Earnings Call Analyzer Documentation

Table of Contents

Overview

The Earnings Call Analyzer is an AI-powered application built with Streamlit that processes PDF transcripts of earnings calls, extracts key topics, generates summaries, and provides an intelligent Q&A interface using Retrieval-Augmented Generation (RAG).

Core Features

- **PDF Processing**: Automated loading and preprocessing of earnings call transcripts
- **Metadata Extraction**: Company details, call dates, and participant information
- **Content Segmentation**: Automatic separation of opening remarks and Q&A sessions
- **Topic Extraction**: AI-powered identification of key discussion topics
- **Summarization**: Topic-focused content summarization
- **RAG-powered Q&A**: Interactive assistant for querying transcript content

Architecture

```
├── app.py              # Main Streamlit application
├── api/
│   ├── embedding.py      # FAISS vector store management
│   └── llm.py          # Language model configurations
└── utils/
    ├── doc_parser.py      # PDF processing and text extraction
    ├── metadata.py       # Metadata extraction pipeline
    ├── create_topics.py   # Topic extraction pipeline
    ├── create_summary.py  # Summarization pipeline
    └── rag.py           # RAG implementation
```

Data Flow

1. **Input**: PDF transcript upload or demo file selection
2. **Processing**: Header/footer reduction, text extraction, dialogue parsing
3. **Segmentation**: Split into "Opening Remarks" and "Q&A Session"
4. **Analysis**: Topic extraction and summarization for each section
5. **Indexing**: Vector embeddings creation for RAG pipeline
6. **Interaction**: User queries processed through RAG system

Installation & Setup

Prerequisites

- Python 3.11+
- Required API keys:
  - `HUGGINGFACE_API_TOKEN` (for embeddings)
  - `GROQ_API_KEY` (for LLM services)

Dependencies

```
faiss_cpu==1.12.0
PyMuPDF==1.26.4
langchain==0.3.27
langchain_community==0.3.29
langchain_core==0.3.76
langchain_groq==0.3.8
langchain_huggingface==0.3.1
langchain_text_splitters==0.3.11
pydantic==2.11.9
python-dotenv==1.1.1
streamlit==1.49.1
streamlit_option_menu==0.4.0
```

Running the Application

streamlit run app.py

Module Documentation

app.py - Main Application

The central Streamlit application implements a multi-page interface.

Key Components:

- **Session State Management**: Prevents expensive re-computations across page reloads
- **Thread Pool Executor**: Background processing for vector store creation
- **Multi-tab Navigation**: Organized workflow through different analysis stages

Critical Functions:

```
 @st.fragment
     def topic_summary():
                """Allows users to select topics and generate a
summary."""
```

utils/doc_parser.py - Document Processing

Primary Functions:

`file_load(path: str, default: bool) -> List[Document]`

Processes PDF files with header/footer reduction to remove boilerplate content.

Complex Logic Explanation:
Uses PyMuPDF to define reduction rectangles for headers/footers
Applies redactions to remove irrelevant content before LangChain processing
Saves to temporary location to avoid overwriting original files

`Pattern_extract(docs: List[Document], management: List[str]) -> List[Document]`

Extracts structured dialogues using regex pattern matching.

```
# Regex Pattern Breakdown:
dialogue_pattern = re.compile(
    r"([A-Z][A-Za-z .'-]+):\s+(.*?)(?=(?:[A-Z][A-Za-z .'-]+:)|\Z)",
    re.S  # DOTALL flag allows . to match newlines
)
# Captures: Speaker name, followed by colon, then speech content until next speaker
```

`split_docs_into_sections(extract_docs) -> Tuple[List[Document], List[Document]]`

Identifies transition from opening remarks to Q&A using pattern recognition.

```
qa_start_pattern = re.compile(r"The first question", re.I)
# Uses case-insensitive matching to detect Q&A session start
```

utils/metadata.py - Metadata Extraction

Implements a robust LangChain pipeline with Pydantic validation.

Error Handling Strategy:

```
def safe_parse(response: str, model: BaseModel, default: dict) -> dict:
    Args:
        response: The raw string response from the LLM.
        model: The Pydantic model to validate against.
        default: The default dictionary to return on parsing failure.
    Returns:
        A dictionary representing the parsed data or the default
dictionary.
    """
    try:
        return model.model_validate_json(response)
    except json.JSONDecodeError as e:
        # Catches cases where the response is not valid JSON.
        return default
    except Exception:
        # Catches other potential validation or parsing errors.
        return default
```

utils/create_topics.py - Topic Extraction

Schema Validation:

```
lass TopicsOutput(BaseModel):
    """Defines the expected JSON output schema for topic extraction.

    This model enforces that the output is a JSON object with a single
key
    "topics" which contains a list of strings. This provides a robust
    schema for validation.
    """
    topics: List[str] = Field(..., description="List of relevant topics
extracted from context")
```

Chain Architecture:

```
extractorchain = prompt | llm_oss | StrOutputParser() | RunnableLambda(
    lambda r: safe_parse(r, TopicsOutput, default_dict)
)
```

LCEL (LangChain Expression Language) chain with error handling

utils/rag.py - RAG Implementation

Similarity Search Strategy:

```
    docs_with_scores = store.similarity_search_with_score(user_input,
k=10)

    docs_with_scores.sort(key=lambda x: x[1], reverse=True)
       #  Prepare a formatted string of the retrieved documents.
    # This string is what gets passed to the LLM in the prompt. It
includes
    # the score and metadata for each document for full context.
    docs_text = "\n\n".join([
        f"[Score: {s:.2f}] {d.page_content} | Metadata: {d.metadata}"
        for d, s in docs_with_scores
```

Context Formatting:

```
   extractorchain = rag_qa_prompt | chat_oss | parser
    result = extractorchain.invoke({
        "question": user_input,
        "documents": docs_text
    })
```

Includes similarity scores and metadata for LLM context

api/embedding.py - Vector Store Management

FAISS Implementation:

```
def create_store(documents: list[Document]):
    """Creates a FAISS vector store from a list of documents and saves
it locally.

    This function takes a list of LangChain Document objects, generates
    embeddings for each, and builds an in-memory FAISS index. The index
is
    then persisted to disk for later use.

    Args:
        documents: A list of Document objects to be added to the vector
store.
    """
    # FAISS.from_documents is a convenient method that handles the
embedding
    # of documents and the creation of the FAISS index in a single
step.
    vector_store = FAISS.from_documents(documents, embeddings)

    # Save the vector store to a local directory named "faiss_index".
    # This allows for the index to be loaded later without
re-processing documents.
    vector_store.save_local("faiss_index")
```

api/llm.py - Language Model Configuration

Model Selection Strategy:

JSON-structured outputs for data extraction
```
llm_groq = ChatGroq(
    model="llama-3.3-70b-versatile",
    temperature=0.0,
    model_kwargs={"response_format": {"type": "json_object"}}
)
```

Conversational responses for RAG
```
groq_chat = ChatGroq(
    model="llama-3.3-70b-versatile",
    temperature=0.0,
)
```

Technical Decisions & Trade-offs

Framework Selection: Streamlit vs React/Angular

Decision: Streamlit for rapid prototyping **Rationale:**

- **Faster Development**: Python-based, minimal frontend code required
- **Proof of Concept Focus**: Prioritized functionality over polished UI
- **Data Science Integration**: Native support for ML/AI workflows
- **Learning Curve**: Lower barrier for data scientists vs full-stack development

Trade-offs:

- **Performance**: Full page re-renders on any interaction
- **Customization**: Limited UI/UX flexibility
- **Scalability**: Not suitable for production-grade applications
- **State Management**: Complex session state handling required

LLM Provider: Groq API

Decision: Groq free tier over OpenAI/Anthropic **Rationale:**

- **Cost Efficiency**: Zero cost for development and testing
- **Performance**: Fast inference speeds
- **Model Variety**: Access to multiple open-source models

Trade-offs:

- **Rate Limits**: Daily usage restrictions on free tier
- **Reliability**: Less stable than enterprise providers
- **Model Quality**: Potentially lower accuracy than GPT-4/Claude
- **Support**: Limited customer support on free tier

Document Processing: PyMuPDF

Decision: PyMuPDF over alternatives (pdfplumber, PDFMiner) **Rationale:**

- **Redaction Capabilities**: Built-in header/footer removal
- **Performance**: Fast processing of large documents
- **LangChain Integration**: Native PyMuPDFLoader support

Trade-offs:

- **Dependency Size**: Larger installation footprint
- **OCR Limitations**: Poor handling of scanned documents
- **Layout Preservation**: May lose complex formatting

Embedding Model: all-mpnet-base-v2

Decision: Sentence Transformers model via HuggingFace **Rationale:**

- **Quality**: High-performing general-purpose embeddings
- **Cost**: Free usage through HuggingFace API
- **Compatibility**: Standard 768-dimensional outputs

Trade-offs:

- **Domain Specificity**: Not optimized for financial text
- **API Dependency**: Requires internet connectivity
- **Speed**: Slower than local embeddings

Known Limitations

1. Streamlit Architecture Issues

Continuous Re-rendering:

- Every user interaction triggers full application reload
- Expensive operations (LLM calls, vector searches) may repeat unnecessarily
- Session state management adds complexity

2. Free Tier API Constraints

Rate Limiting:

- Groq: Limited daily requests
- HuggingFace: API throttling under high load

Service Reliability:

- No SLA guarantees on free tiers
- Potential service interruptions
- Model availability fluctuations

3. Document Processing Limitations

PDF Format Dependencies:

- Assumes text-based PDFs (not scanned images)
- Requires consistent "Speaker: content" formatting
- Headers/footers must follow predictable patterns

Dialogue Extraction Issues:

```
dialogue_pattern = re.compile(
    r"([A-Z][A-Za-z .'-]+):\s+(.*?)(?=(?:[A-Z][A-Za-z .'-]+:)|\Z)",
    re.S
)
# May fail with:
# - Inconsistent speaker name formats
# - Multi-line speaker names
# - Special characters in names
```

4. RAG Pipeline Constraints

Context Window Limitations:

- LLM context limits may truncate large document sets
- No automatic chunking for oversized contexts

Embedding Quality:

- General-purpose embeddings may miss financial domain nuances
- No fine-tuning for earnings call terminology

Vector Store Persistence:

- Local FAISS index not suitable for multi-user deployments
- No automatic index updates when documents change

5. Error Handling Gaps

LLM Response Validation:

```
def safe_parse(response: str, model: BaseModel, default: dict) -> dict:
    # Current implementation catches broad exceptions
    # Missing specific error logging for debugging
    try:
        return model.model_validate_json(response)
    except Exception:  # Too broad - should specify exception types
        return default
```

File Upload Security:

- No validation of PDF content safety
- Potential security risks with malicious files
- No file size limits enforced

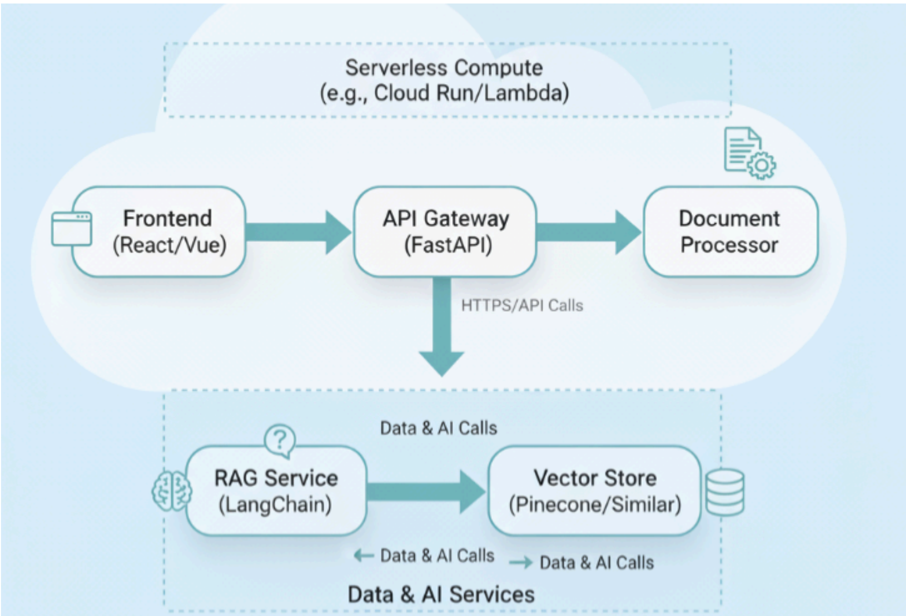
Areas for Improvement

1. Architecture Modernization

Frontend Migration:

- **Recommendation**: Migrate to React/Next.js or Vue.js
- **Benefits**: Better state management, custom UI components, improved performance
- **Implementation**: RESTful API backend with Python, separate frontend

Microservices Architecture:



2. Production-Ready Infrastructure

Database Integration:

- Replace local Embedding with PostgreSQL/MongoDB or any other vectro DB
- Implement user authentication and document management
- Add audit logging for compliance

Scalable Vector Storage:

```
# Current: Local FAISS
vector_store = FAISS.from_documents(documents, embeddings)
vector_store.save_local("faiss_index")

# Recommended: Cloud vector database
import pinecone
index = pinecone.Index("earnings-calls")
index.upsert(vectors=embeddings, metadata=document_metadata)
```

3. Enhanced Document Processing

OCR Integration:

```
# For scanned documents
import pytesseract
from pdf2image import convert_from_path

def process_scanned_pdf(pdf_path: str):
    images = convert_from_path(pdf_path)
    text = ""
    for image in images:
        text += pytesseract.image_to_string(image)
    return text
```

Advanced Pattern Recognition:

```python
# More robust dialogue extraction
import spacy
nlp = spacy.load("en_core_web_sm")

def extract_speakers_nlp(text: str):
    """Use NER for speaker identification"""
    doc = nlp(text)
    speakers = [ent.text for ent in doc.ents if ent.label_ == "PERSON"]
    return speakers
```

4. Model Optimization

Domain-Specific Fine-tuning:

- Fine-tune embeddings on financial documents
- Train custom NER models for financial entity recognition
- Implement sector-specific topic models

Multi-model Ensemble:

```python
class EnsembleTopicExtractor:
    def __init__(self):
        self.models = [
            "llama-3.3-70b-versatile",
            "openai/gpt-oss-120b",
            "mixtral-8x7b-32768"
        ]

    def extract_topics(self, text: str) -> List[str]:
        """Combine outputs from multiple models"""
        all_topics = []
        for model in self.models:
            topics = self.single_model_extract(text, model)
            all_topics.extend(topics)
        return self.deduplicate_and_rank(all_topics)
```

Monitoring and Logging:

```python
import logging
from prometheus_client import Counter, Histogram

# Metrics collection
rag_queries_total = Counter('rag_queries_total', 'Total RAG queries')
rag_response_time = Histogram('rag_response_time_seconds', 'RAG
response time')

@rag_response_time.time()
def monitored_rag_pipeline(query: str):
    rag_queries_total.inc()
    logging.info(f"Processing query: {query[:50]}...")
    return rag_pipeline(query)
```

API Security:

```python
# Rate limiting
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)

@app.post("/analyze")
@limiter.limit("10/minute")
async def analyze_document(request: Request, file: UploadFile):
    # Document analysis endpoint with rate limiting
    pass
```

API Reference

Core Functions

Document Processing

```python
def file_load(path: str, default: bool = True) -> List[Document]
    """Load and preprocess PDF documents.

    Args:
        path: File path to PDF document
        default: Whether to use default demo file
```

```
    Returns:
        List of LangChain Document objects

    Raises:
        FileNotFoundError: If file path is invalid
        PDFProcessingError: If PDF processing fails
    """

def Pattern_extract(docs: List[Document], management: List[str]) ->
List[Document]
    """Extract structured dialogues from documents.

    Args:
        docs: List of raw document objects
        management: List of management participant names

    Returns:
        List of documents with speaker metadata

    Example:
        >>> management = ["John Smith", "Jane Doe"]
        >>> dialogues = Pattern_extract(raw_docs, management)
        >>> print(dialogues[0].metadata)
        {'speaker': 'John Smith', 'role': 'Management', 'order': 1}
    """
```

Analysis Functions

```
def extract_topics(docs: List[Document]) -> dict
    """Extract key topics from document collection.

    Args:
        docs: List of documents to analyze

    Returns:
        Dictionary with 'topics' key containing list of extracted
topics

    Example:
        >>> topics = extract_topics(documents)
        >>> print(topics)
```

```
        {'topics': ['Revenue Growth', 'Market Expansion', 'R&D
Investment']}
    """
def summarizer(input: str, topics: str) -> str
    """Generate topic-focused summary.

    Args:
        input: Text content to summarize
        topics: Comma-separated topics to focus on

    Returns:
        Structured summary organized by topics

    Example:
        >>> summary = summarizer(text, "Revenue, Costs, Guidance")
        >>> print(summary)
        ## Revenue
        - Q2 revenue increased 15% YoY to $100M
        ## Costs
        - Operating costs remained flat at $60M
    """
```

RAG Pipeline

```
def rag_pipeline(user_input: str) -> Tuple[str, List[Tuple[Document,
float]]]
    """Execute RAG pipeline for question answering.

    Args:
        user_input: User's question or query

    Returns:
        Tuple containing:
        - Generated answer (str)
        - List of (document, similarity_score) pairs

    Example:
        >>> answer, docs = rag_pipeline("What was the revenue
guidance?")
        >>> print(f"Answer: {answer}")
        >>> print(f"Based on {len(docs)} retrieved documents")
    """
```

Configuration Parameters

```
# Document Processing
HEADER_REDACTION_HEIGHT = 50  # pixels
FOOTER_REDACTION_HEIGHT = 50  # pixels
MAX_FILE_SIZE = 200 * 1024 * 1024  # 200MB

# RAG Configuration
SIMILARITY_SEARCH_K = 10  # number of documents to retrieve
SIMILARITY_THRESHOLD = 0.7  # minimum similarity score

# LLM Settings
TEMPERATURE = 0.0  # deterministic outputs
MAX_TOKENS = 4000  # response length limit
```