AI-Driven Hyper-Personalization & Recommendations

Table of contents

1. Overview
2. Data Flow
3. Approach
4. Key-Technical considerations
5. Security Considerations
6. Ethical Considerations
7. Insights
8. Business recommendations based on AI-Driven findings

**Overview**

Hyper-personalization is key strategy for businesses to enhance customer experience, increase engagement. Businesses can create unique and engaging experiences for the customers, driving loyalty and growth.

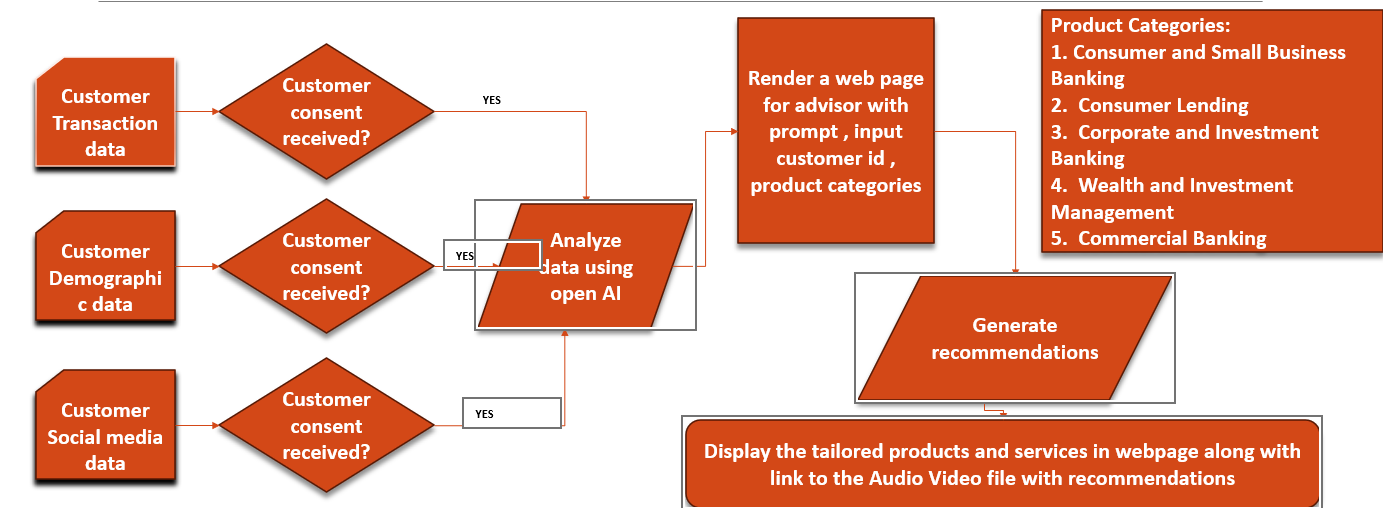
To recommend the personalization choices to customer:

1. Reading Customer data, along with consent of the customer.
2. If consent is provided by customer,

* Uses customer tweets from file generated using Twitter API, conduct sentiment analysis with help of TextBlob.
* Construct personalized prompts using transaction history and sentiment data.
* Utilizes OpenAI API to generate recommendations.
* Converts the generated recommendations into Audio with pyttsx3, to incorporate subtitles, background music, and a watermark. Finalizes the video composition using Moviepy Python library.

**Data flow**

Input (UI/CLI + Excel Files) → Data Merge (Pandas) → Prompt (Customer Data) → OpenAI (Text) → MoviePy (Video) → Output (Text File + Video File + UI)



Input Collection

* Web Mode: User submits a form via Flask (customer\_id, optional prompt, and categories) through the HTML UI.
* CLI Mode: User enters customer\_id via command-line input; default category is "Consumer and Small Business Banking".
* Environment variables (OPENAI\_API\_KEY, EXCEL\_PASSWORD) are loaded using dotenv.

Data Loading

* Source Files: Three encrypted Excel files (customer\_transactions.xlsx, customer\_profile.xlsx, twitter\_data.xlsx) are decrypted using msoffcrypto with EXCEL\_PASSWORD.
* Processing: pandas reads and merges the files into a single DataFrame based on customer\_id, including transaction, profile, and Twitter data. Consent fields (consent, consent\_social\_media) are validated.

Prompt Generation

* User Prompt: Customer data (e.g., name, transactions, sentiment) is extracted and formatted into a string for OpenAI. If no data or consent is missing, an error message is returned.
* System Prompt: Default or user-provided prompt is adjusted with selected categories (e.g., "Consumer Lending").

Recommendation Generation

* OpenAI API: The combined prompt is sent to gpt-4o-mini, generating a recommendation text (max 250 tokens). Results are logged and emitted via SocketIO.
* Output Storage: Text is written to recommendations.txt in artifacts/output.

Video Creation

* Narration: pyttsx3 converts the recommendation text to audio, saved as temp\_audio.mp3.
* Background: pytubefix downloads a YouTube video to temp\_background\_video.mp4.
* Assembly: MoviePy combines:
  + Background video (resized, looped to match audio duration).
  + Text overlays (recommendation split into chunks).
  + Audio (narration + background music from background\_music.mp3).
* Output: Final video is saved as recommendations\_video.mp4 in artifacts/output.

Delivery

* Web Mode: Flask serves the recommendation text and video link via SocketIO events, updating the UI in real-time.
* CLI Mode: Results are logged to the console; video is saved locally.
* Temporary files (temp\_audio.mp3, temp\_background\_video.mp4) are cleaned up.

**Approach**

Approach to Designing the application

1. Modular Design
   * Goal: Break functionality into reusable, independent components.
   * Implementation: Separate functions for data loading (load\_data), prompt generation (generate\_user\_prompt), recommendation generation (get\_recommendation), and video creation (generate\_video\_with\_moviepy). This enhances maintainability and testing.
2. User-Centric Workflow
   * Goal: Cater to both technical (CLI) and non-technical (web) users.
   * Implementation: Dual execution modes: CLI for quick runs with minimal input, and a Flask-based web app with an intuitive UI for interactive use. Real-time feedback via SocketIO keeps users informed.
3. Data-Driven Personalization
   * Goal: Leverage customer data for tailored banking recommendations.
   * Implementation: Use pandas to merge transaction, profile, and Twitter data, feeding it into OpenAI’s gpt-4o-mini with a structured prompt. Consent checks ensure ethical data use.
4. Multimedia Engagement
   * Goal: Enhance recommendations with visual and audio appeal.
   * Implementation: Combine pyttsx3 for narration, pytubefix for background video, and MoviePy to overlay text and mix audio, creating a polished video output.
5. Security and Robustness
   * Goal: Protect sensitive data and ensure reliable execution.
   * Implementation: Encrypt Excel files with msoffcrypto, validate prerequisites (check\_prerequisites), and implement logging for debugging. Temporary file cleanup prevents resource leaks.
6. Iterative Development
   * Goal: Build a flexible, extensible prototype.
   * Implementation: Use Python for rapid prototyping, hardcode initial settings (e.g., YouTube URL), and leave room for future enhancements (e.g., configurable parameters).

Core Philosophy

* Start Simple: Focus on core functionality (data to recommendation to video).
* Integrate Gradually: Layer in web UI, real-time updates, and multimedia.
* Prioritize User Experience: Ensure outputs are engaging and accessible.

**Key-Technical considerations**

Key technical considerations and the high-level technology choices

1. **Data Handling and Security**
   * **Encrypted Excel Files**: The script uses msoffcrypto to decrypt Excel files (customer\_transactions.xlsx, customer\_profile.xlsx, twitter\_data.xlsx) with a password stored in an environment variable (EXCEL\_PASSWORD). This ensures sensitive customer data is protected at rest.
   * **Consent Checks**: The script enforces consent (consent and consent\_social\_media) before processing data, aligning with privacy regulations like GDPR or CCPA. Missing or invalid consent halts processing for a customer.
   * **Pandas for Data Processing**: pandas is used to load, merge, and filter data based on customer\_id. Merging transaction, profile, and Twitter data into a single DataFrame enables comprehensive customer analysis but assumes consistent column naming and data integrity across files.
2. **API Integration**
   * **OpenAI API**: The script leverages OpenAI’s gpt-4o-mini model to generate personalized banking product recommendations. The system prompt is dynamically adjusted with user-selected categories, and the user prompt includes customer data, ensuring context-aware outputs. Error handling for API calls is present but could be more robust (e.g., retry logic for transient failures).
   * **Token and Temperature Settings**: max\_tokens=250 limits output length to fit the 220-word constraint, and temperature=0.7 balances creativity and coherence in recommendations.
3. **Web Application**
   * **Flask and SocketIO**: The script uses Flask as a lightweight web framework and SocketIO for real-time communication. This enables live updates (logs, recommendations, video links) to the frontend, enhancing user experience.
   * **HTML/CSS/JavaScript**: The frontend is a single-page app with a form for input (customer ID, categories, prompt) and dynamic sections for results. JavaScript handles form submission and SocketIO events, while CSS provides a polished UI. However, the template is embedded as a string, which could complicate maintenance for larger UIs.
4. **Video Generation**
   * **MoviePy**: Used to create a video with text overlays (recommendations), background video, narration, and music. Text is split into chunks for readability, synced with audio duration, and styled with ImageMagick for rendering.
   * **pyttsx3**: Generates text-to-speech narration saved as a temporary MP3. The speech rate (140) is hardcoded, which might not suit all content lengths or user preferences.
   * **YouTube Video Download**: pytubefix downloads a background video from a hardcoded URL. Directory management and cleanup are handled, but the process is brittle (e.g., relies on YouTube availability and specific stream formats).
   * **Resource Management**: Temporary files are cleaned up with retry logic for PermissionError, but lingering files could still occur if exceptions are unhandled elsewhere.
5. **Logging and Debugging**
   * **Custom Logging**: The script uses Python’s logging with a custom SocketIOHandler to stream logs to the frontend. This is great for real-time feedback but lacks persistent storage (e.g., file logging) for post-mortem analysis.
   * **Prerequisite Checks**: The check\_prerequisites function validates dependencies, API keys, and file paths, ensuring the script fails fast if setup is incomplete.
6. **File System Management**
   * **Pathlib**: The script uses Path for cross-platform path handling, defining relative paths from the script’s location (BASE\_DIR). This is robust but assumes a specific directory structure (data, artifacts/output, artifacts/temp).
   * **Static File Serving**: Flask serves the generated video from artifacts/output, requiring correct static folder configuration.
7. **Execution Modes**
   * **CLI vs. Web**: The script supports both command-line (--cli) and web modes. CLI mode prompts for a customer ID and uses a default category, while web mode offers a UI with category selection. This dual-mode design adds flexibility but increases complexity.

**High-Level Technology Choices and Rationale**

1. **Python**
   * **Why**: A versatile, high-level language with strong libraries for data processing (pandas), AI (openai), web development (Flask), and multimedia (MoviePy, pyttsx3). Ideal for rapid prototyping and integrating diverse functionalities.
   * **Trade-off**: Slower execution compared to compiled languages, but team is expert in Python and performance isn’t critical for this use case.
2. **Pandas**
   * **Why**: Efficiently handles tabular data (Excel files) with powerful merging and filtering capabilities, essential for combining customer datasets.
   * **Trade-off**: Memory-intensive for large datasets; alternatives like polars could offer better performance for big data.
3. **OpenAI API**
   * **Why**: Provides state-of-the-art natural language generation for personalized recommendations, leveraging customer data and sentiment analysis. The gpt-4o-mini model balances cost and quality.
   * **Trade-off**: Dependency on external API introduces latency, cost, and potential downtime risks. Local models (e.g., via Hugging Face) could reduce this but require more setup.
4. **Flask and SocketIO**
   * **Why**: Flask is lightweight and sufficient for a simple web app, while SocketIO enables real-time updates, critical for a responsive UI during recommendation and video generation.
   * **Trade-off**: Flask isn’t ideal for large-scale apps; a framework like Django might be overkill here but offers more built-in features
5. **MoviePy and pyttsx3**
   * **Why**: MoviePy simplifies video editing (text overlays, audio mixing), and pyttsx3 provides offline text-to-speech, avoiding additional API costs. Together, they create engaging video outputs.
   * **Trade-off**: MoviePy relies on ImageMagick, adding an external dependency, and pyttsx3’s voice quality is basic compared to cloud-based TTS (e.g., Google Text-to-Speech).
6. **pytubefix**
   * **Why**: Enables downloading YouTube videos for background content, keeping the script self-contained without hosting large video files.
   * **Trade-off**: Relies on YouTube’s API stability and terms of service; a local video library would be more reliable but less dynamic.
7. **msoffcrypto**
   * **Why**: Decrypts password-protected Excel files, ensuring data security aligns with banking standards.
   * **Trade-off**: Adds complexity; unencrypted files or a database might simplify but compromise security.

This script is a solid proof-of-concept for personalized banking recommendations with multimedia output, leveraging Python’s ecosystem effectively. With refinements, it could scale to production use.

**Business recommendations based on AI-Driven findings**

Here is one of Business recommendations provided using the created model:

Hello John, I am your Investment Advisor Jeremy Porter. Wells Fargo would like to recommend you a few products that can enhance your banking experience and support your financial goals.

* First, I recommend the Wells Fargo Home Mortgage. Understanding your recent concerns regarding the home loan application process, this mortgage offers tailored solutions that can ease your path to homeownership with competitive rates and flexible terms designed for your unique situation.
* Second, consider the Wells Fargo Personal Loan. With your strong income and positive banking history, this personal loan can provide you with the funds you need for any significant purchases or debt consolidation, all while enjoying fixed monthly payments and no prepayment penalties.
* Finally, I suggest the Wells Fargo Cash Back Rewards Card. This credit card allows you to earn cash back on your everyday purchases, including clothing, which has been a recent focus for you. With no annual fee and the ability to earn rewards, it will complement your spending habits while providing you with financial flexibility.
* These products not only align with your current needs but also set you on the path to achieving your financial aspirations. Let’s take the next steps together.

Visit wellsfargo.com for more details.

**Conclusion**

**Summary:** This script delivers hyper-personalized banking recommendations by integrating customer data, AI, and multimedia output, enhancing customer engagement for Wells Fargo.

**Strengths:**

* 1. Secure data handling with encryption and consent checks.
  2. Real-time web interface via Flask and SocketIO.
  3. Dynamic text and video generation using OpenAI and MoviePy.

**Value**: Combines advanced analytics, natural language processing, and visual storytelling to drive product adoption.

**Future Potential:**

* 1. Scale with async processing for multiple customers.
  2. Enhance with configurable settings and robust error handling.
  3. Explore local AI models for cost efficiency.

**Takeaway:** A powerful proof-of-concept showcasing Python’s versatility in financial services innovation.