11-12-24

Real-Time AI Sales Intelligence and Sentiment-Driven Deal Negotiation Assistant

Introduction

The "Real-Time AI Sales Intelligence and Sentiment-Driven Deal Negotiation Assistant" aims to enhance sales processes by leveraging AI-powered tools to analyze sentiment and generate dynamic responses. Additionally, it recommends optimal deal terms and provides post-call insights to improve outcomes. This system integrates advanced language models (LLMs) like OpenAI's GPT and Meta's LLaMA with tools such as CRM data and Google Sheets to improve conversion rates and sales team efficiency. The core functionality includes real-time sentiment monitoring, customized deal recommendations, and automated call summarization.

Research on LLMs and Sequencing

1. Large Language Models (LLMs)

What are LLMs?

Large Language Models (LLMs) are deep learning models trained on massive amounts of text data to generate human-like language. They can perform a wide range of tasks, including text generation, sentiment analysis, summarization, and translation. LLMs leverage transformers as their underlying architecture.

Different types of LLMs are designed to address various tasks based on their strengths. For example, some are optimized for text generation, while others excel at classification or summarization. Understanding these distinctions can help in selecting the right model for specific applications.

Types of LLMs:

1. **GPT Series (Generative Pre-trained Transformer)**: Developed by OpenAI, these models excel at generating coherent and context-aware text.
2. **BERT (Bidirectional Encoder Representations from Transformers)**: Ideal for tasks like sentiment classification, question answering, and intent recognition.
3. **Meta LLaMA (Large Language Model Meta AI)**: Open-source models designed for research and commercial applications.
4. **T5 (Text-to-Text Transfer Transformer)**: Converts all NLP tasks into a text-to-text format, useful for summarization and text generation.
5. **DistilBERT**: A lightweight version of BERT for faster inference while maintaining performance.

Applications of LLMs in This Project:

- **Sentiment Analysis**: To understand buyer sentiment during calls.
- **Dynamic Text Generation**: To provide real-time negotiation responses.
- **Call Summarization**: To generate concise summaries post-call.

---

2. Sequential Tasks and LLMs Based on Sequencing

Understanding "Sequencing" in LLMs

Sequencing refers to handling data with a time-based or ordered structure. For example, in a sales call, sequencing helps track the flow of conversation from an introduction to objections and finally to closing the deal, allowing the AI to respond appropriately at each stage. In the context of this project, sequencing involves analyzing conversation flows during sales calls. Key tasks include tracking sentiment, generating context-aware replies, and summarizing interactions.

Sequential Tasks for This Project

1. **Real-Time Sentiment Analysis**:

    - **Goal**: Monitor emotional shifts during live calls and adapt the negotiation strategy accordingly.
    - **Approach**: Utilize models like BERT or DistilBERT for classifying sentiment (positive, neutral, negative).
2. **Text Generation for Dynamic Negotiation**:
    - **Goal**: Generate real-time responses that are context-aware and tailored to the buyer's needs.
    - **Approach**: Use GPT models to generate responses based on conversation history.
3. **Call Summarization**:
    - **Goal**: Create concise summaries of sales calls for post-call analysis and follow-ups.
    - **Approach**: Use summarization models like T5 or PEGASUS to distill key insights from the call.

Structure and Function of a Transformer

Overview of the Transformer Architecture

Transformers are the backbone of LLMs and excel at handling sequential tasks without relying on recurrent networks. Unlike recurrent networks, which process data sequentially and can struggle with long-term dependencies, transformers can handle entire sequences simultaneously. This parallelization improves efficiency and allows transformers to capture long-range dependencies more effectively. They consist of encoder and decoder components and utilize self-attention mechanisms to process input data efficiently.

Key Components:

1. **Encoder**: Processes the input sequence and produces contextual representations.
2. **Decoder**: Generates output based on the encoder's representations.
3. **Self-Attention Mechanism**: Allows the model to weigh the importance of different words in a sequence, regardless of their position.

4. **Positional Encoding**: Adds information about the sequence order since transformers lack an inherent sense of time.

Attention Mechanism

The attention mechanism calculates the importance of each word relative to others in a sequence. It involves three key vectors:

- **Query (Q)**
- **Key (K)**
- **Value (V)**

The attention score is calculated as follows:

Transformer Architecture Components

Left Side: Encoder The encoder processes the input data and converts it into a meaningful representation for the decoder to utilize.

1. **Input Embedding:**
    - Converts input tokens (e.g., words) into dense numerical vectors for processing.
2. **Positional Encoding:**
    - Adds information about the token's position in the sequence, since transformers process sequences in parallel and don't inherently understand the order of tokens.
3. **Multi-Head Attention:**
    - Computes relationships between all tokens in the input sequence.
    - Allows the model to focus on different parts of the sequence simultaneously.
4. **Feed Forward Layer:**
    - Applies a fully connected neural network to each token's representation to extract non-linear features.
5. **Add & Norm:**
    - A residual connection adds the input to the output of the preceding layer.
    - Normalizes the values for better training stability.

6. **Stacking (N×):**
    - The encoder consists of multiple identical layers, which are stacked to capture complex relationships.

Right Side: Decoder The decoder generates the output sequence step-by-step while incorporating information from the encoder.

1. **Output Embedding:**
    - Converts the output tokens (e.g., already generated words) into dense numerical vectors.
2. **Positional Encoding:**
    - Adds sequence order information for the output.
3. **Masked Multi-Head Attention:**
    - Similar to the encoder's attention but masks future tokens to ensure predictions depend only on past tokens.
4. **Multi-Head Attention (with Encoder-Decoder Attention):**
    - Attends to the encoder's output, enabling the decoder to focus on relevant parts of the input sequence.
5. **Feed Forward Layer:**
    - Processes the token representations with a fully connected network.
6. **Add & Norm:**
    - As with the encoder, residual connections and normalization are applied.
7. **Softmax & Linear:**
    - A linear layer maps the decoder's output to the vocabulary size.
    - The softmax function computes probabilities for the next token in the sequence.
8. **Stacking (N×):**
    - Like the encoder, the decoder consists of multiple identical layers.

Key Processes in Both Encoder and Decoder:

- **Attention Mechanism:**
  - Allows the model to focus on important parts of the sequence.
  - Self-attention computes relationships within the sequence (encoder or decoder), while encoder-decoder attention links input and output sequences.
- **Feed Forward:**
  - Ensures non-linear transformations and captures complex patterns.

Use of Transformers in This Project:

- **Sentiment Analysis**: The attention mechanism helps identify emotional cues throughout the conversation.
- **Dynamic Recommendations**: By maintaining context, the transformer can provide tailored deal suggestions.
- **Summarization**: The attention mechanism helps distill key points from long conversations.

Hugging Face: Tools and Frameworks

What is Hugging Face?

Hugging Face is a popular platform for building, training, and deploying NLP models. It provides:

- **Transformers Library**: Pre-trained models for tasks like sentiment analysis, text generation, and summarization.
- **Datasets Library**: Ready-to-use datasets for training models.
- **Model Hub**: A repository for thousands of pre-trained models.

Using Hugging Face for This Project

1. **Sentiment Analysis**:
   - **Model**: `distilbert-base-uncased-finetuned-sst-2-english`

2. **Text Generation**:
    - **Model**: `gpt2`
3. **Summarization**:
    - **Model**: `facebook/bart-large-cnn`
4. **Integration with CRM and Google Sheets**:
    - Use the **Google Sheets API** to store and retrieve data.
    - Update deal statuses and track performance insights seamlessly.

Conclusion

This project leverages state-of-the-art LLMs and transformer-based models to enhance real-time sales intelligence. By integrating tools like Hugging Face, OpenAI's GPT, and Google Sheets, the assistant provides sentiment-driven negotiation strategies, dynamic deal recommendations, and post-call insights. This AI-powered approach aims to improve conversion rates, streamline negotiation processes, and enhance the overall efficiency of sales teams.

16-12-2024

Research on Summarization of a Call Using Facebook's BERT/SAMSum Model and Voice-to-Text Models

---

Introduction

The goal of this research is to enable real-time transcription and summarization of sales calls or conversations. The key components of this process include:

1. Converting voice data from calls into text using advanced voice-to-text models.
2. Summarizing the transcribed text into concise, actionable insights using Facebook's BERT and SAMSum-based summarization models.

This document outlines the research conducted on the summarization and voice-to-text models, their relevance, and the methods for integrating them.

Voice-to-Text Model Research

1. OpenAI Whisper

- **Overview**:
  - OpenAI Whisper is a state-of-the-art speech recognition model designed for high accuracy and multilingual transcription.
  - It excels in handling real-world noise, varying accents, and overlapping speech.
- **Features**:
  - Multilingual transcription capabilities.
  - Can operate offline if the model is run locally.
  - High accuracy for complex audio environments (e.g., noisy sales calls).
- **Implementation**:
  - Open-source and easy to integrate using Python.
  - Example Usage:

```
import whisper
import pyaudio
import wave
import os
import time

# Load the Whisper model
model = whisper.load_model("base")

# Audio stream settings
FORMAT = pyaudio.paInt16
```

```
CHANNELS = 1
RATE = 16000
CHUNK = 1024
RECORD_SECONDS = 5  # Duration to listen for each segment

# Create a temporary directory for audio files
temp_dir = "temp_audio"
os.makedirs(temp_dir, exist_ok=True)

# Initialize PyAudio
audio = pyaudio.PyAudio()

# Start the audio stream
stream = audio.open(format=FORMAT, channels=CHANNELS, rate=RATE,
input=True, frames_per_buffer=CHUNK)

print("Starting the speech-to-text service. Press Ctrl+C to stop.")

try:
    while True:
        frames = []
        print(f"Listening for {RECORD_SECONDS} seconds...")

        # Record for a specified duration
        for _ in range(0, int(RATE / CHUNK * RECORD_SECONDS)):
            data = stream.read(CHUNK)
            frames.append(data)

        # Save the recorded frames to a temporary WAV file
        temp_file_path = os.path.join(temp_dir, "temp_audio.wav")
        with wave.open(temp_file_path, 'wb') as wav_file:
            wav_file.setnchannels(CHANNELS)
            wav_file.setsampwidth(2)  # 16-bit audio
            wav_file.setframerate(RATE)
            wav_file.writeframes(b''.join(frames))

        # Transcribe the audio file
        print("Processing the speech...")
        result = model.transcribe(temp_file_path)
        print(f"Transcribed text: {result['text']}\n")

except KeyboardInterrupt:
    print("Speech-to-text service stopped.")

finally:
    # Stop and close the audio stream
    stream.stop_stream()
    stream.close()
```

```
audio.terminate()
```

- **Advantages**:
  - No need for an internet connection when run locally.
  - Handles diverse audio conditions effectively.

2. Google Speech-to-Text API

- **Overview**:
  - Google's Speech-to-Text API provides real-time transcription capabilities via cloud services.
  - Offers speaker diarization (identifying who said what).
- **Features**:
  - Real-time streaming.
  - High accuracy with domain-specific adaptation.
- **Implementation**:
  - Requires internet connectivity.
  - Example Usage:

```
transcription = recognizer.recognize_google(audio)
```

- **Advantages**:
  - Seamless integration with Google's ecosystem.
  - Scalable and suitable for enterprise-grade solutions.

Comparison

Feature	OpenAI Whisper	Google Speech-to-Text
Accuracy	High	High
Multilingual Support	Yes	Limited
Real-Time Capabilities	Yes (with local setup)	Yes (via API)
Offline Functionality	Yes	No
Cost	Free (self-hosted)	Paid (API usage)

Call Summarization Model Research

1. Facebook's BERT

- **Overview**:
  - BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based NLP model developed by Facebook.
  - Pre-trained on large datasets and fine-tuned for specific tasks like text classification, sentiment analysis, and summarization.
- **Features for Summarization**:
  - Extractive summarization: Selects key sentences or phrases from the text.
  - Handles long-form content effectively.
- **Fine-Tuning for Summarization**:
  - BERT can be fine-tuned on domain-specific datasets to adapt to sales or conversational scenarios.
  - Example Process:

- Preprocess data to match the input format.
- Train on annotated datasets for summarization tasks.

2. SAMSum Dataset

- **Overview**:
  - SAMSum is a dataset specifically created for conversational summarization.
  - Includes real-world chat data with summaries, making it ideal for call transcription.
- **Model Fine-Tuned on SAMSum**:
  - Models like `facebook/bart-large` are pre-trained on SAMSum for abstractive summarization.
- **Implementation**:
  - Hugging Face's Transformers library provides pre-trained BART models.
  - Example Usage:

```
from transformers import BartTokenizer, BartForConditionalGeneration
model_name = "philschmid/bart-large-cnn-samsum" # Trained on SAMSum dataset
tokenizer = BartTokenizer.from_pretrained(model_name)
summarizer = BartForConditionalGeneration.from_pretrained(model_name)

def chunk_text_with_tokenizer(text, max_tokens=1000):
    """
    Splits text into chunks respecting the model's max token limit.

    Args:
        text (str): The input text to split.
        max_tokens (int): Maximum tokens per chunk.

    Returns:
        list: List of text chunks.
    """
    tokens = tokenizer(text, truncation=False,
return_tensors="pt")["input_ids"][0]
    chunks = [tokens[i:i + max_tokens] for i in range(0, len(tokens),
max_tokens)]
    return [tokenizer.decode(chunk, skip_special_tokens=True) for chunk in
chunks]

def summarize_large_text(text, max_chunk_size=1000, max_length=250,
min_length=30):
    """
    Summarizes a large text by processing it in chunks.

    Args:
        text (str): The input text to summarize.
        max_chunk_size (int): Maximum size of each text chunk (in tokens).
        max_length (int): Maximum length of each chunk summary.
        min_length (int): Minimum length of each chunk summary.
```

```
    Returns:
        str: The final summary.
    """
    # Split the text into tokenized chunks
    chunks = chunk_text_with_tokenizer(text, max_tokens=max_chunk_size)

    # Summarize each chunk
    chunk_summaries = []
    for idx, chunk in enumerate(chunks):
        print(f"Summarizing chunk {idx + 1}/{len(chunks)}")
        inputs = tokenizer(chunk, return_tensors="pt",
max_length=max_chunk_size, truncation=True)
        summary_ids = summarizer.generate(
            inputs["input_ids"],
            max_length=max_length,
            min_length=min_length,
            length_penalty=2.0,
            num_beams=4,
            early_stopping=True
        )
        summary_text = tokenizer.decode(summary_ids[0],
skip_special_tokens=True)
        chunk_summaries.append(summary_text)

    # Combine all chunk summaries into a final summary
    final_summary = " ".join(chunk_summaries)

    return final_summary

if __name__ == "__main__":
    # Example long transcript
    user_text = """
SR: Good morning, Alex! Thanks for taking the time to speak with me today. How
are you doing?

CL: Good morning, Jamie. I’m doing well, thank you. Let’s get started—I have
about 30 minutes.

SR: Absolutely! I appreciate your time. I wanted to discuss how our project
management software can help streamline your team’s workflows and improve
collaboration. Could you share some of the challenges your team is currently
facing?

CL: Sure. We’re having issues with task tracking and deadline management. Our
current tools don’t integrate well with each other, so our team ends up
duplicating efforts.
```

SR: I understand. That’s a common pain point. Our solution is designed to centralize project tracking, integrate seamlessly with tools like Slack and Google Workspace, and provide real-time updates to keep everyone aligned. How many people are on your team?

CL: We have about 50 people spread across three departments.

SR: That’s great. Our software is built to scale and works well for teams of your size. If I may ask, how are you currently managing inter-departmental projects?

CL: Honestly, it’s chaotic. Different departments use different tools, and we often lose track of who’s responsible for what.

SR: That can be frustrating. One of our features, the centralized dashboard, helps assign tasks across departments while maintaining visibility for everyone involved. Would you like me to demonstrate how it works?

CL: That sounds interesting, but I’m not sure if it’s different from what we’re already using.

SR: Great point! Let me highlight a key differentiator—our software uses AI to analyze project progress and predict potential bottlenecks. This helps managers address issues proactively. For instance, if a team is running behind, the system sends alerts with actionable recommendations.

CL: Hmm, that could be useful. How long does it take to implement?

SR: Implementation typically takes two weeks, depending on your current setup. Our team provides full onboarding support, including training sessions for your staff.

CL: Two weeks is reasonable. But what about cost? Budget is a big concern for us right now.

SR: I completely understand. Our pricing is flexible based on the number of users and features you choose. For a team of 50, the cost would be around $1,500 per month, which includes all integrations, updates, and support.

CL: That’s a bit higher than what we’re paying now.

SR: I hear you. May I ask how much you’re spending currently?

CL: We’re spending about $1,200 per month across three different tools.

SR: Got it. While $1,500 is slightly higher, it consolidates those tools into one platform, saving time and improving efficiency. Additionally, the AI

```
features can significantly reduce delays, which often cost more than the
software itself.

CL: I'll need to discuss it with my team.

SR: Of course! Would it help if I provided a custom proposal detailing how our
solution aligns with your current processes and cost breakdowns?

CL: Yes, that would be helpful.

SR: Perfect. I'll email the proposal by tomorrow morning. Would you be
available for a follow-up call next week to review it?

CL: Sure, let's schedule something for Tuesday afternoon.

SR: Great! Thanks for your time, Alex. I'll follow up with the proposal, and I
look forward to our chat on Tuesday. Have a wonderful day!

CL: Thank you, Jamie. Talk soon.
    """

    # Chunk and summarize
    chunks = chunk_text_with_tokenizer(user_text)
    print(f"Number of chunks: {len(chunks)}")
    for idx, chunk in enumerate(chunks):
        token_count = len(tokenizer(chunk, truncation=False,
return_tensors="pt")["input_ids"][0])
        print(f"Chunk {idx + 1}: Length: {token_count} tokens")

    summary = summarize_large_text(user_text)
    print("Final Summary:", summary)
```

- **Advantages**:
  - Abstractive summarization generates human-like summaries.
  - Tailored for conversational data.

Comparison

Feature	Facebook BERT	BART Fine-Tuned on SAMSum
Summarization Type	Extractive	Abstractive
Use Case	Formal documents	Conversational data
Pre-training Dataset	Generic text corpora	SAMSum dataset
Output Style	Key sentences	Human-like summaries

Proposed Workflow

1. **Voice-to-Text Conversion**:
    - Capture live audio during calls using a service like Twilio or a VoIP provider.
    - Use OpenAI Whisper for transcription if offline or Google STT for cloud-based real-time transcription.
2. **Summarization**:
    - Feed the transcribed text into a summarization model (e.g., `facebook/bart-large-samsum`).
    - Generate concise summaries to assist in post-call analysis or automated note-taking.
3. **Integration**:
    - Combine the transcription and summarization pipelines into a seamless workflow.
    - Deploy the solution as a microservice using Flask or FastAPI for scalability.

Challenges and Mitigations

1. **Accuracy**:
    - Ensure clear audio input to improve transcription accuracy.
    - Fine-tune summarization models for domain-specific language.
2. **Latency**:
    - Optimize model pipelines for faster inference.
    - Use batching where real-time processing is not critical.
3. **Scalability**:
    - Deploy on cloud infrastructure with load balancing for handling concurrent calls.
4. **Privacy and Compliance**:
    - Encrypt audio and text data to ensure privacy.
    - Comply with GDPR and other regional laws for call recording and processing.

Conclusion

This research highlights a robust pipeline for converting voice to text and summarizing conversations using advanced models. OpenAI Whisper and Google Speech-to-Text provide reliable transcription capabilities, while Facebook's BERT and SAMSum-based models excel in summarizing conversational data. The proposed integration ensures a scalable and efficient solution for real-time call processing.

Future work will focus on:

- Fine-tuning models further for domain-specific applications.
- Reducing latency for real-time use cases.

17-12-2024

Understanding Prompt Engineering and Its Application with LLMs

What is Prompt Engineering?

Prompt engineering is the process of designing and optimizing input prompts to achieve desired outputs from large language models (LLMs). Since LLMs generate responses based on the input they receive, the structure, tone, and content of the prompt play a significant role in determining the quality and relevance of the output.

A well-engineered prompt provides the model with clear instructions, context, and constraints, enabling it to perform specific tasks efficiently. Prompt engineering is an essential skill for leveraging LLMs effectively across various domains, including content creation, software development, education, and more.

Key Principles of Prompt Engineering

1. **Clarity and Specificity**: The prompt should clearly articulate the task and expectations. Vague prompts often lead to ambiguous or irrelevant responses.
2. **Contextual Information**: Providing relevant context helps the model understand the nuances of the task, resulting in more accurate outputs.
3. **Instructional Prompts**: Direct instructions like "Write a summary of this text" or "Generate Python code for this task" guide the model effectively.
4. **Iterative Refinement**: Prompts can be refined iteratively to improve outcomes. Testing variations helps identify the most effective format.
5. **Constraints and Examples**: Including constraints (e.g., word count limits, tone requirements) and examples within the prompt can significantly enhance the model's performance.

Applications of Prompt Engineering with LLMs

1. **Content Generation**:
    - Writing articles, blogs, and creative pieces.

- Generating marketing copy and social media posts.
- Drafting emails and professional documents.

2. **Code Assistance**:
   - Debugging and generating code snippets.
   - Explaining programming concepts.
   - Writing algorithms or boilerplate code.
3. **Education and Learning**:
   - Summarizing articles and textbooks.
   - Creating practice questions and quizzes.
   - Explaining complex topics in simple language.
4. **Data Analysis**:
   - Generating SQL queries.
   - Explaining trends in datasets.
   - Drafting reports based on data insights.
5. **Customer Support and Interaction**:
   - Responding to FAQs.
   - Providing step-by-step troubleshooting guidance.
   - Analyzing sentiment in customer reviews.

Free LLMs for Prompt Engineering

Several LLMs are available for free, providing opportunities for experimentation with prompt engineering. Below is a list of notable free LLMs and their key features:

1. **OpenAI's GPT-3.5 (via Free Tier on OpenAI API)**:
   - Offers conversational and generative capabilities.
   - Free tier usage is limited but sufficient for testing.
   - Requires clear, concise prompts for optimal performance.
2. **Hugging Face Models**:
   - Includes a wide range of open-source LLMs like BLOOM, Falcon, and GPT-Neo.
   - Customizable and hosted on Hugging Face's platform.
   - Ideal for research and building tailored applications.
3. **Google's BERT and T5**:
   - Focused on natural language understanding and generation.

- Free to use with TensorFlow or PyTorch implementations.

4. **Cohere's Command R**:
    - Optimized for retrieval-augmented generation.
    - Offers free-tier usage for specific tasks like summarization and content generation.
5. **Meta's LLaMA**:
    - Lightweight and efficient for resource-constrained environments.
    - Open-source models available for customization.
6. **EleutherAI's GPT-NeoX and GPT-J**:
    - Open-source alternatives to OpenAI's GPT models.
    - Suitable for a variety of generative tasks.
7. **Anthropic's Claude (via Free Tier)**:
    - Focused on safe and ethical AI interactions.
    - Available for limited free usage through partnerships or API trials.
8. **Azure OpenAI (Free Tier)**:
    - Provides access to GPT-4 and GPT-3.5 with limited usage under the free tier.
    - Integrated with Microsoft's ecosystem for additional functionalities.

Customizing LLMs with Prompting

Prompt engineering allows users to adapt LLMs to specific use cases without retraining the model. Here are some strategies for customization:

1. **Zero-shot Prompting**:
    - Providing a direct instruction without examples.
    - Example: "Summarize the following text in three sentences."
2. **Few-shot Prompting**:
    - Including examples within the prompt to guide the model.
    - Example: "Translate the following sentences into French:
        1. Hello, how are you?
        2. Good morning, everyone."
3. **Chain-of-Thought Prompting**:
    - Encouraging the model to reason through tasks step-by-step.
    - Example: "Explain why the following statement is true or false with reasoning."

4. **Role-Based Prompting**:
    - Assigning roles to the model for task-specific outputs.
    - Example: "Act as a teacher explaining Newton’s laws to a 10-year-old."

Conclusion

Prompt engineering is a powerful technique for maximizing the utility of LLMs. By understanding how to structure and refine prompts, users can harness the full potential of these models across various applications. With numerous free LLMs available, individuals and organizations can explore and innovate without significant financial investment.