Practice Project: Interactive Food Search and RAG Chatbot System

Estimated time needed: 120 minutes

Interactive Food Search and RAG Chatbot System: Introduction

In this comprehensive lab, you will build an advanced food recommendation system that demonstrates three distinct approaches to similarity search and conversational AI. Using a rich food dataset with detailed nutritional information, ingredients, cooking methods, and taste profiles, you will create an **interactive CLI search interface**, implement **advanced filtering techniques**, and develop a **RAG (Retrieval-Augmented Generation) chatbot** that provides intelligent food recommendations using Chroma DB and natural language processing.

You will learn how to implement **interactive user interfaces**, **metadata filtering**, **similarity scoring**, and **conversational AI systems** that combine vector search with large language model responses. This lab demonstrates real-world applications of vector databases in creating intelligent recommendation systems and chatbots.

Learning objectives

After completing this lab, you will be able to:

- Build **interactive CLI interfaces** for real-time food search and recommendation.
- Implement **advanced search filtering** with cuisine type, calorie restrictions, and ingredient matching.
- Create **RAG systems** that combine similarity search with natural language responses.
- Develop **conversational chatbots** that understand user queries and provide contextual food recommendations.
- Perform **similarity search** with dynamic filtering and real-time user interaction.
- Execute **CRUD operations** on vector databases with immediate search result updates.
- Understand **chatbot conversation flow** and user experience design.
- Implement **CLI-based interfaces** without external web frameworks.
- Compare **traditional search results** with **AI-enhanced responses**.

Prerequisites

- Intermediate knowledge of Python programming
- Understanding of vector embeddings and similarity search concepts
- Basic knowledge of Chroma DB vector database operations
- Familiarity with command-line interfaces and user interaction patterns

Important notice about this lab environment

Skills Network Cloud IDE is an open-source IDE (Integrated Development Environment) that provides an environment for hands-on labs in course and project-related labs.

Please be aware that sessions for this lab environment are not persistent. Every time you connect to this lab, a new environment is created for you.

You will lose data if you exit the environment without saving to GitHub or another external source.

Plan to complete these labs in a single session to avoid losing your data.

Part 1 Task 1: Setting up the Interactive Food Search Environment

Welcome to Part 1 of this comprehensive practice project. In this part, you will create three distinct food recommendation systems: an interactive CLI search interface, an advanced filtered search system, and an intelligent RAG chatbot that provides conversational food recommendations.

Part 1 Task 1: A. Install the Required Packages

1. In your IDE environment, select the **Terminal** tab at the top-right of the window shown at number 1 in the following screenshot, and then select **New Terminal** from the drop-down menu as shown at number 2 in the same screenshot.

2. Install the required Python packages for Chroma DB, embeddings, and AI models. Execute the following commands in the terminal window:

```
pip install numpy==2.3.1
pip install scipy==1.16.0
pip install chromadb==1.0.12
pip install sentence-transformers==4.1.0
pip install ibm-watsonx-ai==1.3.24
```

The packages provide:

- `chromadb`: Vector database for similarity search operations
- `sentence-transformers`: Text embedding generation for semantic search
- `numpy`: Numerical operations for similarity calculations
- `ibm-watsonx-ai`: IBM WatsonX AI SDK for RAG chatbot responses

Part 1 Task 1: B. Download the Food Dataset

3. Download the comprehensive food dataset that will serve as the foundation for all three search systems:

```
wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/sN1PIR8qp1SJ6K7syv72qQ/FoodDataSet.json
```

This dataset contains rich food information including:

- **food_id**: Unique identifier for each food item
- **food_name**: Name of the dish
- **food_description**: Detailed description of the food
- **food_calories_per_serving**: Caloric content per serving
- **food_nutritional_factors**: Breakdown of carbohydrates, protein, and fat
- **food_ingredients**: List of ingredients used
- **food_health_benefits**: Health benefits of the food
- **cooking_method**: How the food is prepared (baking, grilling, etc.)
- **cuisine_type**: Type of cuisine (American, Italian, etc.)
- **food_features**: Taste, texture, appearance, and serving details

Example food item structure:

```
{
    "food_id": 1,
    "food_name": "Apple Pie",
    "food_description": "A classic dessert made with a buttery, flaky crust filled with tender, spiced apples.",
    "food_calories_per_serving": 320,
    "food_nutritional_factors": {
        "carbohydrates": "42g",
        "protein": "2g",
        "fat": "16g"
    },
    "food_ingredients": ["Apples", "Flour", "Butter", "Sugar", "Cinnamon", "Nutmeg"],
    "food_health_benefits": "Rich in antioxidants and dietary fiber",
    "cooking_method": "Baking",
    "cuisine_type": "American",
    "food_features": {
        "taste": "sweet",
        "texture": "crisp and tender",
        "appearance": "golden brown",
        "preparation": "baked",
        "serving_type": "hot"
    }
}
```

Part 2 Task 1: Creating Shared Functions for Food Search

In this section, you will create a shared functions module that will be used by all three search systems. This promotes code reusability and maintains consistency across different interfaces.

Part 2 Task 1: A. Create the Shared Functions File

1. In your IDE environment, create a new Python file by clicking the **New File** symbol in the Explorer panel:

2. After clicking on the **New File** symbol, a pop-up box appears with the default file name **Untitled.txt** as shown in the following screenshot. Replace its default name with `shared_functions.py`.

This file will contain shared functionality used by all three of our search systems.

Part 2 Task 1: B. Import Required Libraries

3. First, add the necessary imports at the top of the `shared_functions.py` file:

```
import chromadb
from chromadb.utils import embedding_functions
import json
import re
import numpy as np
from typing import List, Dict, Any, Optional
# Initialize ChromaDB client
```

```
client = chromadb.Client()
```

These imports provide:

- `chromadb`: The main Chroma DB library for vector database operations
- `embedding_functions`: Functions to create text embeddings
- `json`: For parsing JSON data files
- `typing`: For type hints to make code more readable and maintainable

Part 2 Task 1: C. Create the Data Loading Function

4. Add the function to load and normalize food data from the JSON file:

```
def load_food_data(file_path: str) -> List[Dict]:
    """Load food data from JSON file"""
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            food_data = json.load(file)

        # Ensure each item has required fields and normalize the structure
        for i, item in enumerate(food_data):
            # Normalize food_id to string
            if 'food_id' not in item:
                item['food_id'] = str(i + 1)
            else:
                item['food_id'] = str(item['food_id'])

            # Ensure required fields exist
            if 'food_ingredients' not in item:
                item['food_ingredients'] = []
            if 'food_description' not in item:
                item['food_description'] = ''
            if 'cuisine_type' not in item:
                item['cuisine_type'] = 'Unknown'
            if 'food_calories_per_serving' not in item:
                item['food_calories_per_serving'] = 0

            # Extract taste features from nested food_features if available
            if 'food_features' in item and isinstance(item['food_features'], dict):
                taste_features = []
                for key, value in item['food_features'].items():
                    if value:
                        taste_features.append(str(value))
                item['taste_profile'] = ', '.join(taste_features)
            else:
                item['taste_profile'] = ''

        print(f"Successfully loaded {len(food_data)} food items from {file_path}")
        return food_data

    except Exception as e:
        print(f"Error loading food data: {e}")
        return []
```

This function:

- Opens and parses the JSON file containing food data
- Normalizes the data structure to ensure all required fields exist
- Extracts taste profiles from nested `food_features` objects
- Returns a list of food dictionaries ready for processing

Part 2 Task 1: D. Create the Collection Setup Function

5. Add the function to create a Chroma DB collection with proper embedding configuration:

```
def create_similarity_search_collection(collection_name: str, collection_metadata: dict = None):
    """Create ChromaDB collection with sentence transformer embeddings"""
    try:
        # Try to delete existing collection to start fresh
        client.delete_collection(collection_name)
    except:
        pass

    # Create embedding function
    sentence_transformer_ef = embedding_functions.SentenceTransformerEmbeddingFunction(
        model_name="all-MiniLM-L6-v2"
    )
```

```
    # Create new collection
    return client.create_collection(
        name=collection_name,
        metadata=collection_metadata,
        configuration={
            "hnsw": {"space": "cosine"},
            "embedding_function": sentence_transformer_ef
        }
    )
```

This function:

- Deletes any existing collection with the same name to start fresh
- Creates a SentenceTransformer embedding function using the "all-MiniLM-L6-v2" model
- Returns a new Chroma DB collection configured for similarity search

Part 2 Task 1: E. Create the Data Population Function

6. Add the function to populate the collection with food data and generate embeddings:

```
def populate_similarity_collection(collection, food_items: List[Dict]):
    """Populate collection with food data and generate embeddings"""
    documents = []
    metadatas = []
    ids = []

    # Create unique IDs to avoid duplicates
    used_ids = set()

    for i, food in enumerate(food_items):
        # Create comprehensive text for embedding using rich JSON structure
        text = f"Name: {food['food_name']}. "
        text += f"Description: {food.get('food_description', '')}. "
        text += f"Ingredients: {', '.join(food.get('food_ingredients', []))}. "
        text += f"Cuisine: {food.get('cuisine_type', 'Unknown')}. "
        text += f"Cooking method: {food.get('cooking_method', '')}. "

        # Add taste profile from food_features
        taste_profile = food.get('taste_profile', '')
        if taste_profile:
            text += f"Taste and features: {taste_profile}. "

        # Add health benefits if available
        health_benefits = food.get('food_health_benefits', '')
        if health_benefits:
            text += f"Health benefits: {health_benefits}. "

        # Add nutritional information
        if 'food_nutritional_factors' in food:
            nutrition = food['food_nutritional_factors']
            if isinstance(nutrition, dict):
                nutrition_text = ', '.join([f"{k}: {v}" for k, v in nutrition.items()])
                text += f"Nutrition: {nutrition_text}."

        # Generate unique ID to avoid duplicates
        base_id = str(food.get('food_id', i))
        unique_id = base_id
        counter = 1
        while unique_id in used_ids:
            unique_id = f"{base_id}_{counter}"
            counter += 1
        used_ids.add(unique_id)

        documents.append(text)
        ids.append(unique_id)
        metadatas.append({
            "name": food["food_name"],
            "cuisine_type": food.get("cuisine_type", "Unknown"),
            "ingredients": ", ".join(food.get("food_ingredients", [])),
            "calories": food.get("food_calories_per_serving", 0),
            "description": food.get("food_description", ""),
            "cooking_method": food.get("cooking_method", ""),
            "health_benefits": food.get("food_health_benefits", ""),
            "taste_profile": food.get("taste_profile", "")
        })

    # Add all data to collection
    collection.add(
        documents=documents,
        metadatas=metadatas,
        ids=ids
    )

    print(f"Added {len(food_items)} food items to collection")
```

This function:

- Creates comprehensive text documents by combining multiple food attributes
- Includes name, description, ingredients, cuisine, cooking method, taste features, health benefits, and nutrition
- Stores rich metadata for each food item to enable advanced filtering
- Adds all data to the Chroma DB collection for similarity search

Part 2 Task 1: F. Create the Basic Similarity Search Function

7. Add the function to perform basic similarity search:

```
def perform_similarity_search(collection, query: str, n_results: int = 5) -> List[Dict]:
    """Perform similarity search and return formatted results"""
    try:
        results = collection.query(
            query_texts=[query],
            n_results=n_results
        )

        if not results or not results['ids'] or len(results['ids'][0]) == 0:
            return []

        formatted_results = []
        for i in range(len(results['ids'][0])):
            # Calculate similarity score (1 - distance)
            similarity_score = 1 - results['distances'][0][i]

            result = {
                'food_id': results['ids'][0][i],
                'food_name': results['metadatas'][0][i]['name'],
                'food_description': results['metadatas'][0][i]['description'],
                'cuisine_type': results['metadatas'][0][i]['cuisine_type'],
                'food_calories_per_serving': results['metadatas'][0][i]['calories'],
                'similarity_score': similarity_score,
                'distance': results['distances'][0][i]
            }
            formatted_results.append(result)

        return formatted_results

    except Exception as e:
        print(f"Error in similarity search: {e}")
        return []
```

This function:

- Performs a similarity search using the query text and Chroma DB's `query` method
- Converts distance scores to similarity scores (1 - distance) for better interpretation
- Formats results into a standardized dictionary structure
- Returns an empty list if no results are found or an error occurs

Part 2 Task 1: G. Create the Filtered Similarity Search Function

8. Add the function to perform similarity search with metadata filters:

```
def perform_filtered_similarity_search(collection, query: str, cuisine_filter: str = None,
                                       max_calories: int = None, n_results: int = 5) -> List[Dict]:
    """Perform filtered similarity search with metadata constraints"""
    where_clause = None

    # Build filters list
    filters = []
    if cuisine_filter:
        filters.append({"cuisine_type": cuisine_filter})

    if max_calories:
        filters.append({"calories": {"$lte": max_calories}})

    # Construct where clause based on number of filters
    if len(filters) == 1:
        where_clause = filters[0]
    elif len(filters) > 1:
        where_clause = {"$and": filters}

    try:
        results = collection.query(
            query_texts=[query],
            n_results=n_results,
            where=where_clause
        )

        if not results or not results['ids'] or len(results['ids'][0]) == 0:
            return []
```

```
            formatted_results = []
            for i in range(len(results['ids'][0])):
                similarity_score = 1 - results['distances'][0][i]

                result = {
                    'food_id': results['ids'][0][i],
                    'food_name': results['metadatas'][0][i]['name'],
                    'food_description': results['metadatas'][0][i]['description'],
                    'cuisine_type': results['metadatas'][0][i]['cuisine_type'],
                    'food_calories_per_serving': results['metadatas'][0][i]['calories'],
                    'similarity_score': similarity_score,
                    'distance': results['distances'][0][i]
                }
                formatted_results.append(result)

            return formatted_results

        except Exception as e:
            print(f"Error in filtered search: {e}")
            return []
```

This function:

- Builds individual filter conditions for cuisine type and calorie limits
- Uses Chroma DB's `$and` operator to properly combine multiple filters when needed
- Supports filtering by cuisine type (exact match) and maximum calories (less than or equal)
- Handles single filters directly and multiple filters using the `$and` operator for proper Chroma DB syntax
- Returns formatted results that match both the similarity query and all applied metadata filters

Click on the button below to see the complete `shared_functions.py` script. You can copy-paste the solution into your `shared_functions.py` file.

▶ Click for the complete shared_functions.py script

Part 2 Task 2: Building the Interactive CLI Search System

Now, you will create the first search system: an interactive command-line interface that allows users to search for food items in real-time with immediate feedback.

Part 2 Task 2: A. Create the Interactive Search File

1. Create a new Python file in your IDE by clicking **New File** in the Explorer panel.
2. Name the file `interactive_search.py`. This will contain our interactive CLI search system.

Part 2 Task 2: B. Import Shared Functions and Setup

3. First, import the shared functions module and set up global variables:

```
from shared_functions import *
# Global variable to store loaded food items
food_items = []
```

This imports all the functions we created in the shared_functions module and creates a global variable to store the food data once loaded.

Part 2 Task 2: C. Create the Main Function

4. Add the main function that initializes the interactive search system:

```
def main():
    """Main function for interactive CLI food recommendation system"""
    try:
        print("🍽️  Interactive Food Recommendation System")
        print("=" * 50)
        print("Loading food database...")

        # Load food data from file
        global food_items
        food_items = load_food_data('./FoodDataSet.json')
        print(f"✅ Loaded {len(food_items)} food items successfully")

        # Create and populate search collection
        collection = create_similarity_search_collection(
            "interactive_food_search",
```

```
            {'description': 'A collection for interactive food search'}
        )
        populate_similarity_collection(collection, food_items)

        # Start interactive chatbot
        interactive_food_chatbot(collection)

    except Exception as error:
        print(f"❌ Error initializing system: {error}")
```

This function:

- Displays a welcome message with visual formatting
- Loads the food data from the JSON file using our shared function
- Creates a Chroma DB collection specifically for interactive search
- Populates the collection with food data and embeddings
- Starts the interactive chatbot interface

Part 2 Task 2: D. Create the Interactive Chatbot Function

5. Add the main interactive loop that handles user input:

```
def interactive_food_chatbot(collection):
    """Interactive CLI chatbot for food recommendations"""
    print("\n" + "="*50)
    print("🤖 INTERACTIVE FOOD SEARCH CHATBOT")
    print("="*50)
    print("Commands:")
    print("  • Type any food name or description to search")
    print("  • 'help' - Show available commands")
    print("  • 'quit' or 'exit' - Exit the system")
    print("  • Ctrl+C - Emergency exit")
    print("-" * 50)

    while True:
        try:
            # Get user input
            user_input = input("\n🔍 Search for food: ").strip()

            # Handle empty input
            if not user_input:
                print("   Please enter a search term or 'help' for commands")
                continue

            # Handle exit commands
            if user_input.lower() in ['quit', 'exit', 'q']:
                print("\n👋 Thank you for using the Food Recommendation System!")
                print("   Goodbye!")
                break

            # Handle help command
            elif user_input.lower() in ['help', 'h']:
                show_help_menu()

            # Handle food search
            else:
                handle_food_search(collection, user_input)

        except KeyboardInterrupt:
            print("\n\n👋 System interrupted. Goodbye!")
            break
        except Exception as e:
            print(f"❌ Error processing request: {e}")
```

This function:

- Creates an interactive command-line interface with clear instructions
- Uses an infinite loop to continuously accept user input
- Handles different types of input: search queries, help requests, and exit commands
- Provides graceful error handling for interruptions and exceptions
- Calls appropriate handler functions based on user input

Part 2 Task 2: E. Create the Help Menu Function

6. Add a function to display help information:

```
def show_help_menu():
    """Display help information for users"""
    print("\n📖 HELP MENU")
    print("-" * 30)
    print("Search Examples:")
    print("  • 'chocolate dessert' - Find chocolate desserts")
    print("  • 'Italian food' - Find Italian cuisine")
    print("  • 'sweet treats' - Find sweet desserts")
    print("  • 'baked goods' - Find baked items")
    print("  • 'low calorie' - Find lower-calorie options")
    print("\nCommands:")
    print("  • 'help' - Show this help menu")
    print("  • 'quit' - Exit the system")
```

This function provides examples of search terms that work well with our food dataset and explains the available commands.

Part 2 Task 2: F. Create the Food Search Handler

7. Add the function that processes food searches and displays results:

```
def handle_food_search(collection, query):
    """Handle food similarity search with enhanced display"""
    print(f"\n🔍 Searching for '{query}'...")
    print("   Please wait...")

    # Perform similarity search
    results = perform_similarity_search(collection, query, 5)

    if not results:
        print("❌ No matching foods found.")
        print("💡 Try different keywords like:")
        print("   • Cuisine types: 'Italian', 'American'")
        print("   • Ingredients: 'chocolate', 'flour', 'cheese'")
        print("   • Descriptors: 'sweet', 'baked', 'dessert'")
        return

    # Display results with rich formatting
    print(f"\n✅ Found {len(results)} recommendations:")
    print("=" * 60)

    for i, result in enumerate(results, 1):
        # Calculate percentage score
        percentage_score = result['similarity_score'] * 100

        print(f"\n{i}. 🍽️  {result['food_name']}")
        print(f"   📊 Match Score: {percentage_score:.1f}%")
        print(f"   🥘  Cuisine: {result['cuisine_type']}")
        print(f"   🔥 Calories: {result['food_calories_per_serving']} per serving")
        print(f"   📝 Description: {result['food_description']}")

        # Add visual separator
        if i < len(results):
            print("   " + "-" * 50)

    print("=" * 60)

    # Provide suggestions for further exploration
    suggest_related_searches(results)
```

This function:

- Uses our shared `perform_similarity_search` function to find matching foods
- Provides helpful suggestions if no results are found
- Displays results with rich formatting including emojis and clear sections
- Shows similarity scores as percentages for easier understanding
- Calls a helper function to suggest related searches

Part 2 Task 2: G. Create the Related Search Suggestions Function

8. Add a function that suggests related searches based on current results:

```
def suggest_related_searches(results):
    """Suggest related searches based on current results"""
    if not results:
        return

    # Extract cuisine types from results
```

```
    cuisines = list(set([r['cuisine_type'] for r in results]))

    print("\n💡 Related searches you might like:")
    for cuisine in cuisines[:3]:  # Limit to 3 suggestions
        print(f"   • Try '{cuisine} dishes' for more {cuisine} options")

    # Suggest calorie-based searches
    avg_calories = sum([r['food_calories_per_serving'] for r in results]) / len(results)
    if avg_calories > 350:
        print("   • Try 'low calorie' for lighter options")
    else:
        print("   • Try 'hearty meal' for more substantial dishes")
```

This function analyzes the search results to provide intelligent suggestions for related searches.

Part 2 Task 2: H. Add the Entry Point

9. Finally, add the entry point that runs the main function:

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

This ensures the interactive search system starts when the script is run directly.

Click on the button below to see the complete `interactive_search.py` script. You can copy-paste the solution into your `interactive_search.py` file.

▶ Click for the complete interactive_search.py script

Part 2 Task 2: I. Test the Interactive Search System

10. Run the interactive search system to test its functionality:

```
python3.11 interactive_search.py
```

- **Type `quit` to exit the program.**

11. Try different search queries when prompted:

- "chocolate dessert"
- "Italian food"
- "sweet treats"
- "baked goods"

12. Test the help command and exit functionality.

Part 3 Task 1: Building the Advanced Search System

Now, you will create the second search system that demonstrates advanced filtering capabilities with cuisine types, calorie restrictions, and interactive filtering options. This system will show you how to combine similarity search with metadata filtering to provide more targeted results.

Part 3 Task 1: A. Create the Advanced Search File

1. Create a new Python file in your IDE by clicking **New File** in the Explorer panel.

2. Name the file `advanced_search.py`. This will contain our advanced search system with filtering capabilities.

Part 3 Task 1: B. Import Dependencies and Setup

3. First, import the shared functions module:

```
from shared_functions import *
```

This imports all the functions we created in the shared_functions module, including the enhanced `perform_filtered_similarity_search` function.

Part 3 Task 1: C. Create the Main Function

4. Add the main function that initializes our advanced search system:

```
def main():
    """Main function for advanced search demonstrations"""
    try:
        print("🔬 Advanced Food Search System")
        print("=" * 50)
        print("Loading food database with advanced filtering capabilities...")

        # Load food data from JSON file
        food_items = load_food_data('./FoodDataSet.json')
        print(f"✅ Loaded {len(food_items)} food items successfully")

        # Create collection specifically for advanced search operations
        collection = create_similarity_search_collection(
            "advanced_food_search",
            {'description': 'A collection for advanced search demos'}
        )
        populate_similarity_collection(collection, food_items)

        # Start the interactive advanced search interface
        interactive_advanced_search(collection)

    except Exception as error:
        print(f"❌ Error initializing advanced search system: {error}")
```

This function:

- Loads the food data using our shared function
- Creates a dedicated Chroma DB collection for advanced search operations
- Populates the collection with food data and rich metadata for filtering
- Starts the interactive advanced search interface

Part 3 Task 1: D. Create the Interactive Menu System

5. Add the main menu function that provides various search options:

```
def interactive_advanced_search(collection):
    """Interactive advanced search with filtering options"""
    print("\n" + "="*50)
    print("🔪 ADVANCED SEARCH WITH FILTERS")
    print("="*50)
    print("Search Options:")
    print("  1. Basic similarity search")
    print("  2. Cuisine-filtered search")
    print("  3. Calorie-filtered search")
    print("  4. Combined filters search")
    print("  5. Demonstration mode")
    print("  6. Help")
    print("  7. Exit")
    print("-" * 50)

    while True:
        try:
            choice = input("\n📋 Select option (1-7): ").strip()

            if choice == '1':
                perform_basic_search(collection)
            elif choice == '2':
                perform_cuisine_filtered_search(collection)
            elif choice == '3':
                perform_calorie_filtered_search(collection)
            elif choice == '4':
                perform_combined_filtered_search(collection)
            elif choice == '5':
                run_search_demonstrations(collection)
            elif choice == '6':
                show_advanced_help()
            elif choice == '7':
```

```
                print("👋 Exiting Advanced Search System. Goodbye!")
                break
            else:
                print("❌ Invalid option. Please select 1-7.")

        except KeyboardInterrupt:
            print("\n\n👋 System interrupted. Goodbye!")
            break
        except Exception as e:
            print(f"❌ Error: {e}")
```

This function:

- Provides a clear menu interface with 7 different search options
- Uses a continuous loop to handle user selections
- Calls specific search functions based on user choice
- Handles errors and keyboard interrupts gracefully

Part 3 Task 1: E. Create the Basic Search Function

6. Add the function for performing basic similarity search:

```
def perform_basic_search(collection):
    """Perform basic similarity search without filters"""
    print("\n🔍 BASIC SIMILARITY SEARCH")
    print("-" * 30)

    query = input("Enter search query: ").strip()
    if not query:
        print("❌ Please enter a search term")
        return

    print(f"\n🔍 Searching for '{query}'...")
    results = perform_similarity_search(collection, query, 5)

    display_search_results(results, "Basic Search Results")
```

This function:

- Prompts the user for a search query
- Validates that the query is not empty
- Uses our shared `perform_similarity_search` function to find matches
- Displays the results using a formatted display function

Part 3 Task 1: F. Create the Cuisine-Filtered Search Function

7. Add the function for searching with cuisine filters:

```
def perform_cuisine_filtered_search(collection):
    """Perform cuisine-filtered similarity search"""
    print("\n🍽️ CUISINE-FILTERED SEARCH")
    print("-" * 30)

    # Show available cuisines from our dataset
    cuisines = ["Italian", "Thai", "Mexican", "Indian", "Japanese", "French",
                "Mediterranean", "American", "Health Food", "Dessert"]
    print("Available cuisines:")
    for i, cuisine in enumerate(cuisines, 1):
        print(f"  {i}. {cuisine}")

    query = input("\nEnter search query: ").strip()
    cuisine_choice = input("Enter cuisine number (or cuisine name): ").strip()

    if not query:
        print("❌ Please enter a search term")
        return

    # Handle cuisine selection - accept both number and text input
    cuisine_filter = None
    if cuisine_choice.isdigit():
        idx = int(cuisine_choice) - 1
        if 0 <= idx < len(cuisines):
            cuisine_filter = cuisines[idx]
    else:
        cuisine_filter = cuisine_choice
```

```
    if not cuisine_filter:
        print("❌ Invalid cuisine selection")
        return

    print(f"\n🔍 Searching for '{query}' in {cuisine_filter} cuisine...")
    results = perform_filtered_similarity_search(
        collection, query, cuisine_filter=cuisine_filter, n_results=5
    )

    display_search_results(results, f"Cuisine-Filtered Results ({cuisine_filter})")
```

This function:

- Displays a list of available cuisines from our dataset
- Accepts both numeric choices (1-10) and direct cuisine names
- Uses our shared `perform_filtered_similarity_search` function with cuisine filtering
- Validates both the search query and cuisine selection
- Displays results with cuisine-specific context

Part 3 Task 1: G. Create the Calorie-Filtered Search Function

8. Add the function for searching with calorie restrictions:

```
def perform_calorie_filtered_search(collection):
    """Perform calorie-filtered similarity search"""
    print("\n🔥 CALORIE-FILTERED SEARCH")
    print("-" * 30)

    query = input("Enter search query: ").strip()
    max_calories_input = input("Enter maximum calories (or press Enter for no limit): ").strip()

    if not query:
        print("❌ Please enter a search term")
        return

    max_calories = None
    if max_calories_input.isdigit():
        max_calories = int(max_calories_input)

    print(f"\n🔍 Searching for '{query}'" +
          (f" with max {max_calories} calories..." if max_calories else "..."))

    results = perform_filtered_similarity_search(
        collection, query, max_calories=max_calories, n_results=5
    )

    calorie_text = f"under {max_calories} calories" if max_calories else "any calories"
    display_search_results(results, f"Calorie-Filtered Results ({calorie_text})")
```

This function:

- Prompts for both search query and calorie limit
- Allows optional calorie filtering (user can press Enter to skip)
- Validates the calorie input as a numeric value
- Uses our shared `perform_filtered_similarity_search` function with calorie filtering
- Provides clear feedback about the applied filters

Part 3 Task 1: H. Create the Combined Filters Search Function

9. Add the function for searching with multiple filters:

```
def perform_combined_filtered_search(collection):
    """Perform search with multiple filters combined"""
    print("\n🎯 COMBINED FILTERS SEARCH")
    print("-" * 30)

    query = input("Enter search query: ").strip()
    cuisine = input("Enter cuisine type (optional): ").strip()
    max_calories_input = input("Enter maximum calories (optional): ").strip()

    if not query:
        print("❌ Please enter a search term")
        return

    cuisine_filter = cuisine if cuisine else None
    max_calories = int(max_calories_input) if max_calories_input.isdigit() else None
```

```
    # Build description of applied filters
    filter_description = []
    if cuisine_filter:
        filter_description.append(f"cuisine: {cuisine_filter}")
    if max_calories:
        filter_description.append(f"max calories: {max_calories}")

    filter_text = ", ".join(filter_description) if filter_description else "no filters"

    print(f"\n🔍 Searching for '{query}' with {filter_text}...")

    results = perform_filtered_similarity_search(
        collection, query,
        cuisine_filter=cuisine_filter,
        max_calories=max_calories,
        n_results=5
    )

    display_search_results(results, f"Combined Filtered Results ({filter_text})")
```

This function:

- Allows users to specify multiple filters simultaneously
- Handles optional inputs gracefully (users can skip any filter)
- Builds a descriptive text showing which filters are applied
- Demonstrates the power of combining multiple metadata filters
- Shows how filtering can narrow down search results for specific needs

Part 3 Task 1: I. Create the Demonstration Mode Function

10. Add the function that runs predefined search demonstrations:

```
def run_search_demonstrations(collection):
    """Run predetermined demonstrations of different search types"""
    print("\n📊 SEARCH DEMONSTRATIONS")
    print("=" * 40)

    demonstrations = [
        {
            "title": "Italian Cuisine Search",
            "query": "creamy pasta",
            "cuisine_filter": "Italian",
            "max_calories": None
        },
        {
            "title": "Low-Calorie Healthy Options",
            "query": "healthy meal",
            "cuisine_filter": None,
            "max_calories": 300
        },
        {
            "title": "Asian Light Dishes",
            "query": "light fresh meal",
            "cuisine_filter": "Japanese",
            "max_calories": 250
        }
    ]

    for i, demo in enumerate(demonstrations, 1):
        print(f"\n{i}. {demo['title']}")
        print(f"   Query: '{demo['query']}'")

        filters = []
        if demo['cuisine_filter']:
            filters.append(f"Cuisine: {demo['cuisine_filter']}")
        if demo['max_calories']:
            filters.append(f"Max Calories: {demo['max_calories']}")

        if filters:
            print(f"   Filters: {', '.join(filters)}")

        results = perform_filtered_similarity_search(
            collection,
            demo['query'],
            cuisine_filter=demo['cuisine_filter'],
            max_calories=demo['max_calories'],
            n_results=3
        )

        display_search_results(results, demo['title'], show_details=False)

        input("\n⏸️  Press Enter to continue to next demonstration...")
```

This function:

- Provides three predefined search demonstrations showcasing different filtering capabilities
- Shows examples of Italian cuisine search, low-calorie filtering, and combined filters
- Displays the search parameters for each demonstration clearly
- Pauses between demonstrations for better user experience

Part 3 Task 1: J. Create the Results Display Function

11. Add the function for displaying search results in a formatted way:

```
def display_search_results(results, title, show_details=True):
    """Display search results in a formatted way"""
    print(f"\n📋 {title}")
    print("=" * 50)

    if not results:
        print("❌ No matching results found")
        print("💡 Try adjusting your search terms or filters")
        return

    for i, result in enumerate(results, 1):
        score_percentage = result['similarity_score'] * 100

        if show_details:
            print(f"\n{i}. 🍽️  {result['food_name']}")
            print(f"   📊 Similarity Score: {score_percentage:.1f}%")
            print(f"   🥜  Cuisine: {result['cuisine_type']}")
            print(f"   🔥 Calories: {result['food_calories_per_serving']}")
            print(f"   📝 Description: {result['food_description']}")
        else:
            print(f"   {i}. {result['food_name']} ({score_percentage:.1f}% match)")

    print("=" * 50)
```

This function:

- Provides consistent formatting for all search result displays
- Supports both detailed and summary view modes
- Converts similarity scores to percentages for easier understanding
- Handles empty results gracefully with helpful suggestions

Part 3 Task 1: K. Create the Help Function

12. Add the function that displays help information:

```
def show_advanced_help():
    """Display help information for advanced search"""
    print("\n📖 ADVANCED SEARCH HELP")
    print("=" * 40)
    print("Search Types:")
    print("  1. Basic Search - Standard similarity search")
    print("  2. Cuisine Filter - Search within specific cuisine types")
    print("  3. Calorie Filter - Search for foods under calorie limits")
    print("  4. Combined Filters - Use multiple filters together")
    print("  5. Demonstrations - See predefined search examples")
    print("\nTips:")
    print("  • Use descriptive terms: 'creamy', 'spicy', 'light'")
    print("  • Combine ingredients: 'chicken vegetables'")
    print("  • Try cuisine names: 'Italian', 'Thai', 'Mexican'")
    print("  • Filter by calories for dietary goals")
```

This function provides comprehensive help information about all available search options and useful tips for effective searching.

Part 3 Task 1: L. Add the Entry Point

13. Finally, add the entry point that runs the main function:

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

This ensures the advanced search system starts when the script is run directly.

Click on the button below to see the complete `advanced_search.py` script. You can copy-paste the solution into your `advanced_search.py` file.

▶ Click for the complete advanced_search.py script

Part 3 Task 1: M. Test the Advanced Search System

14. Run the advanced search system:

```
python3.11 advanced_search.py
```

15. Test different search options:

- Option 1: Basic search with "chocolate dessert"
- Option 2: Cuisine search for "sweet" in "American" cuisine
- Option 3: Calorie search for "dessert" under 300 calories
- Option 4: Combined Filters - Use multiple filters together
- Option 5: Run the demonstration mode
- Option 6: Show the help menu
- Option 7: Quit – Exit the program

Part 4 Task 1: Building the RAG Chatbot System

In this enhanced version, you will create a true RAG chatbot that combines ChromaDB similarity search with IBM's Granite language model to provide intelligent, contextual responses about food recommendations.

Understanding RAG Architecture

Before implementing the code, let's understand what makes this a true RAG system:

1. **Retrieval Phase**: Search the vector database for relevant food items based on user query
2. **Context Building**: Extract and structure relevant information from search results
3. **Augmented Generation**: Pass both the user query and retrieved context to an LLM
4. **Response Generation**: The LLM generates a natural, contextual response using the retrieved information

Part 4 Task 1: A. Import Dependencies and Setup LLM Connection

First, create a file called `enhanced_rag_chatbot.py`. Then, import the necessary libraries and establish a connection to the IBM watsonx.ai service, which will give us access to IBM's Granite LLM:

```
from shared_functions import *
from typing import List, Dict, Any
from ibm_watsonx_ai.foundation_models.utils.enums import ModelTypes
from ibm_watsonx_ai.foundation_models import ModelInference
import json
# Global variables
food_items = []
# IBM Watsonx.ai Configuration
my_credentials = {
    "url": "https://us-south.ml.cloud.ibm.com"
}
model_id = 'ibm/granite-3-3-8b-instruct'
gen_parms = {"max_new_tokens": 400}
project_id = "skills-network"  # <--- NOTE: specify "skills-network" as your project_id
space_id = None
verify = False
# Initialize the LLM model
model = ModelInference(
    model_id=model_id,
    credentials=my_credentials,
    params=gen_parms,
    project_id=project_id,
    space_id=space_id,
    verify=verify,
)
```

Explanation: This section connects to IBM watsonx.ai to access the Granite large language model, which will be used to generate intelligent, context-aware responses. The model processes both user queries and relevant food data to provide personalized recommendations.

Part 4 Task 1: B. Create the Main Function and Collection Setup

```
def main():
    """Main function for enhanced RAG chatbot system"""
    try:
        print("🍽️ Enhanced RAG-Powered Food Recommendation Chatbot")
        print("   Powered by IBM Granite & ChromaDB")
        print("=" * 55)

        # Load food data
        global food_items
        food_items = load_food_data('./FoodDataSet.json')
        print(f"✅ Loaded {len(food_items)} food items")

        # Create collection for RAG system
        collection = create_similarity_search_collection(
            "enhanced_rag_food_chatbot",
            {'description': 'Enhanced RAG chatbot with IBM watsonx.ai integration'}
        )
        populate_similarity_collection(collection, food_items)
        print("✅ Vector database ready")

        # Test LLM connection
        print("🔗 Testing LLM connection...")
        test_response = model.generate(prompt="Hello", params=None)
        if test_response and "results" in test_response:
            print("✅ LLM connection established")
        else:
            print("❌ LLM connection failed")
            return

        # Start enhanced RAG chatbot
        enhanced_rag_food_chatbot(collection)

    except Exception as error:
        print(f"❌ Error: {error}")
```

Explanation: The main function sets up the entire RAG system by loading data, creating the vector database collection, testing the LLM connection, and launching the chatbot interface. This ensures all components are working before starting the conversation.

Part 4 Task 1: C. Build the Context Preparation Function

```
def prepare_context_for_llm(query: str, search_results: List[Dict]) -> str:
    """Prepare structured context from search results for LLM"""
    if not search_results:
        return "No relevant food items found in the database."

    context_parts = []
    context_parts.append("Based on your query, here are the most relevant food options from our database:")
    context_parts.append("")

    for i, result in enumerate(search_results[:3], 1):
        food_context = []
        food_context.append(f"Option {i}: {result['food_name']}")
        food_context.append(f"  - Description: {result['food_description']}")
        food_context.append(f"  - Cuisine: {result['cuisine_type']}")
        food_context.append(f"  - Calories: {result['food_calories_per_serving']} per serving")

        if result.get('food_ingredients'):
            ingredients = result['food_ingredients']
            if isinstance(ingredients, list):
                food_context.append(f"  - Key ingredients: {', '.join(ingredients[:5])}")
            else:
                food_context.append(f"  - Key ingredients: {ingredients}")

        if result.get('food_health_benefits'):
            food_context.append(f"  - Health benefits: {result['food_health_benefits']}")

        if result.get('cooking_method'):
            food_context.append(f"  - Cooking method: {result['cooking_method']}")

        if result.get('taste_profile'):
            food_context.append(f"  - Taste profile: {result['taste_profile']}")

        food_context.append(f"  - Similarity score: {result['similarity_score']*100:.1f}%")
        food_context.append("")
```

```
        context_parts.extend(food_context)

    return "\n".join(context_parts)
```

Explanation: This function transforms the raw search results into a structured, readable format that the LLM can understand and use effectively. It extracts key information like descriptions, nutritional data, ingredients, and health benefits, organizing them in a way that helps the LLM generate more informed responses.

Part 4 Task 1: D. Create the LLM Response Generation Function

```
def generate_llm_rag_response(query: str, search_results: List[Dict]) -> str:
    """Generate response using IBM Granite with retrieved context"""
    try:
        # Prepare context from search results
        context = prepare_context_for_llm(query, search_results)

        # Build the prompt for the LLM
        prompt = f'''You are a helpful food recommendation assistant. A user is asking for food recommendations, and I've retrieved relevant opt
User Query: "{query}"
Retrieved Food Information:
{context}
Please provide a helpful, short response that:
1. Acknowledges the user's request
2. Recommends 2-3 specific food items from the retrieved options
3. Explains why these recommendations match their request
4. Includes relevant details like cuisine type, calories, or health benefits
5. Uses a friendly, conversational tone
6. Keeps the response concise but informative
Response:'''
        # Generate response using IBM Granite
        generated_response = model.generate(prompt=prompt, params=None)

        # Extract the generated text
        if generated_response and "results" in generated_response:
            response_text = generated_response["results"][0]["generated_text"]

            # Clean up the response if needed
            response_text = response_text.strip()

            # If response is too short, provide a fallback
            if len(response_text) < 50:
                return generate_fallback_response(query, search_results)

            return response_text
        else:
            return generate_fallback_response(query, search_results)

    except Exception as e:
        print(f"❌ LLM Error: {e}")
        return generate_fallback_response(query, search_results)
```

Explanation: This function implements the core RAG workflow by combining information retrieval with language generation. It constructs an augmented prompt that includes both the user's original prompt and the retrieved contextual data. This prompt is then sent to IBM watsonx.ai, which provides access to the IBM Granite LLM. IBM Granite uses this combined input to generate intelligent, context-aware responses that are natural and helpful.

Part 4 Task 1: E. Create Fallback Response Function

```
def generate_fallback_response(query: str, search_results: List[Dict]) -> str:
    """Generate fallback response when LLM fails"""
    if not search_results:
        return "I couldn't find any food items matching your request. Try describing what you're in the mood for with different words!"

    top_result = search_results[0]
    response_parts = []

    response_parts.append(f"Based on your request for '{query}', I'd recommend {top_result['food_name']}.")
    response_parts.append(f"It's a {top_result['cuisine_type']} dish with {top_result['food_calories_per_serving']} calories per serving.")

    if len(search_results) > 1:
        second_choice = search_results[1]
        response_parts.append(f"Another great option would be {second_choice['food_name']}.")

    return " ".join(response_parts)
```

Explanation: This fallback function ensures the chatbot always provides a response, even if the LLM connection fails. It uses the search results to generate a basic but helpful recommendation, maintaining system reliability.

Part 4 Task 1: F. Build the Main Chatbot Interface

```
def enhanced_rag_food_chatbot(collection):
    """Enhanced RAG-powered conversational food chatbot with IBM Granite"""
    print("\n" + "="*70)
    print("🤖 ENHANCED RAG FOOD RECOMMENDATION CHATBOT")
    print("   Powered by IBM's Granite Model")
    print("="*70)
    print("💬 Ask me about food recommendations using natural language!")
    print("\nExample queries:")
    print("  • 'I want something spicy and healthy for dinner'")
    print("  • 'What Italian dishes do you recommend under 400 calories?'")
    print("  • 'I'm craving comfort food for a cold evening'")
    print("  • 'Suggest some protein-rich breakfast options'")
    print("\nCommands:")
    print("  • 'help' - Show detailed help menu")
    print("  • 'compare' - Compare recommendations for two different queries")
    print("  • 'quit' - Exit the chatbot")
    print("-" * 70)

    conversation_history = []

    while True:
        try:
            user_input = input("\n👤 You: ").strip()

            if not user_input:
                print("🤖 Bot: Please tell me what kind of food you're looking for!")
                continue

            if user_input.lower() in ['quit', 'exit', 'q']:
                print("\n🤖 Bot: Thank you for using the Enhanced RAG Food Chatbot!")
                print("      Hope you found some delicious recommendations! 👋")
                break

            elif user_input.lower() in ['help', 'h']:
                show_enhanced_rag_help()

            elif user_input.lower() in ['compare']:
                handle_enhanced_comparison_mode(collection)

            else:
                # Process the food query with enhanced RAG
                handle_enhanced_rag_query(collection, user_input, conversation_history)
                conversation_history.append(user_input)

                # Keep conversation history manageable
                if len(conversation_history) > 5:
                    conversation_history = conversation_history[-3:]

        except KeyboardInterrupt:
            print("\n\n🤖 Bot: Goodbye! Hope you find something delicious! 👋")
            break
        except Exception as e:
            print(f"❌ Bot: Sorry, I encountered an error: {e}")
```

Explanation: The main chatbot interface provides a conversational experience where users can ask natural language questions about food. It maintains conversation history for context and handles various commands while focusing on the core RAG functionality.

Part 4 Task 1: G. Create the Enhanced Query Handler

```
def handle_enhanced_rag_query(collection, query: str, conversation_history: List[str]):
    """Handle user query with enhanced RAG approach using IBM Granite"""
    print(f"\n🔍 Searching vector database for: '{query}'...")

    # Perform similarity search with more results for better context
    search_results = perform_similarity_search(collection, query, 3)

    if not search_results:
        print("🤖 Bot: I couldn't find any food items matching your request.")
        print("      Try describing what you're in the mood for with different words!")
        return

    print(f"✅ Found {len(search_results)} relevant matches")
    print("🧠 Generating AI-powered response...")

    # Generate enhanced RAG response using IBM Granite
    ai_response = generate_llm_rag_response(query, search_results)

    print(f"\n🤖 Bot: {ai_response}")
```

```
    # Show detailed results for reference
    print(f"\n📊 Search Results Details:")
    print("-" * 45)
    for i, result in enumerate(search_results[:3], 1):
        print(f"{i}. 🍽️  {result['food_name']}")
        print(f"    📍 {result['cuisine_type']} | 🔥 {result['food_calories_per_serving']} cal | 📈 {result['similarity_score']*100:.1f}% match")
        if i < 3:
            print()
```

Explanation: This function orchestrates the complete RAG process: performing vector search, generating AI responses, and displaying results. It provides transparency by showing both the AI-generated response and the underlying search results that informed it.

Part 4 Task 1: H. Add Enhanced Comparison Feature

```
def handle_enhanced_comparison_mode(collection):
    """Enhanced comparison between two food queries using LLM"""
    print("\n🔄 ENHANCED COMPARISON MODE")
    print("   Powered by AI Analysis")
    print("-" * 35)

    query1 = input("Enter first food query: ").strip()
    query2 = input("Enter second food query: ").strip()

    if not query1 or not query2:
        print("❌ Please enter both queries for comparison")
        return

    print(f"\n🔍 Analyzing '{query1}' vs '{query2}' with AI...")

    # Get results for both queries
    results1 = perform_similarity_search(collection, query1, 3)
    results2 = perform_similarity_search(collection, query2, 3)

    # Generate AI-powered comparison
    comparison_response = generate_llm_comparison(query1, query2, results1, results2)

    print(f"\n🤖 AI Analysis: {comparison_response}")

    # Show side-by-side results
    print(f"\n📊 DETAILED COMPARISON")
    print("=" * 60)
    print(f"{'Query 1: ' + query1[:20] + '...' if len(query1) > 20 else 'Query 1: ' + query1:<30} | {'Query 2: ' + query2[:20] + '...' if len(qu
    print("-" * 60)

    max_results = max(len(results1), len(results2))
    for i in range(min(max_results, 3)):
        left = f"{results1[i]['food_name']} ({results1[i]['similarity_score']*100:.0f}%)" if i < len(results1) else "---"
        right = f"{results2[i]['food_name']} ({results2[i]['similarity_score']*100:.0f}%)" if i < len(results2) else "---"
        print(f"{left[:30]:<30} | {right[:30]}")
def generate_llm_comparison(query1: str, query2: str, results1: List[Dict], results2: List[Dict]) -> str:
    """Generate AI-powered comparison between two queries"""
    try:
        context1 = prepare_context_for_llm(query1, results1[:3])
        context2 = prepare_context_for_llm(query2, results2[:3])

        comparison_prompt = f'''You are analyzing and comparing two different food preference queries. Please provide a thoughtful comparison.
Query 1: "{query1}"
Top Results for Query 1:
{context1}
Query 2: "{query2}"
Top Results for Query 2:
{context2}
Please provide a short comparison that:
1. Highlights the key differences between these two food preferences
2. Notes any similarities or overlaps
3. Explains which query might be better for different situations
4. Recommends the best option from each query
5. Keeps the analysis concise but insightful
Comparison:'''
        generated_response = model.generate(prompt=comparison_prompt, params=None)

        if generated_response and "results" in generated_response:
            return generated_response["results"][0]["generated_text"].strip()
        else:
            return generate_simple_comparison(query1, query2, results1, results2)

    except Exception as e:
        return generate_simple_comparison(query1, query2, results1, results2)
def generate_simple_comparison(query1: str, query2: str, results1: List[Dict], results2: List[Dict]) -> str:
    """Simple comparison fallback"""
    if not results1 and not results2:
        return "No results found for either query."
    if not results1:
        return f"Found results for '{query2}' but none for '{query1}'."
    if not results2:
        return f"Found results for '{query1}' but none for '{query2}'."

    return f"For '{query1}', I recommend {results1[0]['food_name']}. For '{query2}', {results2[0]['food_name']} would be perfect."
```

Explanation: The enhanced comparison mode uses the LLM to provide intelligent analysis of the differences and similarities between two food queries, offering insights that go beyond simple result matching.

Part 4 Task 1: I. Add Help Function and Entry Point

```
def show_enhanced_rag_help():
    """Display help information for enhanced RAG chatbot"""
    print("\n📖 ENHANCED RAG CHATBOT HELP")
    print("=" * 45)
    print("🧠 This chatbot uses IBM Granite to understand your")
    print("   food preferences and provide intelligent recommendations.")
    print("\nHow to get the best recommendations:")
    print("  • Be specific: 'healthy Italian pasta under 350 calories'")
    print("  • Mention preferences: 'spicy comfort food for cold weather'")
    print("  • Include context: 'light breakfast for busy morning'")
    print("  • Ask about benefits: 'protein-rich foods for workout recovery'")
    print("\nSpecial features:")
    print("  • 🔍 Vector similarity search finds relevant foods")
    print("  • 🧠 AI analysis provides contextual explanations")
    print("  • 📊 Detailed nutritional and cuisine information")
    print("  • 🔄 Smart comparison between different preferences")
    print("\nCommands:")
    print("  • 'compare' - AI-powered comparison of two queries")
    print("  • 'help' - Show this help menu")
    print("  • 'quit' - Exit the chatbot")
    print("\nTips for better results:")
    print("  • Use natural language - talk like you would to a friend")
    print("  • Mention dietary restrictions or preferences")
    print("  • Include meal timing (breakfast, lunch, dinner)")
    print("  • Specify if you want healthy, comfort, or indulgent options")
if __name__ == "__main__":
    main()
```

Explanation: The help function provides comprehensive guidance on how to interact with the enhanced RAG system, emphasizing the AI capabilities and natural language processing features.

Click on the button below to see the complete `enhanced_rag_chatbot.py` script. You can copy-paste the solution into your `enhanced_rag_chatbot.py` file.

▶ Click for the complete enhanced_rag_chatbot.py script

Part 4 Task 1: J. Testing the Enhanced RAG System

Testing Instructions

1. **Save the Enhanced RAG Chatbot**:
   Save the complete code above as `enhanced_rag_chatbot.py` in your project directory.

2. **Run the Enhancement RAG System**:

   ```
   python3.11 enhanced_rag_chatbot.py
   ```

   Commands:

   - `help` - Show detailed help menu
   - `compare` - Compare recommendations for two different queries
   - `quit` - Exit the chatbot

3. **Test Different Query Types**:

   Natural Language Queries:

   - "I want something healthy and light for lunch"
   - "What Italian comfort food do you recommend?"
   - "I'm looking for protein-rich breakfast options under 300 calories"

Specific Preference Queries:

- "Spicy Asian dishes for dinner"
- "Sweet desserts for a special occasion"
- "Low-calorie snacks for weight management"

Complex Context Queries:

- "I'm feeling sick and want something soothing"
- "Quick meal ideas for busy weeknight"
- "Celebratory foods for a party"

4. **Test the Comparison Feature**:
    - Type `compare` when prompted
    - First query: "chocolate dessert"
    - Second query: "healthy breakfast"
    - Observe how the AI analyzes the differences
5. **Observe the RAG Process**:
    - Notice the "Searching vector database…" message (Retrieval)
    - See "Generating AI-powered response…" (Augmented Generation)
    - Compare the AI response with the detailed search results shown below

Part 5: Testing and Comparing All Three Systems

Now that you have built all three systems, let's test and compare their capabilities to understand the differences and use cases for each approach.

Part 5 Task 1: System Comparison Testing

1. **Test Interactive Search System**:

    ```
    python3.11 interactive_search.py
    ```

    - Try: "healthy salad"
    - Notice: Simple, fast search with immediate results
    - Use case: Quick food discovery with basic similarity
2. **Test Advanced Search System**:

    ```
    python3.11 advanced_search.py
    ```

    - Try Option 2: cuisine search for "pasta" in "Italian"
    - Try Option 3: calorie search for "healthy" under 300 calories
    - Notice: Powerful filtering capabilities for specific requirements
    - Use case: Detailed search with multiple constraints
3. **Test RAG Chatbot System**:

    ```
    python3.11 rag_chatbot.py
    ```

    - Try: "I want something spicy for dinner tonight"
    - Notice: Contextual, conversational responses with explanations
    - Use case: Natural language interaction with intelligent recommendations

Part 5 Task 2: Create a System Comparison Script

4. Create a comparison demonstration script called `system_comparison.py`:

```
from shared_functions import *
import time
def main():
    """Compare all three search systems with the same query"""
    print("🔬 FOOD SEARCH SYSTEMS COMPARISON")
    print("=" * 50)

    # Load data once for all systems
    food_items = load_food_data('./FoodDataSet.json')

    # Create collections for each system
    interactive_collection = create_similarity_search_collection("comparison_interactive")
    advanced_collection = create_similarity_search_collection("comparison_advanced")
    rag_collection = create_similarity_search_collection("comparison_rag")

    # Populate all collections
    populate_similarity_collection(interactive_collection, food_items)
    populate_similarity_collection(advanced_collection, food_items)
    populate_similarity_collection(rag_collection, food_items)

    # Test query
    test_query = "chocolate dessert"

    print(f"\n🔍 Testing query: '{test_query}'")
    print("=" * 50)

    # System 1: Interactive Search Style
    print("\n1️⃣ INTERACTIVE SEARCH APPROACH:")
    print("-" * 30)
    start_time = time.time()
    interactive_results = perform_similarity_search(interactive_collection, test_query, 3)
    interactive_time = time.time() - start_time

    for i, result in enumerate(interactive_results, 1):
        print(f"{i}. {result['food_name']} ({result['similarity_score']*100:.1f}% match)")
        print(f"   {result['food_description']}")
    print(f"⏱️ Response time: {interactive_time:.3f} seconds")

    # System 2: Advanced Search Style
    print("\n2️⃣ ADVANCED SEARCH APPROACH:")
    print("-" * 30)
    start_time = time.time()

    # Show basic search
    basic_results = perform_similarity_search(advanced_collection, test_query, 3)
    print("📋 Basic results:")
    for i, result in enumerate(basic_results, 1):
        print(f"   {i}. {result['food_name']} - {result['cuisine_type']} ({result['food_calories_per_serving']} cal)")

    # Show filtered search
    spicy_results = perform_filtered_similarity_search(
        advanced_collection, test_query, cuisine_filter="Indian", n_results=2
    )
    print("🌶️ Filtered for Indian cuisine:")
    for i, result in enumerate(spicy_results, 1):
        print(f"   {i}. {result['food_name']} ({result['similarity_score']*100:.1f}% match)")

    advanced_time = time.time() - start_time
    print(f"⏱️ Response time: {advanced_time:.3f} seconds")

    # System 3: RAG Chatbot Style
    print("\n3️⃣ RAG CHATBOT APPROACH:")
    print("-" * 30)
    start_time = time.time()

    rag_results = perform_similarity_search(rag_collection, test_query, 3)

    # Generate RAG-style response
    rag_response = f"Perfect! I found some excellent chocolate dessert options for you. "
    rag_response += f"I'd highly recommend the {rag_results[0]['food_name']} - it's a {rag_results[0]['similarity_score']*100:.0f}% match "
    rag_response += f"and offers that sweet, rich flavor you're craving. "
    if rag_results[0]['cuisine_type'] == 'American':
        rag_response += "American desserts are perfect for chocolate lovers! "
    rag_response += f"At {rag_results[0]['food_calories_per_serving']} calories, it's a delightful treat. "
    rag_response += f"You might also enjoy {rag_results[1]['food_name']} as an alternative."

    print(f"🤖 Bot: {rag_response}")

    rag_time = time.time() - start_time
    print(f"⏱️ Response time: {rag_time:.3f} seconds")

    # Comparison Summary
    print("\n📊 SYSTEM COMPARISON SUMMARY:")
    print("=" * 50)
    print("Interactive Search:")
    print("  ✅ Fast and simple")
    print("  ✅ Direct results display")
    print("  ❌ Limited context")

    print("\nAdvanced Search:")
    print("  ✅ Powerful filtering options")
    print("  ✅ Multiple search modes")
    print("  ✅ Precise control")
    print("  ❌ Requires user to know filter options")

    print("\nRAG Chatbot:")
    print("  ✅ Natural language interaction")
    print("  ✅ Contextual explanations")
    print("  ✅ Conversational experience")
```

```
    print("  ❌ More complex implementation")

    print(f"\n⏱️ Performance Comparison:")
    print(f"  Interactive: {interactive_time:.3f}s")
    print(f"  Advanced: {advanced_time:.3f}s")
    print(f"  RAG Chatbot: {rag_time:.3f}s")
if __name__ == "__main__":
    main()
```

5. Run the comparison script:

```
python3.11 system_comparison.py
```

Practice Exercises

Now that you have built and tested all three food search systems, complete these practice exercises to deepen your understanding and extend the functionality.

Exercise 1: Enhance the Interactive Search System

Modify the `interactive_search.py` file to add search history tracking. In essence, you want to keep track of user's previous searches and allow them to view history with a "history" command.

Hint: Add a global variable to store the history, and create new command handlers.

▶ Click here for the partial solution which highlights the lines you must add or modify
▶ Click here for the full solution which features a fully modified interactive_search.py file

Exercise 2: Build a Food Calorie Checker

Create an interactive tool in a new file called `calorie_checker.py` that helps users find foods within their calorie budget:

1. **Ask user for their calorie budget -** Get the maximum calories they want.
2. **Let them search for foods and show if they fit the budget -** Use filtered search to check calorie limits.

Goal: Build a complete interactive program that repeatedly asks for searches and shows budget-friendly results.

▶ Click here for the solution

Exercise 3: Query Different Result Counts

Practice controlling similarity search results by varying the number of results returned in a new file called `result_limiter.py`:

1. **Test different result limits -** Use the `n_results` parameter to get different amounts of results.
2. **Compare result quality -** See how result quality changes with more or fewer results.

Hint: Use the `perform_similarity_search()` function with different `n_results` values (1, 3, 5, 10).

▶ Click here for the solution

Conclusion

After completing this comprehensive lab, you now have hands-on experience with:

- **Interactive CLI Development**: You built a real-time food search interface that responds immediately to user queries, demonstrating how to create engaging command-line applications with Chroma DB integration.
- **Advanced Search Filtering**: You implemented sophisticated filtering capabilities combining similarity search with metadata constraints, showing how to build powerful query systems that meet specific user requirements.
- **RAG (Retrieval-Augmented Generation) Systems**: You created an intelligent chatbot that combines vector similarity search with contextual response generation, demonstrating modern AI applications that provide natural language interfaces to vector databases.
- **System Architecture Comparison**: You built three distinct approaches to the same problem, learning when to use simple similarity search versus advanced filtering versus conversational AI interfaces.

- **Real-world Applications**: These systems demonstrate practical applications of vector databases in recommendation engines, search platforms, and conversational AI systems used in modern applications.

Moreover, in this lab, you developed the following technical skills:

- **Chroma DB Operations**: Creating collections, populating data, performing similarity searches, and applying metadata filters
- **Command-Line Interface (CLI) Design**: Designing user-friendly CLI applications with error handling and interactive loops
- **Similarity Search**: Using cosine similarity to identify documents that closely match a given query
- **Conversational AI Techniques**: Implementing basic context tracking, intent recognition, and response generation
- **Code Architecture**: Building modular code with shared functions and reusable components

Author(s)

Wojciech “Victor” Fulmyk