🏰 Częstochowa City Guide

AI-Powered Question-Answering System using RAG

Course: Neural Networks and Machine Learning
Student: Mehmet Ali Ustaoglu

---

1. Introduction

This project implements an intelligent city guide chatbot for Częstochowa, Poland using **Retrieval-Augmented Generation (RAG)** architecture. The system can answer natural language questions about restaurants, hotels, museums, religious sites, parks, shopping venues, and nightlife in the city.

The primary goal was to demonstrate modern deep learning techniques by building a practical application that combines vector embeddings, semantic search, and large language models to provide accurate, context-aware responses.

> **Key Achievement:** The system indexes 556 Points of Interest (POIs) from OpenStreetMap and uses a locally-running LLM (Gemma) to generate helpful travel recommendations without requiring cloud APIs.

2. Getting Started (Installation Guide)

Follow these steps to run the project from scratch on any machine with Python 3.10+.

2.1 Prerequisites

- **Python 3.10+** - Download from python.org
- **Ollama** - Local LLM runtime
- **~7GB disk space** - For models and dependencies

2.2 Install Ollama

```
# macOS (using Homebrew)
brew install ollama

# Or download from https://ollama.ai for macOS/Windows/Linux

# Start Ollama service
ollama serve

# Pull the Gemma model (in a new terminal)
ollama pull gemma:2b    # Smaller, faster (1.7GB)
# OR
ollama pull gemma:7b    # Larger, better quality (5GB)
```

2.3 Clone and Install Dependencies

```
# Clone the repository
git clone https://github.com/MAUstaoglu/czestochowa-city-guide.git
cd czestochowa-city-guide
```

```
# Create virtual environment
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install Python packages
pip install -r requirements.txt
```

2.4 Prepare Data and Index

```
# Fetch POI data from OpenStreetMap
python3 data/fetch_osm_data.py

# Generate synthetic reviews
python3 data/generate_reviews.py

# Index data into ChromaDB (creates vector embeddings)
python3 rag/vector_store.py
```

2.5 Run the Application

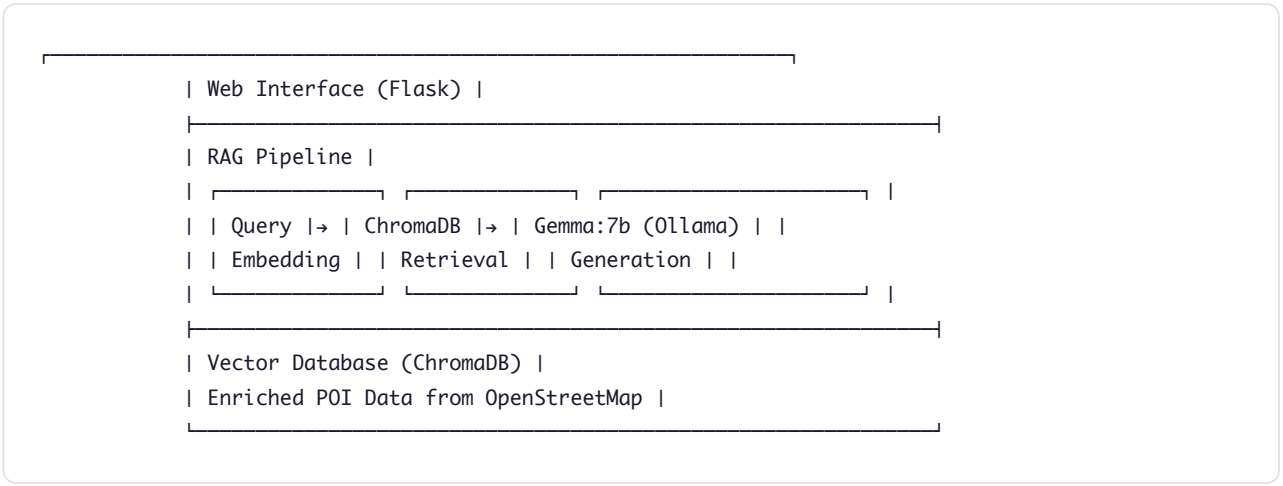
```
# Make sure Ollama is running (ollama serve)
python3 app.py

# Open browser at http://localhost:5001
```

> **Note:** The first run may take a few minutes as it downloads the embedding model (~90MB). Subsequent runs will be faster.

3. System Architecture

The application follows the RAG (Retrieval-Augmented Generation) pattern, which enhances LLM responses by providing relevant context from a knowledge base:



3.1 Data Flow

1. **User Query:** User asks a question in natural language
2. **Embedding:** Query is converted to a 384-dimensional vector using all-MiniLM-L6-v2
3. **Retrieval:** ChromaDB performs semantic similarity search to find relevant POIs
4. **Context Building:** Top-K (default 3) documents are retrieved with metadata

5. **Generation:** Gemma LLM generates a response using the retrieved context
6. **Response:** Answer is displayed with source citations

4. Technology Stack

Category	Technology	Purpose
Language	Python 3.10+	Backend, RAG pipeline, data processing
Web Framework	Flask	REST API and web server
LLM Runtime	Ollama	Local LLM inference engine
LLM Model	Gemma:7b	Google's open-source language model
Embeddings	all-MiniLM-L6-v2	Sentence embeddings (384 dimensions)
Vector Database	ChromaDB	Semantic similarity search
Data Source	OpenStreetMap API	Real-world POI data
Frontend	HTML/CSS/JavaScript	Chat interface

5. Implementation Details

5.1 Data Pipeline

The data preparation involves three stages:

Stage	Script	Output
1. Fetch	`fetch_osm_data.py`	Raw POIs from OpenStreetMap Overpass API
2. Enrich	`generate_reviews.py`	POIs with synthetic reviews and ratings
3. Index	`vector_store.py`	Vector embeddings stored in ChromaDB

5.2 Embedding Model

I use `all-MiniLM-L6-v2` from Sentence-Transformers for creating embeddings. This model produces 384-dimensional vectors that capture semantic meaning, enabling similarity search even when exact keywords don't match. For example, a query about "romantic dinner" can match restaurants described as "cozy" or "intimate."

```
# embeddings.py - Creating sentence embeddings
from sentence_transformers import SentenceTransformer

class EmbeddingModel:
    def __init__(self, model_name="all-MiniLM-L6-v2"):
        self.model = SentenceTransformer(model_name)

    def embed(self, texts: list) -> list:
        """Convert texts to 384-dimensional vectors."""
        return self.model.encode(texts, convert_to_numpy=True)
```

5.3 Vector Database

ChromaDB stores the embedded vectors and enables fast cosine similarity search. When a user asks a question, the query is embedded and compared against all 556 POI vectors to find the most semantically similar documents.

```
# vector_store.py - Semantic search with ChromaDB
import chromadb

class VectorStore:
    def __init__(self):
        self.client = chromadb.PersistentClient(path="./chroma_db")
        self.collection = self.client.get_or_create_collection("pois")

    def search(self, query_embedding, top_k=3):
        """Find top-K most similar documents."""
        results = self.collection.query(
            query_embeddings=[query_embedding],
            n_results=top_k
        )
        return results
```

5.4 Large Language Model

The system uses Google's Gemma model running locally via Ollama. I implemented support for both `gemma:2b` (1.7GB, faster) and `gemma:7b` (5GB, more capable). The UI includes a model switcher to change between available models dynamically.

```
# llm.py - Ollama integration for text generation
import requests

class LLM:
    def __init__(self, model="gemma:7b"):
        self.model = model
        self.base_url = "http://localhost:11434"

    def generate(self, prompt: str, context: str = "") -> str:
        """Generate response using the LLM."""
        response = requests.post(
            f"{self.base_url}/api/generate",
            json={"model": self.model, "prompt": prompt, "stream": False}
        )
        return response.json()["response"]
```

5.5 RAG Prompt Engineering

The LLM receives a carefully crafted prompt that includes:

- System role as a "friendly tourist guide for Częstochowa"
- Instructions to use only the provided context
- Retrieved POI documents with names, addresses, ratings, and reviews
- The user's original question

```
# llm.py - RAG prompt template
def _build_rag_prompt(self, question: str, context: str) -> str:
    prompt = f"""You are a friendly tourist guide for Częstochowa, Poland.

INSTRUCTIONS:
1. Answer using ONLY the information from the Context below.
2. Provide specific details (names, addresses, ratings).
3. If you cannot find relevant information, politely say so.

Context:
{context}

Question: {question}

Answer:"""
    return prompt
```

6. Features

- **Natural Language Queries:** Ask questions in plain English or Polish
- **Category Filtering:** Filter results by restaurant, hotel, museum, etc.
- **Source Citations:** Each answer shows which POIs were used
- **Response Metadata:** Displays latency and document count
- **Model Switching:** Change LLM models from the UI
- **Fully Offline:** Works without internet after initial setup

7. Project Structure

```
project/
├── app.py                # Flask web server & API endpoints
├── config.py             # Configuration settings
```

```
├── requirements.txt       # Python dependencies
|
├── data/
|   ├── fetch_osm_data.py  # OpenStreetMap data fetcher
|   ├── generate_reviews.py # Review generator
|   └── czestochowa_pois.json # Enriched POI data
|
├── rag/
|   ├── embeddings.py      # Sentence embeddings
|   ├── vector_store.py    # ChromaDB integration
|   ├── llm.py             # Ollama/Gemma integration
|   └── pipeline.py        # Complete RAG pipeline
|
├── evaluation/
|   ├── metrics.py         # Evaluation metrics
|   └── run_evaluation.py  # Benchmark runner
|
├── templates/
|   └── index.html         # Chat interface
|
└── static/
    └── style.css          # Styling
```

8. Neural Network Concepts Applied

8.1 Transformer Architecture

Both the embedding model (MiniLM) and the LLM (Gemma) are based on the Transformer architecture. MiniLM uses the encoder portion for creating semantic embeddings, while Gemma uses a decoder-only architecture for text generation.

8.2 Sentence Embeddings

The embedding model maps variable-length text to fixed-size vectors (384 dimensions) in a semantic space where similar meanings are close together. This is achieved through training on sentence pairs using contrastive learning.

8.3 Vector Similarity Search

ChromaDB uses cosine similarity to compare vectors:

```
similarity = (A · B) / (||A|| × ||B||)
```

Higher similarity scores indicate more semantically related documents.

8.4 Language Model Generation

Gemma generates responses autoregressively, predicting one token at a time based on the context (prompt + retrieved documents). Temperature parameter controls randomness in token selection.

9. Results

The system successfully indexes 556 POIs across multiple categories and responds to complex multi-part queries. Example tested queries include:

- "What restaurants are in Częstochowa?" → Returns specific restaurants with ratings
- "Recommend a hotel near Jasna Góra monastery" → Finds hotels with proximity context
- "Where can I go shopping?" → Returns malls, shops, and boutiques

> **Performance:** Average response time ~15-20 seconds with Gemma:7b, ~5-10 seconds with Gemma:2b. Trade-off between quality and speed.

9.1 User Interface

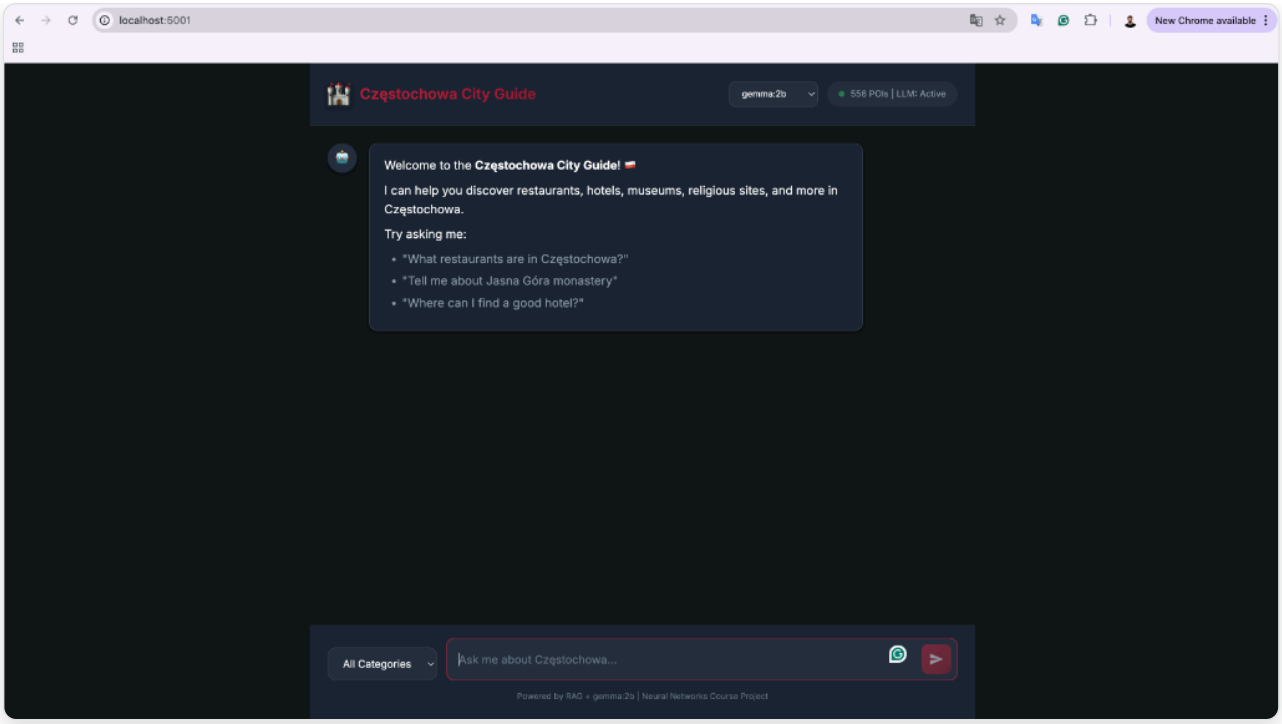


Figure 1: Chat interface showing the model selector, status indicator, and conversation area

10. Challenges & Solutions

Challenge	Solution
Limited POI data from OSM	Generated synthetic reviews to enrich content
LLM hallucination	Strict prompt engineering to use only provided context
Slow inference time	Offer choice between faster (2b) and better (7b) models
Special characters (Polish)	UTF-8 encoding throughout the pipeline

11. Conclusion

This project demonstrates a practical application of modern deep learning techniques including transformer-based embeddings, vector databases, and large language models. The RAG architecture successfully grounds LLM responses in real data, reducing hallucination while providing helpful, accurate travel recommendations.

The fully local deployment using Ollama makes it privacy-friendly and suitable for offline use after initial setup. Future improvements could include real-time data updates, multi-language support, and integration with mapping services.