PersonaPundit.ai: Leveraging AI to Generate User Personas from Product Reviews

Hardik Sodhani (NUID: 002770306), Rucha Chotalia (NUID: 002711888)

August 15, 2024

Abstract

This report presents PersonaPundit.ai, a Streamlit-based application utilizing advanced AI techniques to derive detailed user personas from product reviews. The integration of technologies like Snowflake, Amazon S3, and Google API alongside sophisticated AI models offers comprehensive insights into customer demographics, preferences, and behaviors.

1 Introduction

The application of advanced AI to understand customer behavior presents transformative opportunities for businesses. PersonaPundit.ai leverages generative AI, RAG (Retriever-Augmented Generation), LLMs (Large Language Models), and LangChain to create dynamic, data-driven user personas, revolutionizing traditional market research methods and enhancing strategic marketing initiatives.

2 Project Description

PersonaPundit.ai integrates AI technologies to analyze vast datasets from product reviews, enabling the automatic generation of detailed user personas. By leveraging OpenAI's GPT-3.5 and Gemini models, and utilizing Snowflake for data management, Amazon S3 for data storage, and Google APIs for additional web research, the application ensures a robust process for extracting actionable insights from unstructured data.

3 Technical Implementation

3.1 Generative AI and LLMs

PersonaPundit.ai utilizes generative AI capabilities of GPT-3.5 and Gemini to interpret and synthesize information from product reviews, creating rich, detailed personas that reflect varied customer traits and preferences.

3.2 Retriever-Augmented Generation (RAG)

The application employs RAG to dynamically retrieve relevant information from a Snowflake-managed database during the persona generation process. This method enhances the contextual relevance and accuracy of the generated personas.

3.3 LangChain Integration

LangChain libraries facilitate the seamless integration of LLMs and RAG techniques, orchestrating complex workflows that combine data retrieval, processing, and synthesis within the application's architecture.

3.4 Key Functionalities

- Dynamic retrieval and analysis of review data from Snowflake.
- Integration of generative AI to craft detailed personas.
- Enhanced data interaction using LangChain for efficient processing.

3.5 Data Flow

Data management processes encompass the extraction of data from Snowflake, its enhancement through AI-driven analysis, and the presentation of synthesized insights via the Streamlit interface.

4 Challenges and Solutions

4.1 Integration of Advanced AI Technologies

The initial challenges of integrating advanced AI technologies such as RAG and LangChain were addressed by modularizing the application, improving data flow efficiency, and enhancing system scalability.

4.2 Data Privacy and Security

Enhanced security protocols and compliance with data protection regulations were implemented to safeguard user data, addressing the critical challenges of data privacy and security.

5 Conclusion and Future Scope

PersonaPundit.ai showcases the power of AI in transforming data into strategic insights. Future developments will focus on expanding AI capabilities and integrating more diverse data sources to further refine the accuracy and utility of generated personas.

6 Appendices

6.1 Full Project Code

```
import streamlit as st
import pandas as pd
import snowflake.connector
from openai import OpenAI
import re
import os
from langchain.retrievers.web_research import
    WebResearchRetriever
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.docstore import InMemoryDocstore
from langchain.chat_models import ChatOpenAI
from langchain.utilities import GoogleSearchAPIWrapper
import google.generativeai as genai
import faiss
import boto3
import json

def main():
    st.title("                    PersonaPundit.ai")

    # Assuming the necessary API keys and connection
        details are stored in Streamlit's secrets
    openai_api_key = st.secrets["OPENAI_API_KEY"]
    os.environ['GOOGLE_API_KEY'] = st.secrets["
        GOOGLE_API_KEY"]
    os.environ['GOOGLE_CSE_ID'] = st.secrets["GOOGLE_CSE_ID
        "]
    snowflake_user = st.secrets["connections"]["snowflake"
        ]["user"]
    snowflake_password = st.secrets["connections"]["
        snowflake"]["password"]
    snowflake_account = st.secrets["connections"]["
        snowflake"]["account"]
    snowflake_warehouse = st.secrets["connections"]["
        snowflake"]["warehouse"]
```

```
snowflake_database = st.secrets["connections"]["
    snowflake"]["database"]
snowflake_schema = st.secrets["connections"]["snowflake
    "]["schema"]
aws_access_key_id = st.secrets["AWS_ACCESS_KEY_ID"]
aws_secret_access_key = st.secrets["
    AWS_SECRET_ACCESS_KEY"]
os.environ["GEMINI_API_KEY"] = st.secrets["
    GEMINI_API_KEY"]

search = GoogleSearchAPIWrapper()

# Fetch review data from Snowflake
def fetch_review_data(reviewer_id):
    ctx = snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    )
    cs = ctx.cursor()
    try:
        query = f"""
            SELECT REVIEWERNAME, REVIEWTEXT, SUMMARY,
                TITLE, FEATURE, DESCRIPTION, BRAND,
                PRICE
            FROM AMAZONREVIEW.TOP50_REVIEWERS_DETAILS
            WHERE REVIEWERID = '{reviewer_id}'
        """
        cs.execute(query)
        df = pd.DataFrame(cs.fetchall(), columns=[x[0]
            for x in cs.description])
        return df
    finally:
        cs.close()
        ctx.close()

def analyze_review_data(review_data):
```

```
    # Implement the logic to analyze the review data
      and generate insights.
    # This could include NLP processing, sentiment
      analysis, extracting key phrases, etc.
    # For now, let's just concatenate the data into a
      simple string.
    persona_details = f"""
    Reviewer Name: {review_data['REVIEWERNAME'].iloc
      [0]}
    Product Title: {review_data['TITLE'].iloc[0]}
    Brand: {review_data['BRAND'].iloc[0]}
    Price: {review_data['PRICE'].iloc[0]}
    Review Summary: {review_data['SUMMARY'].iloc[0]}
    Review Text: {review_data['REVIEWTEXT'].iloc[0]}
    """
    return persona_details

def generate_persona_with_gemini(reviewer_id):
    # Fetch review data for the given reviewerID
    review_data = fetch_review_data(reviewer_id)
    if review_data.empty:
        return "No data found for this Reviewer ID."

    # Prepare the prompt for the Gemini model
    prompt = (
        "Generate a detailed customer persona based on
          the following review:\n\n"
        "Reviewer Details:\n"
        f"- Name: {review_data['REVIEWERNAME'].iloc
          [0]}\n"
        f"- Product Purchased: {review_data['TITLE'].
          iloc[0]} by {review_data['BRAND'].iloc[0]}\n
          "
        f"- Price: {review_data['PRICE'].iloc[0]}\n"
        f"- Description: {review_data['DESCRIPTION'].
          iloc[0]}\n"
        "Review Insights:\n"
        f"- Summary: {review_data['SUMMARY'].iloc[0]}\n
          "
```

```
        f"- Detailed Review: {review_data['REVIEWTEXT
            '].iloc[0]}\n"
        "Persona Requirements:\n"
        "- Include demographics, psychographics,
            behavioral traits.\n"
        "- Discuss the customer's needs, goals, and
            potential pain points.\n"
        "- Suggest additional products they might be
            interested in based on the review.\n"
        "- Provide detailed insights into the
            customers likely lifestyle and buying
            behavior."
    )
    try:
        response = model.generate_content(prompt)
        generated_persona = response.text.strip()  #
            Adjust based on the actual structure of the
            response
        return generated_persona
    except Exception as e:
        return f"Error generating persona: {str(e)}"




def load_all_conversations_from_s3():
    s3 = boto3.client('s3', aws_access_key_id=
        aws_access_key_id, aws_secret_access_key=
        aws_secret_access_key)
    bucket_name = 'personapundit'
    conversations = {}
    try:
        response = s3.list_objects_v2(Bucket=
            bucket_name, Prefix='conversations/')
        for item in response.get('Contents', []):
            key = item['Key']
            obj = s3.get_object(Bucket=bucket_name, Key
                =key)
            data = obj['Body'].read().decode('utf-8')
```

```
                conversation_name = key.split('/')[-1].
                    replace('.json', '')
                conversations[conversation_name] = json.
                    loads(data)
            st.sidebar.write("Conversations loaded
                successfully.")
        except Exception as e:
            st.sidebar.error(f"Failed to load conversations
                : {str(e)}")
        return conversations

# Initialize OpenAI client
client = OpenAI(api_key=openai_api_key)
# Initialize session state for conversations and the
    current conversation
if 'conversations' not in st.session_state:
    st.session_state.conversations =
        load_all_conversations_from_s3()
if 'current_conversation' not in st.session_state:
    st.session_state['current_conversation'] = []

def detect_special_command(user_input):
    match = re.search(r"generate a group persona for
        people who love (.+)", user_input.lower())
    if match:
        topic = match.group(1)
        return "generate_group_persona", topic
    return None, None

def fetch_all_personas():
    with snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    ) as conn:
        with conn.cursor() as cur:
```

```
            query = "SELECT PERSONA FROM AMAZONREVIEW.
                PERSONAS_SUBSET"
            cur.execute(query)
            results = cur.fetchall()
            if results:
                return [result[0] for result in results
                    ]
            else:
                return []

# Configure the boto3 client.
s3 = boto3.client(
    's3',
    aws_access_key_id=aws_access_key_id,
    aws_secret_access_key=aws_secret_access_key
)

s3 = boto3.client('s3')
bucket_name = 'personapundit'

def save_conversations_to_s3(conversation_dict,
    file_name):
    """
    Save the conversation dictionary to an S3 bucket as
        a JSON file.
    """
    s3.put_object(
        Bucket=bucket_name,
        Key=f'conversations/{file_name}',
        Body=json.dumps(conversation_dict).encode('utf
            -8')
    )
    st.sidebar.success("Conversation saved to S3!")

# Configure the Gemini API
genai.configure(api_key=os.environ["GEMINI_API_KEY"])
    # Make sure the environment variable name matches
    what you set.

# Initialize the generative model
```

```
model = genai.GenerativeModel('gemini-pro')


# Radio button selection
tab = st.radio(
    "Choose a model:",
    ('GPT-3.5', 'Gemini'))
if tab == 'GPT-3.5':
    def generate_insights_from_personas(personas, topic
        ):
        combined_personas = "\n\n".join(personas)
        prompt = f"""
        Considering the personas of people interested
            in {topic}, which are outlined as follows:
        {combined_personas}

        Generate insights on common characteristics,
            preferences, and potential product
            recommendations for this group, focusing on
            their interest in {topic}.
        """
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "system", "content":
                prompt}]
        )
        if response.choices:
            return response.choices[0].message.content
        else:
            return "No insights were generated."


    # Save, Delete, and Load Conversation Functions
    def save_conversation(name):
        if name and st.session_state['
            current_conversation']:
            # Save to S3
            save_conversations_to_s3(st.session_state['
                current_conversation'], f"{name}.json")
```

```
def delete_conversation(name):
    if name in st.session_state.conversations:
        # Attempt to delete the file from S3 bucket
        try:
            s3.delete_object(Bucket=bucket_name,
                Key=f'conversations/{name}.json')
            # Remove the conversation from session
                state if successfully deleted from
                S3
            del st.session_state.conversations[name
                ]
            st.sidebar.success("Conversation
                Deleted!")
        except Exception as e:
            st.sidebar.error(f"Failed to delete
                conversation: {str(e)}")
        st.experimental_rerun()

def load_conversation(name):
    if name:
        loaded_convo = st.session_state.
            conversations[name]
        st.session_state['current_conversation'] =
            loaded_convo
        st.session_state['loaded_conversation'] =
            True  # Set flag to indicate a
            conversation is loaded

# Conversation Management UI in Sidebar
st.sidebar.header("Conversations Management:")
conversation_name = st.sidebar.text_input("
    Conversation Name:", key="conversation_name")

if st.sidebar.button("Save Conversation"):
    save_conversation(conversation_name)

selected_conversation = st.sidebar.selectbox("
    Select a Conversation", options=list(st.
    session_state['conversations'].keys()))
```

```
if st.sidebar.button("Delete Conversation"):
    delete_conversation(selected_conversation)

if st.sidebar.button("Load Conversation"):
    load_conversation(selected_conversation)

if st.sidebar.button("Start New Conversation"):
    st.session_state['current_conversation'] = []
    st.session_state['loaded_conversation'] = False
        # Reset flag when starting a new
        conversation
    st.experimental_rerun()

# Initialize FAISS and embeddings
def initialize_faiss_and_embeddings():
    embeddings_model = OpenAIEmbeddings(api_key=
        openai_api_key)
    embedding_size = 1536
    index = faiss.IndexFlatL2(embedding_size)
    vectorstore = FAISS(embeddings_model.
        embed_query, index, InMemoryDocstore({}),
        {})
    return vectorstore

vectorstore_public =
    initialize_faiss_and_embeddings()

# Initialize WebResearchRetriever
def initialize_web_research_retriever(vectorstore):
    llm = ChatOpenAI(model_name="gpt-3.5-turbo-16k"
        , api_key=openai_api_key, temperature=0,
        streaming=True)

    # Now, GoogleSearchAPIWrapper is initialized
        without explicit keys
    search = GoogleSearchAPIWrapper()

    web_retriever = WebResearchRetriever.from_llm(
        vectorstore=vectorstore,
        llm=llm,
```

```
        search=search,
        num_search_results=3
    )
    return web_retriever

web_retriever = initialize_web_research_retriever(
    vectorstore_public)

# Fetch review data from Snowflake
def fetch_review_data(reviewer_id):
    ctx = snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    )
    cs = ctx.cursor()
    try:
        query = f"""
            SELECT REVIEWERNAME, REVIEWTEXT,
                SUMMARY, TITLE, FEATURE, DESCRIPTION
                , BRAND, PRICE
            FROM AMAZONREVIEW.
                TOP50_REVIEWERS_DETAILS
            WHERE REVIEWERID = '{reviewer_id}'
        """
        cs.execute(query)
        df = pd.DataFrame(cs.fetchall(), columns=[x
            [0] for x in cs.description])
        return df
    finally:
        cs.close()
        ctx.close()

def analyze_review_data(review_data):
    # Implement the logic to analyze the review
        data and generate insights.
```

```
    # This could include NLP processing, sentiment
        analysis, extracting key phrases, etc.
    # For now, let's just concatenate the data into
        a simple string.
    persona_details = f"""
    Reviewer Name: {review_data['REVIEWERNAME'].
        iloc[0]}
    Product Title: {review_data['TITLE'].iloc[0]}
    Brand: {review_data['BRAND'].iloc[0]}
    Price: {review_data['PRICE'].iloc[0]}
    Review Summary: {review_data['SUMMARY'].iloc
        [0]}
    Review Text: {review_data['REVIEWTEXT'].iloc
        [0]}
    """
    return persona_details

def save_persona_to_db(reviewer_id, persona):
    with snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    ) as conn:
        with conn.cursor() as cur:
            cur.execute("""
                INSERT INTO AMAZONREVIEW.PERSONAS (
                    REVIEWERID, PERSONA)
                VALUES (%s, %s)
            """, (reviewer_id, persona))
            conn.commit()


def fetch_persona_from_db(reviewer_id):
    with snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
```

```
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    ) as conn:
        with conn.cursor() as cur:
            cur.execute("""
                SELECT PERSONA FROM AMAZONREVIEW.
                    PERSONAS
                WHERE REVIEWERID = %s
            """, (reviewer_id,))
            result = cur.fetchone()
            if result:
                return result[0]  # Return the
                    persona text
            else:
                return None


# Function to generate a persona from review data
    using chat.completions
def generate_persona(reviewer_id):
    # Check if a persona already exists for this
        reviewerID
    existing_persona = fetch_persona_from_db(
        reviewer_id)
    if existing_persona:
        return existing_persona  # Return the
            existing persona if found

    # Fetch review data for the given reviewerID
    review_data = fetch_review_data(reviewer_id)
    if review_data.empty:
        return "No data found for this Reviewer ID.
            "

    # Extract details from the review data to
        generate the persona
    reviewer_name = review_data["REVIEWERNAME"].
        iloc[0]
    price = review_data["PRICE"].iloc[0]
```

```
review_text = review_data["REVIEWTEXT"].iloc[0]
title = review_data["TITLE"].iloc[0]
description = review_data["DESCRIPTION"].iloc
    [0]
summary = review_data["SUMMARY"].iloc[0]
brand = review_data["BRAND"].iloc[0]

# Prepare the prompt for the language model
messages = [
    {"role": "system", "content": "You are a
        wise sage providing insights into
        customer personas based on their reviews
        and brand."},
    {"role": "user", "content": f"Based on this
        review by {reviewer_name} (print this
        on the top: Name: {reviewer_name}), who
        paid {price} for the product named {
        title} by {brand} which is described as
        '{description}'. The summary is '{
        summary}'. Here is the review text: '{
        review_text}'. Generate a customer
        persona considering demographics,
        psychographics, behavioral traits, needs
        , goals, and pain points. Dive deeper
        into review text and provide deep
        insights. Use the {title} to suggest
        what other products the user might buy.
        NOTE: Give me 5 Points for each block.
        Arrange it very well. Start with the
        name on the very top"}
]

# Generate the persona using the language model
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages
)

# Extract the generated persona from the
    response
```

```
        generated_persona = response.choices[0].message
            .content

        # Save the new persona to the database
        save_persona_to_db(reviewer_id,
            generated_persona)

        # Return the newly generated persona
        return generated_persona


# Detecting persona request from user input
def detect_special_command(user_input):
        # This tries to find a command pattern for
            generating a group persona based on a topic
            of interest
        match = re.search(r"generate a group persona
            for people who love (.+)", user_input.lower
            ())
        if match:
            topic = match.group(1)
            return "generate_group_persona", topic
        return None, None

# Add a function to process general knowledge
    questions
def handle_general_query(query):
        try:
            response = client.chat.completions.create(
                model="gpt-3.5-turbo",  # Make sure to
                    use the correct model ID
                messages=[{"role": "system", "content":
                    "I am an AI trained to provide
                    information and answer questions."},
                        {"role": "user", "content":
                            query}]
            )

            # Accessing the response correctly
            if response.choices:
```

```
            first_choice = response.choices[0]
            return first_choice.message.content  #
                Correctly access the content
                attribute
        else:
            return "No response was generated."
    except Exception as e:
        return f"An error occurred: {str(e)}"

# Update the function to detect the type of query
    and route it accordingly
def process_input(user_input):
    # First, check for individual reviewer IDs
    reviewer_id_match = re.search(r"reviewerID[:]\s
        (\S+)", user_input, re.IGNORECASE)
    if reviewer_id_match:
        reviewer_id = reviewer_id_match.group(1).
            strip()
        persona = generate_persona(reviewer_id)
        return persona, 'persona'
    else:
        # Then, check for the special command to
            generate a group persona
        command, topic = detect_special_command(
            user_input)
        if command == "generate_group_persona" and
            topic:
            personas = fetch_all_personas()
            if personas:
                insights =
                    generate_insights_from_personas(
                    personas, topic)
                return insights, 'persona_insights'
        # Fallback for other types of queries that
            do not match the above conditions
        general_response = handle_general_query(
            user_input)
        return general_response, 'general'

# Streamlit UI handling logic
```

```
st.subheader("Ask me anything:")

# Initialize 'user_input' to an empty string to
    ensure it's always defined
user_input = ""

# Check if a conversation has been loaded
if 'loaded_conversation' in st.session_state and st
    .session_state['loaded_conversation']:
    st.write("Loaded Conversation:")
    for i, (q, a) in enumerate(st.session_state['
        current_conversation'], start=1):
        st.write(f"Q{i}: {q}")
        st.write(f"A{i}: {a}")
        st.write("---")
else:
    user_input = st.text_area("Enter your request
        here:", key='user_input', value="")

# Assuming you have a function process_input that
    takes the user input, processes it, and returns
    a response and its type ('persona' or 'general')
if user_input:
    # Process the user input
    response, response_type = process_input(
        user_input)

    # Update current conversation with the new Q&A
        pair
    st.session_state['current_conversation'].append
        ((user_input, response))

    # Ensure the response is displayed based on its
        type
    if response_type == 'persona' or response_type
        == 'persona_insights':
        st.write("Generated Persona:")
    elif response_type == 'general':
        st.write("General Knowledge Answer:")
    else:
```

```
            st.write("Response:")

            st.write(response)  # This line actually prints
                the response

    pass

elif tab == 'Gemini':


    reviewer_id = st.text_input("Enter a Reviewer ID to
        generate a persona:", key='gemini_reviewer_id')

    if st.button('Generate Persona with Gemini'):
        if reviewer_id:
            generated_persona =
                generate_persona_with_gemini(reviewer_id
                )
            st.write("Generated Persona:")
            st.write(generated_persona)
        else:
            st.error("Please enter a Reviewer ID to
                generate the persona.")


def fetch_persona_from_db(reviewer_id):
        with snowflake.connector.connect(
            user=snowflake_user,
            password=snowflake_password,
            account=snowflake_account,
            warehouse=snowflake_warehouse,
            database=snowflake_database,
            schema=snowflake_schema
        ) as conn:
            with conn.cursor() as cur:
                cur.execute("""
                    SELECT PERSONA FROM AMAZONREVIEW.
                        PERSONAS
                    WHERE REVIEWERID = %s
                """, (reviewer_id,))
```

```
                result = cur.fetchone()
                if result:
                    return result[0]  # Return the
                        persona text
                else:
                    return None

def generate_persona(reviewer_id):
        # Check if a persona already exists for this
            reviewerID
        existing_persona = fetch_persona_from_db(
            reviewer_id)
        if existing_persona:
            return existing_persona  # Return the
                existing persona if found

        # Fetch review data for the given reviewerID
        review_data = fetch_review_data(reviewer_id)
        if review_data.empty:
            return "No data found for this Reviewer ID.
                "

        # Extract details from the review data to
            generate the persona
        reviewer_name = review_data["REVIEWERNAME"].
            iloc[0]
        price = review_data["PRICE"].iloc[0]
        review_text = review_data["REVIEWTEXT"].iloc[0]
        title = review_data["TITLE"].iloc[0]
        description = review_data["DESCRIPTION"].iloc
            [0]
        summary = review_data["SUMMARY"].iloc[0]
        brand = review_data["BRAND"].iloc[0]

        # Prepare the prompt for the language model
        messages = [
            {"role": "system", "content": "You are a
                wise sage providing insights into
                customer personas based on their reviews
                and brand."},
```

```
            {"role": "user", "content": f"Based on this
                review by {reviewer_name} (print this
                on the top: Name: {reviewer_name}), who
                paid {price} for the product named {
                title} by {brand} which is described as
                '{description}'. The summary is '{
                summary}'. Here is the review text: '{
                review_text}'. Generate a customer
                persona considering demographics,
                psychographics, behavioral traits, needs
                , goals, and pain points. Dive deeper
                into review text and provide deep
                insights. Use the {title} to suggest
                what other products the user might buy.
                NOTE: Give me 5 Points for each block.
                Arrange it very well. Start with the
                name on the very top"}
        ]

def process_input(user_input):
        # First, check for individual reviewer IDs
        reviewer_id_match = re.search(r"reviewerID[:]\s
            (\S+)", user_input, re.IGNORECASE)
        if reviewer_id_match:
            reviewer_id = reviewer_id_match.group(1).
                strip()
            persona = generate_persona(reviewer_id)
            return persona, 'persona'
        else:
            # Then, check for the special command to
                generate a group persona
            command, topic = detect_special_command(
                user_input)
            if command == "generate_group_persona" and
                topic:
                personas = fetch_all_personas()
                if personas:
                    insights =
                        generate_insights_from_personas(
                        personas, topic)
```

```
                return insights, 'persona_insights'
            # Fallback for other types of queries that
                do not match the above conditions
            general_response = handle_general_query(
                user_input)
            return general_response, 'general'


# Move instructions to a sidebar or a static section on
    the main page
st.sidebar.header("Instructions:")
st.sidebar.markdown("""
- To generate a user persona, please type a request
    that includes a specific reviewer ID.
- Example request: "Generate persona for reviewerID:
    A1JMSX54DO3LOP".
- Example request: "generate a group persona for people
    who love books and separated by ReviewerID's"

Note:
- The persona generation leverages both the analysis of
    review data from Snowflake and enriched insights
    through web research.
- Please ensure that the reviewer ID you provide
    matches an existing record in the Snowflake database
    for accurate persona generation.
""")

# Below this line, you could add more functionality or
    information about how the personas are generated,
# tips for interacting with your application, or any
    additional features you provide.

# Example of adding more interactivity or information
st.sidebar.header("About PersonaPundit.ai")
st.sidebar.info("""
PersonaPundit.ai uses advanced AI techniques to
    generate detailed user personas based on product
    review data.
```

```
        By analyzing reviews and supplementing this analysis
            with web research, PersonaPundit.ai provides
            insights into
        the demographics, preferences, and behavior of users,
            helping businesses understand their customers better
            .
        """)

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