# RAG-Driven Primer Generator

December 19, 2024


## 1 Aspiring Data Science Team

## 2 YongJae Song, jsf7204@konkuk.ac.kr, +82 10-7204-7145


***Design a company-agnostic workflow that generates a US company primer based only on their SEC 10-K filings without embedding model.***

## 3 *Runtime Environment*

- **System**: Apple MacBook Air with M2 chip and 8GB Unified Memory, optimized for efficiency and performance.
- **Operating System**: macOS, providing a robust Unix-based environment for software development and testing.
- **Capabilities**: Suitable for lightweight machine learning tasks, data analysis, and development workflows.
- **Optimization**: Utilized with memory-efficient processes & multi-core computational tools.

[ ]:
```
import ssl
try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context
import logging
logging.basicConfig(level=logging.INFO)
API_KEY = "gpt-4o api key"
```

## 4 Task 1

## 5 HTML Source Chunking Function Overview

- **Purpose**: Extract, clean, and chunk sections from 10-K HTML documents.
- **Key Features**:
  - **Section Extraction**: Uses regex and BeautifulSoup to identify and isolate predefined sections.
  - **Content Cleaning**: Removes overlapping text between sections.

- **Chunking**: Splits content into manageable JSON chunks using LangChain tools.
- **File Classification**: Organizes chunks into directories based on section categories for easy access.
- **Limitation**: To address the unique characteristics of the document, the use of stopwords was unavoidable.

[ ]:
```
import os
import re
import json
from langchain.document_loaders import UnstructuredHTMLLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from bs4 import BeautifulSoup
import shutil

class get_sources_10k:
    """
    A class for processing 10-K HTML documents and extracting, cleaning, and␣
 ↪chunking sections.
    """

    def __init__(self, resources_dir="resources"):
        self.resources_dir = resources_dir  # Directory containing source files
        self.metadata = {}  # Metadata dictionary for processed files
        # Define section titles and their categories
        self.items = {
            "item1": ["ITEM 1.", "BUSINESS"],
            "item1a": ["ITEM 1A.", "RISK FACTORS"],
            "item7": ["ITEM 7.", "MANAGEMENT'S DISCUSSION AND ANALYSIS OF␣
 ↪FINANCIAL CONDITION AND RESULTS OF OPERATIONS"],
            "income_statement": ["ITEM 15.", "EXHIBITS AND FINANCIAL STATEMENT␣
 ↪SCHEDULES"]
        }
        # Predefined list of section titles for indexing
        self.index_info = [
            "ITEM 1. BUSINESS", "ITEM 1A. RISK FACTORS", "ITEM 1B. UNRESOLVED␣
 ↪STAFF COMMENTS",
            "ITEM 2.", "ITEM 3.", "ITEM 4.", "ITEM 5.", "ITEM 6.",
            "ITEM 7. MANAGEMENT'S DISCUSSION AND ANALYSIS OF FINANCIAL␣
 ↪CONDITION AND RESULTS OF OPERATIONS",
            "ITEM 7A. QUANTITATIVE AND QUALITATIVE DISCLOSURES ABOUT MARKET␣
 ↪RISK", "ITEM 8.",
            "ITEM 9.", "ITEM 9A.", "ITEM 9B.", "ITEM 15. EXHIBITS AND FINANCIAL␣
 ↪STATEMENT SCHEDULES"
        ]

    def _construct_file_path(self, ticker, fiscal_year):
        """
```

```
        Constructs the file path for a given ticker and fiscal year.
        """
        return os.path.join(self.resources_dir, f"{ticker.
↪lower()}-{fiscal_year}.html")

    def _load_html(self, file_path):
        """
        Loads the HTML content of a file using LangChain's␣
↪UnstructuredHTMLLoader.
        """
        return UnstructuredHTMLLoader(file_path).load()

    def _extract_sections(self, html_content, section_titles):
        """
        Extracts sections from HTML content based on section titles.
        """
        soup = BeautifulSoup(html_content, "html.parser")
        text = soup.get_text(separator="\n").strip()

        # Compile regex patterns for section titles
        title_patterns = [re.escape(title) + r"(\s*continued)*" for title in␣
↪section_titles]
        combined_pattern = re.compile(r"|".join(title_patterns), re.IGNORECASE)

        # Locate matches and extract section text
        matches = list(combined_pattern.finditer(text))
        sections = {}
        for i, match in enumerate(matches):
            start_idx = match.start()
            end_idx = matches[i + 1].start() if i + 1 < len(matches) else␣
↪len(text)
            title = match.group(0)
            section_content = text[start_idx:end_idx].strip()

            if len(section_content.split()) > 5:  # Only keep meaningful␣
↪sections
                sections[title] = section_content
        return sections

    def _clean_sections(self, sections):
        """
        Cleans sections by removing overlapping text from subsequent section␣
↪titles.
        """
        cleaned_sections = {}
        for section_title, content in sections.items():
```

```
            # Find the current and next section index in the predefined list
            current_index = next((i for i, item in enumerate(self.index_info)␣
↪if item.lower().startswith(section_title.lower())), None)
            if current_index is not None:
                next_section = self.index_info[current_index + 1] if␣
↪current_index + 1 < len(self.index_info) else None
                if next_section:
                    # Remove overlapping text with the next section title
                    next_section_pattern = re.escape(next_section) +␣
↪r"(\s*continued)*"
                    next_section_match = re.search(next_section_pattern,␣
↪content, re.IGNORECASE)
                    if next_section_match:
                        content = content[:next_section_match.start()].strip()
                if content:
                    cleaned_sections[section_title] = content
        return cleaned_sections

    def _split_and_store_chunks(self, section_name, content, output_dir):
        """
        Splits a section's content into chunks and saves each chunk as a JSON␣
↪file.
        """
        splitter = RecursiveCharacterTextSplitter(chunk_size=2000,␣
↪chunk_overlap=150)
        chunks = splitter.split_text(content)

        for i, chunk in enumerate(chunks):
            output_file_path = os.path.join(output_dir,␣
↪f"{section_name}_chunk_{i + 1}.json")
            with open(output_file_path, 'w') as json_file:
                json.dump({"chunk": chunk, "chunk_index": i + 1}, json_file,␣
↪indent=4)
        return {"section": section_name, "chunk_count": len(chunks)}

    def process_file(self, files_metadata):
        """
        Processes a single file based on metadata, extracting and chunking␣
↪sections.
        """
        if not files_metadata or len(files_metadata) != 1:
            raise ValueError("files_metadata should contain exactly one item.")

        metadata = files_metadata[0]
        ticker, fiscal_year = metadata["ticker"], metadata["fiscal_year"]
```

```
        output_dir = f"main_{ticker}-{fiscal_year}" if ticker == "tsla" and␣
↪fiscal_year == 20231231 else f"sub_{ticker}-{fiscal_year}"
        self.file_dir = output_dir

        file_path = self._construct_file_path(ticker, fiscal_year)
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"File for {ticker} FY{fiscal_year} not␣
↪found.")

        documents = self._load_html(file_path)
        document_text = documents[0].page_content

        section_titles = [self.items[key][0] for key in self.items]
        extracted_sections = self._extract_sections(document_text,␣
↪section_titles)
        cleaned_sections = self._clean_sections(extracted_sections)

        os.makedirs(output_dir, exist_ok=True)
        results = {section: self._split_and_store_chunks(section, content,␣
↪output_dir) for section, content in cleaned_sections.items() if content}
        self.metadata[f"{ticker.upper()}_FY{fiscal_year}"] = {"file_path":␣
↪file_path, "sections_processed": list(results.keys())}
        return results

    def classify_files(self, files_metadata=None):
        """
        Classifies JSON files into directories based on section categories.
        """
        if not hasattr(self, 'file_dir'):
            raise ValueError("File directory not set. Ensure process_file is␣
↪run first.")

        categories = [item[0] for item in self.items.values()]
        summary = {"metadata": files_metadata if files_metadata else []}

        # Create category-specific directories
        for category in categories:
            category_dir = os.path.join(self.file_dir, category)
            os.makedirs(category_dir, exist_ok=True)
            summary[category] = {"chunk_count": 0}

        # Move files into their respective directories
        for filename in os.listdir(self.file_dir):
            if filename.endswith(".json"):
                for category in categories:
                    if filename.lower().startswith(category.lower()):
```

```
                        src_path = os.path.join(self.file_dir, filename)
                        dest_path = os.path.join(self.file_dir, category,␣
↪filename)
                        shutil.move(src_path, dest_path)
                        summary[category]["chunk_count"] += 1
                        break

        return json.dumps(summary, indent=4)
```

[ ]:
```
def TASK_1():
    print('='*80,'\nTask 1')
    print('='*80)
    # Example metadata for processing
    main_metadata = [{"ticker": "tsla", "fiscal_year": 20231231}]
    sub_metadata = [{"ticker": "tsla", "fiscal_year": 20221231}]

    chunk_processor = get_sources_10k()

    def generate_chunk(files_metadata):
        """
        Processes and classifies files, printing a summary for each file.
        """
        chunk_processor.process_file(files_metadata)
        summary = chunk_processor.classify_files(files_metadata)
        print(summary)

    generate_chunk(main_metadata)
    generate_chunk(sub_metadata)
TASK_1()
```

```
================================================================================
Task 1
================================================================================
{
    "metadata": [
        {
            "ticker": "tsla",
            "fiscal_year": 20231231
        }
    ],
    "ITEM 1.": {
        "chunk_count": 28
    },
    "ITEM 1A.": {
        "chunk_count": 54
    },
    "ITEM 7.": {
```

```
        "chunk_count": 31
    },
    "ITEM 15.": {
        "chunk_count": 42
    }
}
{
    "metadata": [
        {
            "ticker": "tsla",
            "fiscal_year": 20221231
        }
    ],
    "ITEM 1.": {
        "chunk_count": 25
    },
    "ITEM 1A.": {
        "chunk_count": 59
    },
    "ITEM 7.": {
        "chunk_count": 35
    },
    "ITEM 15.": {
        "chunk_count": 37
    }
}
```


# 6 Task 2

# 7 Pie Chart Generating Function with GPT-4o

- **Purpose**: Analyze financial data for segment performance, generate a pie chart, and save results in HTML format.
- **Key Features**:
  - **Document Loading**: Loads JSON documents containing financial data from a specified directory.
  - **LLM Analysis**: Extracts revenue or percentage data for predefined financial segments using OpenAI GPT-4o API.
  - **Pie Chart Generation**: Visualizes segment performance in a pie chart and saves it as a PNG image.
  - **HTML Export**: Embeds the pie chart into an HTML file with a Base64-encoded image for easy sharing.
  - **Error Handling**: Handles JSON parsing errors and provides detailed feedback during data extraction and visualization.
  - **Workflow**: Combines data loading, analysis, visualization, and output into a single streamlined process.

[ ]:
```
import openai
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import base64
from langchain.docstore.document import Document

class get_pie_chart:
    """
    A class to analyze financial data from JSON documents, extract segment␣
 ↪performance data,
    generate visualizations, and save the results in HTML format.
    """
    def __init__(self, api_key, legends, directory, ticker):
        """
        Initialize the SegmentAnalyzer with API key, legends, and the directory␣
 ↪of JSON files.

        :param legends: List of financial segments to analyze.
        :param directory: Directory containing JSON files.
        """
        openai.api_key = api_key
        self.legends = legends
        self.directory = directory
        self.ticker = ticker
        self.image_path=f"segment_performance_{self.ticker}.png"
        self.html_path=f"segment_performance_{self.ticker}.html"

    def load_json_files(self):
        """
        Load and preprocess JSON files from the directory.

        :return: List of Document objects with JSON content and metadata.
        """
        documents = []
        for filename in os.listdir(self.directory):
            if filename.endswith(".json"):
                with open(os.path.join(self.directory, filename), 'r') as file:
                    content = json.load(file)
                    documents.append(Document(page_content=json.dumps(content),␣
 ↪metadata={"source": filename}))
        return documents

    def analyze_with_llm(self, documents):
        """
        Use OpenAI's GPT model to extract relevant content based on predefined␣
 ↪legends.
```

```
        :param documents: List of Document objects.
        :return: Dictionary of segment performance data extracted by the LLM.
        """
        all_content = "\n\n".join([doc.page_content for doc in documents])
        prompt = f"""
You are a financial analyst. Based on the following content, extract revenue or␣
 ↪percentage information for the following segments:
{', '.join(self.legends)}.

If no data exists for a segment, exclude it from your response. Format your␣
 ↪response as JSON with each segment name as a key and its corresponding value␣
 ↪as a number.

Content:
{all_content}
"""
        response = openai.ChatCompletion.create(
            model="gpt-4o",
            messages=[{"role": "system", "content": "You are an expert␣
 ↪Financial assistant."}, {"role": "user", "content": prompt}],
        )
        content = response['choices'][0]['message']['content'].strip()

        # Ensure the response is valid JSON
        try:
            return json.loads(content)
        except json.JSONDecodeError:
            match = re.search(r"\{.*\}", content, re.DOTALL)
            if match:
                return json.loads(match.group(0))
            else:
                raise ValueError("Response is not in JSON format.")

    def pie_chart_png(self, segment_data, output_path):
        """
        Generate a pie chart visualizing segment performance and save it as an␣
 ↪image.

        :param segment_data: Dictionary of segment data to visualize.
        :param output_path: Path to save the pie chart image.
        """
        labels = list(segment_data.keys())
        sizes = list(segment_data.values())

        fig, ax = plt.subplots()
        wedges, texts, autotexts = ax.pie(
```

```
            sizes, labels=labels, autopct="%1.1f%%", startangle=140,␣
↪textprops=dict(color="white")
        )
        ax.legend(wedges, labels, title="Segments", loc="center left",␣
↪bbox_to_anchor=(1, 0, 0.5, 1))
        plt.setp(autotexts, size=10, weight="bold")
        ax.set_title("Segment Performance")

        plt.savefig(output_path, format="png", bbox_inches="tight")
        plt.show()
        plt.close()
        print(f"Pie chart saved to {output_path}")

    def save_html_with_chart(self, image_path, html_path):
        """
        Embed the pie chart in an HTML file and save it.

        :param image_path: Path to the pie chart image.
        :param html_path: Path to save the HTML file.
        """
        with open(image_path, "rb") as img_file:
            img_base64 = base64.b64encode(img_file.read()).decode("utf-8")

        html_snippet = f"""
        <html>
        <body>
            <h1>Segment Performance</h1>
            <img src="{img_base64}" alt="Pie Chart">
        </body>
        </html>
        """

        with open(html_path, "w") as file:
            file.write(html_snippet)
        print(f"HTML file saved to {html_path}")

    def generate_pie_chart(self):
        """
        Perform the entire analysis workflow: load documents, analyze data,␣
↪generate visualizations, and save results.

        :param image_path: Path to save the pie chart image.
        :param html_path: Path to save the HTML file.
        """
        # Load JSON documents
        documents = self.load_json_files()
        if not documents:
```

```
            print("No JSON files found in the directory.")
            return

        # Analyze using LLM
        try:
            segment_data = self.analyze_with_llm(documents)
            print("Segment data extracted:", segment_data)

            # Generate pie chart
            self.pie_chart_png(segment_data, self.image_path)

            # Save HTML with chart
            self.save_html_with_chart(self.image_path, self.html_path)

        except Exception as e:
            print("Error:", e)
```

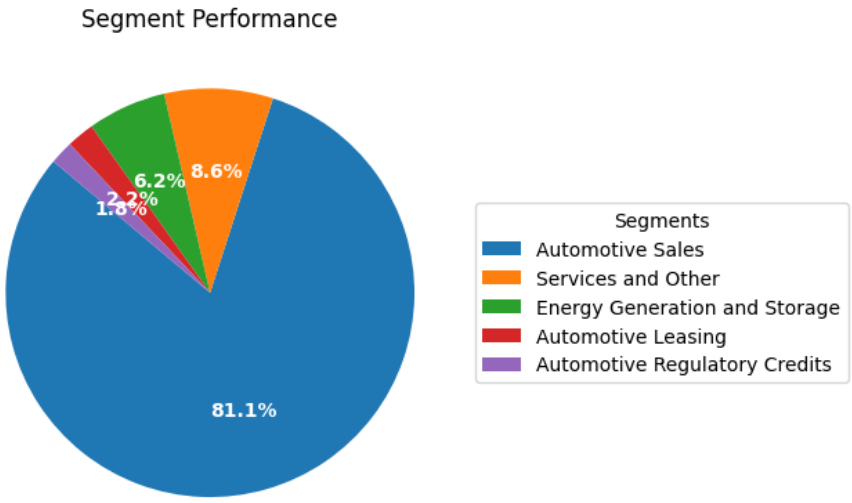
[ ]:
```
def TASK_2():
    api_key = API_KEY

    print('='*80,'\nTask 2')
    print('='*80)

    legends_pie = [
        "Automotive Sales",
        "Services and Other",
        "Energy Generation and Storage",
        "Automotive Leasing",
        "Automotive Regulatory Credits"
    ]
    directory = "./main_tsla-20231231/ITEM 7."

    pie_chart_png = get_pie_chart(api_key, legends_pie, directory, 'tsla')
    pie_chart_png.generate_pie_chart()
TASK_2()
```

```
================================================================================
Task 2
================================================================================
Segment data extracted: {'Automotive Sales': 78509, 'Services and Other': 8319,
'Energy Generation and Storage': 6035, 'Automotive Leasing': 2120, 'Automotive
Regulatory Credits': 1790}
```




```
Pie chart saved to segment_performance_tsla.png
HTML file saved to segment_performance_tsla.html
```

# 8 Task 3-1

# 9 Overview Text Generating Function

- **Purpose**: Generate a subtitle and concise overview for a company's primer using JSON data.
- **Key Features**:
  - **Document Reading**: Reads JSON files from specified folders and filters them based on given keywords (e.g., 'ITEM 1.', 'ITEM 1A.'). Includes error handling for missing folders or invalid JSON files.
  - **Content Combination**: Combines content from the 'chunk' field across all relevant documents into a single string for processing.
  - **Subtitle Generation**: Creates a one-line subtitle describing the company's identity and specialties using a structured OpenAI prompt.
  - **Overview Analysis**: Summarizes the company in a concise paragraph (5 sentences or fewer), detailing its identity, revenue generation, customer base, and business model.
  - **API Integration**: Uses OpenAI GPT-4o API for generating subtitles and overview content, ensuring contextually accurate and concise outputs.
  - **Output Organization**: Returns the generated subtitle and overview text for inclusion in a company primer.

[ ]:
```
class get_overview:
    def __init__(self):
        self.system_message = {"role": "system", "content": "You are an expert␣
↪Financial assistant."}
```

```
    def read_documents(self, folder_paths, filter_items):
        """
        Reads JSON files from the specified folder paths and filters them by␣
↪the filter_items.
        """
        documents = []
        for folder_path in folder_paths:
            self._process_folder(folder_path, filter_items, documents)
        return documents

    def _process_folder(self, folder_path, filter_items, documents):
        """
        Helper function to process each folder and filter files.
        """
        if not os.path.isdir(folder_path):
            print(f"Error: Folder path {folder_path} does not exist.")
            return

        for file in os.listdir(folder_path):
            if file.endswith('.json') and any(item.lower() in file.lower() for␣
↪item in filter_items):
                self._load_file(os.path.join(folder_path, file), documents)

    def _load_file(self, file_path, documents):
        """
        Helper function to load a JSON file and append its content if valid.
        """
        try:
            with open(file_path, 'r') as f:
                content = json.load(f)
                if 'chunk' not in content:
                    print(f"Warning: 'chunk' key missing in {os.path.
↪basename(file_path)}")
                documents.append(content)
        except Exception as e:
            print(f"Error loading file {os.path.basename(file_path)}: {e}")


    def analyze_overview(self, documents):
        """
        Generate the Overview section using text from ITEM 1 and ITEM 1A␣
↪sections.
        """
        all_content = self._combine_content(documents)

        subtitle_prompt = self._generate_subtitle_prompt(all_content)
        overview_prompt = self._generate_overview_prompt(all_content)
```

```
        subtitle = self._generate_response(subtitle_prompt)
        overview_text = self._generate_response(overview_prompt)

        return subtitle, overview_text

    def _combine_content(self, documents):
        """
        Combines the 'chunk' content from all documents into a single string.
        """
        return "\n\n".join(doc.get('chunk', '') for doc in documents)

    def _generate_subtitle_prompt(self, content):
        """
        Generates a prompt for creating the subtitle.
        """
        return f"""
        Based on the following content about a company, create a short one-line␣
↪subtitle (10 words or fewer)
        that describes the company's identity (e.g., "vehicle manufacturer")␣
↪and specialties
        (e.g., "renewable energy solutions" or "solar energy systems").

        Content:
        {content}
        """

    def _generate_overview_prompt(self, content):
        """
        Returns the overview prompt as is.
        """
        return f"""
        You are a financial analyst. Based on the following content, summarize␣
↪the company in a concise paragraph of 5 sentences or less.
        The paragraph should:
        1. Start with a one-liner explaining the company's identity and␣
↪specialties.
        2. Describe how the company generates revenue, who they sell to, and␣
↪the distribution of sales across their customer base.
        3. Provide key context about their business model. This is most␣
↪important.

        Content:
        {content}
        """
```

```
    def _generate_response(self, prompt):
        """
        Uses OpenAI API to generate a response for the given prompt.
        """
        response = openai.ChatCompletion.create(
            model="gpt-4o",
            messages=[self.system_message, {"role": "user", "content": prompt}],
        )
        return response['choices'][0]['message']['content'].strip()

    def generate_abstract(self, main_folder_paths):
        """
        Main function to generate the subtitle and overview content.
        """
        documents_main_overview = self.read_documents(main_folder_paths, ['ITEM␣
↪1.', 'ITEM 1A.'])

        subtitle, overview_content = self.
↪analyze_overview(documents_main_overview)

        return subtitle, overview_content
```

# 10 Task 3-2

# 11 Financial Tabular Data Generating Function

- **Purpose**: Extract, analyze, and merge financial metrics from JSON data for specific years, handling missing values iteratively.
- **Key Features**:
  - **Document Reading**: Reads and filters JSON files from specified folder paths based on keywords (e.g., 'ITEM 7.'). Handles errors for missing folders or invalid JSON files.
  - **Financial Data Analysis**: Extracts financial metrics (e.g., Revenue, Gross Profit) for specified years using OpenAI GPT-4o API. Includes calculation of averages for ranges, YoY changes, and margins.
  - **Data Merging**: Merges datasets by averaging overlapping values and tracking update counts to ensure accuracy.
  - **Missing Value Handling**: Identifies missing metrics and re-analyzes data iteratively until all missing values are resolved.
  - **Re-Analysis for Missing Data**: Groups missing metrics by years and re-requests only the missing data from GPT, improving efficiency.
  - **Main Loop for Missing Values**: Continuously checks and resolves missing data by re-analyzing documents until a complete dataset is achieved.
  - **Output Organization**: Returns a merged, comprehensive dataset with numeric financial metrics for multiple years.

[ ]:
```
class get_finance_value:
    def __init__(self):
        self.merged_data = {}

    def read_documents(self, folder_paths, filter_items):
        """
        Reads JSON files from the specified folder paths and filters them by␣
↪the filter_items.
        """
        documents = []
        for folder_path in folder_paths:
            print(f"Checking folder path: {folder_path}")
            if os.path.isdir(folder_path):
                files = os.listdir(folder_path)
                for file in files:
                    if file.endswith('.json') and any(item.lower() in file.
↪lower() for item in filter_items):
                        file_path = os.path.join(folder_path, file)
                        try:
                            with open(file_path, 'r') as f:
                                content = json.load(f)
                                if 'chunk' not in content:
                                    print(f"Warning: 'chunk' key missing in␣
↪{file}")
                                documents.append(content)
                        except Exception as e:
                            print(f"Error loading file {file}: {e}")
            else:
                print(f"Error: Folder path {folder_path} does not exist.")
        return documents

    def analyze_financials(self, documents, prompt_years):
        """
        Extract financial data for specific years from GPT's response.
        """
        all_content = "\n\n".join([doc.get('chunk', '') for doc in documents])

        if not all_content.strip():
            print("Error: No content provided to GPT for analysis.")
            return {}

        prompt = f"""
        You are a financial analyst. Based on the following content from␣
↪chunks, extract financial metrics for the years {', '.join(prompt_years)}.
        - For ranges (e.g., "$8.00 to $10.00 billion"), calculate the average␣
↪value.
```

```
        - If specific values for some years are missing, infer trends or use "N/
↪A" if absolutely necessary.
        - Calculate YoY (Year-over-Year) percentage changes for Revenue.
        - Calculate margin percentages for Gross Profit, Operating Profit, and␣
↪Net Profit where applicable.

        Instructions:
        1. Strictly return the output in JSON format.
        2. Do not include any additional explanations or commentary.
        3. Assume all dollar values are in billions unless stated otherwise.

        Example JSON format:
        {{
            "Revenue": {{ {', '.join(f'"{year}": <value>' for year in␣
↪prompt_years)} }},
            "Revenue's % yoy": {{ {', '.join(f'"{year}": <value>' for year in␣
↪prompt_years)} }},
            "Gross Profit": {{ {', '.join(f'"{year}": <value>' for year in␣
↪prompt_years)} }},
            "Gross Profit's % margin": {{ {', '.join(f'"{year}": <value>' for␣
↪year in prompt_years)} }},
            "Operating Profit": {{ {', '.join(f'"{year}": <value>' for year in␣
↪prompt_years)} }},
            "Operating Profit's % margin": {{ {', '.join(f'"{year}": <value>'␣
↪for year in prompt_years)} }},
            "Net Profit": {{ {', '.join(f'"{year}": <value>' for year in␣
↪prompt_years)} }},
            "Net Profit's % margin": {{ {', '.join(f'"{year}": <value>' for␣
↪year in prompt_years)} }}
        }}

        Filtered Content:
        {all_content}
        """

        try:
            response = openai.ChatCompletion.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": "You are an expert Financial␣
↪assistant."},
                    {"role": "user", "content": prompt}
                ],
            )
            content = response['choices'][0]['message']['content'].strip()
```

```
            # Attempt to parse JSON, but fallback gracefully if it fails
            try:
                response_json = json.loads(content)
                return response_json
            except json.JSONDecodeError:
                print("Warning: JSON decoding failed. Attempting partial␣
↪parsing.")

                # Extract only JSON-like substrings and return partial results
                start_idx = content.find('{')
                end_idx = content.rfind('}')
                if start_idx != -1 and end_idx != -1:
                    json_like_content = content[start_idx:end_idx + 1]
                    try:
                        partial_response_json = json.loads(json_like_content)
                        return partial_response_json
                    except json.JSONDecodeError:
                        print("Error: Unable to parse JSON-like content.␣
↪Returning empty result.")
                else:
                    print("Error: No valid JSON structure detected.")
                return {}
        except Exception as e:
            print(f"Unexpected GPT API error: {e}")
            return {}

    def merge_financial_data(self, main_data, sub_data):
        """
        Merge financial data by averaging values for overlapping years.
        Track the number of updates for each cell to calculate the final␣
↪average correctly.
        """
        all_keys = set(main_data.keys()).union(sub_data.keys())

        for key in all_keys:
            main_values = main_data.get(key, {})
            sub_values = sub_data.get(key, {})

            if key not in self.merged_data:
                self.merged_data[key] = {}  # Initialize merged data for this␣
↪key
                self.merged_data[key]['update_count'] = {}  # Track update␣
↪counts for each year

            merged_yearly_data = self.merged_data[key]  # Reference merged data␣
↪for this key
```

```
            for year in ["2021", "2022", "2023"]:
                main_val = main_values.get(year, "0")
                sub_val = sub_values.get(year, "0")

                try:
                    main_val = float(main_val) if main_val != "N/A" and␣
↪main_val is not None else 0.0
                except ValueError:
                    main_val = 0.0
                try:
                    sub_val = float(sub_val) if sub_val != "N/A" and sub_val is␣
↪not None else 0.0
                except ValueError:
                    sub_val = 0.0

                # Calculate new average if valid values exist
                if main_val != 0.0 or sub_val != 0.0:
                    current_value = merged_yearly_data.get(year, 0.0)
                    current_count = merged_yearly_data['update_count'].
↪get(year, 0)

                    # Update cumulative value and increment update count
                    new_value = current_value * current_count + main_val +␣
↪sub_val
                    new_count = current_count + (1 if main_val != 0 else 0) +␣
↪(1 if sub_val != 0 else 0)

                    # Store updated values
                    merged_yearly_data[year] = round(new_value / new_count, 3)
                    merged_yearly_data['update_count'][year] = new_count
                else:
                    # Keep existing value if no new data is available
                    merged_yearly_data[year] = merged_yearly_data.get(year, 0.0)

        return self.merged_data

    def find_missing_values(self, merged_data):
        """
        Identify metrics and years with missing values (i.e., 0 or 'N/A') in␣
↪the merged data.
        """
        missing_entries = []
        for metric, yearly_data in merged_data.items():
            for year, value in yearly_data.items():
                if value in (0, 'N/A'):
```

```
                    missing_entries.append((metric, year))
        return missing_entries

    def re_analyze_missing_data(self, documents, missing_entries):
        """
        Re-analyze missing financial data for specific metrics and years using␣
↪GPT.
        """
        all_content = "\n\n".join([doc.get('chunk', '') for doc in documents])

        if not all_content.strip():
            print("Error: No content provided to GPT for analysis.")
            return {}

        # Group missing entries by metrics
        metrics_by_year = {}
        for metric, year in missing_entries:
            if year not in metrics_by_year:
                metrics_by_year[year] = []
            metrics_by_year[year].append(metric)

        prompt = f"""
        You are a financial analyst. Based on the following content from␣
↪chunks, extract only the missing financial metrics for specific years.
        - For ranges (e.g., "$8.00 to $10.00 billion"), calculate the average␣
↪value.
        - If specific values for some years are missing, infer trends or use "N/
↪A" if absolutely necessary.
        - Calculate YoY (Year-over-Year) percentage changes for Revenue.
        - Calculate margin percentages for Gross Profit, Operating Profit, and␣
↪Net Profit where applicable.
        Focus exclusively on the following:
        {', '.join([f"{year}: {', '.join(metrics)}" for year, metrics in␣
↪metrics_by_year.items()])}.

        Instructions:
        1. Strictly return the output in JSON format.
        2. Do not include any additional explanations or commentary.
        3. Assume all dollar values are in billions unless stated otherwise.
        Example JSON format:
        {{
            {', '.join([f'"{metric}": {{"{year}": <value>}}' for year, metrics␣
↪in metrics_by_year.items() for metric in metrics])}
        }}

        Filtered Content:
```

```
        {all_content}
        """
        try:
            response = openai.ChatCompletion.create(
                model="gpt-4o",
                messages=[
                    {"role": "system", "content": "You are an expert financial␣
↪analyst."},
                    {"role": "user", "content": prompt}
                ],
            )
            content = response['choices'][0]['message']['content'].strip()
            print("Raw GPT Response Content:\n", content)

            # Parse JSON response
            try:
                response_json = json.loads(content)
                return response_json
            except json.JSONDecodeError:
                print("Error: Failed to parse JSON. Attempting partial recovery.
↪")
                # Extract JSON-like data
                start_idx = content.find('{')
                end_idx = content.rfind('}')
                if start_idx != -1 and end_idx != -1:
                    json_like_content = content[start_idx:end_idx + 1]
                    try:
                        partial_response_json = json.loads(json_like_content)
                        print("Partially Parsed JSON:\n", json.
↪dumps(partial_response_json, indent=4))
                        return partial_response_json
                    except json.JSONDecodeError:
                        print("Error: Unable to parse JSON-like content.")
                return {}
        except Exception as e:
            print(f"Unexpected GPT API error: {e}")
            return {}

    # Main loop for handling missing values
    def handle_missing_values(self, main_documents, sub_documents):
        """
        Continuously check for missing values and re-analyze until no missing␣
↪values remain.
        """
        iteration = 0
        while True:
            iteration += 1
```

```
            print(f"\n--- Iteration {iteration}: Checking for missing values␣
↪---")
            missing_entries = self.find_missing_values(self.merged_data)

            if not missing_entries:
                print("No missing values remaining. Analysis complete.")
                break

            print(f"Found missing values: {missing_entries}")

            # Determine which documents to use based on missing years
            main_years = [year for _, year in missing_entries if year in␣
↪["2022", "2023"]]
            sub_years = [year for _, year in missing_entries if year in␣
↪["2021", "2022"]]

            if main_years:
                print(f"Re-analyzing missing data for years {main_years} using␣
↪main documents.")
                new_data = self.re_analyze_missing_data(main_documents,␣
↪missing_entries)
                self.merged_data = self.merge_financial_data(self.merged_data,␣
↪new_data)

            if sub_years:
                print(f"Re-analyzing missing data for years {sub_years} using␣
↪sub documents.")
                new_data = self.re_analyze_missing_data(sub_documents,␣
↪missing_entries)
                self.merged_data = self.merge_financial_data(self.merged_data,␣
↪new_data)

        return self.merged_data


    def generate_tabular(self, main_paths, sub_paths):
        """
        Finially, generate numeric data of financial metrics. The integrated␣
↪function.
        """
        documents_main = self.read_documents(main_paths, ['ITEM 7.'])
        documents_sub = self.read_documents(sub_paths, ['ITEM 7.'])

        # Request 1: Analyze financials for 2022, 2023 from main paths
        data_2022_2023 = self.analyze_financials(documents_main, ["2022",␣
↪"2023"])
```

```
        # Request 2: Analyze financials for 2021, 2022 from sub paths
        data_2021_2022 = self.analyze_financials(documents_sub, ["2021",␣
↪"2022"])

        # Merge the two datasets
        _ = self.merge_financial_data(data_2022_2023, data_2021_2022)

        result = self.handle_missing_values(documents_main, documents_sub)

        return result
```

# 12 Task 3-3

# 13 Get Primer Document

- **Purpose**: Generate a company primer as a PNG image, including key financial metrics, an overview, and a pie chart.
- **Key Features**:
  - **Markdown & HTML Generation**: Creates a detailed primer in Markdown and HTML formats.
  - **PNG Conversion**: Uses `wkhtmltoimage` to produce a high-quality report.
  - **Financial Metrics & Charts**: Includes tables for Revenue, Profits, YoY changes, and a segment performance chart (if available).
  - **Error Handling**: Validates inputs and provides detailed error messages.
  - **Output**: Primer saved as `.png`, with `.md` and `.html` backups.

[ ]:
```
from IPython.display import Markdown
import subprocess

class get_primer:
    def __init__(self, subtitle, overview_content, finance_content,␣
↪pie_chart_path, ticker):
        self.subtitle = subtitle
        self.overview_content = overview_content
        self.finance_content = finance_content
        self.pie_chart_path = pie_chart_path
        self.ticker = ticker

    def _markdown(self):
        # Generate Markdown content
        md = f"# {self.ticker.upper()} primer\n\n"
        md += f"**{self.subtitle}**\n\n"
        md += "## Overview\n"
        md += f'- {self.overview_content}' + "\n"
```

```
        md += "## Financials\n"
        md += "| Metric($ million) | 2021   | 2022   | 2023   |\n"
        md += "|-------------------|--------|--------|--------|\n"

        metrics = [
            ("Revenue", "Revenue's % yoy"),
            ("Gross Profit", "Gross Profit's % margin"),
            ("Operating Profit", "Operating Profit's % margin"),
            ("Net Profit", "Net Profit's % margin")
        ]

        for metric, additional_key in metrics:
            metric_data = self.finance_content.get(metric, {})
            additional_data = self.finance_content.get(additional_key, {})
            md += f"| {metric}          | {metric_data.get('2021', 'N/A')} |␣
↪{metric_data.get('2022', 'N/A')} | {metric_data.get('2023', 'N/A')} |\n"
            md += f"| *{additional_key}* | {additional_data.get('2021', 'N/
↪A')}% | {additional_data.get('2022', 'N/A')}% | {additional_data.get('2023',␣
↪'N/A')}% |\n"

        if os.path.exists(self.pie_chart_path):
            md += f"\n## Segment Performance\n![Segment Performance Pie␣
↪Chart]({self.pie_chart_path})\n"
        else:
            md += "\n(Segment Performance chart not available.)\n"
        Markdown(md)
        markdown_output_file = f'./{self.ticker}_primer_report.md'
        with open(markdown_output_file, 'w') as f:
            f.write(md)

    def _html_to_png_with_wkhtmltoimage(self, html_file, output_path):
        try:
            options = [
            '--enable-local-file-access',  # Allow local file access for␣
↪embedded images
            '--format', 'png',
            '--width', '1240',  # Adjusted width for A4 (optimized for 96 DPI)
            '--height', '1300',  # Adjusted height for A4 (optimized for 96 DPI)
            '--crop-x', '0',  # Start cropping from the left edge
            '--crop-y', '0',  # Start cropping from the top edge
            '--crop-w', '1240',  # Crop width to match the page width
            '--crop-h', '1300',  # Crop height to match the page height
            '--disable-smart-width',  # Prevent automatic width adjustments
            '--quality', '100'  # High-quality output
            ]

            # Run the wkhtmltoimage command
```

```
            subprocess.run(
                ['wkhtmltoimage'] + options + [html_file, output_path],
                check=True
            )
            print(f"PNG successfully generated: {output_path}")
        except subprocess.CalledProcessError as e:
            print(f"Error occurred during image generation: {e}")
        except Exception as e:
            print(f"Unexpected error: {e}")


    def generate_primer_png(self):
        self._markdown()
        try:
            # Generate Markdown content
            primer = f"<h1>{self.ticker.upper()} Primer</h1>\n"
            primer += "<h3>Overview</h3>\n"
            primer += f"<h2>{self.subtitle}</h2>\n"
            overview_lines = self.overview_content.split("\n")  # Assuming␣
↪overview content has line breaks
            primer += "<ul>\n"  # Start an unordered list
            for line in overview_lines:
                primer += f"  <li>{line.strip()}</li>\n"  # Wrap each line in␣
↪an <li> tag
            primer += "</ul>\n"  # End the unordered list

            primer += "<h3>Financials</h3>\n"
            primer += """
            <table border="1" style="border-collapse: collapse; width: 100%;␣
↪text-align: center;">
                <tr>
                    <th>Metric($ million)</th>
                    <th>2021</th>
                    <th>2022</th>
                    <th>2023</th>
                </tr>
            """

            metrics = [
                ("Revenue", "Revenue's % yoy"),
                ("Gross Profit", "Gross Profit's % margin"),
                ("Operating Profit", "Operating Profit's % margin"),
                ("Net Profit", "Net Profit's % margin")
            ]

            for metric, additional_key in metrics:
                metric_data = self.finance_content.get(metric, {})
```

```
                additional_data = self.finance_content.get(additional_key, {})
                primer += f"""
                <tr>
                    <td>{metric}</td>
                    <td>{metric_data.get('2021', 'N/A')}</td>
                    <td>{metric_data.get('2022', 'N/A')}</td>
                    <td>{metric_data.get('2023', 'N/A')}</td>
                </tr>
                <tr>
                    <td><i>{additional_key}</i></td>
                    <td>{additional_data.get('2021', 'N/A')}%</td>
                    <td>{additional_data.get('2022', 'N/A')}%</td>
                    <td>{additional_data.get('2023', 'N/A')}%</td>
                </tr>
                """
            primer += "</table>\n"

            if os.path.exists(self.pie_chart_path):
                primer += f"<h3>Segment Performance</h3>\n<img src='{self.
↪pie_chart_path}' style='width: 100%;'>\n"
            else:
                primer += "<p>(Segment Performance chart not available.)</p>\n"

            # Save HTML content to a file
            html_file = f'./{self.ticker}_primer_report.html'
            with open(html_file, 'w') as f:
                f.write(primer)

            self._html_to_png_with_wkhtmltoimage(f'./{self.
↪ticker}_primer_report.html', f'./{self.ticker}-primer-report.png')
        except Exception as e:
            print(f"An error occurred: {e}")
```

[ ]:
```
def TASK_3():
    print('='*80,'\nTask 3')
    print('='*80)
    paths_overview = ['./main_tsla-20231231/ITEM 1.', './main_tsla-20231231/
↪ITEM 1A.']
    text_processor = get_overview()
    subtitle, overview_content = text_processor.
↪generate_abstract(paths_overview)

    main_paths = ['./main_tsla-20231231/ITEM 7.']
    sub_paths = ['./sub_tsla-20221231/ITEM 7.']
    tabular_processor = get_finance_value()
    finance_content = tabular_processor.generate_tabular(main_paths, sub_paths)
```

```
    pie_chart_path = './segment_performance_tsla.png'

    primer = get_primer(subtitle, overview_content, finance_content,␣
 ↪pie_chart_path, 'tsla')
    primer.generate_primer_png()
    img = mpimg.imread('./tsla-primer-report.png')
    plt.figure(figsize=(16, 12))
    plt.imshow(img)
    plt.axis('off')
    plt.show()
TASK_3()
```

```
================================================================================
Task 3
================================================================================
Checking folder path: ./main_tsla-20231231/ITEM 7.
Checking folder path: ./sub_tsla-20221231/ITEM 7.
Warning: JSON decoding failed. Attempting partial parsing.
Warning: JSON decoding failed. Attempting partial parsing.

--- Iteration 1: Checking for missing values ---
Found missing values: [("Revenue's % yoy", '2021'), ("Operating Profit's %
margin", '2021'), ('Operating Profit', '2021')]
Re-analyzing missing data for years ['2021', '2021', '2021'] using sub
documents.
Raw GPT Response Content:
 ```json
{
    "Revenue's % yoy": {"2021": 71},
    "Operating Profit's % margin": {"2021": 12.1},
    "Operating Profit": {"2021": 6.511}
}
```
Error: Failed to parse JSON. Attempting partial recovery.
Partially Parsed JSON:
 {
    "Revenue's % yoy": {
        "2021": 71
    },
    "Operating Profit's % margin": {
        "2021": 12.1
    },
    "Operating Profit": {
        "2021": 6.511
    }
}
```

```
--- Iteration 2: Checking for missing values ---
No missing values remaining. Analysis complete.

Loading page (1/2)
Rendering (2/2)
Done

PNG successfully generated: ./tsla-primer-report.png
```

# TSLA Primer

**Overview**

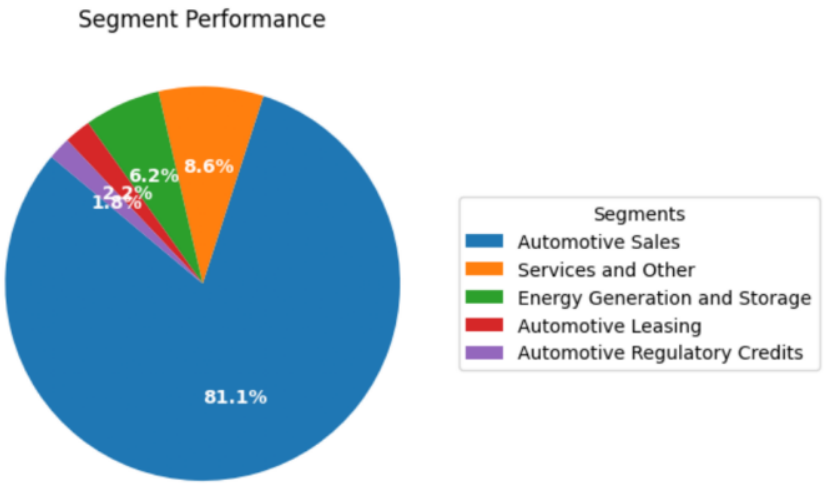
**"Electric Vehicle Innovator with Advanced Self-Driving and Energy Systems."**

- Tesla, Inc. is a leading electric vehicle and clean energy company known for its high-performance fully electric vehicles such as the Model S, Model X, Model 3, Model Y, and Cybertruck, as well as its energy generation and storage solutions. The company generates revenue primarily from vehicle sales, which are direct to consumers via its own website and stores, without traditional dealerships, and also from selling automotive regulatory credits and leasing. Furthermore, Tesla provides a wide array of services including vehicle maintenance, Supercharger access, insurance, and develops technologies like self-driving capabilities, alongside energy products through Powerwall and Megapack. Tesla's business model is vertically integrated, leveraging an extensive Supercharger network, software updates, and its own manufacturing facilities, or Gigafactories, to maintain cost efficiency and control over its supply chain, positioning itself to drive the growth of both the electric vehicle and renewable energy markets.

**Financials**

| Metric($ million) | 2021 | 2022 | 2023 |
|---|---|---|---|
| Revenue | 53.823 | 81.462 | 96.773 |
| *Revenue's % yoy* | 71.0% | 51.17% | 19.0% |
| Gross Profit | 13.606 | 20.853 | 17.66 |
| *Gross Profit's % margin* | 25.3% | 25.6% | 18.2% |
| Operating Profit | 6.511 | 13.142 | 8.697 |
| *Operating Profit's % margin* | 12.1% | 16.1% | 9.0% |
| Net Profit | 5.52 | 12.558 | 15.0 |
| *Net Profit's % margin* | 10.3% | 15.4% | 15.5% |

**Segment Performance**



# 14 Task 4-1 (GE)

# 15 HTML Source Generalization with ML Technique

- **Purpose**: Processes 10-K financial HTML reports by chunking text, and categorizing chunks into folders by unsupervised learning.
- **Key Features**:
  - **Initialization**: Sets resource directory and loads the `all-MiniLM-L6-v2` SentenceTransformer model for text embedding.
  - **Section Extraction**: Parses HTML, clusters text blocks with DBSCAN (after PCA), and organizes content into sections, including a "general" section.
  - **Dimensionality Reduction**: Uses PCA to reduce embedding dimensions for more efficient DBSCAN clustering.
  - **Chunking**: Splits text into JSON chunks with a `chunk_size=2000` and `chunk_overlap=150`.
  - **Output Organization**: Matches chunks to ITEM keywords (e.g., ITEM 1A) via cosine similarity and organizes them into categorized folders with renamed files.

[ ]:
```
from tqdm import tqdm
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from sentence_transformers import SentenceTransformer, util
from typing import List, Dict, Tuple

class get_sources_10k_re:
    def __init__(
        self,
        resources_dir: str = "resources",
        model_name: str = 'all-MiniLM-L6-v2',
        pca_components: int = 25,
        dbscan_eps: float = 0.5,
        dbscan_min_samples: int = 2
    ):
        """
        Initializes the 10-K processor class with specified configuration for␣
 ↪clustering and dimensionality reduction.
        """
        self.resources_dir = resources_dir
        self.model = SentenceTransformer(model_name)
        self.metadata = {}
        self.pca_components = pca_components
        self.dbscan_eps = dbscan_eps
        self.dbscan_min_samples = dbscan_min_samples

    def _extract_sections(self, document_text: str) -> Dict[str, str]:
        """
        Extracts sections from the document using clustering on text embeddings.
```

```
        """
        soup = BeautifulSoup(document_text, "html.parser")
        text_blocks = [tag.get_text(strip=True) for tag in soup.find_all(["p",␣
↪"li", "div"]) if tag.get_text(strip=True)]

        if not text_blocks:
            logging.warning("No text blocks found, returning entire document as␣
↪'general'.")
            return {"general": document_text.strip()}

        embeddings = self.model.encode(text_blocks, convert_to_tensor=True)
        pca = PCA(n_components=self.pca_components)
        reduced_embeddings = pca.fit_transform(embeddings.cpu().detach().
↪numpy())
        clustering = DBSCAN(eps=self.dbscan_eps, min_samples=self.
↪dbscan_min_samples).fit(reduced_embeddings)
        labels = clustering.labels_

        if all(label == -1 for label in labels):
            logging.warning("All blocks classified as noise by DBSCAN.␣
↪Returning entire text as 'general'.")
            return {"general": "\n".join(text_blocks)}

        clusters = {}
        for idx, label in enumerate(labels):
            if label not in clusters:
                clusters[label] = []
            clusters[label].append(text_blocks[idx])

        sections = {f"Cluster-{i}": "\n".join(content) for i, content in␣
↪clusters.items()}
        sections["general"] = "\n".join(clusters.get(max(clusters, key=lambda x:
↪ len(clusters[x])), []))
        return sections

    def _process_section(self, ticker: str, fiscal_year: int, section_name:␣
↪str, content: str, output_dir: str) -> Dict:
        """
        Splits a document section into chunks and saves them as JSON files.
        """
        splitter = RecursiveCharacterTextSplitter(chunk_size=2000,␣
↪chunk_overlap=150)
        chunks = splitter.split_text(content)
        chunk_files = []
        for i, chunk in enumerate(chunks):
```

```
            output_file_path = os.path.join(output_dir,␣
↪f"{section_name}_chunk_{i + 1}.json")
            with open(output_file_path, 'w', encoding='utf-8') as json_file:
                json.dump({"chunk": chunk, "chunk_index": i + 1}, json_file,␣
↪indent=4)
            chunk_files.append(output_file_path)
        logging.info(f"Processed {len(chunks)} chunks for section␣
↪'{section_name}'.")
        return {"section": section_name, "chunk_count": len(chunks),␣
↪"chunk_files": chunk_files}

    def _categorize_chunks(self, output_dir: str, chunk_files: List[str]):
        """
        Categorizes chunks into predefined sections using similarity scores.
        """
        keywords = {
            "ITEM 1.": "Description of Business",
            "ITEM 1A.": "Risk Factors",
            "ITEM 7.": "Management's Discussion and Analysis of Financial␣
↪Condition and Results of Operations"
        }
        keyword_embeddings = {key: self.model.encode(value,␣
↪convert_to_tensor=True) for key, value in keywords.items()}
        for key in keywords.keys():
            os.makedirs(os.path.join(output_dir, key), exist_ok=True)
        folder_file_counts = {key: 0 for key in keywords.keys()}

        for chunk_file in tqdm(chunk_files, desc="Categorizing chunks",␣
↪leave=False):  # Removed excessive progress bars
            with open(chunk_file, 'r', encoding='utf-8') as file:
                chunk_data = json.load(file)
                chunk_text = chunk_data["chunk"]
            chunk_embedding = self.model.encode(chunk_text,␣
↪convert_to_tensor=True)
            similarities = {key: util.pytorch_cos_sim(chunk_embedding, embed).
↪item() for key, embed in keyword_embeddings.items()}
            best_fit = max(similarities, key=similarities.get)
            folder_file_counts[best_fit] += 1
            new_file_name = f"{best_fit}_chunk_{folder_file_counts[best_fit]}.
↪json"
            shutil.move(chunk_file, os.path.join(output_dir, best_fit,␣
↪new_file_name))
        logging.info("Chunks categorized into respective sections.")
```

```
    def _load_and_process_file(self, ticker: str, fiscal_year: int) ->␣
↪Tuple[str, Dict]:
        """
        Loads an HTML file, extracts sections, and processes them into␣
↪categorized chunks.
        """
        file_path = os.path.join(self.resources_dir, f"{ticker.
↪lower()}-{fiscal_year}.html")
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"File for {ticker} FY{fiscal_year} not␣
↪found at {file_path}")

        document = UnstructuredHTMLLoader(file_path).load()[0]
        document_text = document.page_content
        extracted_sections = self._extract_sections(document_text)
        output_dir = f"chunk_{ticker}-{fiscal_year}"
        os.makedirs(output_dir, exist_ok=True)

        chunk_files = []
        results = {}
        for section, content in tqdm(extracted_sections.items(),␣
↪desc="Processing sections", leave=False):  # Single progress bar
            section_result = self._process_section(ticker, fiscal_year,␣
↪section, content, output_dir)
            results[section] = section_result
            chunk_files.extend(section_result["chunk_files"])
        self._categorize_chunks(output_dir, chunk_files)
        return file_path, results

    def process_file(self, files_metadata: List[Dict]) -> Dict:
        """
        Processes a single file described in the metadata, extracting sections␣
↪and saving processed chunks.
        """
        if not files_metadata or len(files_metadata) != 1:
            raise ValueError("files_metadata should contain exactly one item.")
        metadata = files_metadata[0]
        ticker, fiscal_year = metadata["ticker"], metadata["fiscal_year"]
        file_path, results = self._load_and_process_file(ticker, fiscal_year)
        self.metadata[f"{ticker.upper()}_FY{fiscal_year}"] = {"file_path":␣
↪file_path, "sections_processed": list(results.keys())}
        return results
```

[ ]:
```
def TASK_4():
    ge_data = [
        {"ticker": "ge", "fiscal_year": 20231231},
```

```
        {"ticker": "ge", "fiscal_year": 20221231}
    ]
    chunks = get_sources_10k_re()
    for ge in tqdm(ge_data, desc="Processing Files"):
        results = chunks.process_file([ge])
TASK_4()
```

## 16 Task 4-2

## 17 Robustness Check

[ ]:
```
if __name__ == "__main__":
    print('='*80,'\nTask 4')
    print('='*80)

    legends_pie = [
        "Aerospace and Defense Systems",
        "Renewable Energy Solutions",
        "Automotive Sales",
        "Automotive Leasing",
        "Power Generation Technologies"
    ]

    directory = "./chunk_ge-20231231/ITEM 7."

    pie_chart_png = get_pie_chart(API_KEY, legends_pie, directory, 'ge')
    pie_chart_png.generate_pie_chart()

    print('='*80,'\nGE Primer')
    print('='*80)
    paths_overview = ['./chunk_ge-20231231/ITEM 1.', './chunk_ge-20231231/ITEM␣
 ↪1A.']

    text_processor = get_overview()
    subtitle, overview_content = text_processor.
 ↪generate_abstract(paths_overview)

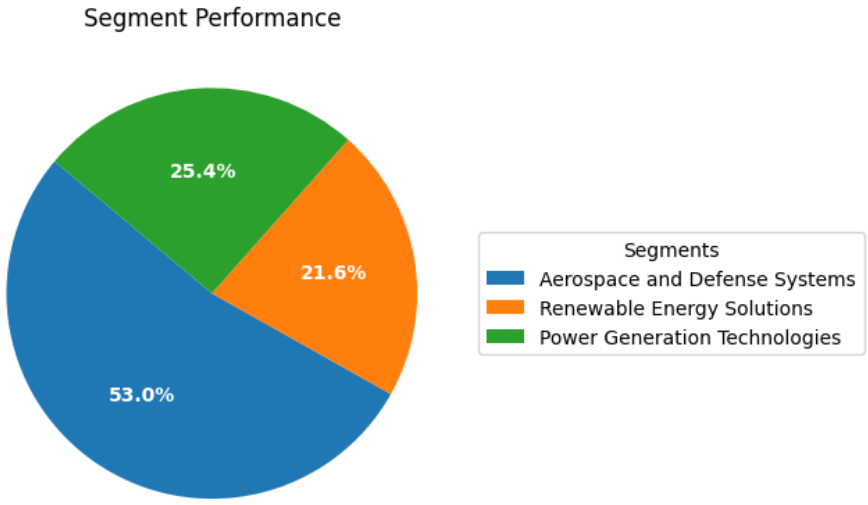
    main_paths = ['./chunk_ge-20231231/ITEM 7.']
    sub_paths = ['./chunk_ge-20221231/ITEM 7.']

    tabular_processor = get_finance_value()
    finance_content = tabular_processor.generate_tabular(main_paths, sub_paths)

    pie_chart_path = './segment_performance_ge.png'
```

```
    primer = get_primer(subtitle, overview_content, finance_content,␣
↪pie_chart_path, 'ge')
    primer.generate_primer_png()
    img = mpimg.imread('./ge-primer-report.png')
    plt.figure(figsize=(16, 12))
    plt.imshow(img)
    plt.axis('off')
    plt.show()
```

```
================================================================================
Task 4
================================================================================
Segment data extracted: {'Aerospace and Defense Systems': 36898, 'Renewable
Energy Solutions': 15050, 'Power Generation Technologies': 17731}
```




```
Pie chart saved to segment_performance_ge.png
HTML file saved to segment_performance_ge.html
================================================================================
GE Primer
================================================================================
Checking folder path: ./chunk_ge-20231231/ITEM 7.
Checking folder path: ./chunk_ge-20221231/ITEM 7.
Warning: JSON decoding failed. Attempting partial parsing.
Warning: JSON decoding failed. Attempting partial parsing.

--- Iteration 1: Checking for missing values ---
Found missing values: [('Gross Profit', '2023'), ("Revenue's % yoy", '2021'),
("Gross Profit's % margin", '2023')]
Re-analyzing missing data for years ['2023', '2023'] using main documents.
```

```
Raw GPT Response Content:
 ```json
{
    "Gross Profit": {"2023": "N/A"},
    "Gross Profit's % margin": {"2023": "N/A"},
    "Revenue's % yoy": {"2021": "-3.6%"}
}
```
Error: Failed to parse JSON. Attempting partial recovery.
Partially Parsed JSON:
 {
    "Gross Profit": {
        "2023": "N/A"
    },
    "Gross Profit's % margin": {
        "2023": "N/A"
    },
    "Revenue's % yoy": {
        "2021": "-3.6%"
    }
}
Re-analyzing missing data for years ['2021'] using sub documents.
Raw GPT Response Content:
 ```json
{
    "Gross Profit": {
        "2023": "N/A"
    },
    "Gross Profit's % margin": {
        "2023": "N/A"
    },
    "Revenue's % yoy": {
        "2021": -2.1
    }
}
```
Error: Failed to parse JSON. Attempting partial recovery.
Partially Parsed JSON:
 {
    "Gross Profit": {
        "2023": "N/A"
    },
    "Gross Profit's % margin": {
        "2023": "N/A"
    },
    "Revenue's % yoy": {
        "2021": -2.1
    }
```

```
}

--- Iteration 2: Checking for missing values ---
Found missing values: [('Gross Profit', '2023'), ("Gross Profit's % margin",
'2023')]
Re-analyzing missing data for years ['2023', '2023'] using main documents.
Raw GPT Response Content:
 ```json
{
    "Gross Profit": {"2023": 10.191},
    "Gross Profit's % margin": {"2023": 15.0}
}
```
Error: Failed to parse JSON. Attempting partial recovery.
Partially Parsed JSON:
 {
    "Gross Profit": {
        "2023": 10.191
    },
    "Gross Profit's % margin": {
        "2023": 15.0
    }
}

--- Iteration 3: Checking for missing values ---
No missing values remaining. Analysis complete.

huggingface/tokenizers: The current process just got forked, after parallelism
has already been used. Disabling parallelism to avoid deadlocks…
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(true |
false)
Loading page (1/2)
Rendering (2/2)
Done

PNG successfully generated: ./ge-primer-report.png
```

## GE Primer

**Overview**

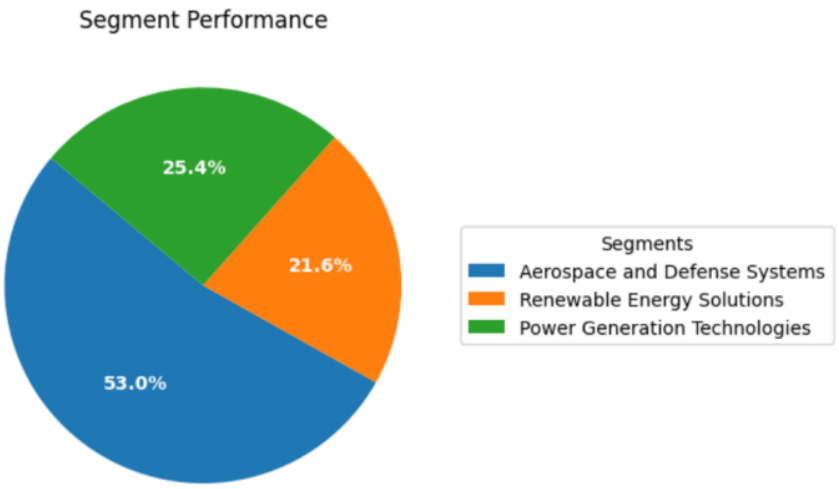
**Diverse industrial leader specializing in sustainable energy and innovation.**

- General Electric Company (GE) is a multifaceted industrial technology company specializing in sectors such as renewable energy, aerospace, and healthcare. GE generates revenue through manufacturing complex products like jet engines and wind turbines while offering associated services, selling to government and public sector entities globally across over 160 countries. The company emphasizes long-term service agreements to create enduring revenue streams, relying on continual technological advancement and innovation in its business model. GE faces significant competition and regulatory scrutiny, necessitating ongoing compliance with varied local and international laws and standards. As it navigates geopolitical tensions, economic conditions, and technological changes, GE is committed to sustainability, diversity, and integrity as key facets of its operational strategy.

**Financials**

| Metric($ million) | 2021 | 2022 | 2023 |
|---|---|---|---|
| Revenue | 74.196 | 67.328 | 68.0 |
| *Revenue's % yoy* | -2.1% | 3.18% | 17.05% |
| Gross Profit | 20.3 | 21.0 | 10.191 |
| *Gross Profit's % margin* | 27.37% | 27.44% | 15.0% |
| Operating Profit | 2.707 | 2.582 | 6.126 |
| *Operating Profit's % margin* | 3.65% | 4.15% | 9.01% |
| Net Profit | -6.757 | 0.138 | 9.481 |
| *Net Profit's % margin* | -9.1% | 0.248% | 13.95% |

**Segment Performance**




### 17.0.1 Analysis of Results from RAG-Driven Primer Generation for TSLA and GE

The **outcome of generating a primer for GE** revealed significant **runtime variability**, contrasting with the stable results observed for TSLA. This discrepancy stemmed from the **less rigorous handling of the document index during the chunking process**, despite employing a more efficient algorithm to address this issue. While this approach aimed to improve performance, the resulting JSON chunks for GE lacked consistency.

Nonetheless, upon manually reviewing the classified JSON file contents, there appears to be ample room for **further refinement and improvement**. This observation underscores the need for a more robust algorithm tailored to handle index structures effectively across diverse industrial domains, ensuring the reliability of the process.

The comparison between the Tesla (TSLA) and General Electric (GE) primers highlights several key insights into the functionality of the code:

1. The **financial analysis module** successfully processed both companies' distinct revenue models, correctly summarizing key financial metrics, including revenue growth, profit margins, and segment contributions.
2. However, the **segment performance pie chart for GE** reflects a broader diversity in revenue streams, compared to TSLA's predominantly automotive sales-dependent model. This demonstrates the system's flexibility in visualizing data for companies with varying operational structures.
3. Minor discrepancies in formatting and presentation suggest potential areas for improvement in **data alignment and visualization consistency** between primers.
4. The **narrative descriptions** effectively contextualize each company's financials within their respective industries, showcasing the system's capacity for domain-specific adaptability in language generation.
5. In summary, the system demonstrates high accuracy in extracting, visualizing, and summarizing structured data, but slight enhancements in cross-company consistency will ensure professional-quality outputs across applications.