**Semantic Cosine Similarity Analysis System**

**Functional Specifications Document**

**System Overview**

**Purpose**: A semantic cosine similarity analysis system designed to compare pairs of business plans through a systematic 4-stage natural language processing pipeline.

**Objective**: Transform raw business plan documents into semantic vectors and calculate quantitative similarity metrics for document comparison and analysis.

**System Architecture**

**Pipeline Controller (0 - Pipeline.py)**

**Role**: Sequential workflow orchestrator

**Function**: Executes the four processing stages in strict order using subprocess calls

**Key Features**:

* Sequential execution of all processing stages
* Error monitoring and status reporting
* Visual feedback with emoji indicators (🚀 for start, ✅ for success, ❌ for errors)
* Automatic failure detection and reporting

**Error Handling**: Catches and reports subprocess failures with detailed error messages

**Dependencies**: Requires all four processing scripts to be present in the same directory

**Technical Implementation**: Uses Python subprocess module for script execution control

**Processing Stage Descriptions**

**Stage 1: Text Normalization (1 - Normalization.py)**

**Purpose**: Converts raw business plan documents into clean, standardized text suitable for NLP analysis

**Key Functions**:

* **Document Reading**: Extracts text content from .docx files in "Testing Data" folder
* **Text Cleaning**: Applies comprehensive normalization including:
  + Converts all text to lowercase
  + Removes stop words, punctuation, and whitespace
  + Performs lemmatization (reduces words to root forms)
* **Output Management**: Saves normalized text as .txt files in "Normalized Data" folder with "\_normalized" suffix
* **Batch Processing**: Handles multiple .docx files automatically

**Technical Implementation**:

* Uses spaCy NLP library with English model (en\_core\_web\_sm)
* NER and parser components disabled for improved performance
* Python-docx library for .docx file reading

**Input Requirements**: .docx business plan files in "Testing Data" directory

**Output Format**: Cleaned .txt files ready for tokenization with "\_normalized" suffix

**Error Handling**: File existence validation and comprehensive exception handling

**Stage 2: Tokenization (2 - Tokenization.py)**

**Purpose**: Breaks normalized text into individual linguistic units (tokens) for semantic processing

**Key Functions**:

* **Text Processing**: Reads normalized .txt files from Stage 1 output
* **Token Extraction**: Converts continuous text into discrete tokens while filtering out whitespace
* **Format Conversion**: Saves tokens as line-separated text files for vectorization input
* **Batch Processing**: Processes all normalized files automatically

**Technical Implementation**:

* Uses full spaCy English model for comprehensive tokenization
* Maintains token integrity while removing non-meaningful whitespace
* One token per line output format

**Input Requirements**: Normalized .txt files from Stage 1

**Output Format**: Tokenized .txt files with "\_tokenized" suffix, one token per line

**Processing Logic**: Preserves all meaningful tokens while filtering whitespace characters

**Stage 3: Semantic Vectorization (3 - Vectorization.py)**

**Purpose**: Transforms tokenized text into high-dimensional numerical vectors that capture semantic meaning

**Key Functions**:

* **Embedding Generation**: Uses SentenceTransformer model to create semantic embeddings
* **Vector Format**: Converts text to numerical vectors that preserve semantic relationships
* **Data Persistence**: Saves vectors as comma-separated values for similarity calculations
* **Batch Processing**: Handles all tokenized files automatically

**Technical Implementation**:

* Uses "all-MiniLM-L6-v2" SentenceTransformer model for semantic embeddings
* Generates high-dimensional vectors representing semantic content
* Comma-separated value format for numerical data storage

**Input Requirements**: Tokenized .txt files from Stage 2

**Output Format**: Vector files with "\_vectorized" suffix containing comma-separated numerical values

**Vector Characteristics**: Fixed-dimension semantic embeddings suitable for similarity calculations

**Stage 4: Cosine Similarity Analysis (4 - Cosine Similarity.py)**

**Purpose**: Calculates semantic similarity scores between pairs of business plan documents

**Key Functions**:

* **File Pairing Logic**: Automatically matches base business plans with their "Final" versions using naming conventions
* **Similarity Calculation**: Computes cosine similarity between vector pairs using scikit-learn
* **Results Generation**: Produces CSV output with similarity scores for each document pair
* **Error Handling**: Validates vector dimensions and handles processing exceptions
* **Quality Control**: Performs dimension matching verification before calculations

**Technical Implementation**:

* Uses scikit-learn's cosine\_similarity function with numpy arrays
* Automatic file pairing based on naming patterns
* Comprehensive error handling for dimension mismatches

**Input Requirements**: Vector files from Stage 3

**Output Format**: "cosine\_similarity\_output.txt" with CSV format containing:

* File names
* Cosine similarity scores (6 decimal precision)

**Pairing Logic**:

* Base files: filename\_normalized\_tokenized\_vectorized.txt
* Final files: filename - Final\_normalized\_tokenized\_vectorized.txt

**Data Flow Architecture**

**Processing Pipeline**

Raw .docx files → Normalization → Tokenization → Vectorization → Similarity Analysis → CSV Results

**File Naming Convention**

Each stage appends a suffix to track processing history:

* filename.docx → filename\_normalized.txt → filename\_normalized\_tokenized.txt → filename\_normalized\_tokenized\_vectorized.txt

**Directory Structure**

* **Input**: "Testing Data" (contains .docx files)
* **Stage 1 Output**: "Normalized Data"
* **Stage 2 Output**: "Tokenized Data"
* **Stage 3 Output**: "Vectorized Data"
* **Final Output**: "Cosine Similarity Results"

**System Requirements**

**Python Dependencies**

* spaCy (with en\_core\_web\_sm model)
* python-docx
* sentence-transformers
* scikit-learn
* numpy
* os (built-in)
* subprocess (built-in)

**Hardware Requirements**

* Sufficient RAM for loading SentenceTransformer models
* Storage space for intermediate processing files
* CPU capable of NLP processing tasks

**Input Format Requirements**

* Business plan documents must be in .docx format
* Files should contain readable text content
* Paired documents should follow naming convention for comparison

**Use Cases and Applications**

**Primary Use Case**

Quantitative comparison of business plan versions to measure semantic similarity and identify changes in content focus or strategy.

**Secondary Applications**

* Document version analysis
* Content similarity assessment
* Business plan evaluation workflows
* Automated document comparison systems

**Technical Specifications**

**Similarity Measurement**

* **Method**: Cosine similarity between semantic vector embeddings
* **Range**: 0.0 (completely dissimilar) to 1.0 (identical)
* **Precision**: 6 decimal places in output

**Processing Characteristics**

* Sequential processing ensures data integrity
* Automatic cleanup of intermediate directories
* Comprehensive error reporting and status tracking
* Batch processing capabilities for multiple documents

**Output Format**

CSV file with columns:

* File Name (base identifier)
* Cosine Similarity (numerical score)

*This system provides automated, quantitative analysis of business plan document similarity using state-of-the-art natural language processing and semantic embedding techniques.*