Large Language Model with Retrieval Augmented Generation

Project Objective and Limitations

i Project Overview

The advent of modern automobile manufacturing has led to increased technical complexity, often resulting in mechanics opting to replace parts rather than diagnose and fix issues. This approach, while convenient for contemporary vehicles, poses a significant challenge for classic cars built 30 to 40 years ago, where replacement parts are scarce or non-existent.

To address this problem, this project aims to leverage Generative AI to create a "virtual mechanic." By utilizing a corpus of text gathered from a classic car forum, this tool will be capable of understanding unstructured questions and providing relevant answers. This solution aims to assist classic car enthusiasts and mechanics by offering expert guidance, thereby preserving the heritage and functionality of vintage automobiles.

ii Objectives

The primary objective of this project is the development of a Natural Language Processing (NLP) model as part of a portfolio of AI projects that can be showcased to potential employers. This will include an outline of the necessary workflow with a comparison and selection of architectures, libraries, and methods. This is a complement to my pursuit of a Master's Degree in Data Science.

iii Use Case

With this code, a user will be able to ask questions in plain, unstructured English and receive answers that are a result of previous similar questions from the forum used to create the corpus. The answers will also include results from pre-trained models, ensuring a rich and informed response. Users will see these answers in plain English. As a programmer, I will have control over the extent to which the answers are sourced from the supplemental corpus versus the pre-trained model.

iv Limitations and Challenges

Where possible, a combination of open source and free resource will be used. Python will be the primary programming language. Google Colab will be used for the notebook with compute resources limited to CPUs. Data storage will be done in a Snowflake database.

iix Workflow for NLP RAG Model

1 Architectures and Models
2 Corpus Development
3 Clean and Preprocess Text
4 Clustering and Summarization
5 Format Text for Training
6 Embedding and Indexing

- Embedding: Convert tokens into numerical representations using embeddings from models like BERT, DistilBERT, or T5.
- Build FAISS Index: Create an index of the embeddings using FAISS for fast similarity searches.

7. Query Processing and Search

- Query Processing: Generate embeddings for new queries.
- Search: Use the FAISS index to find the most similar questions in the corpus.

8. Retrieve and Rank

- Retrieve**:** Fetch the top-N most similar questions and their corresponding answers from the corpus.
- Rank**:** Concatenate the retrieved contexts to form a comprehensive input for the generative model.

9. Answer Generation

- Generation**:** Use a generative model (e.g., T5 or GPT) to generate an answer based on the concatenated context and the query.

10. Evaluation and Tuning

- Evaluate**:** Assess the model's performance using metrics like precision, recall, F1-score, and accuracy.
- Tune**:** Fine-tune the pre-trained models on your specific dataset if necessary.

11. Deploying the Demo (Optional)

- Create UI: Build an interactive UI using tools like Flask or Streamlit.
- Deploy**:** Host the model on a server or cloud service for remote access.

1. Architectures and Frameworks

This document provides an overview of various architectures, models, and tools used in natural language processing tasks. Understanding the strengths and weaknesses of different approaches is crucial for designing effective NLP systems tailored to specific use cases and requirements.

1.1 Machine Learning Architectures

1.1.1 Traditional

1.1.1.1 Bag-of-Words (BoW)

Description:	Represents text data as a collection of unique words and their frequencies.
Example:	Term Frequency-Inverse Document Frequency (TF-IDF)
Pros:	Simple and efficient representation.
	Works well for tasks like sentiment analysis and document classification.
Cons:	Ignores word order and context.
	Doesn't capture semantic meanings well.

1.1.2 Statistical NLP Model

1.1.2.1 Hidden Markov Models (HMM)

Description:	Sequential text models based on hidden state transitions
Example:	hmmlearn (Python library for HMMs)
Pros:	Captures sequential dependencies effectively.
	Suitable for tasks like part-of-speech tagging and named entity recognition.
Cons:	Requires labeled sequential data for training.
	May struggle with capturing complex semantic relationships.

1.1.2.2 Conditional Random Fields (CRF)

Description:	Sequence labeling models
Example:	sklearn-crfsuite (Python library for CRFs based on scikit-learn)
Pros:	Effective for sequential labeling tasks.
	Incorporates feature dependencies between adjacent labels.
Cons:	Requires labeled sequential data for training.
	Less effective for capturing long-range dependencies

1.1.2.3 Support Vector Machines (SVM)

Description:	A supervised learning model used for classification and regression analysis. SVMs find a hyperplane in an N-dimensional space that distinctly classifies data points.
Example:	scikit-learn (Python library for machine learning)
Pros:	Effective in high-dimensional spaces.
	Versatile with different kernel functions for flexibility in decision boundaries.
Cons:	
	Memory-intensive for large datasets.
	May require careful selection of kernel functions and tuning parameters.

1.1.3 Deep Learning Architectures

1.1.3.1 Word Embeddings

Description:	Represent words as dense vectors in a continuous vector space.
Examples:	Word2Vec, GloVe
Pros:	Captures semantic meanings and relationships between words.
	Provides dense vector representations suitable for downstream tasks.
Cons:	Requires large amounts of data for training.
	Struggles with out-of-vocabulary words.

1.1.3.2 Sequence Models

Description:	Models that capture the sequential nature of text data.
Examples:	Hidden Markov Models (HMM), Conditional Random Fields (CRF)
Pros:	Captures sequential dependencies in data.
	Suitable for tasks like named entity recognition and part-of-speech tagging.
Cons:	Requires labeled sequential data for training.
	Can be computationally intensive.

1.1.3.3 Recurrent Neural Networks (RNN)

Description:	Neural networks that process sequences by iterating through elements.
Examples:	Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU)
Pros:	Effective for capturing sequential dependencies in data.
	Suitable for tasks like language modeling and machine translation.
Cons:	Vulnerable to vanishing and exploding gradient problems
	Computationally expensive to train.

1.1.4 Transformers

1.1.4.1 Transformer Models

Description:	Neural network architecture based entirely on self-attention mechanisms.
Examples:	BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pre-trained Transformer), T5 (Text-To-Text Transfer Transformer)
Pros:	Captures long-range dependencies effectively.
	Parallelizable training process.
Cons:	Requires large amounts of computational resources.
	Limited interpretability compared to traditional models.

1.1.4.2 Pre-trained Models

Description:	Models pre-trained on large corpora and fine-tuned for specific tasks.
Examples:	BERT, GPT, T5
Pros:	Leverage large amounts of unlabeled data for pre-training.
	Achieve state-of-the-art performance on various NLP tasks.
Cons:	Resource-intensive pre-training process.
	May require substantial computational resources for fine-tuning.

1.1.5 Additional Models and Techniques

1.1.5.1 Retriever-Generator Models

Description:	Models that combine retrieval and generation components for text generation tasks.
Examples:	RAG
Pros:	Incorporates both structured and unstructured information for generation.
	Produces more diverse and contextually relevant responses.
Cons:	Requires efficient retrieval mechanisms.
	Increased complexity in model architecture.

1.1.5.2 Knowledge-Enhanced Retrieval-Augmented Generation (KERAG)

Description: A variant of RAG that incorporates knowledge graphs to enhance retrieval and generation.

Examples:	Graph-BERT
Pros:	Integrates structured knowledge for improved understanding and generation.
	Enables more coherent and contextually relevant responses.
Cons:	Requires high-quality and curated knowledge graphs.
	Increased computational complexity compared to standard RAG.

1.1.5.3 Elastic Search

Description:	Distributed search and analytics engine for indexing and searching large volumes of data.
Examples:	Elasticsearch, Apache Solr
Pros:	Scalable and distributed architecture.
	Supports full-text search and complex query structures.
Cons:	
	Requires infrastructure for deployment and maintenance.
	Indexing and search performance may degrade with large datasets.

Model/Architecture	Performance	Contextual Accuracy	Ease of Use	Cost	Scalability	Compatibility	Total
RAG	2	2	1	1	2	2	10
Pre-trained	2	1	2	1	2	2	10
BoW	1	0	2	2	2	2	9
KERAG	2	2	0	1	2	2	9
Transformer	2	1	1	1	2	2	9
Word Embeddings	1	0	2	2	2	2	9
CRF	1	1	1	2	1	2	8
Elastic Search	1	0	1	2	2	2	8
HMM	1	1	1	2	1	2	8
RNN	2	1	1	1	1	2	8
Sequence	1	1	1	2	1	2	8
SVM	2	1	1	1	1	2	8

0 = Doesn’t meet
1 = Partially meets
2 = Full meets

Conclusion

Given the project's requirements, the higher complexity lower ease of use cost of RAG are justified by its superior performance in generating human-like responses. Therefore, RAG remains the best choice for creating a robust virtual mechanic. For clarity, this RAG will make use of Word Embeddings and a Pre-trained model.

1.2 Frameworks and Tools

1.2.1 TensorFlow

Description:	Open-source machine learning framework developed by Google for building and deploying ML models.
Pros:	Comprehensive ecosystem with support for various deep learning architectures.
	Scalable and efficient execution on both CPUs and GPUs.
Cons:	Steeper learning curve compared to some other frameworks.
	Limited support for dynamic computation graphs.

1.2.2 PyTorch

Description:	Open-source deep learning framework developed by Facebook's AI Research lab.
Pros:	Pythonic and intuitive interface for model development.
	Dynamic computation graph enables easier debugging and experimentation.
Cons:	Less optimized for deployment in production compared to TensorFlow.
	Limited built-in support for distributed training.

1.2.3 scikit-learn

Description:	Simple and efficient machine learning library built on NumPy, SciPy, and matplotlib.
Pros:	Easy-to-use API for common machine learning tasks.
	Comprehensive documentation and community support.
Cons:	
	Limited support for deep learning models and architectures.
	Less flexibility for customization compared to deep learning frameworks.

1.3 Data Storage and Database

Efficient data storage and management are pivotal for the project, focusing on accommodating extensive unstructured data from various sources. The project explores two main classes of storage solutions: Cloud Storage and Local Storage, each offering unique benefits and challenges.

1.3.1 Cloud Storage

Cloud storage solutions offer scalability, reliability, and remote access, making them suitable for projects with dynamic data needs and global access requirements.

- Tools: Snowflake (for relational data), MongoDB Atlas (for NoSQL data)
  - Pros:
    - Scalability: Easily scales to meet growing data demands without the need for physical infrastructure management.
    - Accessibility: Provides global access to the data, facilitating collaboration and remote work.
    - Maintenance and Security: Cloud providers manage the security, backups, and maintenance, reducing the administrative burden.
  - Cons:
    - Cost: While scalable, costs can increase significantly with data volume and throughput.
    - Internet Dependence: Requires consistent internet access, which might be a limitation in some scenarios.
    - Data Sovereignty: Data stored in the cloud may be subject to the laws and regulations of the host country, raising concerns about compliance and privacy.

1.3.2 Local Storage

Local storage solutions rely on on-premises or personal hardware, providing full control over the data and its management but requiring more direct oversight.

- Tools: MySQL (for relational data), MongoDB (Local installation for NoSQL data)
  - Pros:
    - Control: Complete control over the data storage environment and configurations.
    - Cost: No ongoing costs related to data storage size or access rates, aside from initial hardware and setup.
    - Connectivity: No reliance on internet connectivity for access, ensuring data availability even in offline scenarios.
  - Cons:
    - Scalability: Physical limits to scalability; expanding storage capacity requires additional hardware.
    - Maintenance: Requires dedicated resources for maintenance, backups, and security, increasing the administrative burden.

- Accessibility: Data is not as easily accessible from remote locations, potentially hindering collaboration and remote access needs.

Conclusion: I will be using Snowflake to store my corpus.

2 Develop Corpus

2.1 Data Ethics

The data collected here is a collection of posts from widely available public forum. However, should this project move into public distribution, additional step will be necessary to endure PII is obfuscated or removed. In addition, this document shall serve as full disclosure of the projects goals and data gathering process.

2.2 Data Collection

The project leverages user-generated content from a domain-specific online forum as the training corpus. This data is largely unstructured, with minimal metadata available. The following tools were considered to gather the source text for the corpus:

2.2.1 Web Scraping

Tools:	Beautiful Soup, online SaaS products
Pros:	Direct Access to Targeted Data: Enables precise extraction of user-generated content from specific sections or threads within the forum.
	Efficiency in Data Collection: Automated scripts can gather large volumes of data in a short amount of time, making it suitable for assembling significant datasets for NLP.
Cons:	Potential for Incomplete Data: May miss embedded content or dynamically loaded data, depending on the website's structure.
	Ethical and Legal Considerations: Scraping data from forums may raise concerns about user privacy and must adhere to the terms of service of the website.
	Very Platform Dependent: Forum specific solutions result in forum specific data schemas that must be reverse engineered to for successful text extraction.

2.2.2 Forum-specific APIs

Tools:	Python (`requests` library for API calls and `json` library for handling responses)
Pros:	Structured and Reliable Data Retrieval: APIs provide structured data, making it easier to process and integrate into your project.
	Efficient and Direct Access: Directly accessing the forum's data through its API is efficient, bypassing the need for HTML parsing.
	Compliance and Ethical Data Use: Utilizing APIs respects the forum's data policies and ensures access is in line with user agreements.
Cons:	Rate Limiting: APIs often have limitations on the number of requests that can be made in a certain timeframe, which could slow down data collection.
	API Changes: Dependence on the forum's API structure means that changes or deprecation could disrupt your data collection pipeline.

Access Restrictions: Some data or functionalities might be restricted or require authentication, posing additional challenges for comprehensive data collection.

3 Clean and Preprocess Text

a. Remove Unnecessary Characters
b. Convert Text to Lowercase
c. Remove Stop Words
d. Deduplication
e. Lemmatization
f. Entity Recognition and Anonymization: Identify and anonymize personal information or specific entity names to maintain privacy.

4 Clustering and Summarization

4.1 Summarization Strategies

Summarization in NLP involves condensing large texts into shorter versions, capturing the most critical information. This can be approached through multiple options. For this effort, the following solutions were scored to reduce the potential solution set:

Provider	OpenAI	Anthropic	Meta	Google	Amazon
Package	GPT (GPT-2)	Claude (Claude 3)	DistilBART (sshleifer/distilbart-cnn-12-6)	T5 (t5-small)	AWS Comprehend
Strengths	Optimal for CPUs, free access	Latest version, enhanced safety features	Optimized for CPU usage, efficient resource usage	Efficient CPU usage, flexible, and powerful	Integrated AWS service, scalability
CPU Performance	2 (GPT-2 optimized for CPUs)	2 (Enhanced capabilities in Claude 3)	2 (DistilBART optimized for CPUs)	2 (Efficient on CPUs)	1 (General cloud performance)
Free	2 (GPT-2 is free)	0 (Research access mainly)	2 (Open-source and free)	2 (Open-source and free)	0 (Paid service)
API Independence	1 (Can be run locally, no API needed)	1 (Research and conditional access)	2 (Completely API independent)	2 (Can be run locally)	0 (API-dependent service)
Handling Varying Text Length	2 (Good at handling different lengths)	2 (Claude 3 designed for robust handling)	2 (Handles varying lengths well)	2 (Handles varying lengths efficiently)	2 (Designed to scale)
Ease of Integration	1 (Integration flexibility varies)	1 (Varies by deployment)	2 (Good documentation and community support)	2 (Good documentation and support)	2 (Tightly integrated with AWS ecosystem)
Total Score	8	6	10	10	5

### Criteria Explanations
1. CPU Performance: Measures how well the solution performs on standard CPUs.
2. Free: Indicates if the solution is free to use.
3. API Independence: Indicates if the solution can be used without relying on an external API.
4. Handling Varying Text Length: Assesses how well the solution can manage different lengths of text input.
5. Ease of Integration: Measures the flexibility and support for integrating the solution into various environments.

### Scoring Explanations
0: Does not meet
2: Fully meets
1: Partially meets

### A test of solutions can be found in the Appendix. T5 was chosen based on better ROUGE scores than BART.

5 Format Text for Training

Structure the text into suitable format

Tokenization Strategies

Tokenization is a crucial preprocessing step in NLP, segmenting text into manageable units for further analysis or model training. The choice of tokenization strategy affects both the complexity of the model and its ability to understand the text.

### Word-level Tokenization
- Tools: NLTK, spaCy, TensorFlow/Keras Tokenizers, BPE, Hugging Face Tokenizers
  - Pros:
    - Preserves word integrity and semantic meaning, crucial for comprehension tasks.
    - Subword tokenization methods like BPE can efficiently handle unknown words.
  - Cons:
    - Can result in a large vocabulary size, increasing memory and processing requirements.
    - May overlook nuances in character-level variations.

### Character-level Tokenization
- Tools: Supported by deep learning frameworks like TensorFlow and Keras
  - Pros:
    - Captures morphological nuances at the character level, aiding languages with rich morphology.
    - Simplifies the vocabulary to a set of unique characters, reducing model complexity.
  - Cons:
    - Leads to longer input sequences, which can increase computational costs.
    - Loses direct access to semantic information encoded in words or phrases.

### Subword Tokenization (BPE and Hugging Face Tokenizers)
- Tools A blend of word-level and character-level tokenization, aiming to balance vocabulary size and semantic richness.
  - Pros:
    - Offers a middle ground, effectively managing vocabulary size while preserving semantic information.
    - Facilitates handling of rare or unknown words by breaking them down into recognizable subwords.
  - Cons:
    - Requires preprocessing to establish a subword vocabulary, adding complexity.
    - Generated subwords may lack standalone meaning, complicating interpretation.

### Model-Specific Tokenization
- Tools: Hugging Face's transformers library provides access to pre-built tokenizers corresponding to each pre-trained model, ensuring that tokenization is consistent with the model's original training data.

Examples:

- For BERT: AutoTokenizer.from_pretrained('bert-base-uncased')

- For GPT-2: AutoTokenizer.from_pretrained('gpt2')

- For T5: AutoTokenizer.from_pretrained('t5-small')

6 Embedding and Indexing

- Embedding: Convert tokens into numerical representations using embeddings from models like BERT, DistilBERT, or T5.
- Build FAISS Index: Create an index of the embeddings using FAISS for fast similarity searches.

7 Query Processing and Search

- Query Processing: Generate embeddings for new queries.
- Search: Use the FAISS index to find the most similar questions in the corpus.

8 Retrieve and Rank

- Retrieve: Fetch the top-N most similar questions and their corresponding answers from the corpus.
- Rank: Concatenate the retrieved contexts to form a comprehensive input for the generative model.

9 Answer Generation

- Generation: Use a generative model (e.g., T5 or GPT) to generate an answer based on the concatenated context and the query.

10 Evaluation and Tuning

- Evaluate: Assess the model's performance using metrics like precision, recall, F1-score, and accuracy.
- Tune: Fine-tune the pre-trained models on your specific dataset if necessary.

1. PyTorch and Hugging Face Transformers
   - Pros: PyTorch offers dynamic computation graphs that are intuitive for RAG model development. Hugging Face's Transformers library provides easy access to pre-trained models and tokenizers, facilitating both training and evaluation with extensive support for RAG architectures.
   - Cons: While highly flexible, this combination might require a steep learning curve for those not familiar with PyTorch or the Transformers library.

2. TensorFlow and T5
   - Pros: TensorFlow provides robust tools for model development and deployment, with T5 being a versatile model for text-to-text tasks, adaptable for RAG purposes. TensorFlow's extensive ecosystem includes TensorBoard for monitoring training processes.
   - Cons: TensorFlow's static computation graph can be less intuitive than PyTorch's dynamic graphs. T5's text-to-text format might require additional preprocessing steps.

3. JAX and Flax/Haiku
   - Pros: JAX offers accelerated NumPy operations and automatic differentiation, making it efficient for large-scale model training. Flax and Haiku provide neural network libraries for JAX, supporting complex RAG model architectures.
   - Cons: JAX's ecosystem is less mature, with fewer pre-trained models and community resources available compared to PyTorch and TensorFlow. This can make development and troubleshooting more challenging.

10.1 Hyperparameter Tuning

Tuning Strategy: Outline your strategy for hyperparameter tuning, including the tools or techniques (like grid search or random search) you plan to use.

10.2 Model Evaluation

Evaluation Metrics: Detail the metrics you will use to evaluate your model, such as F1 score, precision, recall, and explain why each is important for your project’s success.

The project's success will be assessed based on the accuracy and speed of responses generated by the language model.

1. PyTorch and Hugging Face Transformers
   - Pros: PyTorch offers dynamic computation graphs that are intuitive for RAG model development. Hugging Face's Transformers library provides easy access to pre-trained models and tokenizers, facilitating both training and evaluation with extensive support for RAG architectures.
   - Cons: While highly flexible, this combination might require a steep learning curve for those not familiar with PyTorch or the Transformers library.

2. TensorFlow and T5
   - Pros: TensorFlow provides robust tools for model development and deployment, with T5 being a versatile model for text-to-text tasks, adaptable for RAG purposes. TensorFlow's extensive ecosystem includes TensorBoard for monitoring training processes.
   - Cons: TensorFlow's static computation graph can be less intuitive than PyTorch's dynamic graphs. T5's text-to-text format might require additional preprocessing steps.

3. JAX and Flax/Haiku
   - Pros: JAX offers accelerated NumPy operations and automatic differentiation, making it efficient for large-scale model training. Flax and Haiku provide neural network libraries for JAX, supporting complex RAG model architectures.
   - Cons: JAX's ecosystem is less mature, with fewer pre-trained models and community resources available compared to PyTorch and TensorFlow. This can make development and troubleshooting more challenging.

11 Deployment

- Create UI: Build an interactive UI using tools like Flask or Streamlit.
- Deploy: Host the model on a server or cloud service for remote access..

11.1.1 Deployment Tools

# Deployment and Serving Infrastructure Selection

1. Hugging Face Spaces
   - Pros: Provides a simple and direct way to deploy and share machine learning models, including RAG models. It supports interactive web-based applications and API endpoints, making it ideal for showcasing projects.
   - Cons: While convenient for prototypes and demonstrations, it might not offer the scalability and control needed for high-demand production environments.

2. AWS SageMaker
   - Pros: Offers a fully managed service that enables data scientists and developers to build, train, and deploy machine learning models at scale. SageMaker supports direct deployment of PyTorch models, including those built with the Hugging Face Transformers library, with robust monitoring and security features.
   - Cons: Can be more expensive and requires familiarity with AWS services. The setup and management might be complex for smaller projects or those new to cloud services.

3. Docker + Kubernetes
   - Pros: This combination offers flexibility and scalability for deploying machine learning models. Docker containers make it easy to package your RAG model with all its dependencies, while Kubernetes provides orchestration to manage and scale your deployment across multiple instances or cloud providers.
   - Cons: Requires significant DevOps knowledge to setup, manage, and scale. It might be overkill for simple or one-off deployments.

## Decision
For deploying a RAG model, especially within an academic or portfolio context where ease of use, accessibility, and cost-effectiveness are key considerations, Hugging Face Spaces is highly recommended. It allows you to quickly deploy your models with minimal setup and offers a user-friendly platform for showcasing your work to a wide audience. For projects that might evolve into more scalable or commercial applications, starting with Docker for containerization and then moving to a Kubernetes-based deployment as needs grow could be a strategic approach. This path provides a balance between initial simplicity and long-term scalability.

12 Appendix

Reflection and Learning

Challenges Faced

Lessons Learned

Future Work and Improvements

Potential Enhancements

Areas for Further Research