ChatGPT

Emotion-Aware, Anti-Addiction Entertainment Recommendation System

A comprehensive plan for building an **Emotion-Aware, Anti-Addiction Entertainment Recommendation System**. This system will recommend entertainment content (videos, music, games, etc.) tailored to the user's preferences and current emotional state, while also actively preventing overuse and fatigue. The design is broken into phased tasks, each with clear objectives, file structure changes, implementation steps, and success criteria.

Global AI Assistant Instructions

- **Phase-by-Phase Execution:** Follow each phase in order from 0 through 10. Complete all tasks in one phase before moving to the next.
- **Repository Structure:** Use a consistent, modular project layout (e.g., following Cookiecutter Data Science best practices [1]). Common top-level folders include `data/`, `src/`, `models/`, `backend/`, `frontend/`, `notebooks/`, `docs/`, and configuration files (e.g., `README.md`, `requirements.txt` or `pyproject.toml`).
- **Code Quality:** Write clean, well-documented code. Include unit tests and type annotations. Use version control commits at the end of each phase.
- **Tools and Frameworks:** Python is the primary language. Use libraries/frameworks as needed (e.g., Pandas/PySpark for data ingestion, Scikit-learn or TensorFlow/PyTorch for models, FastAPI for backend, React/Vue/Angular for frontend).
- **Reporting:** At the end of each phase, ensure the code and functionality meet the success criteria before proceeding. Document any assumptions or decisions.

Phase 0: Repository Initialization

- **Goal:** Set up the project repository with a clear, modular structure and initial configuration files. Establish development environment and CI/CD basics.
- **Files & Folders:**
- Root: `README.md`, `LICENSE`, `Makefile` or `pyproject.toml` / `requirements.txt`, environment setup (e.g. `environment.yml` or `requirements.txt`).
- `data/`: subfolders `raw/`, `interim/`, `processed/`, and `external/` for dataset stages [1].
- `src/`: source code modules (e.g., `features.py`, `models.py`, `recommendation.py`).
- `backend/`: API code (will contain FastAPI app).
- `frontend/`: web app code.
- `notebooks/`: exploratory Jupyter notebooks (labeled with numbering).
- `models/`: serialized models and artifacts.
- `tests/`: automated tests.
- **Tasks:**
- Initialize a Git repository. Add `.gitignore` (Python, IDE, OS files).

- Create initial `README.md` with project overview.
- Establish virtual environment (e.g. `python -m venv venv`). Lock dependencies (`requirements.txt` or `pyproject.toml`).
- Follow a data science project layout (e.g., Cookiecutter Data Science structure) to organize code and data [1].
- Set up a basic Makefile or scripts (e.g., `make setup`, `make test`) for common tasks.
- (Optional) Configure CI pipeline (e.g., GitHub Actions) for linting and testing on each commit.
- **Output / Success:**
- Repository is initialized with the directory structure above, and the project builds without errors (e.g., `pip install -r requirements.txt`).
- Basic README and license in place.
- Development tools (formatter, linter, test runner) are configured.

Phase 1: Data Ingestion & Feature Schema

- **Goal:** Define what data will be used and how to process it. Ingest or simulate data sources (user profiles, content metadata, interaction logs, emotion signals) and design feature schemas.
- **Files & Folders:**
- `data/`: place raw datasets (`data/raw/`) and intermediate files (`data/interim/`).
- `src/data_ingestion.py`: module to load and preprocess data.
- `src/feature_schema.py`: definitions of user/item features, context features, and emotion signals.
- **Tasks:**
- **Identify Data Sources:** Decide on required inputs: user demographics, content metadata (genre, length, tags), historical user-item interactions (views, likes, time spent), emotion signals (e.g. facial images, sensor data, text sentiment), and usage patterns (session lengths, time-of-day). If real data is unavailable, create synthetic or use public datasets (e.g., movie ratings with timestamps) to simulate interactions.
- **Data Ingestion Pipeline:** Implement loading routines (`src/data_ingestion.py`) to read and merge raw data. Clean data (remove duplicates, handle missing values). Store processed data in `data/processed/`.
- **Feature Schema Definition:** In `src/feature_schema.py`, define feature sets for:
    - **User features:** e.g. age, demographics, historical preferences.
    - **Item features:** e.g. category/genre, content length, popularity.
    - **Context features:** e.g. time-of-day, day of week, device type. Contextual signals greatly influence behavior [2].
    - **Emotion features:** e.g. current mood (valence/arousal), detected via sensors or analysis.
    - **Anti-addiction signals:** e.g. cumulative usage time today, count of continuous viewing sessions, or engagement counters (clicks per session).
    - Utilize feature engineering best practices: turn raw behavior logs into counters and rates (e.g. click-through rates, view counts) [3] [4]. For example, build user-item interaction rates and time-based counters.
    - Encode categorical features (e.g. one-hot genres, embeddings for users/items).
- **Feature Store (Optional):** If needed, set up a simple feature store or caching mechanism to compute and retrieve features efficiently.
- **Output / Success:**
- A working ingestion script that produces cleaned datasets in `data/processed/`.

- A documented feature schema listing each feature and its purpose (user/item/context/emotion).
- Sanity checks passed: e.g. sample features extracted correctly.

Phase 2: Baseline Recommendation Engine

- **Goal:** Build a basic recommendation engine to serve as the foundation. At this stage, it should recommend relevant items without considering emotion or fatigue.
- **Files & Folders:**
- `src/recommender/baseline.py`: code for the baseline model.
- `src/models/` or `src/recommender/models.py`: scripts to train collaborative filtering or content-based models.
- `models/baseline/`: saved model files.
- **Tasks:**
- **Candidate Generation (Retrieval):** Implement a simple retrieval mechanism, such as popularity-based (top items) or user-based collaborative filtering (find similar users by past likes, then recommend items they liked). Tools like Surprise or implicit matrix factorization can be used. This produces a set of candidate items for each user.
- **Ranking Model:** Build a ranking model to score candidates. Initially, use a simple approach (e.g. logistic regression or gradient-boosted trees on the features defined in Phase 1). The features might include user-item interaction history, content similarity, and context features. As noted in industry, combining watch history with genre preferences and time-of-day patterns improves recommendations [5].
- **Evaluation:** Evaluate this baseline using offline metrics (Precision@K, Recall@K, NDCG, MAP [6]) on a test split of historical data. Ensure the recommender returns sensible results (e.g. top-N lists for test users).
- **Output / Success:**
- A script or module that, given a user ID (and optional context), outputs a ranked list of recommended content IDs.
- Baseline performance metrics recorded (e.g., Precision@10 $\geq$ random baseline).
- Smoke tests verifying the engine runs end-to-end.

Phase 3: Emotion Inference Module

- **Goal:** Develop a module to infer the user's current emotional state from available signals (e.g., facial expression, voice tone, text, physiological data).
- **Files & Folders:**
- `src/emotion/`: folder for emotion-related models and utilities.
- `src/emotion/face_emotion.py`: module for facial emotion recognition (if camera input is used).
- `src/emotion/text_sentiment.py`: module for text-based sentiment analysis (if chat or text input).
- `src/emotion/sensor.py`: module for physiological signal processing (if using wearable data).
- **Tasks:**
- **Choose Modalities:** Decide which modalities to use. For example: facial images (via webcam) using a CNN-based FER model; text sentiment (e.g., using Transformers) from user input; or audio analysis (if voice is available). Facial FER is a popular non-invasive option because cameras are ubiquitous [7].

- **Implement Inference:** Use or train models to output an emotion label or valence/arousal score. For instance, apply a pre-trained model (e.g., from the `deepface` library or `fer` package) for facial emotion detection. Alternatively, use a sentiment analysis API for text.
- **Integrate Module:** Create a function `infer_emotion(input) -> emotion_vector`. Ensure it runs in real-time (or near-real-time) with minimal latency.
- **Testing:** Validate the emotion inference on sample inputs to ensure reasonable outputs (e.g., test that happy faces yield "happy").
- **Output / Success:**
- A callable module (`infer_emotion`) that processes input data and returns an emotion state (e.g., `{happiness:0.8, sadness:0.1, ...}` or valence/arousal floats).
- The system can read sample inputs (images/text) and produce consistent emotion predictions.

[7] *Figure: Multi-modal emotion detection (e.g., facial expression recognition) can feed user mood into the system.*

Phase 4: User State Vector Construction

- **Goal:** Build a unified "user state vector" that captures the user's current profile, context, and mood. This vector will serve as input to the final ranking model.
- **Files & Folders:**
- `src/state.py`: code to assemble the state vector.
- (Optionally) `src/embedding/`: code for embeddings (e.g. user/item embedding lookup).
- **Tasks:**
- **Combine Signals:** In `src/state.py`, implement a function that takes user ID and current context and returns a state vector. This should include:
    - **User Profile:** Static features (demographics, long-term preferences).
    - **Recent Interactions:** e.g. one-hot or embeddings of recently watched/liked items.
    - **Emotion:** The output from Phase 3.
    - **Fatigue/Usage:** Derived features like `today_watch_time` or `sessions_in_row`.
    - **Contextual:** Time-of-day, device, location, etc. (Studies show that usage patterns change by context; e.g. morning short-form vs. evening long-form [2].)
- **Dimensionality:** Ensure the state vector is a fixed-size numeric array. You may use embeddings or hash features for high-cardinality fields.
- **Normalization:** Apply scaling or normalization to features as needed.
- **Sanity Check:** Log or test a few state vectors to confirm they incorporate all parts (profile, mood, etc.).
- **Output / Success:**
- A function `get_user_state(user_id, context)` that returns a consistent vector. Example tests: if the same inputs are given, the vector is reproducible; different contexts produce different vectors.
- The state vector should reflect intuitive changes (e.g., if current emotion changes, some dimensions differ).

[8] [5] *Figure: In sequential recommendation research, the user's state vector (based on history and context) is the input to the action policy. We similarly build a state embedding capturing preferences, context, and emotion.*

Phase 5: Anti-Addiction & Fatigue Modeling

- **Goal:** Incorporate user fatigue and addiction-prevention logic. The system should detect when a user may be getting tired or excessively engaged and adjust recommendations accordingly.
- **Files & Folders:**
- `src/anti_addiction.py`: model or rules for fatigue/addiction.
- `models/anti_addiction/`: (Optional) trained model artifacts if using ML.
- **Tasks:**
- **Define Fatigue:** Based on research, fatigue can be measured by negative feelings (boredom, tiredness) from repetitive content [9]. Implement metrics such as:
    - **Repetition Index:** How many similar items has the user seen recently (e.g. count same genre in last N items).
    - **Session Length:** Duration or number of items consumed in current session.
    - **Time-on-App:** Total usage time today.
- **Fatigue Score:** Create a function that computes a `fatigue_score` from these signals (e.g., a weighted sum or a small neural net). You may start with simple thresholds or linear model and later refine. Studies show CTR drops when users see too much of the same category [10].
- **Interventions:** Define how to respond to high fatigue: for example, if `fatigue_score` exceeds a threshold, modify recommendations by injecting novel content or even pausing suggestions. Also consider user well-being principles: e.g., "take a break" notifications or limiting screen time [11].
- **Model Training (Optional):** If using a machine learning approach (e.g., time-series model or reinforcement learning), train it to predict future disengagement.
- **Output / Success:**
- A function `compute_fatigue(user_state)` that outputs a numerical fatigue indicator.
- Demonstrated behavior: e.g. simulate a user binge-watching similar content and show the system increases fatigue score.
- (Optional) Rules for intervention are documented (e.g., "After 3 hours of continuous use, do X" based on research recommendations [11]).

[9] [11] *Figure: Research defines fatigue as negative feelings (tiredness, boredom) from repetitive content [9]. Ethical design suggests adding break reminders and prioritizing user well-being over endless engagement [11].*

Phase 6: Final Ranking Engine

- **Goal:** Combine all signals (baseline score, emotion, fatigue) to produce the final ranked list of recommendations. This is a multi-criteria ranking step that balances relevance, personalization, and anti-addiction.
- **Files & Folders:**
- `src/recommender/final_ranker.py`: logic to compute final scores and sort items.
- If using a learned model: `src/recommender/ranker_model.py` and saved model in `models/ranker/`.
- **Tasks:**
- **Score Adjustment:** Start with the baseline ranking score. Apply adjustments based on emotion and fatigue: for instance, if a user is stressed, prioritize relaxing content; if fatigue is high for a category, down-weight those items (to diversify).

- **Re-ranking Filters:** Apply rule-based filters if needed (e.g., remove content flagged as “addictive” or very long if user is tired). You can also transform scores: e.g., multiply the baseline score by a factor that decreases with fatigue.
- **Diversity & Fairness:** Incorporate diversity: avoid recommending many similar items in a row (Google suggests re-ranking by genre/metadata to maintain diversity) [12] . For example, if top-K are too similar, shuffle or insert different genres. This also addresses fatigue [10] .
- **Optimization (Optional):** If you implement a learning-to-rank model, train it using combined features (state vector + candidate features) so that it inherently balances these factors.
- **Output / Success:**
- The final sorted recommendation list should respect the adjustments: e.g. repeated categories should be spaced out, emotional relevance increased, and high-fatigue items lowered.
- Test scenarios: Show that if a user has high fatigue for comedy movies, the final list has fewer comedies.
- The re-ranked list remains high-quality according to metrics, while satisfying the new constraints.

[13] [12] *Figure: The final stage re-ranking can filter or adjust scores by criteria (e.g., freshness or diversity)* [13] *. Ensuring diversity (e.g., by genre) keeps the list from becoming “boring” and helps reduce fatigue* [12] *.*

Phase 7: Explainability Layer (XAI)

- **Goal:** Provide explanations for each recommendation to improve user trust and understanding. The system should output human-readable reasons or feature attributions for why an item was recommended.
- **Files & Folders:**
- `src/explainability.py`: functions to generate explanations.
- Possibly submodules for different explanation techniques (e.g., `lime_explainer.py`, `attention_visualizer.py`).
- **Tasks:**
- **Choose XAI Methods:** Decide on an approach. Options include:
    - **Intrinsic Models:** Use an interpretable model (e.g., attention-based networks, factorization with item attributes) that can highlight important factors.
    - **Post-hoc Explainers:** Use libraries like LIME or SHAP to get feature importance for the final recommendation score.
    - **Template Explanations:** For example, “This content is recommended because you liked similar {genre} and we detect your current mood as {mood}.” Templates can fill in keywords (genre, mood, etc.).
- **Implement Explanation:** In `src/explainability.py`, write a function `explain(item_id, user_state)` that returns an explanation string or structure. For example, extract the top features from SHAP values or take note of model weights.
- **Evaluation:** Ensure the explanations are sensible. For example, if recommending an action movie because the user liked other action movies and user’s mood is adventurous, the explanation should mention those.
- **Output / Success:**
- Every recommended item is accompanied by a textual (or visual) explanation. For instance: “Recommended because you enjoyed action films and seem excited (happy mood).”
- Demonstrated boost in interpretability: e.g. unit tests that check `explain()` runs and returns non-empty rationale.

- **Note:** Explainability enhances transparency and trust [14]. Ensure the explanations are concise and relevant to the user's state and item features.

[14] *Figure: Explainable recommendation systems generate human-readable justifications for suggestions, increasing user trust and understanding of the model's decisions [14].*

Phase 8: Backend API (FastAPI)

- **Goal:** Expose the recommendation system via a web API so clients can request recommendations, submit emotion data, and retrieve explanations.
- **Files & Folders:**
- `backend/app.py`: main FastAPI application.
- `backend/routers/`: folder for route definitions (e.g., `recommend.py`, `emotion.py`, `user.py`).
- `backend/schemas/`: Pydantic models for request/response formats.
- **Tasks:**
- **Set up FastAPI App:** Initialize a FastAPI instance (`app = FastAPI()`) in `backend/app.py`.
- **Define Endpoints:**
    - **POST /recommend:** Accepts user ID (and optional context/emotion) and returns a list of recommended items with scores and explanations.
    - **POST /emotion:** (Optional) Endpoint to submit raw emotion input (e.g. image or text); returns inferred emotion.
    - **GET /state:** Returns the current user state vector or its summary.
    - **GET /health:** A simple healthcheck endpoint.
    - **Other:** e.g., `GET /items/{id}` for item details.
      Use Pydantic models for request and response bodies as shown in examples [15] [16].
- **Integrate Modules:** In each endpoint handler, call the appropriate code from previous phases: e.g., `/recommend` calls the final ranker and explainability.
- **Run and Test:** Use Uvicorn to run the API locally. Verify endpoints using the interactive docs (`/docs`) that FastAPI provides.
- **Output / Success:**
- A running FastAPI service with the above endpoints. For example, a `curl` to `POST /recommend` returns a valid JSON with recommended item IDs and explanations.
- Automated tests (using pytest) for each endpoint (mocking underlying logic if needed).
- Endpoint documentation auto-generated by FastAPI (via Pydantic schemas) works correctly [15].

[15] *Figure: FastAPI is a modern Python framework for quickly building backend APIs. We can define routes (e.g., POST /recommend) and use Pydantic for request/response models as shown above [15] [16].*

Phase 9: Frontend Dashboard

- **Goal:** Create a user-facing interface that displays recommendations, explanations, and usage stats. The dashboard should allow interaction (e.g., giving feedback, viewing state, enabling break suggestions).
- **Files & Folders:**
- `frontend/`: directory for the web app (e.g., React or Vue project).
- Components: e.g., `Dashboard`, `RecommendationList`, `EmotionInput`, `UsageChart`.

- `frontend/public/` and `frontend/src/`.
- **Tasks:**
- **Framework Setup:** Initialize a front-end project (e.g. `create-react-app`, Next.js, Vue CLI). Install needed libraries for HTTP requests (Axios/Fetch) and visualization (e.g., Chart.js, Recharts).
- **UI Components:**
    - **Recommendation List:** Displays recommended items (title, thumbnail) and their explanations.
    - **Emotion/Mood Input:** Optionally allow the user to input or upload data for mood detection (e.g., take a photo).
    - **Usage/Notifications:** Show user's usage stats (daily screen time, fatigue score) in charts or gauges. Display alerts if too much screen time (leveraging anti-addiction logic).
    - **Settings:** Controls for preferences and anti-addiction (e.g., "Enable break reminders").
- **API Integration:** Connect frontend to FastAPI endpoints. On user actions (login or page load), fetch recommendations via `/recommend`. Send emotion inputs to `/emotion`, and update UI based on API responses.
- **Styling and UX:** Ensure the UI is clean and responsive. Use a UI library (e.g., Material-UI or Bootstrap) for layout. Visualization of user state (e.g., fatigue gauge) can make the anti-addiction feature clear.
- **Output / Success:**
- A running web application displaying dynamic recommendations and explanations for a logged-in user.
- The dashboard updates when the user's emotion or context changes.
- Anti-addiction notifications appear appropriately (e.g., "Time for a short break!" after prolonged use).

Phase 10: Evaluation & Reporting

- **Goal:** Thoroughly evaluate the recommendation system on both accuracy and user-centric metrics, and generate a report of the results.
- **Files & Folders:**
- `src/evaluation/`: evaluation scripts.
- `reports/`: generated evaluation reports and plots.
- **Tasks:**
- **Offline Metrics:** Compute traditional recommender metrics on a held-out test set: Precision@K, Recall@K, NDCG, MRR, MAP [6]. Compare the baseline (Phase 2) vs. the final engine (Phase 6) to quantify improvements.
- **Behavioral Metrics:** Evaluate engagement and diversity: e.g., session CTR, average session length, category entropy. Include diversity/novelty scores as mentioned in ranking literature [6]. Specifically measure if anti-addiction logic reduces overspecialization (i.e., increases diversity when needed).
- **User Study (Optional):** If feasible, simulate or collect user feedback on satisfaction. For example, A/B test showing explanations vs. none, or measuring self-reported fatigue levels.
- **Report Generation:** Produce a Markdown or PDF report summarizing methodology and results. Include plots (e.g., Precision@K curves, fatigue score trends, user engagement before/after interventions).
- **Output / Success:**
- A documented report in `reports/` that details the evaluation setup, metrics, and analysis.
- Quantitative results showing that the system meets design goals (e.g., maintains high relevance while improving diversity or reducing user time-on-app).

- Confirmation that anti-addiction features have a measurable effect (e.g., users take breaks more often or fatigue score is lowered).

[6] *Figure: Evaluate the recommender using ranking metrics like Precision@K or NDCG [6], and also consider behavior-oriented metrics (diversity, novelty) to assess user experience beyond accuracy.*

Summary: By following these phases, the AI assistant will build a complete entertainment recommendation system that personalizes to emotions while promoting healthy usage. Each phase's output builds the foundation for the next, resulting in a working prototype ready for testing and deployment.

---

[1] Cookiecutter Data Science
https://cookiecutter-data-science.drivendata.org/

[2] [3] Feature Engineering for Recommendation Systems – Part 1 - AI Infrastructure Alliance
https://ai-infrastructure.org/feature-engineering-for-recommendation-systems-part-1/

[4] [5] What is the role of feature engineering in recommender systems?
https://milvus.io/ai-quick-reference/what-is-the-role-of-feature-engineering-in-recommender-systems

[6] 10 metrics to evaluate recommender and ranking systems
https://www.evidentlyai.com/ranking-metrics/evaluating-recommender-systems

[7] ijisrt.com
https://www.ijisrt.com/assets/upload/files/IJISRT25AUG1648.pdf

[8] A social image recommendation system based on deep reinforcement learning | PLOS One
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0300059

[9] Making algorithmic app use a virtuous cycle: Influence of user gratification and fatigue on algorithmic app dependence | Humanities and Social Sciences Communications
https://www.nature.com/articles/s41599-024-03221-z?error=cookies_not_supported&code=c81ca11b-89c3-4657-9884-7b1753efcc78

[10] [2405.11764] Modeling User Fatigue for Sequential Recommendation
https://ar5iv.org/pdf/2405.11764

[11] Social Media Algorithms and Teen Addiction: Neurophysiological Impact and Ethical Considerations - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11804976/

[12] [13] Re-ranking | Machine Learning | Google for Developers
https://developers.google.com/machine-learning/recommendation/dnn/re-ranking

[14] Explainable Recommendation Systems
https://www.emergentmind.com/topics/explainable-recommendation-systems

[15] [16] Build a movie recommendation app using Python, Fast API and React — Part 2 | by Alfurquan Zahedi | Medium
https://medium.com/@zahedialfurquan20/build-a-movie-recommendation-app-using-python-fast-api-and-react-part-2-b8ee4660c7ce