

UNplugged.AI: An Offline-First Multi-Modal Local LLM Workbench with Integrated Emotion Analytics

Rounak Dey

Department of AI

FET.Jain Deemed-to-be University

Bengaluru-562112

rounakriju2003@gmail.com

Janaki kandasamy

Department of CSE (AI)

FET.Jain Deemed-to-be University

Bengaluru-562112

k.janaki@jainuniversity.ac.in

Abstract—Many services around the cloud-based use of large language models (LLMs), such as concerns about user privacy, internet dependency, and breaking apart or separating workflows among different tools and services, have received special attention. We introduce a fully-designed, implemented, and benchmarked AI workbench, dubbed as UNplugged.ai, based on Streamlit and integrated with the runtime Ollama, serving as a standalone multi-modal offline-first application for AI workloads to model inference and execution on the local machine and in runtime. Howe's system allows for various modalities of interaction, such as free-form conversational chatting over a single interface or document-assisted question-answering for PDF / DOCX, raw text ingestion, or image-based prompting, with voice-based interaction. Document processing and document inferences are always done locally on the host machine and the sensitive content remains on-device throughout the interaction journey. The platform also has an emotion classification module from scikit-learn to classify the emotions on the fly based on the user prompt and/or based on the model's response, then visualizing the emotion probability distributions as an interactive bar chart using Altair. Document grounding relies on a direct method of "context stuffing" without the necessity of embedding or having a vector database. In conclusion, the experimental analysis has proven that the platform can serve as a viable alternative to cloud-based AI productivity applications with equivalent performance

in new areas of voice input, image prompting and context scaling while also revealing individual areas where further enhancements can be made. It addresses architectural design, fast construction, a modality specific pipeline, performance challenges, known limitations to date and opportunities for improvement. The section above contained the following key terms: Offline LLM, Ollama, Streamlit, multimodal prompting, document question-answering, privacy-preserving AI, emotion classification, context injection, local inference, human-AI interaction, speech recognition, sentiment analysis, edge deployment.

Keywords—Probably Phrases— Offline LLM, Ollama, Streamlit, Multimodal Prompting, Document Q&A, Privacy-Preserving AI, Emotion Classification, Context Injection, Local Inference, Human-AI Interaction, Speech Recognition, Sentiment Analysis, Edge Deployment

I. INTRODUCTION

Large language models have given new meaning to the concept of access in the era of conversational AI. Large language models have rewritten the rules of the game when it comes to conversational AI's access. They have shown enough coverage and complexity to be practically used to deduce knowledge, summary documents, code generation, and creativity aiding in various professional and personal processes. Dominant architectures for these systems, however, involve the

taking to a cloud inference system of user queries, documents and conversational histories for processing. The pattern is non-trivial in terms of privacy issues, regulatory compliance problems to some sensitive domains, and is also very dependent on network availability and makes such tools ineffective in operational settings with limited bandwidth, with no Internet connectivity at all. There are unfortunately non-trivial privacy concerns and regulatory compliance issues to some sensitive domains in this pattern, it also has a hard dependency on network availability and makes such tools ineffective in operational settings with limited bandwidth, no Internet connectivity at all. In the last year, open weight models have made great steps forward, while local inference runtime has also improved significantly, allowing for the deployment of powerful language and vision models on commodity hardware, without relying on the cloud. Some frameworks like Ollama [2] have significantly reduced the deployment cost for local use of models by packaging quantization, memory management, and model hardware acceleration behind a standard API that's compatible with common patterns of HTTP and python based clients. However, the consumer AI tooling landscape is lacking a seamless way of integrating various forms of interaction in a single locally-executable application. Yet the greater the consumer tooling ecosystem has failed to align its superficially multiform interface to an ability to integrate various forms of interaction in a single execution unit [3]. The existing multi-modal AI workflows that are spread across individual cloud services create an extra friction; the user performs an image description, a conversational reasoning, and a PDF description in three different services, which results in data duplication, coordination overhead and user data is exposed to multiple service providers. Toward this vision, the present contributions aims to write a unified, deployable workbench in the Streamlit rapid application development framework [4] that is capable of orchestrating a locally-running Ollama backend across six different modalities for interactions.

A. Objectives

To achieve this objective in this study, it is designed and implemented a multi-modal Streamlit interface that switches between conversational, document, image, and voice input types to the locally-hosted Ollama inference backend and constructed the modality-specific context

ingestion pipelines designed for PDF, DOCX, TXT, and image input formats; a pre-trained scikit-learn emotion classification overlay is developed and integrated; and the resulting system is evaluated with respect to the tradeoffs of the architecture, privacy guarantees, performance characteristics and operational limitations.

B. Principal Contributions

This is the main contribution of this work: (i) It allows for using local LLM runtime, namely Ollama, while adding also the following modalities “chat”, “PDF Q&A”, “DOCX Q&A”, “text ingestion”, “image prompting” and “voice interaction” to a single AI workbench. (iii) Modular tab based design where a distinct modality would be loaded during the session, while the context (e.g. chat history, numbers, texts entered by the end-user) would be persistent even after changing the modality for the next session. (e.g. by storing it in a runtime data-store like Streamlit's `st.session_state`) If the input sentence is a prompt construction frame intended for a particular field, stuff it directly. If the input sentence is a prompt construction frame for a specific domain, embed the frame rightly with a corresponding domain specific wrapper. (iv) Interactions with the emotion Classifier from the scikit-learn library powered by text serialization, which will return emotion class predictions along with emotion class probability distributions per text, in the form of interactive Altair bar charts. (v) A systematic description of privacy guarantees, limitations and architectural tradeoffs for the system.

II. RELATED WORK

A. Local Inference Runtimes and Open Weight Model Ecosystems

LLMs have been released using two licenses for which an environment of local inference runtimes has sprung up around them side by side. Large Language Models, which are released under a permissive license, have spurred a parallel ecosystem of local inference runtimes. By utilizing quantization, it has been shown that multi-billion parameter models can operate at interactive response rates without requiring GPUs, running them on regular C/C++ inference code supported by the efficient llama.cpp library developed for consumer CPUs [5]. Ollama [2] offers an integrated model server with REST

API support that includes local model management, which can be used with families such as Llama, Mistral, Phi, DeepSeek, Qwen, and others. Both LM Studio and GPT4All take initial steps towards developing a user friendly interface for interacting with local models, but they are mostly chat modality interfaces so far with limited extensibility. LM Studio and GPT4All are parallel projects that aim to create user friendly frontends for the interaction with local models, but given that, thus far, they rely primarily on a single modality “chat”, with no programmatic extensibility.

B. Rapid Application Development Frameworks for AI Interfaces

The Streamlit framework [4] is currently the most popular platform of choice for those looking to make new web applications with the ability to integrate based on AI and data. In the current Python Programmer's Lifestyle, Streamlit is the most popular Web Application Construction Application being used to speedily develop new Web Applications based on AI and Data. It uses a declarative model, meaning it defines widgets in each state without being conflated with the rest of the other states; it has automatic rerun semantics when the state changes, meaning it determines when to rerun when anything in the system changes state; it also has a built in support for session-scoped state persistence via `st.session_state`, which makes it viable for implementing stateful conversational interfaces [7] out of the box. Still another way is the one taken by Gradio [8] which offers abstractions on the component level defined to meet the requirements of model demonstration scenarios. Previous research that developed LLM based Chat Interfaces on Streamlit has shown the viability of multi-turn conversation management using session state's message history, albeit with less studies on integrating multi-modal context pipelines and local inference backends.

C. Document Question Answering and Context Injection

The retrieval-augmented generation (RAG) approach [9] has received extensive attention the last few years as a document-grounded Q&A model that splits up documents into chunks, embeds them and looks them up at query time if they contain semantically related passages. There are some popular abstractions on top of

the LLM backends to build RAG pipelines: LangChain [10] and LlamaIndex [11]. Embedding models, utilizing vector databases and integrating indexing pipelines, however, introduces a significant degree of complexity and cost to infrastructure in RAG architectures. A simple approach, that has few disadvantages, is the direct-context injection; the primary disadvantage being a limited context window, which can be limited [12].

D. Sentiment and Emotion Analysis in Human AI Interaction

The research on affective computing has shown that emotion-aware interfaces can improve the user experience by revealing in-depth emotion dimensions of man-machine interactions [13]. The classification on emotions in the context of texts has been explored on both classical methods like logistic regression pipelines over the TF-IDF features [14] and modern methods like transformer based models. Emotion classification overlays are still a field of application research on conversational AI systems that can be used in user experience monitoring, in generating adaptive responses, or in assessing the quality of interactions [15].

III. SYSTEM ARCHITECTURE

A. Overview

The suggested architecture is basically made of five main layers, kind of forming a processing stack that is run per each user action, you know: (1) the Streamlit UI layer that's in charge of showing modality-specific tabs and also rendering the results that come from annotating the model output; (2) the Context Ingestion layer, which does modality-specific operations for reading, processing, and encoding the input data and all that; (3) the Prompt Construction layer where the request is assembled from a user query together with the document context; (4) the Local Inference Backend layer that takes care of delivering the messages payload to the Ollama daemon and (5) the Emotion Analytics layer that ends up doing classification plus visualization of the results.

TABLE I. End-to-End Processing Pipeline

Layer	Component	Output
1	Streamlit UI (app.py + tabs/*.py)	User input, uploaded file bytes
2	Context Ingestion (PyPDF2, python-docx)	Extracted text / encoded image string
3	Prompt Builder (prepare_context_prompt)	Context-wrapped messages[] payload
4	Ollama Python client (ollama.chat)	LLM response text + timing
5	Emotion Classifier (scikit-learn + Altair)	Emotion label, confidence, chart

B. Runtime Session State Model

The application uses Streamlits dictionary `st.session_state` to provide a runtime data store for the application. This will allow it to save the application's transient state during user interaction that exists across one or multiple sessions within the same browser window (i.e., local). All session state is divided into three categories based on their function. State associated with conversational and contextual buffers includes a list of message role-content dictionaries created for every turn, and four content buffer fields which are stored as plain string values: `pdf_text`, `docx_text`, `text_file_text`, and `manual_text`. These fields contain the document content that has been extrated from the documents. An additional field, `image_data`, stores the base64 encoded PNG version of any images that have been loaded onto the system [16]. State associated with voice interaction is managed using two Boolean flags and a single string value, `status_message`. State associated with model selection is tracked using three

fields: `selected_model`, `selected_variation_size`, and `ollama_list_cache`.

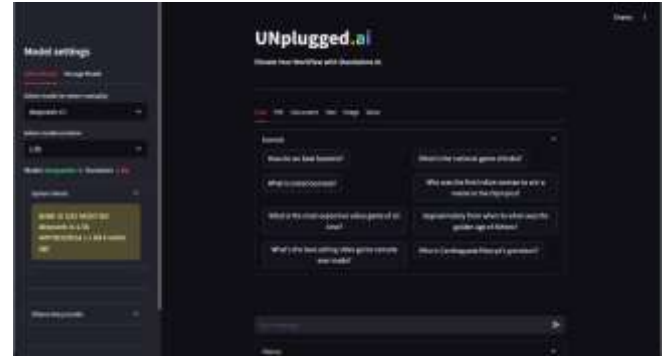


Fig. 1. System output: Dashboard allows users to interact with a locally hosted AI model and access features such as chat, document analysis, image processing, and voice assistance.

C. Model Selection and Identifier Resolution

The application provides an inventory of open-weight models that include permitted size variation suffixes for model prefixes as well as includes some common weight classes such as DeepSeek-R1, Llama 3.2, Llama 3.3, Phi-4, Mistral, Qwen 2.5, Qwen 2.5-Coder and LLaVA in sizes from 500 million to 671 billion parameters. The fully qualified name of the model is created by combining the selected prefix or family ID with the selected variation or size ID string using the Ollama model tag syntax. Before executing the model (i.e., before calling the model's inference function) the application optionally checks if the resolved model ID exists in the cached list of available models produced during execution of the ollama command. If the user has requested use of a non-curated model, he can pass in a completely arbitrary identifier string that will be used instead of a valid model ID.

TABLE II. Curated Model Families and Size Configurations

Model Family Prefix	Available Size Variations
deepseek-r1:	1.5b, 7b, 8b, 14b, 32b, 70b, 671b
llama3.2:	1b, 3b
llama3.2-	11b, 90b

vision:	
mistral:	7b
phi4:	14b
qwen2.5:	0.5b, 1.5b, 3b, 7b, 14b, 32b, 72b
llava:	7b, 13b, 34b

IV. METHODOLOGY

A Modality-specific context ingestion pipelines There is one ingestion pipeline per modality (6 total) that are each defined in their respective tab modules (as tabs/) under the /package directory.

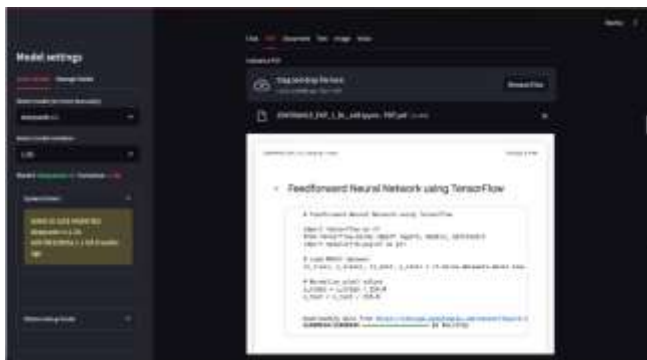


Fig. 2. System output: The platform enables PDF uploads and displays document content for AI assisted reading and analysis, as shown with notebook.

The ingestion pipeline for PDF utilizes PyPDF2.PdfReader to loop through every single page on the uploaded file; it takes the text from pdf_reader.getPage().extractText(); then stores this data into the pdf_text session state buffer; finally an inline base64 encoded iframe preview is created within the tab [17].

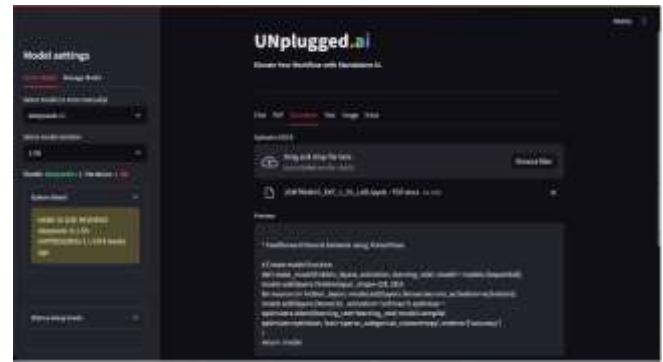


Fig. 3. System output: The user uploads a DOCX file, and the AI previews the document content for analysis or processing.

The DOCX pipeline uses the python-docx library to iterate over paragraph objects, with the text properties of each paragraph joined with newline delimiters into the docx_text buffer. The Text pipeline has two sources of input, a file uploader for .txt files and a manual textarea widget. The resulting context is the concatenation of sources available.

B. Prompt Construction and Context Injection

The prompt construction layer uses a simple context-stuffing approach, avoiding embedding-based retrieval in favor of directly injecting entire documents into the model's context window. The function prepare_context_prompt takes a user query string and a context_type identifier as input and returns a context-wrapped prompt string using a modality-specific template. In the case of the PDF context, the wrapper concatenates the pdf_text buffer with a fixed preamble that instructs the model to answer the question using the content of the PDF given. The context types DOCX and Text use similar templates. If you want to provide image context, you can embed the image_data string (base64-encoded image data) in the prompt body. This works depending on Ollama model you use, and whether it understands base64-encoded image data as part of the prompt text.

Every user prompt and Bot. gets added to the messages list stored in session state.

Stored in session state, messages are appended to the messages list after every user prompt and assistant response. Each of these appends takes the form of a dict with role/content keys. Because conversation history is passed along for every subsequent request after the first

in a session, later requests in the conversation have access to all previous messages in their context. This allows for multiple-turn conversations with injected documents, but also means that effective prompt length increases cumulatively throughout a session. Eventually, this may cause it to reach the limits of the context window if the injected document is large or the conversation has many turns [12].

C. Emotion Classification Module

The emotion analytics module is implemented in sentiment/emotion.py and provides two primary functions: predict_emotions, which returns the highest-probability emotion label for an input text string, and get_prediction_proba, which returns the full probability distribution over recognized emotion classes as a NumPy array. Both functions operate over a scikit-learn pipeline serialized to disk as text_emotion.pkl and loaded at module initialization via joblib. The emotion label set includes classes such as anger, joy, and neutral, each mapped to a

The Image pipeline converts the uploaded file to PNG-format bytes in memory and produces a base64-encoded string stored in image_data. The Voice pipeline employs pyttsx3 for text-to-speech output and the SpeechRecognition library for speech-to-text input.

corresponding emoji character in a visualization dictionary [14].



Fig. 4. System output: The system processes a user query, predicts the associated emotion with confidence scores, visualizes the results in a chart, and generates an AI response.

The analyze_sentiment function in app.py handles the entire classification and visualization pipeline: it calls

predict_emotions and get_prediction_proba, calculates the confidence score as the max class probability, reloads the serialized model again to access the classes_ attribute for creating class labels, creates a pandas DataFrame with columns for emotion and probability, and displays the results as an Altair horizontal bar chart. By default, sentiment visualization is enabled for the Chat tab and disabled for document Q&A context tabs to avoid analytically irrelevant predictions over extracted document content [13].

D. Inference Backend Integration

Calling ollama sends inference through the ollama client library for Python.chat with the resolved model id. messages list. The function blocks synchronously until the Ollama daemon returns the complete response, then the response text is extracted by inspecting the response dict structure and trying multiple known access patterns to work around differences across Ollama client library versions. The wall-clock latency is measured by taking timestamps before and after the ollama.chat call, and formatted as a minutes:seconds string displayed to the user. A simulated typing animation is added to the rendered response by chunking the response text in 20 character chunks and progressively concatenating them with a 100 ms delay [2].

V. SYSTEM IMPLEMENTATION

A. Development Environment and Dependencies

The full application is written in Python 3.8 or later. The core stack of dependencies is: Streamlit for the web application framework and session state management; the Ollama Python client library for communication with the local inference daemon; PyPDF2 for extracting text from PDFs; python-docx for extracting paragraphs from DOCX files; Pillow for converting image formats; pyttsx3 for offline text-to-speech synthesis;



Fig. 5. System output: The Voice tab is open, showing voice interaction settings where users can adjust speech speed and start voice commands.

- SpeechRecognition for speech to text transcription
- Pandas and Numpy for tabular data building
- Altair for declarative interactive chart rendering
- Scikit-learn for the emotion classification pipeline
- Joblib for model serialization and deserialization [4][8]

B. Source Code Organization

The repository is divided into four main filesystem locations. The app.py application entry point (at the repository root) contains session state initialization, sidebar model management controls, the analyze_sentiment orchestration function, and the send_prompt inference dispatch function. There are 6 module files in the tabs/ directory. The sentiment/ subdirectory contains the emotion.py and the pre-trained text_emotion.pkl model artifact. The complete source code map is shown in Table III.

TABLE III. Source Code Organization and Component Responsibilities

Module / File	Responsibility
app.py	Session init, sidebar, send_prompt, analyze_sentiment
tabs/chat_tab.py	Free-form chat UI, example prompt buttons
tabs/pdf_tab.py	PDF upload, PyPDF2 extraction, Q&A
tabs/document_tab.py	DOCX upload, python-docx extraction, Q&A
tabs/text_tab.py	TXT upload + manual

	textarea, context Q&A
tabs/image_tab.py	Image upload, base64 encoding, prompting
tabs/voice_tab.py	STT (SpeechRecognition), TTS (pyttsx3)
sentiment/emotion.py	predict_emotions, get_prediction_proba

C. Core Interaction Loop

The central coordination function send_prompt handles the whole interaction lifecycle for one user turn. The function first checks that a model identifier has been selected and, if the Ollama list cache exists, checks that the identifier is in the cache. If a non-null context_type argument is passed in, the function calls prepare_context_prompt to wrap the user query with the right document context. The wrapped prompt is appended to the messages list as a user role entry. If sentiment analysis is enabled, analyze_sentiment is called on the unwrapped user query and the chart is displayed. The law .Send chat call with current messages list , extract response text , and perform animated rendering . The assistant response is then added to the list of messages.

VI. RESULTS AND DISCUSSION

A. Privacy and Offline Capability Assessment

The system works really well offline for its functions like understanding what people mean and processing documents. When you have the Ollama model, on your computer it can do its job without sending any information to servers. It can look at PDF, DOCX and TXT files. Get the information it needs from them without ever sending the files anywhere else. This is because it uses tools that can look at the files right on your computer.

The system can also turn text into speech using a tool called pyttsx3. It does this on your computer too.. There is one thing that does not work offline: when you talk to the system it sends what you say to Googles servers so they can figure out what you mean. This means that you need to be connected to the internet for this part to work.

B. Performance and Compute Considerations

System response time mainly depends on three things:

- * the size of the model chosen
- * the total number of tokens, in the message sent to Ollama
- *. The hardware setup of the computer.

Smaller models, around 1 to 3 billion parameters usually give answers in a seconds on todays CPU.

Larger models in the 70 billion parameter range need a GPU to respond quickly.

The way we handle context for questions and answers adds to the delay.

As we chat and add more conversation history the prompt gets longer.

This increases the time it takes to process. Also risks exceeding the models limit on how much it can handle.

TABLE IV. System Performance and Privacy Characteristics by Modality

Modality	Offline	Data Retained Locally	Scalability Risk
Chat	Yes	Session only	Low
PDF Q&A	Yes	Session only	High (large docs)
DOCX Q&A	Yes	Session only	High (large docs)
Text	Yes	Session only	Medium
Image	Partial	Session only	Medium
Voice TTS	Yes	N/A	Low
Voice STT	No	Sent to Google	Low

C. Known Limitations

Five clear limitations of the system need to be shared.

First the voice input part is not fully offline.

The recognize_google function sends audio to a service. This goes against the privacy promises for voice input.

* Second there is an issue with the image part.

A NameError happens when running the image tab module.

This is because Image.open is used without importing Image from the Pillow library.

This problem needs to be fixed before images can work properly.

Thirld the way base64 images are prompted is not ideal.

The image data is added as text in the string.

It would be better, as an attachment.

This might affect how well the Ollama model works.

Fourth there is no document chunking and retrieval.

This means long documents will not be handled well.

The model can only understand much text at a time.

Users are not told if the response is not good.

Fifth the emotion classifier model is provided without details.

There is no model card, training data description or bias information.

This limits how much we can trust the models results.

VII. CONCLUSION

This paper is about UNplugged.ai, which's a special kind of computer program that lets you use artificial intelligence without being connected to the internet. It is a -modal AI workbench that combines six different ways of interacting with the program into one application. This application uses a thing called Streamlit. It works with a locally-running Ollama inference daemon.

The good thing about this system is that it shows we can make intelligence tools that protect your privacy and run on your own computer. We do not need to use cloud computers or special databases to make it work. We used a way to make the program understand what you are talking about which is good for documents that are not too long. This works well if your computer is fast enough to handle it.

There are some things that we need to make. We want to replace the part that recognizes your speech with something that can run on your computer. We also need to fix a problem with the image part of the program.. We want to make it so the program can handle longer documents.

In the future we want to add features to the program. We want to make it so you can attach pictures to your messages and the program can understand them. We also want to make it so you can save your conversations and come back to them later.. We want to do a study to see if people like using this program better, than the ones that run on big cloud computers.

REFERENCES

- [1] A. Narayanan and V. Shmatikov, "Robust De-anonymization of Large Datasets (How to Break Anonymity of the Netflix Prize Dataset)," in Proc. IEEE Symp. Security and Privacy, Oakland, CA, 2008, pp. 111–125.
- [2] Ollama, "Ollama: Get up and running with large language models locally," 2023. [Online]. Available: <https://ollama.com>
- [3] T. B. Brown et al., "Language Models are Few-Shot Learners," Advances in Neural Information Processing Systems (NeurIPS), vol. 33, pp. 1877–1901, 2020.
- [4] A. Streamlit Inc., "Streamlit: The fastest way to build and share data apps," 2019. [Online]. Available: <https://streamlit.io>
- [5] G. Gerganov, "llama.cpp: LLM inference in C/C++," GitHub repository, 2023. [Online]. Available: <https://github.com/ggerganov/llama.cpp>
- [6] Y. Chang et al., "A Survey on Evaluation of Large Language Models," ACM Trans. Intelligent Systems and Technology, vol. 15, no. 3, pp. 1–45, 2024.
- [7] R. Taori et al., "Alpaca: A Strong, Replicable Instruction-Following Model," Stanford CRFM Blog, 2023. [Online]. Available: <https://crfm.stanford.edu/2023/03/13/alpaca.html>
- [8] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan, and J. Zou, "Gradio: Hassle-free sharing and testing of ML models in the wild," arXiv preprint arXiv:1906.02569, 2019.
- [9] P. Lewis, E. Perez, A. Piktus et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems (NeurIPS), vol. 33, pp. 9459–9474, 2020.
- [10] H. Chase, "LangChain: Building applications with LLMs through composability," GitHub repository, 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>
- [11] J. Liu, "LlamaIndex: A data framework for LLM-based applications," GitHub repository, 2022. [Online]. Available: https://github.com/run-llama/llama_index
- [12] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, "Large Language Models Struggle to Learn Long-Tail Knowledge," in Proc. Int. Conf. Machine Learning (ICML), 2023, pp. 15696–15707.
- [13] R. Picard, Affective Computing. Cambridge, MA: MIT Press, 1997.
- [14] S. Mohammad and P. Turney, "Emotions Evoked by Common Words and Phrases: Using Mechanical Turk to Create an Emotion Lexicon," in Proc. NAACL HLT Workshop on Computational Approaches to Analysis and Generation of Emotion in Text, 2010, pp. 26–34.
- [15] C. Strapparava and A. Valitutti, "WordNet Affect: An Affective Extension of WordNet," in Proc. Language Resources and Evaluation Conf. (LREC), Lisbon, 2004, pp. 1083–1086.
- [16] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
- [17] P. K. Janert, Data Analysis with Open Source Tools. Sebastopol, CA: O'Reilly Media, 2010.
- [18] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust Speech Recognition via Large-Scale Weak Supervision," in Proc. Int. Conf. Machine Learning (ICML), 2023, pp. 28492–28518.