03_RAG

December 14, 2025

1 NER RAG

1.1 1. Setup & Configuration

1.1.1 1.1 Import Dependencies

[1]:
```
# Import core libraries
import json
import os
import sys
import time
from loguru import logger
from pathlib import Path
from datetime import datetime
from typing import List, Dict, Any, Optional, Tuple
from dataclasses import dataclass
from contextlib import contextmanager

import numpy as np
import pandas as pd
from tqdm.auto import tqdm

# LangChain imports
from langchain_ollama import OllamaEmbeddings, OllamaLLM
from langchain_chroma import Chroma
from langchain_core.documents import Document

# Import evaluation functions
sys.path.insert(0, str(Path.cwd().parent))
from src.utils.evaluation import (
    calculate_accuracy,
    print_comparison_table,
    parse_ner_response
)


# Suppress verbose logs from httpx and chromadb
import logging
```

```
logging.getLogger("httpx").setLevel(logging.WARNING)
logging.getLogger("chromadb").setLevel(logging.WARNING)
```

1.1.2 1.2. Configuration

[2]:
```
# Configuration
CONFIG = {
    # Ollama configuration
    "ollama_host": os.getenv("OLLAMA_HOST", "http://127.0.0.1:11434"),
    "embedding_model": os.getenv("EMBEDDING_MODEL", "nomic-embed-text:latest"),
    "llm_model": os.getenv("LLM_MODEL", "mistral:7b"),  # Using smaller model␣
 ↪for stability

    # Chroma configuration
    "chroma_persist_dir": os.getenv("CHROMA_PERSIST_DIR", "./chroma_db"),
    "chroma_collection": os.getenv("CHROMA_COLLECTION", "ner_kb"),

    # Chunking configuration
    "chunk_size": int(os.getenv("CHUNK_SIZE", "1500")),  # 800 chars = safe for␣
 ↪embedding
    "chunk_overlap": int(os.getenv("CHUNK_OVERLAP", "100")),  # More overlap␣
 ↪preserves context
    "min_chunk_size": int(os.getenv("MIN_CHUNK_SIZE", "50")),  # Minimum useful␣
 ↪chunk

    # Retrieval configuration
    "top_k_retrieval": int(os.getenv("TOP_K_RETRIEVAL", "5")),  # Reduced for␣
 ↪smaller KB
    "retrieval_threshold": float(os.getenv("RETRIEVAL_THRESHOLD", "0.5")),

    # LLM inference configuration
    "temperature": float(os.getenv("TEMPERATURE", "0.1")),
    "max_tokens": int(os.getenv("MAX_TOKENS", "1024")),
    "top_p": float(os.getenv("TOP_P", "0.9")),
    "top_k": int(os.getenv("TOP_K", "40")),
}

logger.info("  Configuration:")
logger.info(json.dumps(CONFIG, indent=2))
```

```
2025-12-14 10:23:18.509 | INFO     |
__main__:<module>:28 -   Configuration:
2025-12-14 10:23:18.510 | INFO     |
```

```
__main__:<module>:29 - {
  "ollama_host": "http://127.0.0.1:11434",
  "embedding_model": "nomic-embed-text:latest",
  "llm_model": "mistral:7b",
  "chroma_persist_dir": "./chroma_db",
  "chroma_collection": "ner_kb",
  "chunk_size": 1500,
  "chunk_overlap": 100,
  "min_chunk_size": 50,
  "top_k_retrieval": 5,
  "retrieval_threshold": 0.5,
  "temperature": 0.1,
  "max_tokens": 1024,
  "top_p": 0.9,
  "top_k": 40
}
2025-12-14 10:23:18.510 | INFO     |
__main__:<module>:29 - {
  "ollama_host": "http://127.0.0.1:11434",
  "embedding_model": "nomic-embed-text:latest",
  "llm_model": "mistral:7b",
  "chroma_persist_dir": "./chroma_db",
  "chroma_collection": "ner_kb",
  "chunk_size": 1500,
  "chunk_overlap": 100,
  "min_chunk_size": 50,
  "top_k_retrieval": 5,
  "retrieval_threshold": 0.5,
  "temperature": 0.1,
  "max_tokens": 1024,
  "top_p": 0.9,
  "top_k": 40
}
```

1.2 2. Document Chunking Pipeline

Define `VietnameseDocumentChunker` class to chunk Vietnamese text into smaller chunks with metadata

[3]:
```python
import re
from underthesea import sent_tokenize

class VietnameseDocumentChunker:
    """Production-grade Vietnamese document chunker using underthesea"""

    def __init__(self,
                 chunk_size: int = 1500,
                 overlap: int = 100,
                 min_chunk_size: int = 50):
        self.chunk_size = chunk_size
        self.overlap = overlap
        self.min_chunk_size = min_chunk_size

    def chunk_by_sentences(self, text: str) -> List[str]:
        """Split text by Vietnamese sentences using underthesea"""
        try:
            # Vietnamese sentence segmentation
            sentences = sent_tokenize(text)
            return [s.strip() for s in sentences if s.strip()]
        except Exception as e:
            logger.warning(f"Underthesea failed, falling back to regex: {e}")
            # Fallback to regex-based splitting
            sentence_markers = r'([.!?]|â€¦|\n\n)'
            parts = re.split(sentence_markers, text)
            sentences = []
            for i in range(0, len(parts)-1, 2):
                sent = parts[i] + (parts[i+1] if i+1 < len(parts) else "")
                if sent.strip():
                    sentences.append(sent.strip())
            return sentences

    def chunk_with_metadata(self,
                            text: str,
                            doc_id: str,
                            metadata: Dict = None) -> List[Dict[str, Any]]:
        """Chunk text with metadata and context overlap"""

        sentences = self.chunk_by_sentences(text)
        chunks = []
        current_chunk = []
        current_length = 0
```

```
        for sent in sentences:
            sent_len = len(sent)  # CHARACTER count (FIXED: was word count!)

            if current_length + sent_len <= self.chunk_size:
                current_chunk.append(sent)
                current_length += sent_len
            else:
                # Save current chunk if large enough
                if current_length >= self.min_chunk_size:
                    chunk_text = " ".join(current_chunk)
                    chunks.append({
                        "text": chunk_text,
                        "doc_id": doc_id,
                        "chunk_index": len(chunks),
                        "char_count": current_length,  # Changed from word_count
                        "metadata": metadata or {}
                    })

                # Start new chunk with overlap (last 2 sentences)
                overlap_sents = current_chunk[-2:] if len(current_chunk) >= 2␣
↪else current_chunk[-1:]
                current_chunk = overlap_sents + [sent]
                current_length = sum(len(s) for s in current_chunk)  #␣
↪CHARACTER count

        # Save last chunk
        if current_chunk and current_length >= self.min_chunk_size:
            chunks.append({
                "text": " ".join(current_chunk),
                "doc_id": doc_id,
                "chunk_index": len(chunks),
                "char_count": current_length,  # Changed from word_count
                "metadata": metadata or {}
            })

        return chunks

# Initialize chunker
```

Create a chunker instance with the configured parameters

[4]:
```
chunker = VietnameseDocumentChunker(
    chunk_size=CONFIG["chunk_size"],
    overlap=CONFIG["chunk_overlap"],
    min_chunk_size=CONFIG["min_chunk_size"]
)
logger.success("Chunker initialized with parameters:")
```

```
logger.info(f"  Chunk size: {CONFIG['chunk_size']}")
logger.info(f"  Overlap: {CONFIG['chunk_overlap']}")
logger.info(f"  Min chunk size: {CONFIG['min_chunk_size']}")
```

```
2025-12-14 10:23:19.702 | SUCCESS  |
__main__:<module>:6 - Chunker initialized
with parameters:
2025-12-14 10:23:19.702 | INFO     |
__main__:<module>:7 -   Chunk size: 1500
2025-12-14 10:23:19.703 | INFO     |
__main__:<module>:8 -   Overlap: 100
2025-12-14 10:23:19.703 | INFO     |
__main__:<module>:9 -   Min chunk size: 50
2025-12-14 10:23:19.702 | INFO     |
__main__:<module>:7 -   Chunk size: 1500
2025-12-14 10:23:19.703 | INFO     |
__main__:<module>:8 -   Overlap: 100
2025-12-14 10:23:19.703 | INFO     |
__main__:<module>:9 -   Min chunk size: 50
```

1.3 3. Embedding Setup

[5]:
```
# Initialize embedder
embedder = OllamaEmbeddings(
    model=CONFIG["embedding_model"],
    base_url=CONFIG["ollama_host"]
)

# Test embedding
try:
    test_embedding = embedder.embed_query("Kiểm tra")
    logger.success(f"Embedding model initialized: {CONFIG['embedding_model']}")
    logger.info(f"   Vector dimension: {len(test_embedding)}")
except Exception as e:
    logger.error(f"Failed to initialize embedding model: {e}")
    raise
```

```
2025-12-14 10:23:19.728 | SUCCESS  |
__main__:<module>:10 - Embedding model
initialized: nomic-embed-text:latest
2025-12-14 10:23:19.729 | INFO     |
__main__:<module>:11 -    Vector dimension:
768
2025-12-14 10:23:19.729 | INFO     |
```

```
__main__:<module>:11 -    Vector dimension:
768
```

1.4 4. Vector Store (Chroma)

[6]:
```python
# Initialize Chroma vector store
try:
    # Create persist directory if it doesn't exist
    persist_dir = Path(CONFIG["chroma_persist_dir"])
    persist_dir.mkdir(parents=True, exist_ok=True)

    vector_store = Chroma(
        collection_name=CONFIG["chroma_collection"],
        embedding_function=embedder,
        persist_directory=str(persist_dir)
    )

    # Get collection stats
    collection = vector_store._collection
    count = collection.count()

    logger.success("" Chroma VectorStore initialized")
    logger.info(f"  Collection: {CONFIG['chroma_collection']}")
    logger.info(f"  Persist directory: {persist_dir}")

except Exception as e:
    logger.error(f"- Chroma initialization error: {e}")
    raise
```

```
2025-12-14 10:23:19.888 | SUCCESS  |
__main__:<module>:17 - " Chroma VectorStore
initialized
2025-12-14 10:23:19.889 | INFO     |
__main__:<module>:18 -   Collection: ner_kb
2025-12-14 10:23:19.889 | INFO     |
__main__:<module>:19 -   Persist directory:
chroma_db
2025-12-14 10:23:19.889 | INFO     |
__main__:<module>:18 -   Collection: ner_kb
2025-12-14 10:23:19.889 | INFO     |
__main__:<module>:19 -   Persist directory:
chroma_db
```

1.5 5. Prompt Engineering

[7]:
```
class RAGPromptBuilder:
    """Build context-aware RAG prompts for NER"""

    SYSTEM_PROMPT = """You are an expert Vietnamese Named Entity Recognition␣
 ↪(NER) system.

Task: Extract named entities from Vietnamese text into three categories.

Entity Categories:
1. person: Names of people, individuals
2. organizations: Company names, institutions, government agencies
3. address: Geographic locations, addresses, place names

Requirements:
1. Extract ALL entity mentions (even if they appear multiple times)
2. Preserve exact original Vietnamese text (no normalization)
3. Include titles and descriptors when they're part of the entity
4. Return ONLY valid JSON output

Output Format:
{
    "person": ["Entity 1", "Entity 2"],
    "organizations": ["Entity 1", "Entity 2"],
    "address": ["Entity 1", "Entity 2"]
}"""

    @staticmethod
    def build_rag_prompt(input_text: str,
                         retrieved_docs: List[Document],
                         top_k: int = 5) -> str:
        """Build RAG prompt with similar examples"""

        # Format retrieved examples
        examples_section = "### Similar Examples from Knowledge Base:\n"
        for i, doc in enumerate(retrieved_docs[:top_k], 1):
            # text_preview = doc.page_content[:300]
            # examples_section += f"\nExample {i}:\n"
            # examples_section += f"Text: {text_preview}\n"
            examples_section += f"\nExample {i}:\n"
            examples_section += f"Text: {doc.page_content}\n"

        if not retrieved_docs:
            examples_section = "### Note: No similar examples found in␣
 ↪knowledge base.\n"
```

```
        # Build full prompt
        prompt = f"""{RAGPromptBuilder.SYSTEM_PROMPT}

{examples_section}

# ## Input Text:
\"\"\"
{input_text}
\"\"\"

# ## Output (JSON only, no explanation):
"""

        return prompt

# Test prompt builder
test_docs = [
    Document(page_content="Nguyễn Văn A là CEO của công ty B. Anh sinh năm 1980␣
 ↪tại Hà Nội.")
]
test_prompt = RAGPromptBuilder.build_rag_prompt(
    input_text="Tôi là Phạm Thị C, làm việc tại Đại học X",
    retrieved_docs=test_docs
)
```

1.6 6. NER Extractor

[8]:
```
class RAGNERExtractor:
    """Production RAG-enhanced NER extractor"""

    def __init__(self,
                 embedder: OllamaEmbeddings,
                 vector_store: Chroma,
                 llm_model: str = "ministral-3:14b",
                 ollama_host: str = "http://127.0.0.1:11434",
                 temperature: float = 0.1,
                 top_k_retrieval: int = 3):

        self.embedder = embedder
        self.vector_store = vector_store
        self.top_k_retrieval = top_k_retrieval

        # Initialize LLM
        try:
            self.llm = OllamaLLM(
                base_url=ollama_host,
                model=llm_model,
```

```
                temperature=temperature,
                top_p=0.9,
                top_k=40,
                num_predict=1024
            )
        except Exception as e:
            logger.error(f"Failed to initialize LLM: {e}")
            raise

    def _deduplicate_chunks(self, docs: List[Document], max_per_doc: int = 2)␣
↪-> List[Document]:
        """
        Deduplicate chunks from same document to avoid redundancy.

        Strategy:
        - Group chunks by doc_id
        - Keep max_per_doc best chunks from each document
        - If multiple chunks from same doc are consecutive, merge them

        Args:
            docs: Retrieved documents (chunks)
            max_per_doc: Maximum chunks to keep from same document

        Returns:
            Deduplicated list of documents
        """
        if not docs:
            return docs

        # Group chunks by doc_id
        from collections import defaultdict
        doc_groups = defaultdict(list)

        for idx, doc in enumerate(docs):
            doc_id = doc.metadata.get("doc_id", f"unknown_{idx}")
            doc_groups[doc_id].append((idx, doc))

        # Keep top chunks per document
        deduplicated = []
        seen_doc_ids = set()

        for doc in docs:
            doc_id = doc.metadata.get("doc_id", None)

            if doc_id not in seen_doc_ids:
                # First chunk from this document - always keep
                deduplicated.append(doc)
```

```
                seen_doc_ids.add(doc_id)
            elif len([d for d in deduplicated if d.metadata.get("doc_id") ==␣
↪doc_id]) < max_per_doc:
                # Add more chunks up to limit
                deduplicated.append(doc)


        return deduplicated

    def extract_with_rag(self,
                         text: str,
                         use_rag: bool = True,
                         return_context: bool = False) -> Dict[str, Any]:
        """Extract entities using RAG"""

        result = {
            "text": text,
            "entities": {"person": [], "organizations": [], "address": []},
            "retrieval_info": None,
            "error": None
        }

        try:
            start_time = time.time()

            # Step 1: Retrieve similar chunks
            retrieved_docs = []
            retrieval_time = 0

            if use_rag:
                ret_start = time.time()
                try:
                    # Similarity search
                    # Truncate query to prevent errors
                    # Max 1500 chars for embedding
                    query_text = text[:1500] if len(text) > 1500 else text

                    retrieved_docs = self.vector_store.similarity_search(
                        query=query_text,
                        k=self.top_k_retrieval
                    )
                    retrieval_time = time.time() - ret_start

                    # Deduplicate chunks
                    retrieved_docs = self._deduplicate_chunks(retrieved_docs,␣
↪max_per_doc=1)
                except Exception as e:
```

```
                    logger.warning(f"Retrieval failed, proceeding without RAG:␣
↪{e}")
                    use_rag = False

            # Step 2: Build prompt
            prompt = RAGPromptBuilder.build_rag_prompt(
                input_text=text,
                retrieved_docs=retrieved_docs,
                top_k=self.top_k_retrieval
            )

            # Step 3: Call LLM
            llm_start = time.time()
            response = self.llm.invoke(prompt)
            llm_time = time.time() - llm_start

            # Step 4: Parse response
            entities = self._parse_response(response)
            result["entities"] = entities

            # Step 5: Collect metadata
            total_time = time.time() - start_time
            result["retrieval_info"] = {
                "num_chunks": len(retrieved_docs),
                "retrieval_time": retrieval_time,
                "llm_time": llm_time,
                "total_time": total_time,
                "used_rag": use_rag
            }

            if not return_context:
                result.pop("retrieval_info")

        except Exception as e:
            result["error"] = str(e)
            logger.error(f"Extraction error: {e}", exc_info=True)

        return result

    @staticmethod
    def _parse_response(response: str) -> Dict[str, List[str]]:
        """Parse LLM JSON response"""

        import re

        # Extract JSON from response
```

```
        json_match = re.search(r'\{[^{}]*(?:\{[^{}]*\}[^{}]*)*\}', response, re.
↪DOTALL)

        default_result = {
            "person": [],
            "organizations": [],
            "address": []
        }

        if not json_match:
            logger.warning("No JSON found in LLM response")
            return default_result

        try:
            parsed = json.loads(json_match.group())

            result = {
                "person": parsed.get("person", []),
                "organizations": parsed.get("organizations", []),
                "address": parsed.get("address", [])
            }

            # Ensure all values are lists
            for key in result:
                if not isinstance(result[key], list):
                    result[key] = [str(result[key])] if result[key] else []

            return result

        except json.JSONDecodeError as e:
            logger.warning(f"JSON parse error: {e}")
            return default_result

# Initialize extractor
```

[9]:
```
try:
    ner_extractor = RAGNERExtractor(
        embedder=embedder,
        vector_store=vector_store,
        llm_model=CONFIG["llm_model"],
        ollama_host=CONFIG["ollama_host"],
        temperature=CONFIG["temperature"],
        top_k_retrieval=CONFIG["top_k_retrieval"]
    )
    logger.success("" RAGNERExtractor initialized")
except Exception as e:
    logger.error(f"- RAGNERExtractor error: {e}")
```

```
    logger.info(f"  Make sure the LLM model is available:")
    logger.info(f"  ollama pull {CONFIG['llm_model']}")
```

```
2025-12-14 10:23:19.919 | SUCCESS  |
__main__:<module>:10 - " RAGNERExtractor
initialized
```

1.7 7. Data Preparation

[10]:
```
import sys
from pathlib import Path

# Add project root to path
project_root = Path.cwd().parent
if str(project_root) not in sys.path:
    sys.path.insert(0, str(project_root))

# Import data loading functions
from src.data import load_processed_data

# Load VLSP 2018 NER dataset
data_splits = load_processed_data()

logger.info(f"Dataset loaded:")
for split_name, split_data in data_splits.items():
    logger.info(f"  {split_name:>5}: {len(split_data):>4} examples")

# Use train split for RAG knowledge base
train_data = data_splits['train']
dev_data = data_splits['dev']
test_data = data_splits['test']

# Create articles from train data for RAG
training_articles = []
for item in train_data:
    article = {
        "id": str(item['id']),
        "text": item['text'],
        "source": "vlsp_2018_train",
        "domain": item.get('topic', 'general'),
        "ground_truth": item['ground_truth']  # Store ground truth for reference
    }
    training_articles.append(article)

logger.info(f"Prepared {len(training_articles)} articles from training set for␣
 ↪RAG knowledge base")
```

```
# Display sample
sample = training_articles[0]
logger.info(f"Sample article:")
```

```
2025-12-14 10:23:19.952 | INFO     |
__main__:<module>:15 - Dataset loaded:
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -   train:  781 examples
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -     dev:  260 examples
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -    test:  241 examples
2025-12-14 10:23:19.954 | INFO     |
__main__:<module>:36 - Prepared 781 articles from
training set for RAG knowledge base
2025-12-14 10:23:19.954 | INFO     |
__main__:<module>:40 - Sample article:
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -   train:  781 examples
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -     dev:  260 examples
2025-12-14 10:23:19.953 | INFO     |
__main__:<module>:17 -    test:  241 examples
2025-12-14 10:23:19.954 | INFO     |
__main__:<module>:36 - Prepared 781 articles from
training set for RAG knowledge base
2025-12-14 10:23:19.954 | INFO     |
__main__:<module>:40 - Sample article:
```

1.8 8. Knowledge Base Ingestion

Define `KnowledgeBaseManager`

[11]:
```
class KnowledgeBaseManager:
    """Production-grade knowledge base ingestion manager with crash recovery"""

    def __init__(self,
                 vector_store: Chroma,
                 chunker: VietnameseDocumentChunker,
                 batch_size: int = 5,
                 max_retries: int = 3,
                 retry_delay: float = 2.0):
        self.vector_store = vector_store
        self.chunker = chunker
        self.batch_size = batch_size
        self.max_retries = max_retries
```

```
        self.retry_delay = retry_delay

    def check_collection_status(self) -> Dict[str, Any]:
        """Check current collection status"""
        try:
            collection = self.vector_store._collection
            count = collection.count()
            return {
                "exists": True,
                "count": count,
                "collection_name": collection.name
            }
        except Exception as e:
            logger.error(f"Failed to check collection status: {e}")
            return {"exists": False, "count": 0}

    def clear_collection(self, confirm: bool = False):
        """Clear all documents from collection"""
        if not confirm:
            raise ValueError("Must set confirm=True to clear collection")

        try:
            collection = self.vector_store._collection
            current_count = collection.count()

            if current_count > 0:
                collection.delete(where={})
                logger.info(f"Cleared {current_count} documents from␣
↪collection")
            else:
                logger.info("Collection already empty")

        except Exception as e:
            logger.error(f"Failed to clear collection: {e}")
            raise

    def ingest_articles(self,
                        articles: List[Dict],
                        strategy: str = "full",
                        ) -> Dict[str, Any]:
        """
        Ingest articles with automatic crash recovery

        Args:
            articles: List of article dicts
            strategy: 'chunked' or 'full'
```

```
        Returns:
            Ingestion statistics
        """


        start_time = time.time()
        documents = []
        stats = {
            "articles_processed": 0,
            "chunks_created": 0,
            "documents_indexed": 0,
            "failed": 0,
            "failed_ids": [],
            "strategy": strategy,
        }

        # Step 1: Prepare documents
        logger.info("Step 1/2: Preparing documents...")

        for article in tqdm(articles, desc="Preparing documents"):
            try:
                if strategy == "chunked":
                    chunks = self.chunker.chunk_with_metadata(
                        text=article["text"],
                        doc_id=article["id"],
                        metadata={
                            "source": article["source"],
                            "domain": article["domain"]
                        }
                    )

                    for chunk in chunks:
                        doc = Document(
                            page_content=chunk["text"],
                            metadata={
                                "doc_id": chunk["doc_id"],
                                "chunk_index": chunk["chunk_index"],
                                "char_count": chunk["char_count"],
                                **chunk["metadata"]
                            }
                        )
                        documents.append(doc)
                        stats["chunks_created"] += 1

                elif strategy == "full":
                    # Max 1000 chars per chunk
                    text = article["text"][:1000]
```

```
                    doc = Document(
                        page_content=text,
                        metadata={
                            "doc_id": article["id"],
                            "source": article["source"],
                            "domain": article["domain"],
                            "full_article": True
                        }
                    )
                    documents.append(doc)
                    stats["chunks_created"] += 1

                stats["articles_processed"] += 1

            except Exception as e:
                logger.error(f"Failed to prepare article {article['id']}: {e}")
                stats["failed"] += 1
                stats["failed_ids"].append(article["id"])


        if len(documents) == 0:
            logger.warning("No documents to ingest!")
            return stats

        # Index with retry
        logger.info("Step 2/2: Indexing with crash recovery...")

        consecutive_failures = 0
        max_consecutive_failures = 3

        for i in tqdm(range(0, len(documents), self.batch_size),␣
↪desc="Indexing"):
            batch = documents[i:i+self.batch_size]
            batch_success = False

            for attempt in range(self.max_retries):
                try:
                    # Batch embedding
                    self.vector_store.add_documents(batch)
                    stats["documents_indexed"] += len(batch)
                    batch_success = True
                    consecutive_failures = 0  # Reset on success

                    # Conservative delay for stability
                    time.sleep(1.0)  # 1 second between batches
                    break
```

```
                except Exception as e:
                    error_msg = str(e)

                    if attempt < self.max_retries - 1:
                        wait_time = self.retry_delay * (2.0 ** attempt)  # Aggressive backoff
                        logger.warning(f"Batch {i//self.batch_size} failed (attempt {attempt+1}), waiting {wait_time:.1f}s")
                        time.sleep(wait_time)
                    else:
                        logger.error(f"Batch {i//self.batch_size} failed permanently")
                        stats["failed"] += len(batch)
                        for doc in batch:
                            stats["failed_ids"].append(doc.metadata.get("doc_id", "unknown"))
                        consecutive_failures += 1

        # Final stats
        elapsed_time = time.time() - start_time
        stats["elapsed_time"] = elapsed_time
        stats["docs_per_second"] = stats["documents_indexed"] / elapsed_time if elapsed_time > 0 else 0


        return stats
```

Initialize KnowledgeBaseManager

[12]:
```
kb_manager = KnowledgeBaseManager(
    vector_store=vector_store,
    chunker=chunker,
    batch_size=2,
    max_retries=5,
    retry_delay=3.0
)

status = kb_manager.check_collection_status()
logger.success("KnowledgeBaseManager initialized with auto-recovery")
logger.info(f"Current collection status: {status}")
```

```
2025-12-14 10:23:19.967 | SUCCESS  |
__main__:<module>:10 - KnowledgeBaseManager
initialized with auto-recovery
2025-12-14 10:23:19.967 | INFO     |
```

```
__main__:<module>:11 - Current collection status:
```

```
{'exists': True, 'count': 1674, 'collection_name': 'ner_kb'}
2025-12-14 10:23:19.967 | INFO     |
__main__:<module>:11 - Current collection status:
```

```
{'exists': True, 'count': 1674, 'collection_name': 'ner_kb'}
```

Initialize embedder

[13]:
```
embedder = OllamaEmbeddings(
    model=CONFIG["embedding_model"],
    base_url=CONFIG["ollama_host"]
)

# Test embedding
try:
    test_embedding = embedder.embed_query("Kiểm tra")
    logger.success(f"Embedding model initialized: {CONFIG['embedding_model']}")
    logger.info(f"   Vector dimension: {len(test_embedding)}")
except Exception as e:
    logger.error(f"Failed to initialize embedding model: {e}")
    raise
```

```
2025-12-14 10:23:19.990 | SUCCESS  |
__main__:<module>:9 - Embedding model
```

```
initialized: nomic-embed-text:latest
2025-12-14 10:23:19.990 | INFO     |
__main__:<module>:10 -    Vector dimension:
```

```
768
2025-12-14 10:23:19.990 | INFO     |
__main__:<module>:10 -    Vector dimension:
```

```
768
```

Ingest training data into knowledge base

[14]:
```
# Check if we should ingest
status = kb_manager.check_collection_status()
current_count = status['count']


if current_count > 0:
    logger.warning(f"Warning: Collection already contains {current_count}␣
 ↪documents!")

else:
    logger.info("Collection is empty, starting ingestion...")
```

```
    # Using chunked strategy
    INGESTION_STRATEGY = "chunked"  # Preserves ALL content (0% data loss)



    # Run ingestion
    ingestion_stats = kb_manager.ingest_articles(
        articles=training_articles,
        strategy=INGESTION_STRATEGY
    )

    # Print results
    logger.success("INGESTION COMPLETE")
    logger.info(f"Results:")
    logger.info(f"  Strategy: {ingestion_stats['strategy']}")
    logger.info(f"  Articles processed: {ingestion_stats['articles_processed']}/
↪{len(training_articles)}")
    logger.info(f"  Documents created: {ingestion_stats['chunks_created']}")
    logger.info(f"  Documents indexed: {ingestion_stats['documents_indexed']}")
    logger.error(f"  Failed: {ingestion_stats['failed']}")

    # Check success rate
    if ingestion_stats['chunks_created'] > 0:
        success_rate = (ingestion_stats['documents_indexed'] /␣
↪ingestion_stats['chunks_created']) * 100
        logger.success(f"  Success rate: {success_rate:.1f}%")

        if success_rate < 90:
            logger.warning(f"Warning: Low success rate!")
        else:
            logger.success(f"Excellent success rate!")

    # Verify final count
    final_status = kb_manager.check_collection_status()
    logger.info(f"Final collection count: {final_status['count']} documents")

    logger.info("=" * 80)
```

```
2025-12-14 10:23:19.996 | WARNING  |
__main__:<module>:7 - Warning: Collection

already contains 1674 documents!
```

1.9 9. Test NER Extraction

[15]:
```
# Test input
test_input = """Lê Văn E là giám đốc của Công ty Cổ phần Phần mềm ABC. Anh ấy␣
 ↪sinh năm 1985 tại
Sài Gòn. Công ty ABC có văn phòng chính tại đường Nguyễn Hữu Cảnh, Hồ Chí Minh.
 ↪"""

logger.info(f"Input Text: {test_input}")
```

```
2025-12-14 10:23:19.999 | INFO     |
__main__:<module>:5 - Input Text: Lê Văn E là
giám đốc của Công ty Cổ phần Phần mềm ABC. Anh ấy sinh năm 1985 tại
Sài Gòn. Công ty ABC có văn phòng chính tại đường Nguyễn Hữu Cảnh, Hồ Chí
Minh.
```

[16]:
```
# Extract without RAG (baseline)
baseline_result = ner_extractor.extract_with_rag(
    text=test_input,
    use_rag=False,
    return_context=True
)

if baseline_result["error"]:
    logger.error(f"Error: {baseline_result['error']}")
else:
    logger.info(json.dumps(baseline_result["entities"], ensure_ascii=False,␣
 ↪indent=2))
    if baseline_result["retrieval_info"]:
        logger.info(f"Timing: {baseline_result['retrieval_info']['total_time']:.
 ↪3f}s")
```

```
2025-12-14 10:23:21.653 | INFO     |
```

```
__main__:<module>:11 - {
  "person": [
    "Lê Văn E",
    "Anh ấy"
  ],
  "organizations": [
    "Công ty Cổ phần Phần mềm ABC"
  ],
  "address": [
    "Sài Gòn",
    "đường Nguyễn Hữu Cảnh, Hồ Chí Minh"
  ]
}
2025-12-14 10:23:21.654 | INFO     |
__main__:<module>:13 - Timing: 1.647s
2025-12-14 10:23:21.654 | INFO     |
__main__:<module>:13 - Timing: 1.647s
```

[17]:
```python
# Extract with RAG
logger.info("-"*40)

rag_result = ner_extractor.extract_with_rag(
    text=test_input,
    use_rag=True,
    return_context=True
)

if rag_result["error"]:
    logger.error(f"Error: {rag_result['error']}")
else:
    logger.info(json.dumps(rag_result["entities"], ensure_ascii=False,␣
 ↪indent=2))
    if rag_result["retrieval_info"]:
        info = rag_result["retrieval_info"]
        logger.info(f"Retrieval Info:")
        logger.info(f"  Chunks retrieved: {info['num_chunks']}")
        logger.info(f"  Retrieval time: {info['retrieval_time']:.3f}s")
        logger.info(f"  LLM time: {info['llm_time']:.3f}s")
        logger.info(f"  Total time: {info['total_time']:.3f}s")
```

```
2025-12-14 10:23:21.659 | INFO     |
__main__:<module>:2 -
```

```
----------------------------------------
2025-12-14 10:23:23.849 | INFO     |
__main__:<module>:13 - {
  "person": [
    "Lê Văn E"
  ],
  "organizations": [
    "Công ty Cổ phần Phần mềm ABC"
  ],
  "address": [
    "đường Nguyễn Hữu Cảnh, Hồ Chí Minh"
  ]
}
2025-12-14 10:23:23.850 | INFO     |
__main__:<module>:16 - Retrieval Info:
2025-12-14 10:23:23.850 | INFO     |
__main__:<module>:17 -   Chunks retrieved: 4
2025-12-14 10:23:23.850 | INFO     |
__main__:<module>:18 -   Retrieval time:
0.013s
2025-12-14 10:23:23.851 | INFO     |
__main__:<module>:19 -   LLM time: 2.176s
2025-12-14 10:23:23.851 | INFO     |
__main__:<module>:20 -   Total time: 2.190s
```

1.10 10. Evaluation & Metrics

[18]:
```python
# Prepare validation set from dev data
# Use a subset for faster evaluation
import random

# Set seed for reproducibility
random.seed(42)

# Sample validation examples from dev set
val_size = 30  # Use 30 examples for validation
val_indices = random.sample(range(len(dev_data)), min(val_size, len(dev_data)))

validation_examples = []
for idx in val_indices:
    item = dev_data[idx]
    validation_examples.append({
```

```
        "id": item['id'],
        "text": item['text'],
        "ground_truth": item['ground_truth']
    })

logger.info(f"Prepared validation set: {len(validation_examples)} examples from␣
 ↪dev split")
total_entities = sum(sum(len(v) for v in ex['ground_truth'].values()) for ex in␣
 ↪validation_examples)

# Show entity distribution
entity_counts = {'person': 0, 'organizations': 0, 'address': 0}
for ex in validation_examples:
    for entity_type in entity_counts:
        entity_counts[entity_type] += len(ex['ground_truth'].get(entity_type,␣
 ↪[]))

for entity_type, count in entity_counts.items():
    logger.info(f"  {entity_type:15s}: {count:4d} ({count/total_entities*100:.
 ↪1f}%)")

# Display sample
sample = validation_examples[0]
logger.info(f"Sample validation example:")
logger.info(f"  Entities: {sum(len(v) for v in sample['ground_truth'].
 ↪values())} total")
```

```
2025-12-14 10:23:23.856 | INFO     |
__main__:<module>:21 - Prepared validation set:
30 examples from dev split
2025-12-14 10:23:23.856 | INFO     |
__main__:<module>:31 -   person         :  140
(36.5%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:31 -   organizations  :   92
(24.0%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:31 -   address        :  152
(39.6%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:35 - Sample validation
example:
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:36 -   Entities: 8 total
```

```
2025-12-14 10:23:23.856 | INFO     |
__main__:<module>:31 -   person          :  140

(36.5%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:31 -   organizations   :   92

(24.0%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:31 -   address         :  152

(39.6%)
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:35 - Sample validation

example:
2025-12-14 10:23:23.857 | INFO     |
__main__:<module>:36 -   Entities: 8 total
```

[ ]:
```python
# Evaluate both methods on validation set

baseline_predictions = []
rag_predictions = []

start_time = time.time()

for i, example in enumerate(tqdm(validation_examples, desc="Evaluating")):
    # Baseline (without RAG)
    base = ner_extractor.extract_with_rag(example["text"], use_rag=False)
    baseline_predictions.append(base["entities"])

    # With RAG
    rag = ner_extractor.extract_with_rag(example["text"], use_rag=True)
    rag_predictions.append(rag["entities"])

elapsed = time.time() - start_time
ground_truths = [ex["ground_truth"] for ex in validation_examples]

logger.success(f"Evaluation complete in {elapsed:.1f}s ({elapsed/
 ↪len(validation_examples):.1f}s per example)")

# Calculate metrics
logger.info("Calculating metrics...")
baseline_metrics = calculate_accuracy(baseline_predictions, ground_truths)
rag_metrics = calculate_accuracy(rag_predictions, ground_truths)

logger.info("RESULTS")

logger.info(f"  Exact Match Accuracy: {baseline_metrics['accuracy']:.1%}")
```

```
logger.info(f"  Overall Precision:␣
 ↪{baseline_metrics['overall_entity_metrics']['precision']:.1%}")
logger.info(f"  Overall Recall:␣
 ↪{baseline_metrics['overall_entity_metrics']['recall']:.1%}")
logger.info(f"  Overall F1: {baseline_metrics['overall_entity_metrics']['f1']:.
 ↪1%}")

logger.info("  Per-entity-type metrics:")
for entity_type, metrics in baseline_metrics['per_entity_type'].items():
    logger.info(f"    {entity_type:15s}: P={metrics['precision']:.1%},␣
 ↪R={metrics['recall']:.1%}, F1={metrics['f1']:.1%}")

logger.info(f"  Exact Match Accuracy: {rag_metrics['accuracy']:.1%}")
logger.info(f"  Overall Precision:␣
 ↪{rag_metrics['overall_entity_metrics']['precision']:.1%}")
logger.info(f"  Overall Recall:␣
 ↪{rag_metrics['overall_entity_metrics']['recall']:.1%}")
logger.info(f"  Overall F1: {rag_metrics['overall_entity_metrics']['f1']:.1%}")

logger.info("  Per-entity-type metrics:")
for entity_type, metrics in rag_metrics['per_entity_type'].items():
    logger.info(f"    {entity_type:15s}: P={metrics['precision']:.1%},␣
 ↪R={metrics['recall']:.1%}, F1={metrics['f1']:.1%}")

logger.info("3. IMPROVEMENT:")
rag_f1 = baseline_metrics['overall_entity_metrics']['f1']
baseline_f1 = rag_metrics['overall_entity_metrics']['f1']
overall_improvement = (rag_f1 - baseline_f1) * 100

logger.info(f"  Overall F1 improvement: {overall_improvement:+.1f}%")
logger.info(f"  Exact Match improvement: {(rag_metrics['accuracy'] -␣
 ↪baseline_metrics['accuracy']) * 100:+.1f}%")

logger.info("  Per-entity-type F1 improvement:")
for entity_type in baseline_metrics['per_entity_type'].keys():
    baseline_et_f1 = baseline_metrics['per_entity_type'][entity_type]['f1']
    rag_et_f1 = rag_metrics['per_entity_type'][entity_type]['f1']
    improvement = (rag_et_f1 - baseline_et_f1) * 100
    logger.info(f"    {entity_type:15s}: {improvement:+.1f}%")

# Print comparison table

comparison_dict = {
    'Baseline (No RAG)': baseline_metrics,
    'With RAG': rag_metrics
}
```

```
print_comparison_table(comparison_dict)


# Save results
results = {
    'config': CONFIG,
    'validation_size': len(validation_examples),
    'baseline_metrics': baseline_metrics,
    'rag_metrics': rag_metrics,
    'improvement': {
        'overall_f1': overall_improvement,
        'exact_match': (rag_metrics['accuracy'] - baseline_metrics['accuracy'])␣
 ↪* 100
    },
    'elapsed_time': elapsed
}

logger.info(f" Evaluation results ready")
logger.info(f"  Baseline F1: {baseline_f1:.1%}")
logger.info(f"  RAG F1: {rag_f1:.1%}")
```

1.11 11. Performance Profiling

[23]:
```
import time
from collections import defaultdict

class PerformanceMonitor:
    """Monitor RAG pipeline performance"""

    def __init__(self):
        self.timings = defaultdict(list)

    def record(self, operation: str, duration: float):
        self.timings[operation].append(duration)

    def report(self) -> pd.DataFrame:
        """Generate performance report"""

        rows = []
        for operation, times in self.timings.items():
            rows.append({
                "Operation": operation,
                "Count": len(times),
                "Mean (ms)": np.mean(times) * 1000,
                "Median (ms)": np.median(times) * 1000,
                "Min (ms)": np.min(times) * 1000,
```

```
                "Max (ms)": np.max(times) * 1000,
                "Std (ms)": np.std(times) * 1000
            })

        return pd.DataFrame(rows)

# Profile extraction
```

[24]:
```
monitor = PerformanceMonitor()

logger.info("Performance Profiling (10 iterations):")

for i in range(10):
    # Baseline
    start = time.time()
    ner_extractor.extract_with_rag(test_input, use_rag=False)
    baseline_time = time.time() - start
    monitor.record("Baseline (No RAG)", baseline_time)

    # RAG
    start = time.time()
    ner_extractor.extract_with_rag(test_input, use_rag=True)
    rag_time = time.time() - start
    monitor.record("With RAG", rag_time)

logger.info(monitor.report().to_string())
```

```
2025-12-14 10:34:22.938 | INFO     |
__main__:<module>:3 - Performance Profiling (10
iterations):
2025-12-14 10:34:59.776 | INFO     |
__main__:<module>:18 -            Operation
Count    Mean (ms)  Median (ms)     Min (ms)     Max (ms)   Std (ms)
0  Baseline (No RAG)     10  1494.510007  1480.045795  1462.295532  1589.990854
35.897936
1           With RAG     10  2188.977551  2188.072205  2183.880329  2197.580814
3.407714
```