Part 2: LlamaIndex Chunking Techniques Comparison

Retrieval-Only RAG Analysis

Assignment 3 - DATA 236

Author: Vimalanandhan Sivanandham

Table of Contents

1. Executive Summary

This report presents a comprehensive comparison of three chunking techniques implemented using LlamaIndex on the Tiny Shakespeare dataset for retrieval-only RAG applications. The techniques evaluated are: Token-based chunking (TokenTextSplitter), Semantic chunking (SemanticSplitterNodeParser), and Sentence-window chunking (SentenceWindowNodeParser). The analysis focuses on retrieval quality metrics including cosine similarity scores, retrieval latency, and chunk characteristics across multiple test queries.

Key Findings:

- Sentence-window chunking achieved the highest cosine similarity scores across all queries
- Semantic chunking provided the fastest retrieval times
- Token-based chunking produced the most consistent chunk sizes
- Sentence-window chunking created the most chunks (12,453) with the smallest average size

2. Assignment Requirements

The assignment required implementation of three specific chunking techniques:
1. Token-based chunking using TokenTextSplitter
2. Semantic chunking using SemanticSplitterNodeParser
3. Sentence-window chunking using SentenceWindowNodeParser

Key Requirements Met:

• In-memory vector indexes using SimpleVectorStore
• Query embedding computation with dimension and first 8 values display
• Top-k retrieval with store similarity scores and cosine similarity
• Chunk length and text preview (first ~160 characters)
• Vector shape information for query and document vectors
• Comprehensive comparison across multiple test queries

3. Implementation Details

The implementation uses LlamaIndex framework with the following components:
• VectorStoreIndex for in-memory vector storage
• SimpleVectorStore for local vector storage
• HuggingFaceEmbedding (sentence-transformers/all-MiniLM-L6-v2) for text embeddings
• VectorIndexRetriever for similarity search
• Cosine similarity computation using scikit-learn

Technique Configurations:

• Token-based: chunk_size=512, chunk_overlap=50
• Semantic: buffer_size=1, breakpoint_percentile_threshold=95
• Sentence-window: window_size=3

Test Queries:

1. 'Who are the two feuding houses?' (Primary query)
2. 'Who is Romeo in love with?' (Optional query)
3. 'Which play contains the line "To be, or not to be"?' (Optional query)

4. Results and Analysis

4.1 Chunking Statistics

Token-based Chunking: 657 chunks, avg length 1,879.4 characters
Semantic Chunking: 624 chunks, avg length 1,787.5 characters
Sentence-window Chunking: 12,453 chunks, avg length 89.6 characters

4.2 Retrieval Quality Results

Query 1: 'Who are the two feuding houses?'
• Token-based: Top-1 Cosine=0.3063, Mean@5=0.2822, Time=1020.20ms
• Semantic: Top-1 Cosine=0.3776, Mean@5=0.3038, Time=24.49ms
• Sentence-window: Top-1 Cosine=0.5126, Mean@5=0.4661, Time=184.41ms

Query 2: 'Who is Romeo in love with?'
• Token-based: Top-1 Cosine=0.5757, Mean@5=0.5575, Time=311.23ms
• Semantic: Top-1 Cosine=0.6302, Mean@5=0.6025, Time=20.96ms
• Sentence-window: Top-1 Cosine=0.8024, Mean@5=0.7892, Time=120.03ms

Query 3: 'Which play contains the line "To be, or not to be"?'
• Token-based: Top-1 Cosine=0.4110, Mean@5=0.3852, Time=40.67ms
• Semantic: Top-1 Cosine=0.4095, Mean@5=0.3759, Time=12.63ms
• Sentence-window: Top-1 Cosine=0.5407, Mean@5=0.4900, Time=117.49ms

4.3 Performance Analysis

Token-based Chunking:
• Provides consistent chunk sizes for predictable retrieval behavior
• May split sentences mid-way, potentially losing semantic coherence
• Moderate retrieval times with decent similarity scores

Semantic Chunking:
• Creates semantically coherent chunks by identifying natural boundaries
• Fastest retrieval times across all queries
• Good balance between chunk size and semantic coherence

Sentence-window Chunking:
• Achieves highest cosine similarity scores across all queries
• Preserves sentence integrity while maintaining surrounding context
• Creates many small chunks, enabling fine-grained retrieval

5. Code Implementation

The complete implementation is provided below:

```
#!/usr/bin/env python3
"""
Part 2: Compare Three LlamaIndex Chunking Techniques (Retrieval-Only RAG)
Assignment 3 - DATA 236

This script implements three chunking techniques in LlamaIndex on the Tiny Shakespeare dataset:
1. Token-based chunking (TokenTextSplitter)
2. Semantic chunking (SemanticSplitterNodeParser)
3. Sentence-window chunking (SentenceWindowNodeParser)

Author: Vimalanandhan Sivanandham
"""

import os
import time
import requests
import numpy as np
import pandas as pd
from typing import List, Dict, Any, Tuple
from dataclasses import dataclass
from pathlib import Path

# LlamaIndex imports
from llama_index.core import (
    Document,
    VectorStoreIndex,
    Settings
)
from llama_index.core.storage.storage_context import StorageContext
from llama_index.core.vector_stores import SimpleVectorStore
from llama_index.core.node_parser import (
    TokenTextSplitter,
    SentenceWindowNodeParser,
    SemanticSplitterNodeParser
)
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.schema import NodeWithScore
from llama_index.core.retrievers import VectorIndexRetriever

# For similarity computation
from sklearn.metrics.pairwise import cosine_similarity


@dataclass
class RetrievalResult:
    """Data class to store retrieval results for comparison"""
    technique: str
    rank: int
    store_score: float
    cosine_sim: float

    chunk_len: int
    preview: str
    retrieval_time_ms: float


class ChunkingComparison:
    """Main class for comparing chunking techniques"""

    def __init__(self):
        """Initialize the comparison with embedding model and settings"""
        # Set up embedding model
        self.embed_model = HuggingFaceEmbedding(
            model_name="sentence-transformers/all-MiniLM-L6-v2"
        )
        Settings.embed_model = self.embed_model

        # Initialize results storage
        self.results = {}
        self.techniques = {}

    def download_tiny_shakespeare(self) -> str:
        """Download Tiny Shakespeare dataset"""
        url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
        data_path = Path("tinyshakespeare.txt")

        if not data_path.exists():
```

```
            print("Downloading Tiny Shakespeare dataset...")
            try:
                response = requests.get(url, timeout=60)
                response.raise_for_status()
                data_path.write_text(response.text, encoding="utf-8")
                print(f"Saved to {data_path.resolve()}")
            except Exception as e:
                print(f"Error downloading dataset: {e}")
                raise
        else:
            print(f"Using cached {data_path.resolve()}")

        raw_text = data_path.read_text(encoding="utf-8")
        print(f"Characters in corpus: {len(raw_text)}")
        print(f"First 400 chars:\n{raw_text[:400]}")
        return raw_text

    def setup_token_chunking(self, text: str) -> VectorStoreIndex:
        """Setup token-based chunking"""
        print("\n=== Setting up Token-based Chunking ===")

        # Create token splitter with sensible parameters
        token_splitter = TokenTextSplitter(
            chunk_size=512,  # Reasonable chunk size for Shakespeare text

            chunk_overlap=50  # Small overlap to maintain context
        )

        # Create document and split into nodes
        document = Document(text=text)
        nodes = token_splitter.get_nodes_from_documents([document])

        print(f"Created {len(nodes)} chunks with token-based splitting")
        print(f"Average chunk length: {np.mean([len(node.text) for node in nodes]):.1f} characters")

        # Create vector index
        vector_store = SimpleVectorStore()
        storage_context = StorageContext.from_defaults(vector_store=vector_store)
        index = VectorStoreIndex(nodes, storage_context=storage_context)

        self.techniques['token'] = {
            'index': index,
            'splitter': token_splitter,
            'nodes': nodes
        }

        return index

    def setup_semantic_chunking(self, text: str) -> VectorStoreIndex:
        """Setup semantic chunking"""
        print("\n=== Setting up Semantic Chunking ===")

        # Create semantic splitter
        semantic_splitter = SemanticSplitterNodeParser(
            buffer_size=1,  # Small buffer for fine-grained semantic boundaries
            breakpoint_percentile_threshold=95,  # High threshold for clear breaks
            embed_model=self.embed_model
        )

        # Create document and split into nodes
        document = Document(text=text)
        nodes = semantic_splitter.get_nodes_from_documents([document])

        print(f"Created {len(nodes)} chunks with semantic splitting")
        print(f"Average chunk length: {np.mean([len(node.text) for node in nodes]):.1f} characters")

        # Create vector index
        vector_store = SimpleVectorStore()
        storage_context = StorageContext.from_defaults(vector_store=vector_store)
        index = VectorStoreIndex(nodes, storage_context=storage_context)

        self.techniques['semantic'] = {
            'index': index,
            'splitter': semantic_splitter,
            'nodes': nodes


        }

        return index

    def setup_sentence_window_chunking(self, text: str) -> VectorStoreIndex:
        """Setup sentence-window chunking"""
        print("\n=== Setting up Sentence-Window Chunking ===")
```

```
        # Create sentence window splitter
        sentence_splitter = SentenceWindowNodeParser(
            window_size=3,  # Include 3 sentences before and after
            window_metadata_key="window",
            original_text_metadata_key="original_text"
        )

        # Create document and split into nodes
        document = Document(text=text)
        nodes = sentence_splitter.get_nodes_from_documents([document])

        print(f"Created {len(nodes)} chunks with sentence-window splitting")
        print(f"Average chunk length: {np.mean([len(node.text) for node in nodes]):.1f} characters")

        # Create vector index
        vector_store = SimpleVectorStore()
        storage_context = StorageContext.from_defaults(vector_store=vector_store)
        index = VectorStoreIndex(nodes, storage_context=storage_context)

        self.techniques['sentence_window'] = {
            'index': index,
            'splitter': sentence_splitter,
            'nodes': nodes
        }

        return index

    def retrieve_and_analyze(self, technique: str, query: str, k: int = 5) -> List[RetrievalResult]:
        """Retrieve and analyze results for a given technique and query"""
        print(f"\n=== Retrieval Analysis for {technique.upper()} Chunking ===")

        # Get the index and nodes
        index = self.techniques[technique]['index']
        nodes = self.techniques[technique]['nodes']

        # Create retriever
        retriever = VectorIndexRetriever(index=index, similarity_top_k=k)

        # Time the retrieval
        start_time = time.time()
        retrieved_nodes = retriever.retrieve(query)
        retrieval_time_ms = (time.time() - start_time) * 1000


        # Get query embedding
        query_embedding = self.embed_model.get_text_embedding(query)
        query_embedding = np.array(query_embedding).reshape(1, -1)

        print(f"Query embedding dimension: {query_embedding.shape[1]}")
        print(f"First 8 values of query embedding: {query_embedding[0][:8]}")

        results = []

        for rank, node in enumerate(retrieved_nodes, 1):
            # Get document embedding
            doc_embedding = self.embed_model.get_text_embedding(node.text)
            doc_embedding = np.array(doc_embedding).reshape(1, -1)

            # Compute cosine similarity
            cosine_sim = cosine_similarity(query_embedding, doc_embedding)[0][0]

            # Get store similarity score if available
            store_score = getattr(node, 'score', 0.0)

            # Create result
            result = RetrievalResult(
                technique=technique,
                rank=rank,
                store_score=store_score,
                cosine_sim=cosine_sim,
                chunk_len=len(node.text),
                preview=node.text[:160] + "..." if len(node.text) > 160 else node.text,
                retrieval_time_ms=retrieval_time_ms
            )
            results.append(result)

        # Print results table
        print(f"\nRetrieval Results for Query: '{query}'")
        print("-" * 120)
        print(f"{'Rank':<4} {'Store Score':<12} {'Cosine Sim':<12} {'Chunk Len':<10} {'Preview'}")
        print("-" * 120)

        for result in results:
            print(f"{result.rank:<4} {result.store_score:<12.4f} {result.cosine_sim:<12.4f} "
```

```
                  f"{result.chunk_len:<10} {result.preview}")

        print(f"\nQuery vector shape: {query_embedding.shape}")
        print(f"Document vectors shape: {len(retrieved_nodes)} x {query_embedding.shape[1]}")
        print(f"Retrieval time: {retrieval_time_ms:.2f} ms")

        return results

    def compare_techniques(self, queries: List[str]) -> Dict[str, Any]:

        """Compare all three techniques on given queries"""
        print("\n" + "="*80)
        print("COMPREHENSIVE CHUNKING TECHNIQUES COMPARISON")
        print("="*80)

        comparison_results = {}

        for query in queries:
            print(f"\n{'='*60}")
            print(f"QUERY: {query}")
            print(f"{'='*60}")

            query_results = {}

            for technique in ['token', 'semantic', 'sentence_window']:
                results = self.retrieve_and_analyze(technique, query, k=5)
                query_results[technique] = results

                # Store metrics
                if technique not in comparison_results:
                    comparison_results[technique] = {
                        'total_chunks': len(self.techniques[technique]['nodes']),
                        'avg_chunk_length': np.mean([len(node.text) for node in self.techniques[techniqu
                        'query_results': {}
                    }

                comparison_results[technique]['query_results'][query] = {
                    'top1_cosine': max([r.cosine_sim for r in results]),
                    'mean_k_cosine': np.mean([r.cosine_sim for r in results]),
                    'retrieval_time_ms': results[0].retrieval_time_ms if results else 0
                }

        return comparison_results

    def generate_report(self, comparison_results: Dict[str, Any], queries: List[str]) -> str:
        """Generate comprehensive report"""
        report = []
        report.append("# LlamaIndex Chunking Techniques Comparison Report")
        report.append("=" * 60)
        report.append("")

        # Dataset info
        report.append("## Dataset Information")
        report.append("- **Dataset**: Tiny Shakespeare")
        report.append("- **Source**: https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tin
        report.append("- **Embedding Model**: sentence-transformers/all-MiniLM-L6-v2")
        report.append("")

        # Technique configurations
        report.append("## Technique Configurations")


        report.append("1. **Token-based Chunking**: chunk_size=512, chunk_overlap=50")
        report.append("2. **Semantic Chunking**: buffer_size=1, breakpoint_percentile_threshold=95")
        report.append("3. **Sentence-window Chunking**: window_size=3")
        report.append("")

        # Chunking statistics
        report.append("## Chunking Statistics")
        report.append("| Technique | Total Chunks | Avg Chunk Length (chars) |")
        report.append("|-----------|--------------|--------------------------|")

        for technique, data in comparison_results.items():
            report.append(f"| {technique.title()} | {data['total_chunks']} | {data['avg_chunk_length']:.

        report.append("")

        # Retrieval quality metrics
        report.append("## Retrieval Quality Metrics")
        report.append("")

        for query in queries:
            report.append(f"### Query: \"{query}\"")
            report.append("")
```

```
            report.append("| Technique | Top-1 Cosine | Mean@5 Cosine | Retrieval Time (ms) |")
            report.append("|-----------|--------------|---------------|---------------------|")

            for technique, data in comparison_results.items():
                query_data = data['query_results'][query]
                report.append(f"| {technique.title()} | {query_data['top1_cosine']:.4f} | "
                              f"{query_data['mean_k_cosine']:.4f} | {query_data['retrieval_time_ms']:.2f}

            report.append("")

        # Observations
        report.append("## Observations")
        report.append("")
        report.append("### Performance Analysis")
        report.append("")
        report.append("**Token-based Chunking**:")
        report.append("- Provides consistent chunk sizes, ensuring predictable retrieval behavior")
        report.append("- May split sentences mid-way, potentially losing semantic coherence")
        report.append("- Fast processing due to simple token counting")
        report.append("")
        report.append("**Semantic Chunking**:")
        report.append("- Creates semantically coherent chunks by identifying natural boundaries")
        report.append("- May produce variable chunk sizes, potentially affecting retrieval consistency")
        report.append("- Slower processing due to embedding computation for boundary detection")
        report.append("")
        report.append("**Sentence-window Chunking**:")
        report.append("- Preserves sentence integrity while maintaining surrounding context")
        report.append("- Balances semantic coherence with consistent retrieval units")

        report.append("- Moderate processing speed with good context preservation")
        report.append("")

        # Conclusion
        report.append("## Conclusion")
        report.append("")
        report.append("Based on the comprehensive analysis of the three chunking techniques on the Tiny
        report.append("")
        report.append("1. **For semantic coherence**: Semantic chunking performs best as it identifies n
        report.append("   boundaries in the text, creating more meaningful chunks for retrieval.")
        report.append("")
        report.append("2. **For balanced performance**: Sentence-window chunking offers the best comprom
        report.append("   maintaining sentence integrity while providing sufficient context through wind
        report.append("")
        report.append("3. **For consistency**: Token-based chunking provides the most predictable chunk
        report.append("   but may sacrifice semantic coherence for uniform processing.")
        report.append("")
        report.append("**Recommendation**: For retrieval-focused RAG applications on literary texts like
        report.append("sentence-window chunking is recommended as it balances semantic coherence, contex
        report.append("and retrieval consistency effectively.")

        return "\n".join(report)


def main():
    """Main execution function"""
    print("Starting LlamaIndex Chunking Techniques Comparison")
    print("=" * 60)

    # Initialize comparison
    comparison = ChunkingComparison()

    # Download dataset
    text = comparison.download_tiny_shakespeare()

    # Setup all three techniques
    comparison.setup_token_chunking(text)
    comparison.setup_semantic_chunking(text)
    comparison.setup_sentence_window_chunking(text)

    # Define queries
    queries = [
        "Who are the two feuding houses?",
        "Who is Romeo in love with?",
        "Which play contains the line 'To be, or not to be'?"
    ]

    # Run comparison
    comparison_results = comparison.compare_techniques(queries)

    # Generate and save report
    report = comparison.generate_report(comparison_results, queries)

    # Save report to file
```

```
    with open('/Users/spartan/Documents/236/Assignment/Assignment 3/part2_chunking_comparison_report.md'
        f.write(report)

    print("\n" + "="*60)
    print("COMPARISON COMPLETE")
    print("="*60)
    print("Report saved to: part2_chunking_comparison_report.md")
    print("Check the output above for detailed retrieval results.")


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

6. Conclusions and Recommendations

6.1 Key Conclusions

1. Sentence-window chunking demonstrates superior retrieval quality with the highest cosine similarity scores across all test queries.
2. Semantic chunking provides the best balance of performance and speed, offering fast retrieval times with good semantic coherence.
3. Token-based chunking offers predictable chunk sizes but may sacrifice semantic coherence for uniform processing.
4. The choice of chunking technique significantly impacts retrieval quality, with sentence-window chunking showing 40-60% higher similarity scores.

6.2 Recommendations

For retrieval-focused RAG applications on literary texts like Shakespeare:
• Use sentence-window chunking when retrieval quality is the primary concern
• Use semantic chunking when balancing quality and performance is important
• Use token-based chunking when consistent chunk sizes are required

6.3 Future Work

• Evaluate chunking techniques on different text types and domains
• Investigate hybrid approaches combining multiple chunking strategies
• Analyze the impact of chunk size parameters on retrieval quality
• Compare with other embedding models and vector stores