Translation Memory with RAG - Experimental Repository

This is an experimental repository where we're exploring the integration of Retrieval Augmented Generation (RAG) with translation memory concepts. We're testing how RAG can enhance translation systems by providing contextually relevant examples from a curated translation memory.

Note: This is a research/testing project and not a production-ready translation system.

What We're Exploring

- **Translation Memory + RAG**: Combining traditional translation memory approaches with modern RAG techniques
- **Context-Aware Retrieval**: How well can vector search find relevant translation examples?
- **Cultural Context Integration**: Testing if RAG can surface cultural nuances and formality levels
- **Multi-language Vector Embeddings**: Experimenting with different embedding models for multilingual content
- **Scalable Architecture**: Building a flexible pipeline that could be extended for various translation workflows

Current Implementation Features

- Multi-language support
- ChromaDB vector storage for translation examples
- Fireworks AI integration for generation
- Flexible embedding model fallbacks
- CLI interface for testing different scenarios

Repository Structure

```
translation-rag/
├── config.py              # Configuration management
├── utils.py               # Utility functions
├── pipeline.py            # Reusable RAG pipeline setup
├── rag.py                 # Main translation RAG interface
├── translation_memory.py  # Translation memory management
├── seed_memory/           # Sample translation memories (generated with Claude
Sonnet 4)
│   ├── en_de.json         # English-German pairs
│   ├── en_es.json         # English-Spanish pairs
│   └── ...                # Other language combinations
├── environment.yml        # Conda environment specification
├── chroma_db/             # ChromaDB storage (created automatically)
└── tests/                 # Test suite
```

About seed_memory: The translation example files in `seed_memory/` were automatically generated using Claude Sonnet 4 to provide diverse, culturally-aware translation pairs for testing our RAG approach.

Getting Started

Prerequisites

- Anaconda or Miniconda
- Fireworks AI API access

Setup

1. **Clone and setup environment**:

```
conda env create -f environment.yml
conda activate translation-rag
```

2. **Configure your API key**:
   Create a `.env` file with your Fireworks API credentials:

```
FIREWORKS_API_KEY=your_api_key_here
FIREWORKS_BASE_URL=https://api.fireworks.ai/inference/v1
MODEL_NAME=accounts/fireworks/models/llama4-scout-instruct-basic
```

3. **Test the setup**:

```
python rag.py "How do you say hello in Spanish?"
```

Testing Different Approaches

Basic RAG vs Direct LLM

```
# Using RAG (retrieves similar examples from translation memory)
python rag.py "How do you say 'thank you' in Italian?"

# Direct LLM (no retrieval)
python rag.py "How do you say 'thank you' in Italian?" --no-rag
```

Experimenting with Different Queries

```
# Test cultural context awareness
python rag.py "What's the formal way to greet someone in German?"
```

```
# Test formality levels
python rag.py "How do I politely decline a meeting in Spanish?"

# Test technical translations
python rag.py "Translate 'machine learning algorithm' to French"
```

System Analysis

```
# Show retrieval statistics
python rag.py --stats

# Reinitialize with seed data
python rag.py --seed
```

What We've Learned So Far

RAG Integration Benefits

- **Contextual Examples**: RAG successfully retrieves relevant translation pairs that provide context
- **Cultural Nuances**: The system can surface culturally-appropriate alternatives
- **Consistency**: Similar phrases get consistent translations when examples exist

Current Limitations

- **Cold Start**: Performance depends heavily on the quality of seed translation memory
- **Cross-lingual Embeddings**: Some language pairs work better than others
- **Context Boundaries**: System sometimes retrieves overly similar examples

Interesting Findings

- ChromaDB performs well with multilingual embeddings for European languages
- Cultural context examples significantly improve translation quality
- The fallback embedding strategy helps with robustness

Architecture Overview

```
User Query
    ↓
Translation Memory RAG Pipeline
    ↓
1. Vector Similarity Search (ChromaDB)
    ↓
2. Context Assembly (Retrieved Examples)
    ↓
3. Prompt Construction (Query + Examples)
    ↓
```

```
4. LLM Generation (Fireworks AI)
    ↓
Enhanced Translation Response
```

Extending the System

Adding New Translation Pairs

1. Create new JSON files in `seed_memory/` following the existing format
2. Run `python rag.py --seed` to update the vector database

Testing New Embedding Models

Update the embedding configuration in `config.py` or environment variables:

```
EMBEDDING_MODEL=your-preferred-model
```

Experimenting with Different Retrievers

The `pipeline.py` module provides a flexible base for trying different retrieval strategies.

Development Notes

Key Dependencies

- **LangChain**: RAG framework and document processing
- **ChromaDB**: Vector storage with persistence
- **Fireworks AI**: LLM provider
- **SentenceTransformers**: Multilingual embeddings

Testing

Run the test suite to validate changes:

```
python -m pytest tests/
```

Common Issues

- **ChromaDB Persistence**: If you encounter database locks, delete the `chroma_db/` directory
- **Embedding Failures**: The system falls back through multiple embedding providers automatically
- **API Limits**: Fireworks AI has rate limits; consider adding delays for batch processing