you do so.

3. **Stopping the Container:** If running in foreground, `Ctrl+C` will stop it. If you ran it detached (with `-d`), use `docker ps` to find the container ID and then `docker stop <container_id>`.

**Docker Tips:** - If you plan to deploy this publicly, consider implementing authentication for the Streamlit app or running it behind a web server, because by default it's open to anyone who can access the URL. - Monitor resource usage. The NLP models mean the container will consume a few hundred MB of memory. It's typically fine on a modern PC or small cloud VM, but keep an eye if running many concurrent users. - Docker makes it easy to update the app: just rebuild the image with new code changes and redeploy the container.

By using Docker, you ensure consistency ("it works on my machine and everywhere else"). This is often the preferred deployment method for cloud providers (you could use AWS ECS, Google Cloud Run, Heroku, etc., all of which support deploying containers directly).

## Troubleshooting & FAQ

Despite best efforts, you may encounter some issues. Here are common problems and how to address them:

- **OSError: [WinError 10013] An attempt was made to access a socket in a way forbidden by its access permissions** – This is a Windows-specific error that can occur when the app tries to open a local server for Spotify OAuth or access network resources. Essentially, the network socket is being blocked. Causes could be:
- Another instance is already running and using the port.
- Your firewall or antivirus is blocking the connection.
- Corporate group policies preventing opening of certain ports.

**Solution:** First, ensure no other process is occupying the port 8888 (the default redirect URI port) or 8501 (Streamlit's port). If so, change the port (you can modify the redirect URI to a different port and update Spotify's dashboard). If firewall/AV is the issue, try running Python/Streamlit as an administrator, or adjust the firewall settings to allow the app. Some users solved this by using a different redirect method (e.g., setting `open_browser=False` in `SpotifyOAuth` and manually copying the auth URL into a browser) [29]. If you're seeing this in a Streamlit context, make sure you're not on a network that blocks internal HTTP calls. Running the app locally (localhost) usually avoids such restrictions. Lastly, you can try the Client Credentials flow to skip the local server OAuth – that often sidesteps this error entirely. On Windows, also ensure that the `localhost` or `127.0.0.1` is not being blocked by some IT policy. This error is essentially a permissions issue with networking – run as admin or adjust security settings accordingly [30] [31].

- **Spotify API Rate Limits:** The Spotify API allows only a certain number of requests per second/minute. If you use the app heavily (or many users are using it), you might hit rate limits. Symptoms include getting HTTP 429 errors from the API or Spotipy exceptions indicating rate limit reached. The system as-is doesn't implement automatic retries with backoff, so you might see failures in retrieving songs if this happens.

**Solution:** If you suspect rate limiting: - Slow down the requests. Avoid triggering the app repeatedly in a very short time. Insert delays if needed. - Implement caching for identical requests. For example, if the same genre "pop" is looked up multiple times, cache the results for a minute or two. - Spotipy's `Spotify` object has a property for `max_retries` you can set, or you can catch exceptions and wait. - Usually, for

development or single-user use, you won't hit limits easily. But in a multi-user scenario, consider upgrading to Spotify's commercial API if needed (rarely necessary).

Also note that Spotify's *Recommendations* endpoint has its own rate limiting and sometimes may return fewer results for very niche genres. That's not a "rate limit" issue per se, but a limitation of their data.

- **Auth and Cache Issues:** You might see messages like *"Couldn't read cache at: .cache"* repeated, or the app keeps asking you to authorize. The `.cache` file is used by Spotipy to store the OAuth token so you don't have to log in every run. If the app cannot read or write this file (due to permissions or path issues), or if the token expired and refresh failed, you may get into a loop.

**Solution:** - **Ensure Writable Directory:** If running via Docker or certain environments, the working directory might be read-only. Make sure the app runs in a folder where it can create the .cache file. You can specify a `cache_path="path\to\.cache"` in `SpotifyOAuth` [32] to control where it goes. In Streamlit, the working directory is typically the app folder. - **Delete Stale Cache:** If the cache file exists but something's wrong (e.g., maybe you changed the scope or client secret), delete the `.cache` file and re-run the auth flow fresh. - **Use Client Credentials:** As mentioned, using the client credentials flow avoids the need for user auth and cache entirely (no .cache is used in that case). If your use-case doesn't need user-specific data, this is a simpler route. - **Double-check Redirect URI:** If the auth flow isn't completing, it could be that the redirect URI in your Spotify app doesn't exactly match what the app is using. Update one or the other so they match, otherwise Spotify will refuse the connection and you'll not get a token (leading to repeated auth attempts and cache misses).

In general, once you successfully log in once, the cache should store your access and refresh tokens, and you shouldn't need to login frequently. The cache messages on startup are normal if no cache exists; they're warnings rather than errors (the app then proceeds to get a new token).

- **No Music Found for Emotion:** In rare cases, the system might print *"All systems failed... nothing left to give you."* This means none of the methods found a track. Possible reasons:
- The emotion mapped to a very obscure genre that Spotify doesn't recognize or has no tracks with previews.
- There was a network connectivity issue that caused all API calls to fail (e.g., no internet at that time).
- The Spotify credentials were not correct (though that usually triggers an auth error earlier, not an empty result).

**Solution:** Verify your internet connection. Try a more common emotion/input to see if it works (to isolate if it's the specific input's issue). If it's due to an obscure genre, consider updating the `genre_fallbacks` mapping: this mapping in the code ensures that each custom genre is converted to a valid seed genre [33] [34]. For example, *"punk-rock"* is converted to *"rock"* internally because Spotify doesn't have a seed for "punk-rock" (it has "punk" or "rock"). The validation function printed messages for each emotion's genre validity [35]. If you added a new genre mapping, you might need to add a fallback for it too. In short, double-check that the genre being searched is in the `VALID_SEEDS` list. Adjust if necessary.

- **Model Download/Initialization Errors:** If the Hugging Face models fail to download (maybe due to firewall or no internet), the app will not be able to classify emotions. You might get a pipeline error. Ensure you can access huggingface.co from your environment. If you need to use an offline environment, you could download the model weights separately and point the code to the local

path. This is advanced, but Transformers allows you to use a local folder for a model (by specifying the folder path in place of the model name).

- **Streamlit App not updating:** If you enter text in the Streamlit app and nothing happens or the output doesn't change, check if you need to click a submit button. Some apps require pressing a button after text input. Also check the terminal where Streamlit runs for any error tracebacks and resolve accordingly. Streamlit will show errors in the webpage if they occur during a callback.

If you encounter an issue not covered above, it might help to run the application in a verbose/debug mode: - Add print statements or logs in the code to see how far it gets. - Use try/except around Spotify calls to catch exceptions and print them (some of this is already done in the code with those printouts). - Consult the documentation of Spotipy or the Spotify API if the issue is related to data/limits.

Remember that the community is often a great resource – if you face a weird error, searching the exact error message (e.g., *"WinError 10013 Spotify Spotipy"*) can yield forum discussions or GitHub issues where others have solved it.

## Sample Output

To illustrate the end-to-end functionality, here's a sample run in console mode:

**User input:**

```
Tell me how you're feeling: I just got dumped and everything tastes like dust.
```

**System output (excerpt):**

```
Top Detected Emotions:
  disappointment: 39.27%
  disgust: 13.84%
  embarrassment: 9.62%

Mapped Mood: pensive
Suggested Genre: indie

No luck with direct track search. Trying public playlists...

Falling back to public playlists for genre: 'indie'
  Found 8 previewable tracks in "Indie Breakup Songs".
Here's your playlist scraped from public collections:

         Track                       Artist
Preview                               Spotify Link
0  Somebody Else              The 1975              https://p.scdn.co/mp3-
```

```
preview/...   https://open.spotify.com/track/3W0D...
1  All I Want                  Kodaline             https://p.scdn.co/mp3-
preview/...   https://open.spotify.com/track/1O1O...
2  ... (and so on) ...
```

In this scenario, the input text was quite sorrowful. The system detected *disappointment* as the strongest emotion (around 39.3% confidence) [8], which mapped to a *pensive* mood and the *indie* genre [4]. The direct genre track search didn't find previewable tracks (perhaps due to a network restriction, as indicated by the WinError in the earlier logs), so it fell back to playlist search. It found a playlist titled "Indie Breakup Songs" (for example) and pulled some tracks from there, which it then displayed. The user ends up with a list of indie songs that match the somber mood.

*(If the direct search had succeeded, the output would have been under "Here's your personalized playlist:" with tracks from the indie genre right away. If everything had failed, it would have attempted an AI search on terms like "disappointment" which might find songs like "Disappointment" by certain artists, etc.)*

In a Streamlit app, the same result would be presented perhaps in a more visual way, but the content (the choice of songs) would be the same.

The above example demonstrates how the system deals with a complex emotional statement and still manages to produce a relevant playlist. You can try various inputs – happy, angry, nostalgic, excited, etc. – and observe how the system responds. It's quite interesting to see the **AI's interpretation of emotions and the resulting music mix**.

## Contribution Guidelines

Contributions to the project are welcome! If you have ideas for improvement or new features, feel free to fork the repository and open a pull request. Here are some guidelines for contributing:

- **Discussion:** It's often a good idea to open an issue first to discuss your planned changes, especially for major features. This allows maintainers and the community to provide feedback and ensures effort isn't duplicated.
- **Code Style:** Try to follow the coding style of the existing code. Use clear variable names and add comments where necessary, especially if introducing complex logic.
- **Testing:** If possible, test your changes with a variety of inputs to ensure nothing breaks. If you add a new mapping, test that the genre is valid. If you modify the model pipeline, test at least the basic emotions.
- **Documentation:** Update the README or documentation files to reflect your changes. If you add a new feature, make sure to mention how to use it. (This project aims to be beginner-friendly, so documentation changes are important.)
- **Pull Request:** In your PR description, clearly outline what you changed and why. If it fixes a bug, reference the issue number. If it adds a feature, describe the use-case.

By contributing to this project, you agree to abide by the Code of Conduct (see below). All contributions, no matter how small (even fixing a typo!), are appreciated.

## License

This project is licensed under the **MIT License**. This means you are free to use, modify, and distribute the code, even in commercial projects, as long as you include the original license notice. A full copy of the license text is available in the repository (usually in a `LICENSE` file).

In short, MIT license gives you a lot of freedom: you can fork the code, build upon it, and do pretty much anything, provided you attribute the original source and don't hold the authors liable for anything.

*(If you use this project in your own work or research, a shout-out or citation is always welcome but not required under the license.)*

## Code of Conduct

All contributors and participants in the project's community are expected to adhere to a simple but important **Code of Conduct**. This project follows the common sense principles of respect and professionalism, loosely based on the Contributor Covenant.

**Our Pledge:** We as contributors and maintainers pledge to make participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, race, ethnicity, age, religion, or nationality.

**Standards:** - Be respectful and considerate in your communication. - Provide constructive feedback and be open to feedback on your contributions. - Do not engage in personal attacks, trolling, or insulting language. - Respect differing viewpoints and experiences. - Aim to resolve conflicts amicably and seek help from maintainers if needed.

**Enforcement:** Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the project maintainers (you can typically find contact info in the repository or you can open an issue if appropriate). Maintainers have the right to remove, edit, or reject comments, commits, code, issues, and other contributions that don't align with this Code of Conduct, or to ban temporarily or permanently any contributor for such behaviors.

By participating in this project, you agree to uphold this code. Let's foster a positive, inclusive environment together.

---

*Thank you for using and contributing to the Emotion-Aware Music Recommendation System! We hope this documentation helps you get the most out of the project. Enjoy the music, and may it always suit your mood!* [1] [2]

---

[1] GitHub - ChitturiSaiSuman/Emotion-Aware-Music-Recommendation-System: Recommends music to users based on their current emotion
https://github.com/ChitturiSaiSuman/Emotion-Aware-Music-Recommendation-System

2 19 20 21 boltuix/bert-emotion · Hugging Face
https://huggingface.co/boltuix/bert-emotion

3 4 6 8 9 10 11 12 13 14 15 16 17 18 22 23 24 25 26 27 28 32 33 34 35 Emotion_aware_music_system.pdf
file://file-QGyNEvbPVDtFbmkAuLNQE2

5 Web API Reference | Spotify for Developers
https://developer.spotify.com/documentation/web-api/reference/search

7 facebook/bart-large-mnli · Hugging Face
https://huggingface.co/facebook/bart-large-mnli

29 python - socket.error: [Errno 10013] An attempt was made to access ...
https://stackoverflow.com/questions/2778840/socket-error-errno-10013-an-attempt-was-made-to-access-a-socket-in-a-way-forb

30 Unable to launch Tautulli Windows - WinError 10013 - Reddit
https://www.reddit.com/r/Tautulli/comments/u60h85/unable_to_launch_tautulli_windows_winerror_10013/

31 Socket 10013 error when trying to connect some programs
https://answers.microsoft.com/en-us/windows/forum/all/socket-10013-error-when-trying-to-connect-some/2cc71cb6-24b2-4b63-8f60-32bb444eb8fd