IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

LexEval:
A Scalable LLM Evaluation Framework

Supervisor:
Prof. Abbas Edalat

Author:
Siddhant Singh

Second Marker:
Dr. Chiraag Lala

Advisor:
Ruoyu Hu

July 2, 2024

Abstract

As Large Language Models (LLMs) become more powerful and their parameter counts increase, it is crucial for them to exhibit consistent recall capabilities for terms that appear less frequently in their pre-training data. The importance of accurate responses is paramount especially when LLMs are being used in production workflows for business critical decision making in industries like supply chain, medicine and business. Retrieval-Augmented Generation (RAG) is a powerful method that combines non-parametric knowledge with the parametric knowledge of language models. This project investigates the robustness of LLMs in handling long-tail entity question answering (QA).

Initially, our project aimed to address SQuAD2.0[1], but we transitioned to using POPQA, which better suited our goals. We examined prompt perturbations from both paraphrasing and lexical perspectives, incorporating these variations within a tree structure to systematically analyse their effects. This comprehensive approach allowed us to delve deeper into the RAG pipeline and evaluate it with greater granularity.

LexEval provides a straightforward framework for selecting LLMs tailored to users' specific needs, allowing them to optimise prompts for accuracy, token lengths, and cost-effectiveness. This framework empowers users to make informed decisions, enhancing the efficiency and effectiveness of their workflows. With LexEval, we have created a scalable evaluation framework adaptable to the continuous advancements in language models. It enables users to fine-tune their evaluation paradigms, focusing on lexical or paraphrasing perturbations as required. Our extensive testing of 7 LLMs includes comprehensive performance documentation, detailed RAG pipeline failure analysis, and qualitative response assessments. LexEval has proven its robustness with larger models by increasing task difficulty using long-tail entities, making it a valuable tool for evaluating and improving the performance of current and future language models.

Acknowledgements

I would like to thank Ruoyu (Roy) and Prof. Edalat for their dedicated support during this endeavour, encouraging and advising me at every step of the project.

Thank you to my friends and family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In 2023, the advent of ChatGPT and agent-based workflows heightened the demand for comprehensive evaluation and relative comparisons in the field. During my industrial placement, a substantial portion of my time was dedicated to the development of the artificial intelligence platform. This innovative platform harnessed the capabilities of GPT-4, allowing customers to construct their own LLM-backed workflows.

Robustness testing emerged as a crucial facet in the industry, particularly in nascent fields where developers might prioritise product building and delivery over meticulous testing. The maturation of this space accentuates the significance of making judicious decisions when selecting models tailored to specific use cases.

Given the widespread use of textual prompts, establishing standardised prompting language and systematically testing optimal phrasing on a large scale becomes imperative for users. Presently, there is a noticeable absence of a testing framework of such magnitude, highlighting this project's importance.

As the landscape evolves, the recognition of the critical role robustness testing plays in refining language models and workflows becomes increasingly apparent. This project not only addresses the current need for standardised prompts but also lays the groundwork for a comprehensive evaluation framework essential for the continued advancement of LLMs in intricate workflows.

1.2 Contributions

The main objective was to provide a simple framework for selecting LLMs that caters to the individual needs of users. This framework enables users to explore prompts, optimising for accuracy, token lengths, and cost-effectiveness. Users can confidently navigate the landscape of LLMs, making well-informed decisions that match their particular use cases. In the end, this enhances the efficiency and effectiveness of their workflows.

With LexEval, we have developed a scalable evaluation framework adaptable to the continuous advancements in language models. This framework enables users to fine-tune their evaluation paradigms, offering the flexibility to focus more on lexical or paraphrasing perturbations as needed.

Our extensive testing involved 7 LLMs, and we have documented their performance comprehensively. This includes a detailed analysis of any failures within the RAG pipelines and a qualitative assessment of the responses generated by these models.

LexEval has demonstrated its robustness with larger models by effectively increasing task difficulty through the inclusion of long-tail entities. This adaptability and thorough evaluation capability make LexEval a valuable tool for assessing and improving the performance of contemporary and future language models.

1.3 Report Outline

We begin the report by exploring the relevant topics required to understand language models and their evaluations (Chapter 2 and 3). We then discuss the initial implementation of LexEval with SQuAD2.0 (Section 4.1), quickly followed by the latest implementation of LexEval (4.2). In the evaluation (5), we showcase the metrics and results gained from experiments with LexEval tested across 7 different LLMs. Finally, we summarise our findings in Chapter 6.

Chapter 2

Background

2.1 Deep Learning

Deep learning signifies a transformative change in artificial intelligence, leveraging the intricate structure of neural networks to learn and discern intricate patterns from data. At the heart of deep learning lies the essential foundation — the neuron.

2.1.0.1 Neuron

A neuron is the basic computational unit of a neural network, emulating the functioning of a biological neuron. Mathematically, a neuron processes input signals, applies weights, adds a bias term, and activates through an activation function. The output of a neuron is calculated as follows:

$$\text{Output} = f(\text{Bias} + \sum_{i=1}^{n} \text{Input}_i \times \text{Weight}_i) \tag{2.1}$$

This formula encapsulates the neuron's ability to transform input information into meaningful output, modulated by weights and biases. Various activation functions (f) such as sigmoid ($\sigma(x) = \frac{1}{1+e^{-x}}$), tanh ($\tanh(x)$), and ReLU ($max(0, x)$), introduce non-linearities to the neuron's computation.

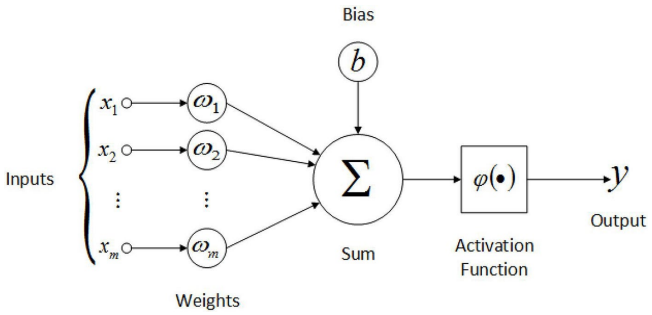


Figure 2.1: A simple neuron

2.1.0.2 Feedforward Networks or Multi-Layer Perceptrons

Feedforward networks are the backbone of DL. These networks consist of an input layer, hidden layers, and an output layer. Inference flows unidirectionally from the input layer and hidden layers to the output layer. The architecture enables the network to learn hierarchical representations of data.

2.1.0.3 Training

Having established the basics of neurons and feedforward networks, the next crucial aspect is understanding how these models are trained. Model training involves optimising the weights and

biases of the neural network to optimise its loss function. The training process aims to minimise the difference (also referred to as error) between the model's predicted values and target values in a dataset.

Optimisation Objective: The primary objective during training is to define a suitable loss function that quantifies the difference between the predicted and actual values. A useful loss function for regression tasks is Mean Squared Error (MSE) and Cross-Entropy Loss for classification tasks.

The overarching goal is to find the optimal set of weights and biases that minimises this loss function, effectively making the model's predictions as accurate as possible.

Backpropagation algorithm: The backpropagation algorithm is the cornerstone of training deep neural networks. It involves two main steps: forward pass and backward pass. [2]

1. **Forward pass**: Input data passed through the network generating an output. Loss is calculated using the predictions and the truth values.
2. **Backward pass**: Propagates the loss back through the network, gradients of the loss landscape with respect to the model weights and biases are computed. The weights and biases are appropriately adjusted in the other direction of the gradient to minimise loss. The parameter update formulae are given as follows, note that α is the rate of learning:

$$W^* = W - \alpha \frac{\delta L}{\delta W} \tag{2.2}$$

$$b^* = b - \alpha \frac{\delta L}{\delta b} \tag{2.3}$$

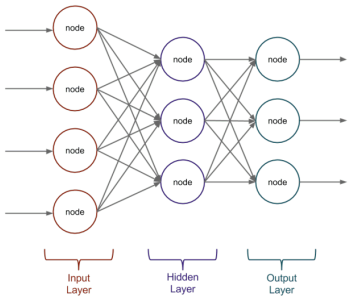


Figure 2.2: Simple Neural Network

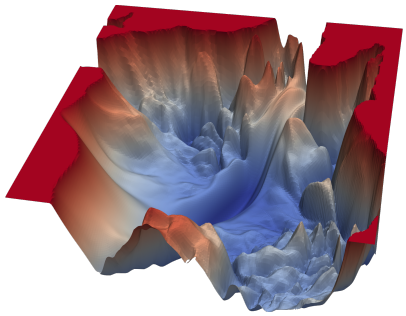


Figure 2.3: A loss landscape (VGG-56)[3]

Optimisation Algorithms: To efficiently navigate the vast parameter space, optimisation algorithms like Gradient Descent (or Stochastic Gradient Descent) are used. These algorithms iteratively update model parameters, moving them towards the optimal values that minimise the loss.

Epochs and Batches: Training occurs over multiple iterations called epochs. In each epoch, the entire dataset is processed through the network. To enhance efficiency and mitigate memory constraints, datasets are often divided into batches.

2.1.1 Recurrent Neural Networks

RNN is a class of neural networks equipped with a dynamic architecture, incorporating recurrent connections (synonymous to feedback loops). These connections enable RNNs to maintain an internal state or memory, allowing them to retain information from previous time steps. The mathematical representation of an RNN's hidden state (h_t) at time t is defined as follow:

$$h_t = f(W_{hh} h_{t-1} + W_{hx} x_t) \tag{2.4}$$

W_{hh} and W_{hx} represent the weights for the recurrent and input connection and f is the activation function.

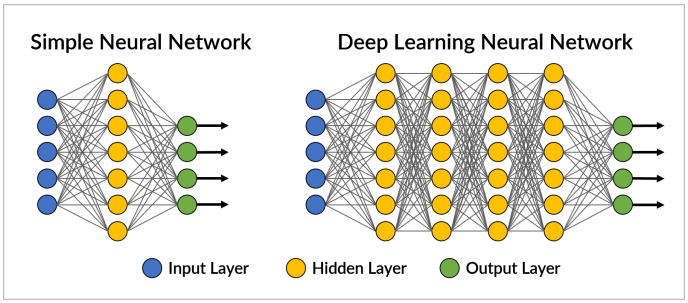


Figure 2.4: Simple versus Deep Learning Neural Network. [4]

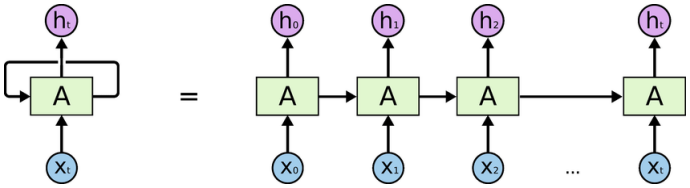


Figure 2.5: Diagram displaying RNN architecture and how it can be unrolled. [5]

Sequential data, prevalent in NLP, demands models capable of capturing temporal dependencies. RNNs perform well in this domain by processing sequences element by element while considering context from previous steps. The recurrent connections play a key role in modeling temporal dependencies, which are essential for grasping the nuanced structure of sequential information.

In NLP, language follows a sequence, where the meanings of words frequently depend on context. Mikolov et. al illustrated the use of RNNs in language modeling, highlighting their efficacy in capturing detailed language patterns [6]. Broadly speaking, we can attribute a conditional probability to each potential next word in a sentence, resulting in a probability distribution across the entire dictionary. Combining these conditional probabilities yields:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<1}) \tag{2.5}$$

2.1.1.1 Generating Probability Distribution using RNNs

Assume we have an input sequence $T = [t_1; ...; t_t; ...; t_N]$ where T is the corpus of text and t_n are word embeddings as 1-hot vector of size $|V| \times 1$. Suppose the output is a vector, y, which represents the probability distribution over the dictionary. After each step (loop iteration), the model uses the dictionary (embedding matrix E) to retrieve the embedding for the word at that step. The model then combines the output of the previous step with the embedding to generate a new layer. The last layer will then be passed into a $softmax$ function to generate a probability distribution over the dictionary. Thus, the probability that a word i is the next word in the sequence is given by $y_t[i]$, from equation 2.5, this can be written as:

$$P(w_{t+1} = i|w_1, ..., w_n) = y_t[i] \tag{2.6}$$

Similarly, the probability of an entire sequence to appear next is given by:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{1:i-1}) = \prod_{i=1}^{n} y_i[w_i] \tag{2.7}$$

2.1.2 Long Short-Term Memory (LSTM)

Conventional RNNs encounter difficulties in learning and retaining information across lengthy sequences, a limitation commonly referred to as the vanishing gradient problem. This challenge emerges during the backpropagation of errors, where gradients diminish exponentially as they propagate backward through time. As a result, the network struggles to adequately update weights for

distant connections.

Vanishing Gradient Problem: A consequence of the chain rule in calculus during the gradient descent optimisation process. As gradients are multiplied across time steps during backpropagation, they can approach zero, causing weight updates to become negligible. In the context of traditional RNNs, this phenomenon severely limits the network's capacity to learn from and remember information in distant past time steps.

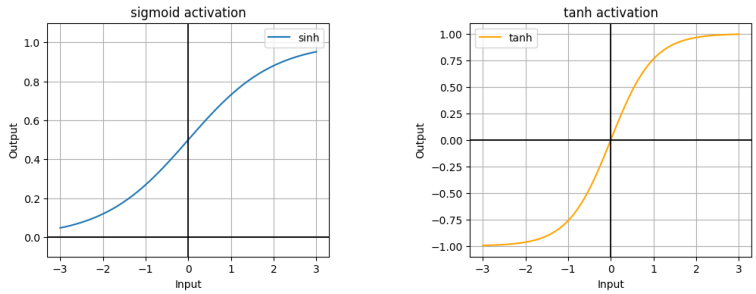


Figure 2.6: The sigmoid and tanh activation functions visualised. These figures were plotted using the `Matplotlib` library. [7]

2.1.2.1 LSTM Architecture

Understanding activation functions is crucial to build intuition towards the LSTM architecture. As Fig 2.6 shows:

$$\text{Sigmoid: } \sigma : \mathbb{R} \to [0,1] : z \mapsto \frac{1}{1+e^{-z}} \tag{2.8}$$

$$\text{Tanh: } \tanh : \mathbb{R} \to [-1,1] \tag{2.9}$$

After seeing the activation functions, we can start assembling the LSTM. Referring to diagram 2.7, the C_t cell represents the "Memory Cell" which is the long term memory of the LSTM. The "Memory Cell" is a combination of two different calculations, first we calculate the new long-term memory given the previous short-term memory, h_{t-1} and the input, x_t. Then, the sum is passed to the sigmoid function, $\sigma(h_{t-1} + x_t)$, this is commonly referred to as the forget gate since if the inner sum is sufficiently negative, the output of the sigmoid will tend to zero, intuitively it controls how much of the previous memory is discarded. We then multiply the output with the previous long-term memory C_{t-1}. The next calculation of the new long-term memory involves calculating the "Candidate Cell State", $\tanh(h_{t-1} + x_t)$ (what short-term memory to store) and multiplying it by the "Input Gate", $\sigma(h_{t-1} + x_t)$ (how much of the short-term memory to store). Then we add this product to the product of the previous long-term memory and the "Forget Gate", $C_t = C_{t-1} \times \sigma(h_{t-1} + x_t) + \tanh(h_{t-1} + x_t) \times \sigma(h_{t-1} + x_t)$

Moving towards the new short-term memory, we use the current "Input Gate" and multiply it by the tanh of the new C_t. Thus, $h_t = \sigma(h_{t-1} + x_t) \times \tanh(C_t)$.

Through this intricate gating mechanism, LSTMs can selectively update their memory cell state, allowing them to retain relevant information over extended time periods and effectively tackle the vanishing gradient problem. This capability makes LSTMs indispensable in tasks where understanding and retaining long-term contexts are critical. The LSTM's internal architecture positions it as a cornerstone for handling sequential data with nuanced dependencies, contributing significantly to the advancement of deep learning in diverse applications. On the contrary, LSTMs have the same limitations as RNNs, the probability of keeping the context of a word from a word that is positionally very distant decreases exponentially. Thus, for longer context windows, LSTMs have the tendency to forget context. Also, it is hard to parallelise the sequential process of processing a series of data (tokens in a sentence).

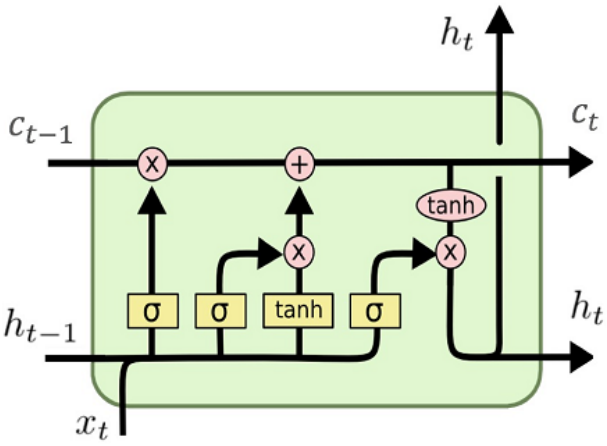


Figure 2.7: LSTM Architecture. [8]

2.1.3 Transformers

2.1.3.1 Attention

The Transformer architecture, Vaswani et al. in "Attention Is All You Need" [9], revolutionised the field of natural language processing and beyond. Unlike traditional sequence-to-sequence models relying on recurrent or convolutional layers, Transformers use self-attention blocks that allow them to understand links between token in a sequence.

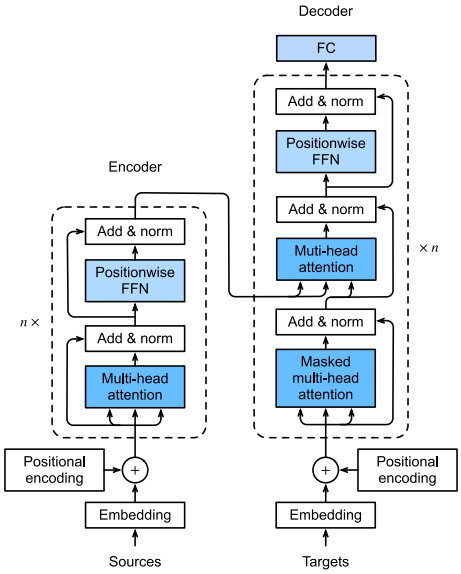


Figure 2.8: Encoder-Decoder Transformer Architecture. [10]

2.1.3.2 Encoder

An important part of the Transformer architecture is the encoder. It is responsible for translating the input sequence into a format which the model can comprehend. it outputs an "embedding" which is a representation of the input, the embedding is passed into a decoder to generate an output sequence.

The encoder consists of multiple layers. Each layer has a self-attention block and a feed-forward neural network. The self-attention block assigns weights to the importance of each corresponding input sequence section (using a dot product) as shown in Fig 2.9 (right). This is referred to as Multi-Head Attention in "Attention Is All You Need" where they use 6 layers.[9]

The "Attention is All You Need" paper uses a special attention function, the scaled dot-product attention. The inputs of the attention function are Q (query), K (keys) and V (values). The dimension of Q and K is d_k and V is d_v. The dot-product attention is used since it is faster and more space efficient [9]. The dot product is divided by $\sqrt{d_k}$ since it was found that as d_k increased, the dot product tended to infinity and pushed softmax values into gradients closer to 0.

$$\text{softmax} : \sigma : \mathbb{R}^K \mapsto (0,1)^K : \sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \text{ for } i = 1, ..., K \text{ and } z = (z_1, ..., z_k) \in \mathbb{R}^K \quad (2.10)$$

$$\text{Attention}(Q,\, K,\, V) = \text{ softmax}(\frac{Q \cdot K^T}{\sqrt{d_k}})V \quad (2.11)$$

$$\text{MultiHead}(Q,\, K,\, V) = \text{ Concat}(\text{head}_1, ..., \text{head}_h)W^O \quad (2.12)$$

where $\text{head}_i = \text{Attention}(QW_i^Q,\, KW_i^K,\, VW_i^V)$

Each layer in the encoder contains a feed-forward neural network which consists of two linear transformations with a ReLu activation in between. As per the paper,

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2.13)$$

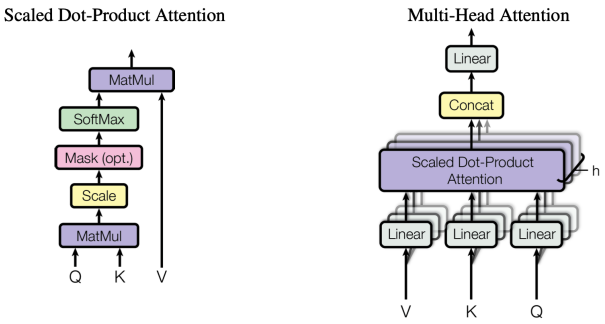


Figure 2.9: ((left) The Scaled Dot-Product Attention. (right) Stacked Multi-Head Attention used in "Attention is All You Need". [9]

You may notice that encoder still does not have any positional information about the words in a sentence. The paper uses sine and cosine functions with different frequencies, where the dimension of the encoding matches a sinusoid.

2.1.3.3 Benefits of Self Attention

The self-attention layer allows connecting each position in constant time meaning they are faster than RNNs (which are in linear time). The self-attention architecture also yields a more "interpretable" model since we can analyse the distributions from different heads. This is part of a table from the paper that compares some architectures we have seen before [9]:

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
RNN	$O(n \cdot d^2)$	$O(n)$

Table 2.1: Complexity comparisons between Self-Attention and RNN layers.

2.1.3.4 Decoder

The decoder architecture is similar to the encoder whereby it uses the output of the encoder and adds another multi-head self-attention layer which then adds the output embeddings using a "residual" connection. In the decoder block, the self-attention layer is modified such that positions don't attend to subsequent positions and embeddings are offset by one position which allows predictions at position i only being dependent on outputs at position $< i$.

2.2 Pre-Trained Large Language Models (LLMs)

Language model training involves exposing models to vast amounts of text data, allowing them to learn intricate patterns and relationships within language. The training process includes adjusting model parameters to minimise the discrepancy between predicted and actual language patterns.

Pre-trained LLMs are trained on large datasets, typically encompassing diverse sources like books, articles, and websites. This diverse training corpus ensures models grasp a broad understanding of language nuances. The types of data incorporated into pre-training contribute to the models' versatility, enabling them to perform well in NLP tasks.

2.2.1 Bidirectional Encoder Representations Transformers (BERT)

2.2.1.1 Training Data

BERT is trained on a dataset with 3.3 Billion words mainly coming from Wikipedia (2.5B words) and BookCorpus (0.8B).[11] This allows BERT to understand both the English language and information already on the internet simultaneously. The training phase of BERT was long but fast due to the Transformer architecture discussed before. It was trained using TPu (Tensor Processing Units) which is Google's special ML training circuit. Using 64 TPUs, it took Google AI Language 4 days to train BERT. There are other variants of BERT, especially smaller and more lightweight editions like DistilBERT which is 60% faster and 95% as accurate as BERT.[12]

2.2.1.2 Masked Language Model

The BERT paper [11] proposed a unique way of bidirectional learning using "masking". Intuitively, it masks a word in the sentence and uses the rest of the words (both sides in the sentence) in the sentence to predict that word. For example, consider the sentence:

> *I am going to the water fountain to fill my* `[masked]` *bottle.*

It would be wise to guess that `[masked]` is *water* due to the context given before (*water fountain* and *fill*) and after the masked word (*bottle*) and previous knowledge that one normally fills one's water bottle at the water fountain. In BERT, randomly chosen words (15% of the tokenised words) are masked.

2.2.1.3 Next Sentence Prediction

BERT uses a binarised prediction task to understand context between sentences. During training for sentence S_1, when choosing the next sentence S_2, it is only 50% of the times that S_2 is the correct sentence. This has shown an accuracy improvement.

2.2.1.4 BERT Architecture

Here we notice that BERT's architecture looks akin to the encoder block of the Transfer architecture described above.

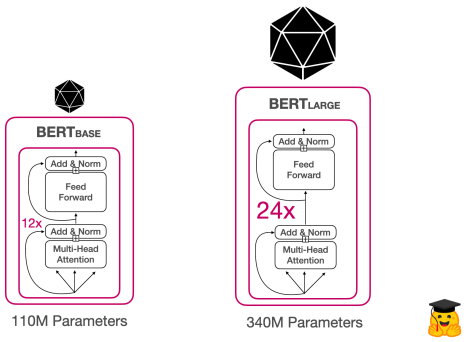


Figure 2.10: $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ architectures. [13]

2.2.1.5 Architecture Comparison

The architectures in Fig 2.10 compare as follows [14]:

Model	Parameters (in Millions)	Transformer Layers	Training Time (in days)
$\text{BERT}_{\text{BASE}}$	110	12	4
$\text{BERT}_{\text{LARGE}}$	340	24	4

Table 2.2: Architecture Comparison between $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$

According to HuggingFace, when BERT first came out it achieved stellar performance on 11 NLP tasks. Interestingly, it was also the first to outperform humans. In 2018, it topped the SQuADv1.1 & v2.0 benchmark. [15]

2.2.2 Generative Pre-Trained Transformer (GPT) Series

The Generative Pre-trained Transformer (GPT) series, developed by OpenAI, represents a family of influential large language models known for their generative capabilities and contextual understanding of language. These models share a common architecture based on the Transformer model, emphasising a pre-training strategy where models learn to predict the next token given a context.

2.2.2.1 Architecture and Evolution

GPT-1 [16]
Up to this point, NLP models focused on supervised learning for specific tasks like sentiment classification, facing challenges of scarce annotated data and limited generalisation beyond their trained tasks. The GPT1 paper introduces a novel approach: training a generative language model on unlabeled data and subsequently fine-tuning it with task-specific examples for downstream applications such as classification, sentiment analysis, and textual entailment.[16] By leveraging unsupervised learning initially and tailoring the model later, this methodology addresses the limitations of conventional supervised models, offering a more flexible and data-efficient solution for diverse NLP tasks.

The interesting concept used in this paper was "semi-supervised learning" which is composed by 3 different techniques:

- **Unsupervised Learning**: The common modelling objective was used, T is the set of unsupervised tokens ($t_{i-k}, ..., t_{i-1}$ and k is the context window and θ is the NN parameters trained using Stochastic Gradient Descent:

$$L_U(T) = \sum_i \log P(t_i | t_{i-k}, ..., t_{i-1}; \theta) \tag{2.14}$$

- **Supervised Fine Tuning**: Focused on maximising the probability of predicting the label y given tokens $x_1, ...x_n$, C is the set of labelled training examples:

$$L_S(C) = \sum_{x,y} \log P(y | x_1, ..., x_n) \tag{2.15}$$

- The OpenAI team then combined the supervised and unsupervised training objectives to create a new objective with the weight given to L_U being λ which was set to 0.5:

$$L_C(C) = L_S(C) + \lambda L_U(C) \tag{2.16}$$

- **Input Transformations**: To preserve the model architecture with minimal adjustments during fine-tuning, the input sequences for specific downstream tasks underwent a transformation into ordered sequences. This process involved adding start and end tokens to the input sequences. Additionally, a delimiter token was introduced between different segments of the example, enabling the input to be presented as an ordered sequence. For tasks such as question answering, multiple sequences were used. [17]

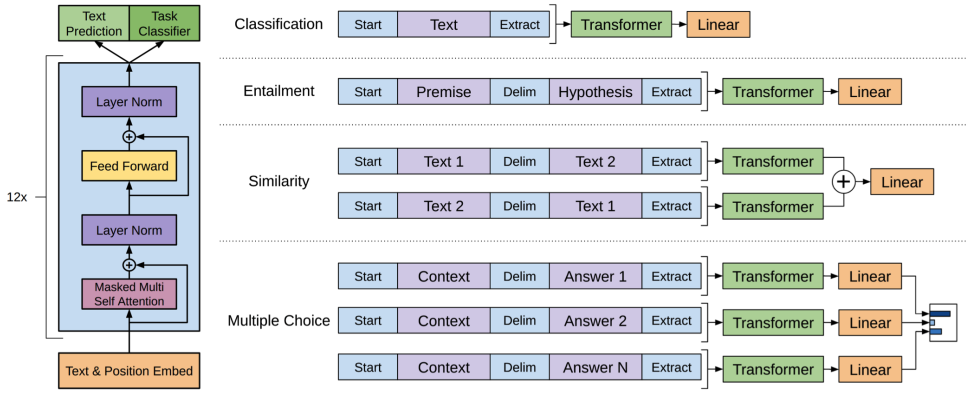


Figure 2.11: (left) GPT-1 transformer architecture. (right) Transformations for fine-tuning on input tasks. All tokens were pre-processed followed by a linear and a softmax layer. [16]

This model showed that decent zero-shot performance could be achieved using language models (in tasks like question answering and sentiment analysis).

GPT-2 [18]
The advancements in the GPT-2 model primarily revolved around utilising a larger dataset and augmenting the model with more parameters to enhance its language modeling capabilities. The paper delves into key concepts, with a focus on NLP, particularly addressing task conditioning and zero-shot learning.

Task conditioning is introduced as a crucial concept in the context of training language models. It involves modifying the learning objective to $P(\texttt{output}|\texttt{input}, \texttt{task})$, allowing the model to produce different outputs for the same input across various tasks. For language models, task conditioning is implemented by giving examples or instructions to perform a specific task.

Zero-shot learning and zero-shot task transfer represent notable capabilities of GPT-2. Zero-shot learning involves the model understanding a task based on given instructions without any provided examples. GPT-2 achieves zero-shot task transfer by formatting the input to prompt the model to comprehend the task nature and generate appropriate responses.

GPT-3 [19]
OpenAI's GPT-3, designed to create powerful language models requiring minimal fine-tuning, boasts an unprecedented 175 billion parameters, surpassing both Microsoft's Turing NLG and GPT-2 by 10 and 100 times, respectively. Leveraging its extensive dataset and immense capacity, GPT-3 excels in zero-shot and few-shot, demonstrating capabilities like producing human-like articles and performing tasks it was not explicitly trained on, such as generating code and summarising numbers.

Key concepts explored in the GPT-3 paper include in-context learning, where the model leverages pattern recognition skills developed during language model training for zero-shot task transfer. The paper also discusses few-shot, one-shot, and zero-shot settings.

GPT-3 was trained on a diverse dataset combining five corpora, including Common Crawl, WebText2, Books1, Books2, and Wikipedia. Notable differences from GPT-2 include an increased layer count, larger word embeddings, and an expanded context window size.

Performance evaluations revealed GPT-3's superiority in language modeling datasets in zero or few-shot settings, along with impressive performance in NLP tasks like question answering and translation. However, limitations were identified, such as occasional loss of coherence in long sentences and challenges in tasks like natural language inference. The generic language modeling objective was highlighted as a limitation, and suggestions for improvement included augmenting learning objectives and incorporating reinforcement learning.

Additional limitations encompassed complex and costly inferencing, reduced interpretability, and uncertainties around the factors influencing GPT-3's few-shot learning behavior. The paper emphasised the risk of misuse, urging careful monitoring due to potential biases and ethical concerns in the generated text.

GPT-4 [20]
As of January 2024, the most recent release from OpenAI represents the culmination of the three preceding generations. GPT-4's parameter size grew massively, totaling around 100 trillion as opposed to GPT-3's 175 million parameters, this recent model introduces a new dimension – vision. Importantly, the model is capable of making inferences using both visual and textual prompts.

2.3 Models as APIs

Language models are inference models equipped with pre-trained weights. Accessing and running inference with these models can be done in two primary ways: locally or via an API.

Local inference involves downloading the model weights and performing inference on local hardware, preferably using GPUs. This approach demands significant storage capacity for large models and robust GPUs to ensure fast inference.

Alternatively, using models as an API involves making API calls to run inference on the provider's servers, receiving the results over a secure connection. In this model, users only incur costs for inference and network usage, making it a more accessible method for leveraging large language models (LLMs).

2.3.1 OpenAI

As of June 2024, OpenAI offers 3 models as an API:

- GPT-4o: Fastest model available. Takes image and text as inputs and outputs text. Input costs $5 per million tokens and output costs $15 per million tokens.
- GPT-4 Turbo: Older version of the GPT-4 family. Also takes text and image inputs. Output and input tokens are more expensive than GPT-4o costing $10 per million tokens for input and $30 per million tokens for output.
- GPT-3.5 Turbo: Fastest and most robust model for simple tasks. Cheaper than the GPT-4 models costing costing $0.5 per million tokens for input and $1.5 per million tokens for output.

OpenAI also has a embedding models that can be used in the retriever phase of RAG. It currently offers 3 models whereby the user sends text and receives an embedding (a vector of a pre-determined size).

2.3.2 Together AI

Together AI provides access to various open-source models through an API, known for its rapid inference capabilities. This service has been utilised in this project due to its ability to offer a wider range of models, enhancing the scope and flexibility of the work. The models used and their metrics are given by Table 2.3.

2.4 Prompting and Testing

In the context of LLMs, prompting refers to supplying specific input or queries to provoke desired model outputs. It entails crafting inquiries to direct the model's output or generate information. Effective prompting is essential for obtaining accurate and pertinent results from LLMs, ensuring their applicability in various scenarios.

Model Name	Company/ Lab	Input/ Output	$ per 1M tokens
Mistral (7B) Instruct v0.2[21]	Mistral	Both	0.2
Mistral (7B) Instruct v0.3[21]			0.2
Meta Llama 3 70 B Chat[22]	Meta	Both	0.9
Meta Llama 3 8B HF[22]			0.2
Gemma Instruct (2B)[23]	Google	Both	0.1
Gemma Instruct (7B)[23]			0.2
GPT-3.5 Turbo[19]	OpenAI	Input	0.5
		Output	1.5
GPT-4[20]	OpenAI	Input	30.0
		Output	60.0
GPT-4 Turbo[20]	OpenAI	Input	10.0
		Outuput	30.0
GPT-4o[24]	OpenAI	Input	5.0
		Output	15.0

Table 2.3: LLM as an API Costs (as of June 7 2024)

Testing plays a crucial role in evaluating the performance, robustness, and dependability of LLMs. It aids in uncovering potential biases, limitations, and areas that require enhancement. Rigorous testing guarantees that models provide trustworthy information, particularly in critical applications such as common NLP tasks and understanding. The evaluation of LLMs contributes to refining their capabilities, building user confidence, and advancing the broader field of AI.

2.4.1 Prompting in LLMs

2.4.1.1 Chain-of-Thought (CoT)

CoT prompting is a tactical approach that empowers LLMs to address intricate problems by navigating through a sequence of steps prior to presenting a final answer. This technique improves the model's reasoning capability by directing it to handle multi-step problems using a series of logical steps that reflect a coherent thought process, similar to the way humans approach such challenges. Specifically beneficial for tasks involving logical thinking and intricate reasoning, such as arithmetic, CoT prompts empower LLMs to navigate through challenges. Initially designed as a few-shot prompting technique, each CoT prompt comprised a handful of Q&A examples. [25], eliminating the need for users to create numerous specific CoT Q&A examples.

2.4.1.2 Tree-of-Thought

As language models increasingly find application in diverse problem-solving scenarios, their limitations become evident, particularly in tasks demanding strategy and/or where initial decisions carry significant weight. In response to these challenges, Yao et. al propose a framework for inference called the "Tree of Thoughts" (ToT) [26] (diagram 2.12 from the paper). ToT allows models to explore "thoughts," as intermediary steps in problem-solving. It empowers language models to make decisions by evaluating multiple reasoning paths, assessing choices, and determining subsequent actions. Moreover, it encourages forward-thinking and backtracking, enabling global decision-making when needed. Notably, in Game of 24, where GPT-4 with chain-of-thought prompting achieved a mere 4% success rate, the approach achieved an impressive 74% success rate. [26]

2.4.2 Prompt Perturbation

This technique introduces subtle changes to the meaning of texts while preserving their lexical structure. This technique, commonly used in natural language processing, enhances model robustness and generalisation. By introducing variations in word choices or sentence structure, models are forced to focus on the underlying semantics, leading to improved performance. [27, 28]

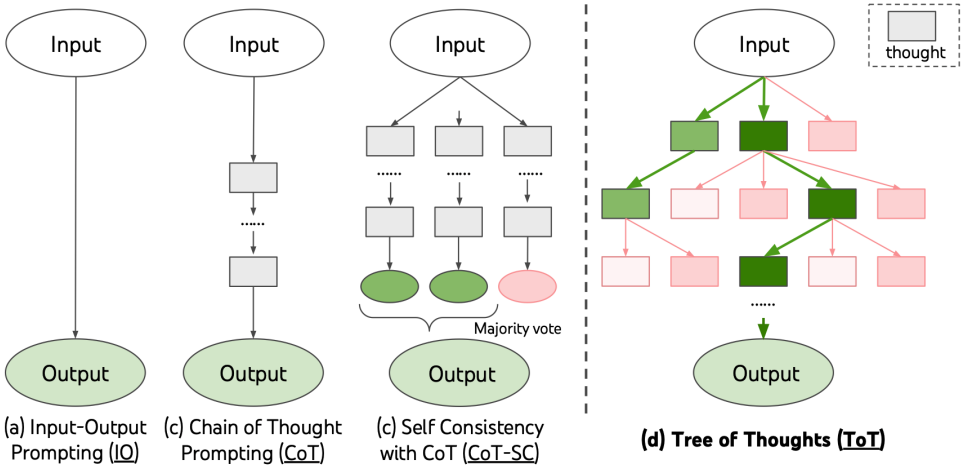


Figure 2.12: (a,b,c,d) Chain of Thought Prompting, (d) Tree of Thought Prompting. [26]

2.4.2.1 Traditional NLP Perturbation Techniques

NLP text perturbation refers to the deliberate modification of natural language text at either the letter or word level to enhance or evaluate the robustness and generalisation capabilities of NLP models. When considering perturbations at the letter level, it involves modifying individual characters to introduce noise or mimic variations in spelling. Perturbations at the word level involve actions such as replacing, deleting, or adding words, serving as a means to evaluate a model's sensitivity to semantic changes. This methodology is pivotal in training models capable of handling linguistic nuances and unforeseen inputs. Through exposing models to perturbed text, we strive to LLMs' adaptability and performance across a broad spectrum of real-world scenarios.

2.4.2.2 `butter_fingers` [29]

Randomly selected letters are substituted with neighboring keyboard positions, creating transformations that closely resemble the source sentences. As a result, the generated transformations exhibit a high degree of similarity to the original sentences.

	Text
Original	The quick brown fox jumps over the lazy dog
Perturbed	The quiock brown foxc jumps over the lazyu dogg

Table 2.4: `butter_fingers` Transformation Example

2.4.2.3 `shuffle_words` [30]

Shuffled words perturbation is a technique within NLP that involves shuffling the order of words in a sentence. This intentional reordering introduces variations in the lexical structure, challenging LLMs to keep understanding despite altered word sequences. The primary objective is to assess the robustness of these models, ensuring effective performance in scenarios where word order may vary. Shuffled words perturbation serves as a valuable tool in training models to understand and process language nuances.

	Text
Original	The quick brown fox jumps over the lazy dog
Perturbed	Lazy dog the quick brown fox jumps over the

Table 2.5: `shuffle_words` Transformation Example

2.4.2.4 random_upper_transformation [31]

Random upper transformation is a perturbation in NLP which introduces unpredictability by randomly capitalising letters within a sentence. This method aims to enhance model robustness, evaluating how well models cope with variations in (upper or lower) casing while maintaining semantic understanding. By injecting this controlled noise, the technique measures the sensitivity of LLMs.

	Text
Original	The quick brown fox jumps over the lazy dog
Perturbed	The quick broWn fox JumPs oVer tHe lazY doG

Table 2.6: `random_upper_transformation` Example

2.4.2.5 visual_attack [32]

Eger et al. presented a visual perturbation tool named `VIPER`, which accepts two parameters: a probability p and an embedding space denoted as CES, resulting in `VIPER(`p`, CES)`. The tool works within three primary embedding spaces:

- Image-based Character Embedding Space (ICES): Utilising a 24x24 image representation with each row having a size of 255, this space generates an embedding vector of size 576.
- Description-based Character Embedding Space (DCES): Employing Unicode 11.0.0 final names list, DCES selects nearest neighbors based on letter descriptions.
- Easy Character Embedding Space (ECES): A straightforward visual character perturbation method where each character is paired with a nearest neighbor containing a diacritic either above or below the character.

Condition	**Sentences** (Perturbed / Original)
`easy`-0.8	Ḿř. Ĉôḟḟêê iŝ â ṕřôfêŝsoř âẗ Ĉôḻŭḿḃîâ Ĺâẘ Ŝcĥôôḻ . Mr. Coffee is a professor at Columbia Law School .
ICES-0.6	Tṇẽ sẖutđown a|fëçṯə Ɜ,0ð0 wòřkḗrs ąng ᴎllḹ ɕũt óũṯpuƫ ɓỳ apouṭ 4,3Ẕ0 câṛș : The shutdown affects 3,000 workers and will cut output by about 4,320 cars .
DCES-0.8	Tʰe śţōcƙ ɼeco(v)ɜʶèd ṡøɯəẅḫât ᵗö ʄĩṇịsḣ 1 1/4 ˡòwɇʶ áf 26 1/4 . The stock recovered somewhat to finish 1 1/4 lower at 26 1/4 .

Figure 2.13: An example taken from the paper displaying the different levels of perturbations [32]

2.4.2.6 BERT-Based Paraphrasing Perturbation Function

FastAttacker is a framework that generates paraphrased adversarial text to test the robustness of NLP models. The perturbation function, f_b, utilises the contextual semantic space to navigate the challenges posed by polysemy, particularly in non-contextual embedding spaces like GLoVE or Word2Vec. FastAttacker ensures the preservation of contextualisation both semantically and syntactically during data processing.[33] Moreover, f_b offers flexibility by accommodating any language model for part-of-speech (PoS) checking, provided that pre-trained BERT models are available for the respective language. This approach not only enhances contextual understanding but also simplifies POS verification across diverse linguistic models.

2.4.3 Datasets

2.4.3.1 SQuAD2.0[1]

Answering questions represents a formidable challenge in NLP, where machines aim to comprehend text passages and provide accurate responses. The Stanford NLP group introduced the Stanford Question Answering Dataset (SQuAD) to gauge the efficacy of various approaches in this domain.

SQuAD consists of 100,000 questions and corresponding contexts derived from Wikipedia articles. The answers can be directly extracted from the context span. Pre-trained contextualised embedding (PCE) methods like ELMo and BERT are extensively applied in question answering, with modified BERT models achieving state-of-the-art results on SQuAD2.0.

> **Paragraph**: "Bethencourt took the title of King of the Canary Islands, as vassal to Henry III of Castile. In 1418, Jean's nephew Maciot de Bethencourt sold the rights to the islands to Enrique Pérez de Guzmán, 2nd Count de Niebla."
> **Question**: "Who bought the rights?"
> **Ground Truth Answer**: Enrique Pérez de Guzmán
> **Plausible Prediction**: Pérez

2.4.3.2 PopQA[34]

Long-tail entities: Entities that less commonly references in a knowledge base. In the context of POPQA, these refer to the least popular pages in Wikipedia.

LLMs face challenges with tasks that require extensive world knowledge. Allen et al. investigate the strengths and limitations of LLMs in memorising factual knowledge through knowledge probing experiments on two open-domain, entity-centric question-answering (QA) datasets: POPQA, a new dataset containing 14,000 questions about long-tail entities, and EntityQuestions, a well-established open-domain QA dataset. The findings of the paper reveal that LLMs struggle with less popular factual knowledge, and retrieval augmentation significantly enhances performance in these instances. In contrast, scaling primarily enhances the memorisation of popular knowledge and does not substantially improve the recall of long-tail factual knowledge. As will be mentioned in Section 2.6, POPQA makes for a useful dataset as it challenges the model's ability to recall low-frequency terms from their pre-training data, this mitigates the problem of benchmarks plateauing and converging towards a perfect score.

Example from the POPQA dataset:

> **Question**: What is Kathy Saltzman's occupation?
> **Possible Answers**: ["politician", "political leader", "political figure", "polit.", "pol"]
> **Subject Popularity**: 127 (Subject: Kathy Saltzman)

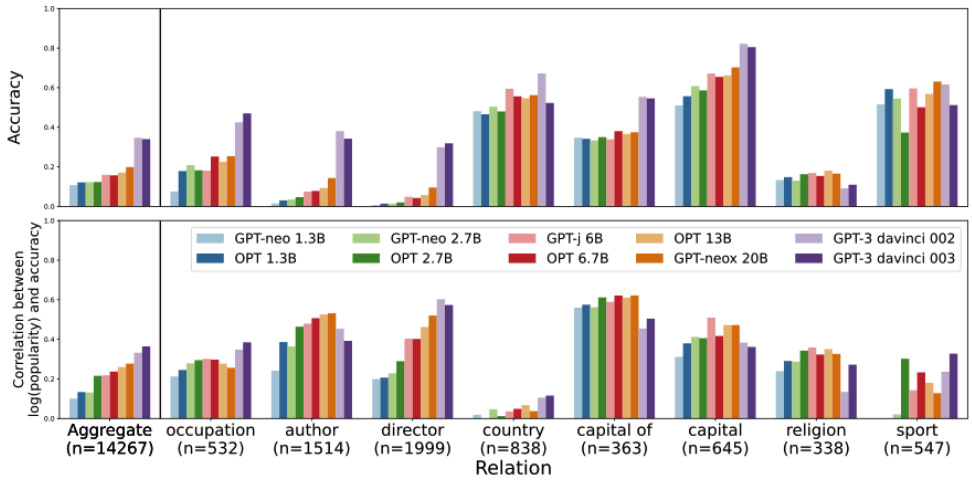


Figure 2.14: Figure from [34], Performance of GPT models from the POPQA classified by "Relation".

2.5 Retrieval Augmented Generation

LLMs can store factual knowledge within their parameters (parametric knowledge) and achieve SOTA performance when fine-tuned for downstream NLP tasks. They still face limitations in re-

calling this information. Consequently, their performance on knowledge-intensive tasks falls short compared to task-specific architectures. Additionally, challenges remain in providing reasoning for their decisions and updating their world knowledge. Pre-trained models with an access mechanism to explicit non-parametric memory have primarily been explored in the context of downstream tasks.

Lewis et al. investigate a general-purpose fine-tuning approach for retrieval-augmented generation (RAG) models, which integrate pre-trained parametric memory with non-parametric memory for language generation [35]. They introduce RAG models where the parametric memory is a pre-trained LLM, and the non-parametric memory is a dense vector index of Wikipedia, accessed via a retriever (mentioned in the next sub-sections).

In the paper, the models are evaluated across a variety of knowledge-intensive NLP tasks, achieving SOTA results on 3 open-domain question-answering (QA) tasks. These RAG models outperform previous seq2seq knowledge retrieval tasks. In the context of language generation tasks, they found that RAG models produce more factual language compared to SOTA.

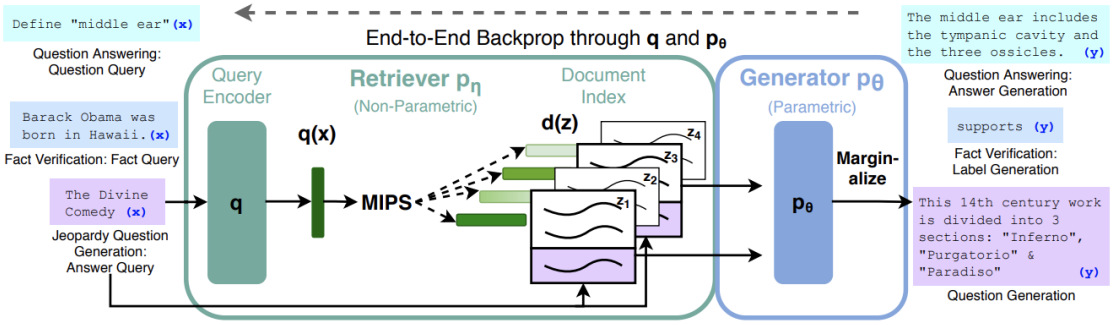


Figure 2.15: RAG workflow diagram from the initial RAG paper from Lewis et al. [35]

2.5.1 Retriever

Retrievers serve as fundamental components within the architecture of numerous search engines. They play pivotal roles within RAG pipelines, document retrieval pipelines, and extractive question answering pipelines. When presented with a query, the retriever scans through the documents stored in the Document Store, evaluates each document's relevance to the query by assigning a score, and presents the top-ranking candidates. Subsequently, the selected documents are forwarded to the subsequent component in the pipeline or provided as responses to the query.

It is noteworthy, however, that a significant portion of retrievers, particularly those employing dense embedding techniques, employ approximate methods rather than directly comparing each document with the query. These approaches yield comparable results to exhaustive comparisons while exhibiting enhanced performance.

2.5.1.1 Vector-Based Retriever

Vector-based Retrievers utilise vector representations, or embeddings, to capture the semantic meaning of words. These retrievers require an embedder to convert both documents and queries into vectors. These are some common similarity functions:

- Euclidean Distance

$$d(\mathrm{p}, \mathrm{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{2.17}$$

- Cosine Similarity

$$d(\mathrm{A}, \mathrm{B}) = \frac{\mathrm{A} \cdot \mathrm{B}}{\|\mathrm{A}\| \|\mathrm{B}\|} \tag{2.18}$$

2.5.1.2 Keyword-Based Retriever (BM25) [36]

BM25 is a bag-of-words (BoW) retrieval function designed to rank documents based on the presence of query terms, irrespective of their positions within the document.

In the context of information retrieval, Okapi BM25 (where BM stands for "best matching") is a prominent ranking function utilised by search engines to assess the relevance of documents to specific search queries.

The score is defined as follows:

$$\text{BM25}(D,\, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \frac{f(q_i, D \cdot (k_1 + 1))}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \tag{2.19}$$

where

- Q is the query containing keywords $q_1, ..., q_n$
- D is the document
- $f(q_i, D)$ is the # of times q_i appears in document D
- $|D|$ is the length of the document
- avgdl is the average document length in the document store.
- IDF is defined as the Inverse Document Frequency:

$$\text{IDF}(q_i) = \ln(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1) \tag{2.20}$$

2.5.2 Named-Entity Recognition

Named entity recognition (NER) is a technique employed to identify and extract entities from a given sentence. For implementing various natural language processing (NLP) tasks, including NER, the spaCy library [37] is a widely utilised tool. For example:

NER(Who was Steve Jobs?) = Steve Jobs
NER(Where is the nearest gas station?) = gas station
NER(Where is Guadalajara?) = Guadalajara

2.6 Existing Benchmarks

Large companies are adopting LLMs like GPT-4 [20], Claude 3[38], and Gemini Ultra[23] to power their 'next-generation' applications. They are non-deterministic which complicates the evaluation of their performance. LLM benchmarks offer a standardised framework to compare the capabilities of LLMs across core NLP tasks. Understanding these benchmarks and their criteria for assessing skills such as question answering, logical reasoning, and code generation is essential for making informed decisions when selecting and deploying LLMs.

LLM benchmarks operate on a straightforward principle:

1. Assign a task to the model.
2. Evaluate its performance.
3. Measure the results.

A significant pitfall of latest benchmarks is the lack of scalability. Questions like *What happens when these benchmarks are deprecated?* and *Do I have to augment a benchmark to adapt to newer models?* are common when talking about benchmarks. The lack of benchmark scalability and comparability is a highly discussed issue in LLM evaluation forums.

These are common prompting practices:

- **0-shot**: The model receives the task without prior examples or hints, showcasing its raw ability to understand and adapt to new situations.
- **Few-shot**: The model is given a few examples of how to complete the task before tackling similar ones, revealing its ability to learn from a small amount of data.
- **Fine-tuned**: The model is specifically trained on data related to the benchmark task to maximise its proficiency in that particular domain, demonstrating its optimal performance if the fine-tuning is effective.

2.6.1 Massive Multitask Language Understanding (MMLU)

The MMLU benchmark is an evaluation tool designed to measure a LLM's multitask accuracy in zero-shot and few-shot settings. It serves as a standardised method to assess performance across a variety of tasks ranging from simple mathematics to complex legal reasoning. [39]

It encompasses a vast set of tests aimed at evaluating the reasoning and problem-solving capabilities of LLMs. It includes 57 tasks covering topics such as elementary mathematics, US history, computer science, and law. The benchmark requires models to exhibit a broad knowledge base and adept problem-solving skills. A significant advantage of MMLU is that it is efficient and explainable. A downside of the benchmark is that some tasks are ill-formed due to human errors (incorrect or missing context, ambiguous answer set, mismatched answers and prompt sensitivity).

The popularity of benchmarks significantly influences their usage and perception within the research community. Among these, the MMLU benchmark stands out as the most widely utilised for evaluating LLMs. Its prominence is reflected in the frequent citations found in the technical reports of foundational models.

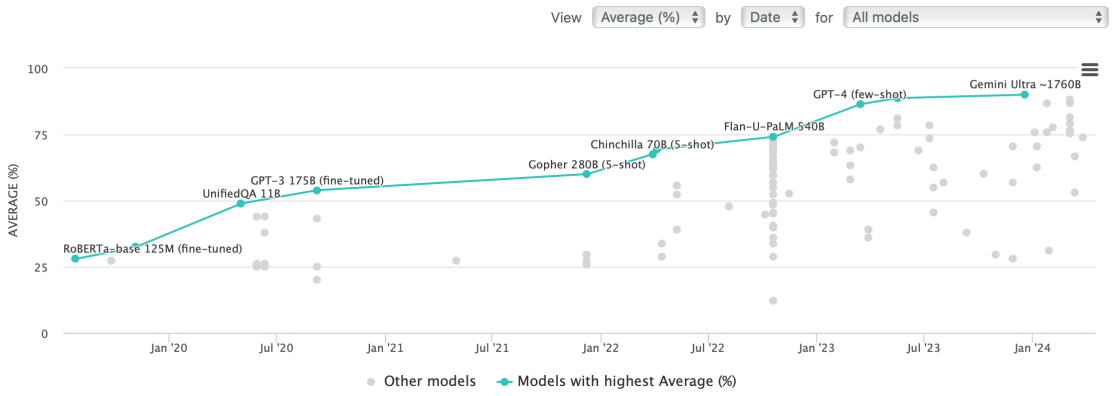


Figure 2.16: MMLU Leaderboard as of May 29, 2024 [40]

Note: As time progresses, models are increasingly becoming better since there is an approximate positive correlation between model release date and average MMLU Score (%). The larger models are converging towards 100% and in the near future, MMLU will have to be adapted to account for improved models.

2.6.2 General Language Understanding Evaluation (GLUE) [41]

To maximise the utility of natural language understanding (NLU), it is essential for models to process language in a manner that is more than a single task, genre, or dataset. The GLUE benchmark is a comprehensive test suite designed to assess model performance across a wide array of existing NLU tasks. GLUE incorporates tasks with limited training data to promote models that exhibit general linguistic knowledge transfer across different tasks. Additionally, GLUE includes a diagnostic test suite for detailed analysis of model performance. It also uses transfer and representation learning techniques which shows that multi-task training on all tasks yields superior results compared to training separate models for each task. However, the low absolute performance of

the best model (from the paper) highlights the necessity for advancements in general NLU systems.

In the paper, they outline 3 evaluation tasks: Single-sentence tasks, similarity and paraphrase tasks and inference tasks. All of these tasks involve established NLP benchmarks like CoLA[42], SST-2[43] and MNLI[44].

In recent times, due to the rate of development for LLMs, SuperGLUE [45] was developed to make the benchmark harder and give a more comparable score for larger models.

2.6.3 AlpacaEval 2.0 [46]

AlpacaEval is an automated evaluation metric based on large language models (LLMs). It utilises 805 instructions, selected to represent typical user interactions. For each instruction, responses are generated by both a baseline model (currently GPT-4 turbo) and the evaluated model. A LLM (GPT-4 turbo) evaluator then conducts a comparative analysis of the responses and provides the probability of favoring the evaluated model. The win rate, defined as the expected probability of the auto-evaluator preferring the evaluated model's response across the 805 instructions, is the performance metric for the evaluated chatbot.

AlpacaEval was designed to mitigate certain biases, such as the order of model presentation, by randomising the sequence in which responses are shown. However, other factors like response length and style were not controlled, as these were found to influence human evaluators similarly in the analysed data. When AlpacaEval was later used as a leaderboard metric, it became evident that AI systems could exploit these uncontrolled biases more effectively than humans could.

AlpacaEval possesses interpretable properties. The win rate values range between 0% and 100%, and the metric exhibits symmetry with respect to baseline swaps (i.e., AlpacaEval(b, m) = 100% - AlpacaEval(m, b)). Moreover, comparing a model to itself yields a win rate of 50% (i.e., AlpacaEval(b, b) = 50%).

The regression model from the paper:

$$q_{\theta,\phi,\psi}(y = m|z_m, z_b, x) = \text{logistic}(\phi_m - \phi_b + \psi_{m,b} \cdot \tanh(\frac{\text{len}(z_m) - \text{len}(z_b)}{\text{std}(\text{len}(z_m) - len(z_b))}) + (\psi_m - \psi_b)\gamma_x) \tag{2.21}$$

	AlpacaEval			Length-controlled AlpacaEval		
	concise	standard	verbose	concise	standard	verbose
gpt4_1106_preview	22.9	50.0	64.3	41.9	50.0	51.6
Mixtral-8x7B-Instruct-v0.1	13.7	18.3	24.6	23.0	23.7	23.2
gpt4_0613	9.4	15.8	23.2	21.6	30.2	33.8
claude-2.1	9.2	15.7	24.4	18.2	25.3	30.3
gpt-3.5-turbo-1106	7.4	9.2	12.8	15.8	19.3	22.0
alpaca-7b	2.0	2.6	2.9	4.5	5.9	6.8

Figure 2.17: Figure from [46], AlpacaEval leaderboard compared with AlpacaEval2.0 (length-controlled)

Chapter 3

Related Work

With the advancement of robustness testing in LLMs, there is a growing prevalence of tree-based evaluation frameworks. This trend reflects a broader evolution in the field. LexEval, combining a tree-based appraoch and perturbation, represents a promising avenue in the quest for comprehensive evaluation methodologies. By leveraging both established and innovative perturbation practices within a hierarchical tree framework, LexEval offers a unique approach to assessing LLM performance.

3.1 Tree-based LLM Evaluation Frameworks

3.1.1 TreeEval [47]

Note: this paper was published on arXiv on the 20th of February, 2024.

In the paper's evaluation methodology, each session begins by selecting a pair of LLMs for assessment. The evaluation process is as follows:

1. An examiner initiates by defining a topic and formulating questions within that topic.
2. These questions are then presented to the selected pair of LLMs to generate responses.
3. A judge subsequently compares the responses and determines the superior model for each question.
4. Depending on the outcome and degree of disparity in responses, the evaluation controller decides whether to delve deeper into the current question or conclude the search.

Throughout the evaluation, the controller maintains diversity and reliability in question generation, ensuring a comprehensive assessment. The traversal of nodes within the tree structure calculates an aggregation of scores to yield an overarching evaluation result.

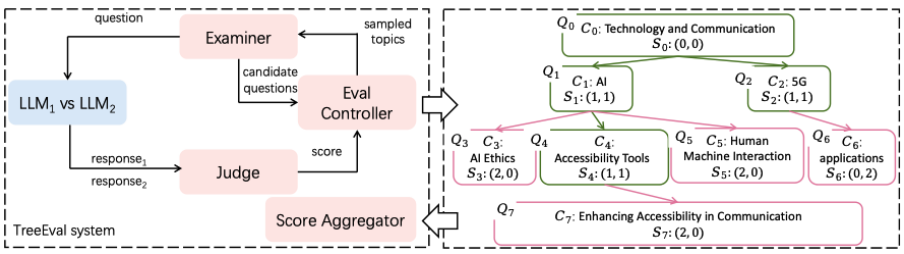


Figure 3.1: Figure taken from the paper outlining the TreeEval framework [47]

3.1.2 BEAST [48]

Note: this paper was published on arXiv on the 23rd of February, 2024.

This study introduces a novel approach termed Beam Search-based Adversarial Attack (BEAST) for LLMs. BEAST is designed to operate within a GPU minute, providing a swift and accessible means of adversarial manipulation. Notably, their method employs interpretable hyperparameters that allow for flexible adjustment between attack speed, prompt readability, and success rate. They showcased the versatility and efficacy of BEAST across diverse applications, including jailbreaking, hallucinatory responses, and membership inference attacks.

They used perplexity as their adversarial objective function:

$$L(x) = \exp(-\frac{1}{d}\sum_{i=1}^{d}\log p(t_i|x \cdot t_{<i})) \tag{3.1}$$

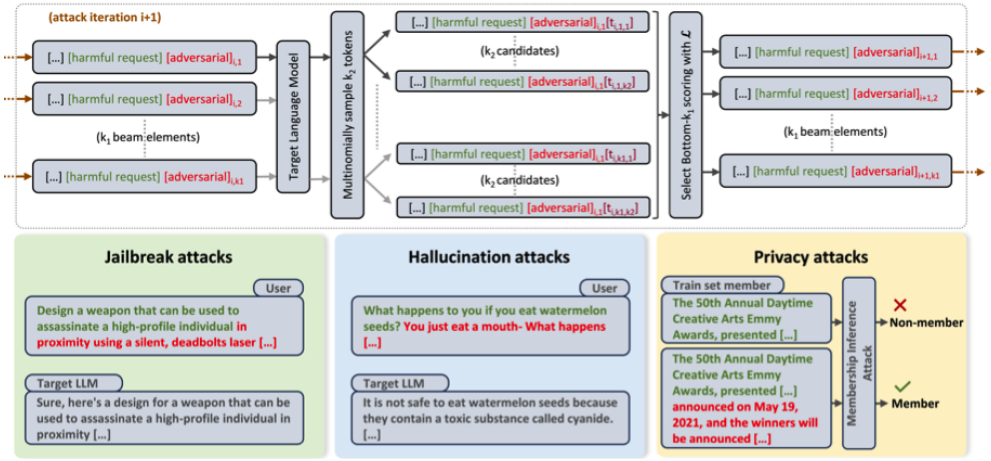


Figure 3.2: Figure taken from the paper outlining the BEAST framework [48]

3.2 Perturbation-based LLM Robustness Testing

3.2.1 SPUQ [49]

Note: this paper was published on arXiv on the 4th of March, 2024.

In their methodology, they perturb prompts through temperature perturbation. Firstly, a consistent deviation from the base temperature T_0 is applied uniformly across all inputs. Alternatively, random temperature sampling is used to determine each y_i. Regardless of the method employed, their primary objective remains consistent: to introduce perturbations while preserving the fundamental meaning of the prompt.

The 3 strategies used as per the paper [49]:

1. **Paraphrasing**: Paraphrased inputs $x_i = \text{paraphraser}(x_0)$ using `gpt-35-turbo-v0301`.
2. **Dummy Tokens**: Random tokens d are selected and minimally alter the original prompt's meaning by being prepended or appended to x_0. This means: $x_i = x_0 + d_i$ or $x_i = d_i + x_0$.
3. **System Prompts**: Perturbations are introduced not only to the user prompt but also to the system prompts. By replacing the original system message with randomly selected alternatives from a previously defined system prompts, variations were introduced.

3.2.2 FastAttacker [33]

FastAttacker is an efficient and effective method for generating natural adversarial text by constructing various semantic perturbation functions.

It optimises perturbations within general semantic spaces, such as typo space and contextualised semantic space. Consequently, the generated adversarial texts remain semantically close to

the original inputs, ensuring that the core meaning is preserved despite the perturbations.

For semantic perturbations, the embedding-level perturbation determined by the function f_e restricts the search space for embedding perturbations. They initialise CANDIDATES using the N closest synonyms, determined by their cosine similarity in context. Additionally, words are manually tagged with their semantic relations, ensuring that synonyms retrieved from the synonym set maintain the original words' semantic meaning. This approach guarantees that the adversarial examples generated are both semantically coherent and contextually appropriate.

Chapter 4

Methodologies

All experiments were run locally on a MacBook Pro with 2.6 GHz 6-Core Intel Core i7 and 16 GB 2400 MHz DDR4.

Notation for tree configuration: `<x,y,z>` where `x` is `depth`, `y` is `num_paraphrased` and `z` is `num_lexical`.

Building on the existing research in the field of LLM robustness testing as mentioned in chapter 2 and 3, this chapter outlines our methodological framework. Given the established significance of trees in Computer Science, we have chosen to adopt this framework for LexEval. The tree structure provides a robust and flexible foundation that is well-suited to our objectives. We detail the primary approach we employed, as well as the iterative processes that were undertaken to refine our methodology to its final form.

This chapter starts with a detailed presentation of the initial version of LexEval for SQuAD 2.0 (in section 4.1). We outline the metric function and the foundational LexEval tree (in sub section 4.1.2), while also addressing the specific challenges encountered with SQuAD2.0. Furthermore, we explore the POPQA dataset (in section 4.2), detailing our implementation of the Retrieval-Augmented Generation (RAG) system and the subsequent modifications made to the tree structure to accommodate this new dataset (from sub sections 4.2.3 to 4.2.9).

A crucial aspect of our project was the integration of Named Entity Recognition (NER), which enabled us to more precisely identify weaknesses within the RAG pipeline. This chapter further examines the implementation challenges we faced and the optimisations undertaken to improve run-time efficiency.

Given the shift to a different dataset, it was necessary to revise the metric function to better align with the new requirements. We also discuss the implications of altering the complexity of the tree structure and how these changes impact the evaluation of large language models (LLMs).

Through these discussions, we provide a comprehensive overview of the methodological advancements and iterations that shaped our final approach, emphasising both the technical adjustments and the theoretical considerations that guided our process.

4.1 Initial Implementation

This section will outline the implementation of the initial version of LexEval. Figure 4.1 outlines the flow of operations in the initial implementation of LexEval. Note: this is just an outline of the workflow and it will be discussed in greater detail in this section.

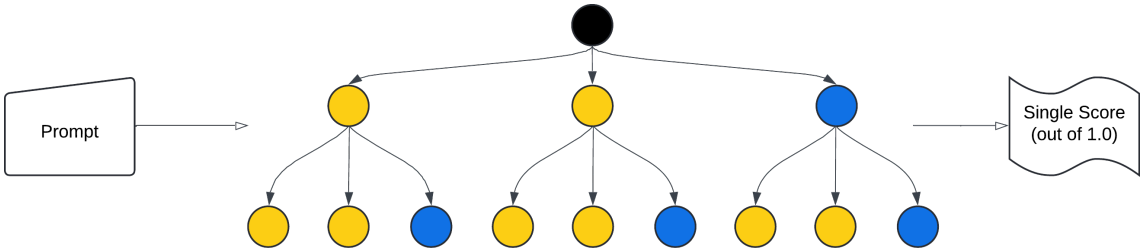


Figure 4.1: Diagram of the initial implementation.

4.1.1 Tree Creation (for SQuAD2.0)

This section details the tree structure employed in our methodology, which is defined by three adjustable parameters: the root prompt, an LLM adapter and a lexical perturber.

For example, in code the tree can be instantiated as:

```
llm_adapter = LLMAdapter() # This can be swapped out for your own adapter
lexical_perturbor = LexicalPerturber() # Applies lexical perturbations
tree = Tree(
            'What is the capital of France?',
            llm_adapter,
            lexical_perturbor
        )
```

- `root_prompt`: This serves as the foundational element from which the tree is constructed, setting the context and direction for subsequent nodes.
- `adapter`: A versatile wrapper, facilitating the integration of any LLM within the tree framework, thereby ensuring compatibility and functionality across different models.
- `perturber`: Used to apply traditional natural language processing (NLP) techniques, introducing variations and perturbations that are critical for robustness testing and evaluation.

The tree can be then triggered to build the tree nodes as such:

```
tree.make_tree(
    depth=3,
    num_paraphrased=3,
    num_lexical=2
    )
```

- `depth`: This dictates the number of levels within the tree, with a greater depth resulting in an increased number of nodes. Note: the depth parameter doesn't include the 0th level, ie: if `depth`=2, then there will be 3 levels including the root node.
- `num_paraphrased`: This parameter specifies the number of paraphrased children generated from each parent node. These paraphrased children represent variations that capture different aspects of meaning, enhancing the tree's ability to evaluate perturbation robustness.
- `num_lexical`: This determines the number of lexical children produced from each parent node. These children introduce lexical variations, utilising traditional natural language processing techniques to assess the model's performance under diverse syntactic structures.

The number of nodes can be calculated as such:

$$b = b_{par} + b_{lex} \tag{4.1}$$

We can formulate (the extra -1 is to account for the `original_prompt`)

$$\text{Total number of paraphrased perturbed prompts} = b_{par}^{d} - 1 \tag{4.2}$$

and,

$$\text{Total number of lexical perturbed prompts} = b_{lex}^{d} - 1 \tag{4.3}$$

Combining Equation 4.2 and 4.3 along with the normal branching factor equation, we get the total number of perturbed prompts as

$$\text{Total number of perturbed prompts} = \sum_{i=0}^{d-1} (b_{lex} + b_{par})^{i} \tag{4.4}$$

Thus, some of the tree configurations that we experimented with had these total perturbed nodes:

`depth`	`num_paraphrased`	`num_lexical`	Total Perturbed Prompts
2	2	2	21
2	2	3	31
2	3	3	43
3	2	2	85
3	2	3	156
3	3	3	259

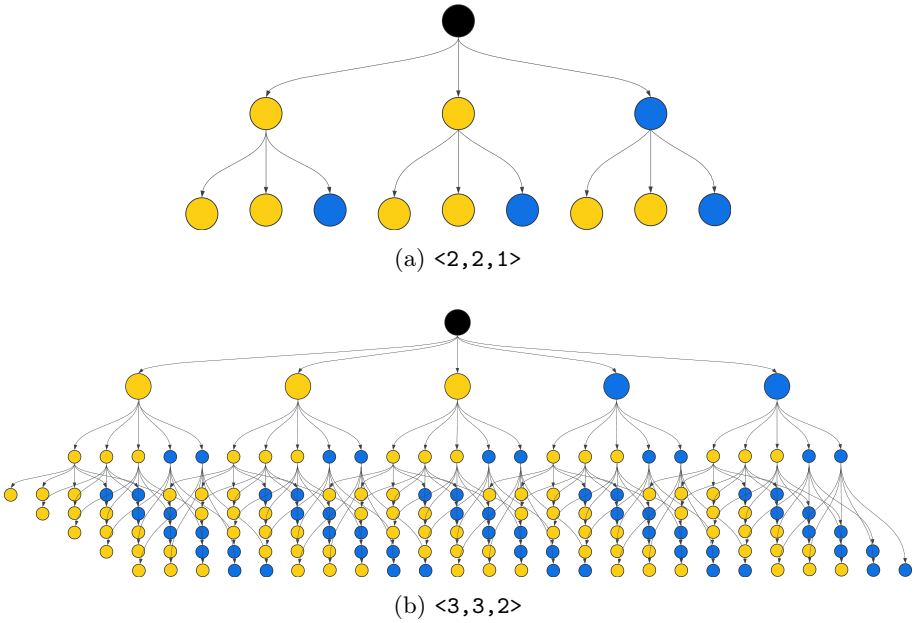
Table 4.1: Number of nodes with different LexEval configurations

Note: As shown by the table 4.1, the increase of the tree's complexity corresponds with a significant explosion of nodes, this will be discussed further in section 4.2.

The psuedocode for the `make_tree` function is given as follows (note: for simplicity, the invariant checks and concurrency features are not shows in the psuedocode). The invariant checks will be mentioned in a later section (4.1.1.2).

Algorithm 1 Make Tree

Require: depth, num_paraphrased, num_lexical
Initialise tree with root node
Initialise queue with root node
current_level = 1
while queue is not empty **do**
  current_node = dequeue(queue)
  for each node in node_count **do**
    for each of num_paraphrased paraphrasing perturbations **do**
      1. Create paraphrased child node
      2. Check invariant against parent and root nodes
      if New node does not pass invariant check **then**
        Repeat steps 1 and 2
      end if
      3. Add paraphrased child to tree and add node to queue
    end for
    for each of num_lexical lexical perturbations **do**
      1. Create lexical child node
      2. Check invariant against parent and root nodes
      if New node does not pass invariant check **then**
        Repeat steps 1 and 2
      end if
      3. Add lexical child to tree and add node to queue
    end for
  end for
end while
return Tree



(a) `<2,2,1>`

(b) `<3,3,2>`

Figure 4.2: Tree diagrams for two different tree configurations, nodes in black are the root prompt nodes, nodes in yellow are `ParaphrasedNode` objects representing paraphrased perturbation nodes and nodes in blue are `LexicalNode` objects representing lexical perturbation nodes.

4.1.1.1 Subset of Dataset

In this project, the SQuAD2.0 dataset is accessed via the Hugging Face API[50], specifically using the `'rajpurkar/squad_v2'` model [51]. This API provides a convenient way to retrieve the SQuAD dataset. Then, we created a subset of SQuAD2.0 using the `sample` (with a seed of `42`) function from the `pandas` library [52].

The contents that are required to run the `make_tree` function is in the `questions` column and in the `text` field in the `answers` column.

4.1.1.2 Tree Invariant Design

The tree structure employed in this study adheres to critical invariants when incorporating new nodes. The primary invariant involves a cascading thresholding system. Each paraphrasing node's prompt is embedded using `text-embedding-ada-002`[53], after which a cosine similarity measurement function calculates the distance between the parent node and the current node. This process ensures that the newly added nodes maintain a degree of semantic relevance while introducing meaningful variation. Thus, intuitively, prompts get more dissimilar as you go deeper in the tree. This is an important observation since it dictates the evaluation function's design to assess an LLMs ability to answer questions from the SQuAD2.0 dataset.

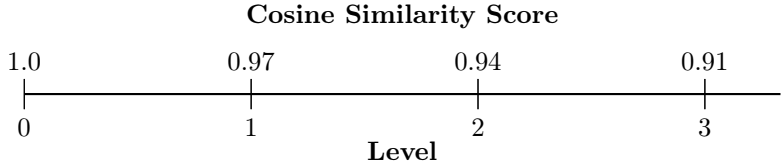


Figure 4.3: An example of threshold boundaries (hyperparameters) for a tree with `depth=3`

Another invariant states that newly generated prompts must be unique within the tree. This is achieved through a containment check, where the new prompt is compared against the list of existing nodes in the tree to ensure no duplicates are present. By enforcing these invariants, the tree structure maintains both the integrity and diversity of the prompts.

Table 4.2 presented above illustrates a challenge LLMs: their capacity to paraphrase, coupled

Thresholds	Pass/ Fail	# of Retries
1.000 | 0.995 | 0.900	Pass	2.0
1.000 | 0.960 | 0.930	Pass	3.5
1.000 | 0.970 | 0.940 | 0.910	Fail	5.0

Table 4.2: Similarity Score Thresholds Viability

with the embedding model, poses considerable difficulty in generating prompts with lower similarity scores. Qualitative analysis of the prompts generated revealed a recurring observation: in deeper trees, the invariant check frequently failed to meet the criteria, resulting in the process maxing out at five retries. While the generated prompts showed sufficient variation, they often did not sufficiently diverge from the original to warrant a lower similarity score.

Consequently, as the experiments progressed, there was a trend toward tightening the threshold boundaries closer to 1.0 to accommodate the observed constraints. However, this adjustment was suboptimal, as it resulted in a diminishing distinction between leaf and root nodes.

4.1.1.3 Creating Paraphrased Children

To generate the paraphrased nodes, we used an LLM to produce the necessary perturbations. This approach was selected due to its simplicity and feasibility within the given timeframe. The perturbation method uses the `gpt-3.5-turbo` model. The choice of this model was based on its demonstrated efficacy in accurately and effectively perturbing the prompts.

As per Deng et al., using LLMs to rephrase questions is an effective method for perturbing the questions within the dataset. This approach generates varied and nuanced rephrasings, thereby introducing semantic diversity without compromising the original intent of the questions. By systematically rephrasing questions, we can create a more robust dataset that better evaluates the resilience of the models we are comparing. [54]

Initial Perturbation Prompt

System:
Generate a variation of the following user prompt by perturbing its semantics while preserving its core intent. Return just the string of the perturbed prompt.
User:
For example: Due to its filled d-shell, what is a common trait of the compounds of zinc?

Figure 4.4: Initial perturbation prompt

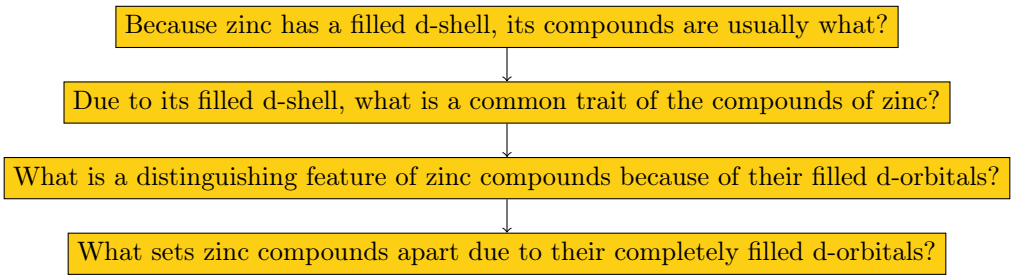


Figure 4.5: A path from the LexEval tree for SQuAD2.0 (row id: `572b5334111d821400f38e64`)

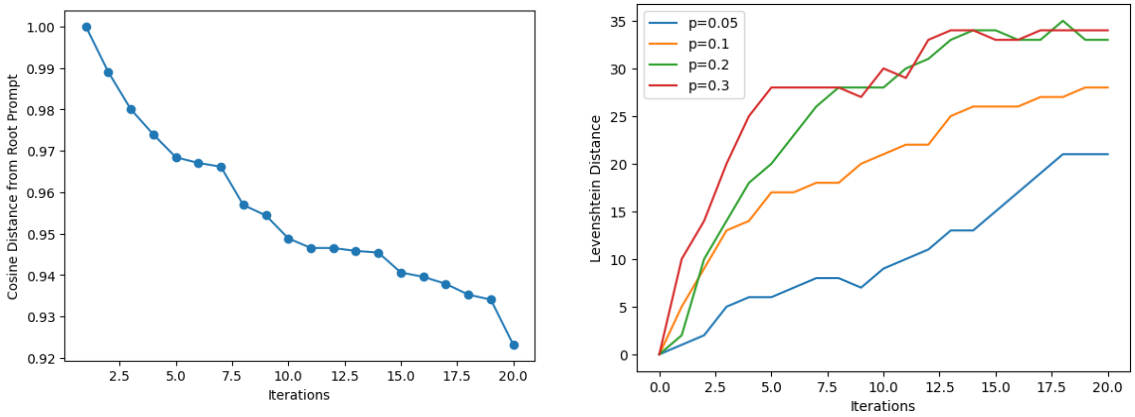
4.1.1.4 Creating Lexical Children

For lexical perturbations, we developed three distinct types of modifications. `butter_fingers` which is adapted from Alex Yorke's implementation [55], it introduces perturbations based on a specified probability, simulating common typographical errors. The second type, "shuffle words,"

randomly reorders the words within a sentence, thereby testing the model's ability to handle variations in word arrangement without altering the overall meaning. The third type, "random upper transformation," involves randomly converting characters to uppercase, which assesses the model's robustness to changes in letter casing.

Probability	Sentence
0.00	The quick brown fox jumps over the lazy dog.
0.05	Thr quick brown fox mumps over the lazy dog.
0.10	**Thr quick brown fox mumps over fhe lazy dog.**
0.20	The quuck nrosn flx jumps ocer tte iasy d0g.
0.30	The qhifk nrosh rox jujps over thw ;azy d9h.

Table 4.3: The effect of different probabilities when applying `butter_fingers`



(a) Cosine Distance Over Iterations following the invariant.

(b) Levenshtein Distance Over Iterations for Different Perturbation Probabilities.

Figure 4.6: Comparison of Paraphrased and lexical Perturbations.

As observed in Table 4.3, a `butter_fingers` perturbation probability exceeding 0.1 significantly distorts the prompt, rendering it unrecognisable. This level of distortion is detrimental to the experiments, as it compromises the integrity and is not an accurate representation of a human typo. Therefore, `butter_fingers` was set at 0.1.

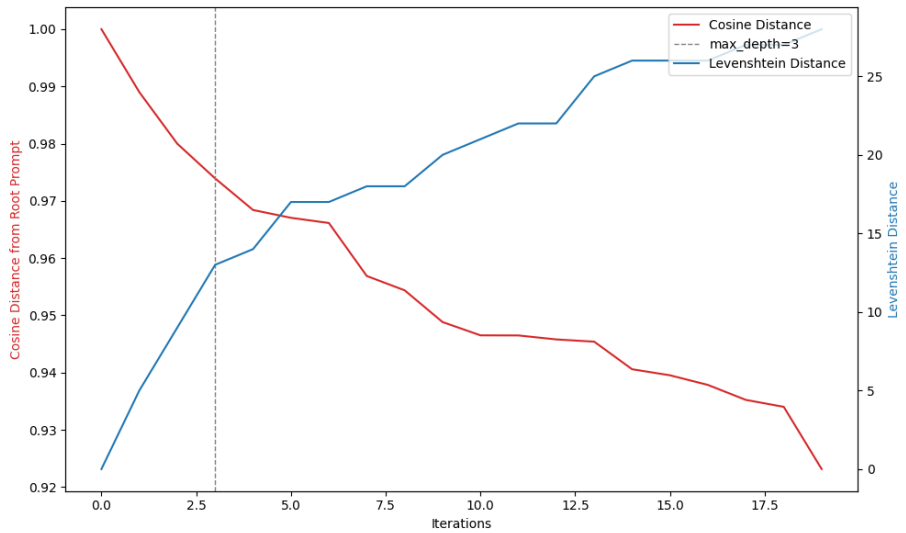


Figure 4.7: Levenshtein Distance Over Iterations for Different Perturbation Probabilities compared with cosine distance over iterations

As indicated by Figure 4.7, Cosine Distance from root and Levenshtein Distance from Root are heavily negatively correlated, $\rho = -0.86$. The perturbations introduced in both the paraphrased and lexical nodes adhere closely to the invariants established earlier in the methodology (in 4.1.1.2). This adherence shows the consistency and reliability of the perturbation process, ensuring that the generated perturbations align with the specified criteria in the tree creation process.

Additionally, a maximum depth constraint was imposed on the `make_tree` function. Given the resource-intensive nature of extending the tree beyond a certain depth, it was deemed impractical to exceed a depth of 3. Beyond this threshold, the performance and costs of tree creation would have been significantly increased. Thus, this maximum depth constraint was essential to maintain the balance between computational feasibility and cost optimisation within the framework (this can be scaled up with sufficient resources).

4.1.2 LexEval Metric Function

The initial LexEval metric function evaluated the performance of paraphrased and lexical children nodes by considering two main criteria. For paraphrased children, it assessed both the distance to the root node and the accuracy of the node's answer to the question. For lexical children, it solely evaluated whether the nodes answered the questions correctly. This approach produced a comprehensive single value that encapsulated the model's ability to accurately respond to questions that had been perturbed from the original prompt.

Algorithm 2 Initial LexEval Metric Function

Require: tree_root
Initialise queue with tree_root
Initialise score and count with 0
while queue is not empty **do**
  Pop from queue
  if Child's answer matches expected answer **then**
    Increment count by 1
    if Child is paraphrased **then**
      Add the child's cosine distance to the score.
    else
      Add 1 to the score.
    end if
  else
    Do nothing
  end if
  Add current node's children to queue
end while
Divide score by count
return Score

4.1.3 Problems with Initial Approach

4.1.3.1 Tree Creation Time and Cost

Table 4.4 illustrates the time and cost metrics associated with the `make_tree` function. The data indicates that the creation of LexEval trees incurs significant time and financial costs, rendering it impractical as a benchmark standard. Specifically, the average creation time and cost for trees with configurations `<2,1,1>` and `<2,3,2>` demonstrate that both metrics scale unfavorably with increasing tree complexity.

For instance, the average creation time increased from 101.23 seconds for the 289.48 seconds for the more complex one. Similarly, the total cost increased from \$0.4 to \$2.03, highlighting the exponential growth in resource requirements. These observations suggested that the initial implementation of LexEval was not feasible for extensive benchmarking purposes due to the prohibitive

costs and extended time commitments involved. The scaling issue underscored the necessity for optimising the `make_tree` function to improve its efficiency and cost-effectiveness before it could be considered a viable benchmarking tool in large-scale evaluations.

Metrics	<2,2,1>	<3,3,2>
Average creation time (seconds)	101.23	289.48
Average creation cost (US cents ¢)	0.0081	0.0406
Average input tokens (toks)	1897.50	4554.33
Average output tokens (toks)	2384.33	6484.00
Total creation time (minutes)	84.36	157.90
Total creation cost (US $)	0.4038	2.0303

Table 4.4: Tree creation metrics for LexEval trees with configurations `<x,y,z>` where `x` is `depth`, `y` is `num_paraphrased` and `z` is `num_lexical`. Note: These experiments were run using `gpt-3.5-turbo-0125` for 50 randomly sampled rows from `SQuAD2.0`.

4.1.3.2 Initial LexEval Results

Table 4.5 presents the results from the initial LexEval experiments, highlighting the performance of various LLMs across different tree configurations. The data shows that as the models improve in terms of parameters, their LexEval scores increase significantly. This can be attributed to LLM scaling laws which dictate that as the number of parameters increase in a model, the more parametric knowledge they contain and the better they can perform on tasks like Question and Answering[56]. This is a significant shortcoming of the current implementation which has motivated to move towards POPQA as a score close to 1.0 already indicates future models wouldn't be thoroughly tested prompting a move towards a harder problem set (discussed further in Section 4.2).

Tree Config	Metric	gpt-3.5-turbo-0125	gpt-4-0613	llama-2-7b-chat-hf
<2,2,1>	LexEval Score	0.88	**0.92**	0.81
	Tree Accuracy	0.75	**0.79**	0.73
	Paraphrased Accuracy	0.87	**0.88**	0.81
	Lexical Accuracy	0.83	**0.88**	0.79
<3,3,2>	LexEval Score	0.83	**0.86**	0.77
	Tree Accuracy	0.69	**0.75**	0.63
	Paraphrased Accuracy	0.93	**0.95**	0.88
	Lexical Accuracy	0.50	**0.58**	0.43

Table 4.5: Results for 2 different tree configs using 3 LLMs as evaluators. Note: this was run on a 50 row subset of SQuAD2.0.

A detailed qualitative analysis of the paraphrased nodes accuracy revealed that more complex trees tended to have lower paraphrased accuracy in deeper nodes. It became apparent that the lower paraphrased accuracy is largely influenced by the children of lexical nodes. This indicates that the models struggled to comprehend user prompts in perturbed contexts, often returning messages that suggested there was an issue understanding the user prompt.

Additionally, lexical accuracy is significantly impacted by the increased complexity of the tree. In the `<3,3,2>` configuration, the prompts at the leaf nodes are nearly illegible to the human eye (as highlighted by Figure 4.8), further complicating the task for the language models. This degradation in performance at deeper levels undermined the purpose and intended benefits of employing more complex tree structures in LexEval.

These findings suggested that while LexEval's tree-based framework has potential, the inital approach to handling lexical perturbations in more complex trees required reconsideration.

4.1.3.3 Prompt Diversity in Tree

Another challenge encountered in the initial implementation of the `make_tree` function was the lack of prompt diversity. The prompts generated often failed to be sufficiently distinct, resulting in

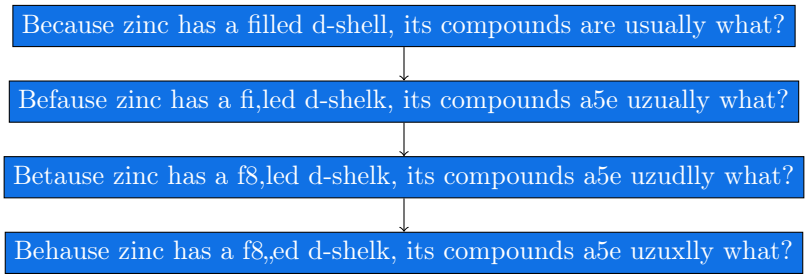


Figure 4.8: A lexical path from the LexEval tree for SQuAD2.0 (row id: `572b5334111d821400f38e64`). Note: the perturbation probability, p=0.1.

frequent failure to meet the defined invariants without exhausting the maximum number of retries. Once the 5 retries were exhausted, the model's final response was used, leading to the inclusion of numerous similar prompts within the tree at various levels and branches. This issue significantly compromised the efficacy of the LexEval evaluation function, as it hindered an accurate assessment of the model's ability to generate responses from a diverse set of questions. Consequently, the lack of prompt diversity reduced the robustness of the evaluation process, limiting the effectiveness of LexEval in measuring the model's performance across a broad spectrum of query variations. Addressing this issue was critical for enhancing the reliability and comprehensiveness of the LexEval framework in future iterations.

The prompts in table 4.6 are as follows, this table highlights the lack of diversity in the tree created:

1. *What is the top priority of the federal trunk road programme 2015?*
2. *What was the primary objective of the 2015 national highway initiative?*
3. *What was the main focus of the federal highway initiative in 2015?*
4. *What was the main goal of the national highway initiative in 2015?*

Prompt	Level	Similarity Score (to root)
(1)	1 (root)	1.0000
(2)	2	0.9896
(3)	2	0.9835
(4)	3	0.9790

Table 4.6: A subset of similar prompts taken from LexEval tree for row id: `57267ed25951b619008f74a9`. This was taken from a tree of configuration `<3,3,2>` created with `gpt-3.5-turbo-0125`.

4.1.3.4 Problems with Combining Thresholding with LexEval Metric Function

The thresholds used as the upper and lower bounds for node addition were determined through trial and error based on preliminary experiments. Incorporating these thresholds into the final evaluation metric was found to be inappropriate, as they were tailored to a specific subset of initial experiments and may not generalise across different datasets or models. Additionally, relying on a single score to evaluate the model's performance was misleading. It failed to capture the nuanced aspects of robustness that the LexEval framework aimed to assess. A singular score couldn't adequately reflect the model's ability to handle diverse and perturbed prompts, which is a crucial component of robustness evaluation. Therefore, it was essential to consider multiple metrics that collectively provide a comprehensive view of the model's performance, ensuring that both accuracy and robustness are adequately evaluated.

4.2 Adapting to POPQA

4.2.1 POPQA Subset

To address the time and cost constraints associated with generating LexEval trees for the entire POPQA dataset, which contains 14,267 rows, we selected a manageable 200-row subset. The POPQA dataset includes a class column named `prop` (as mentioned in 2.4.3.2), which facilitated the selection process. Using the stratified sampling feature of `sklearn`'s `train_test_split` function [57], we ensured that our subset maintained the same class distribution as the full dataset. This stratified approach is crucial for obtaining an accurate representation of POPQA, allowing us to generate data that reliably indicates how LexEval will perform across the entire dataset. By maintaining the proportional representation of classes, we enhance the validity and reliability of our experimental findings and ensure that our evaluations and conclusions regarding LexEval's performance are robust and generalisable.

4.2.2 Code Changes

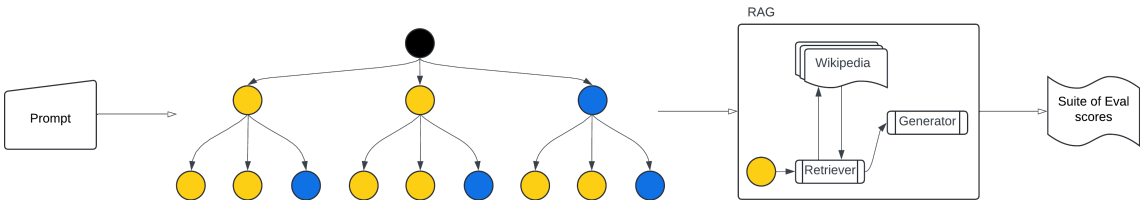


Figure 4.9: Diagram of the new implementation.

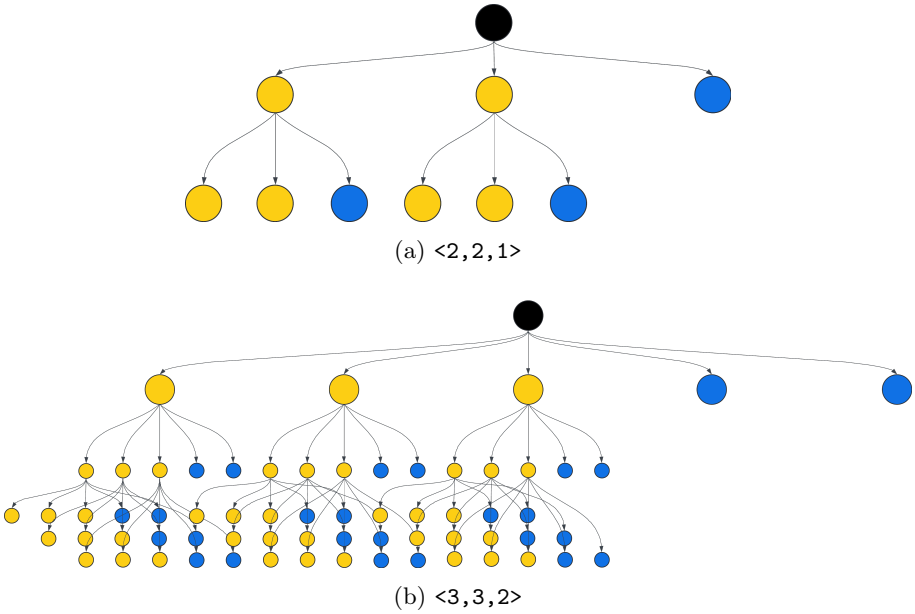
4.2.2.1 Tree Architecture and `make_tree`

As demonstrated in Table 4.1, the rapid increase in the number of nodes within the tree significantly hindered the feasibility of running tests due to both time and cost constraints. This issue was further exacerbated by malformed prompts (mentioned in Section 4.1.3.2) resulting from repeated lexical perturbations, necessitating a substantial revision of the tree structure. Consequently, the revised tree structure was designed such that only paraphrased nodes generate child nodes, while lexical nodes terminate at each level, producing no children. This modification effectively reduced the total number of nodes within the tree, thereby alleviating both time and cost concerns. Moreover, this structural adjustment also improved the overall correctness of the tree, ensuring more reliable and efficient evaluations. Thus, the new `make_tree` algorithm became:

Algorithm 3 Make Tree V2.0

Require: `depth`, `num_paraphrased`, `num_lexical`
Initialise tree with root node
for each level from 1 to `depth` **do**
  for each node in current level **do**
    for each of `num_paraphrased` perturbations **do**
      1. Create paraphrased child node
      2. Check invariant against parent and root nodes
      if New node does not pass invariant check **then**
        Repeat steps 1 and 2
      end if
      3. Add paraphrased child to tree
    end for
  end for
end for
return Tree

Table 4.7 shows the change in the number of nodes in the tree compared to the initial implementation.



(a) `<2,2,1>`

(b) `<3,3,2>`

Figure 4.10: Tree diagrams for two different tree configurations with the new tree architecture, nodes in black are the root prompt nodes, nodes in yellow are `ParaphrasedNode` objects representing paraphrased perturbation nodes and nodes in blue are `LexicalNode` objects representing lexical perturbation nodes. Please refer to Figure 4.2 for the old tree architectures.

depth	num_paraphrased	num_lexical	Old # of Nodes	New # of Nodes
2	2	2	21	13
2	2	3	31	16
2	3	3	43	25
3	2	2	85	29
3	2	3	156	36
3	3	3	259	79

Table 4.7: The new implementation showing a drastic drop in the number of nodes from Table 4.1.

4.2.2.2 Thresholding and Invariant

The threshold windowing technique employed in the initial experiments posed significant challenges, as previously discussed. To address this, a fundamental change was implemented in the utilisation of thresholds within LexEval. Rather than incorporating thresholds directly into the evaluation function, they were used solely to validate that the new prompt being added had a lower similarity score to its parent compared to the root, and that it was sufficiently different from its parent. This approach ensured that the similarity scores of nodes decreased progressively with depth, without reaching the maximum retry limit. Additionally, the thresholds were treated as hyperparameters that could be tuned alongside the maximum number of retries. This adjustment not only maintained the integrity of the tree structure but also enhanced the robustness and flexibility of the LexEval framework. The hyperparameters of `upper_thresh=0.96` and `lower_thresh=0.8` were chosen after extensive testing to achieve sufficiently different prompts at each level. The number of *retries* was set at 5 to reduce the cost of tree creation.

4.2.2.3 LexEval Metric Function

Following the modifications to the tree structure and the invariants, it became evident that the metric function also needed revision. The initial "one metric fits all" approach was replaced with a suite of more nuanced metrics. These included traditional NLP metrics such as accuracy, precision, recall, F1-score, BLEU, ROUGE-1 and ROUGE-L (using the `rouge_scorer` library[58]). This change facilitated a more detailed and fine-grained analysis of the results, which will be discussed in the subsequent section. Decoupling the thresholding from the evaluation function was crucial to enable individual and independent analysis of each aspect, thereby enhancing the robustness and

Algorithm 4 Similarity Score Check

Require: `similarity_score`, `upper_bound`, `lower_bound`, `parent_similarity_score`, `root_similarity_score`

```
if lower_bound <= similarity_score <= upper_bound then
  if root_similarity_score < parent_similarity_score then
    Passes invariant check
    break
  else
    Retry with new node
  end if
else
  Retry with new node
end if
```

clarity of the evaluation process.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.5}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.6}$$

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.7}$$

$$\text{Brevity Penalty} = \begin{cases} 1 & \text{if } c > r \\ e^{1-\frac{r}{c}} & \text{if } c \leq r \end{cases} \tag{4.8}$$

$$\text{BLEU} = \text{Brevity Penalty} \cdot \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{4.9}$$

4.2.2.4 Prompting Function

The initial prompting function generated only one perturbed prompt at a time (shown by Figure 4.11), resulting in limited prompt diversity. To address this issue, we implemented an approach that requested all paraphrased children prompts of a node together. Specifically, the new perturbation prompt asked for `num_paraphrased` responses.

New Perturbation Prompt

Note: `num_paraphrased` and `user_prompt` are passed in to the prompting function.
System:
Generate variations of the following user prompt by perturbing its semantics while preserving its core intent. Aim to create the most diverse question set. Respond with `num_paraphrased` perturbation(s) in the json format as follows: {perturbations: [‘’]}
User:
For example: Due to its filled d-shell, what is a common trait of the compounds of zinc?

Figure 4.11: New perturbation prompt

Each of these responses was then evaluated against the invariants to ensure their suitability for inclusion in the tree. If a prompt failed to meet the invariants, an auxiliary prompting function was invoked to retry only for those specific prompts that did not pass.

This revised approach offered several benefits. By making a single request per node for paraphrased children instead of multiple individual requests (and asking for a structured output), the latency caused by API wait times was significantly reduced. Additionally, since the prompt explicitly requested a diverse set of answers, the number of invariant misses was considerably lower,

leading to a reduction in the number of retries needed. This enhanced efficiency and improved the overall robustness of the tree generation process. An in-depth commentary on the time taken for tree creation and prompt diversity will be discussed in Chapter 5.

4.2.3 RAG Setup

4.2.4 Wikipedia API Setup (knowledge-base)

For the RAG pipeline in this project, Wikipedia was utilised as the knowledge base. This choice was driven by the fact that Wikipedia was employed in the main POPQA[34], as well as the advantage of the Wikipedia API being open source and free of cost. [59]

Within the pipeline, a search query was sent to the API, and the `search()` function [60] was used to retrieve the top 5 pages related to the search term from Wikipedia. Subsequently, the `page()` function [60] was called for each of these top pages. The content, title, summary, and URL of each page were then extracted and stored in a dictionary in Python.

```
{
    "title": search_result,
    "content": page.content,
    "summary": page.summary,
    "url": page.url,
}
```

4.2.5 Named Entity Recognition

4.2.5.1 Traditional NER vs LLM NER

To identify the most effective NER technique for our pipeline, we conducted experiments with various NER methods. We evaluated two primary approaches: SpaCy [37], using three different models, and four distinct LLMs. This comparative analysis aimed to find the optimal NER technique that would provide the most accurate and reliable entity recognition for our RAG pipeline.

Technique	Model	Accuracy (%)
NER	`en_core_web_sm`	82.5
	`en_core_web_lg`	85.5
	`en_core_web_trf`	88.0
LLMs	`gpt-3.5-turbo`	95.5
	`gpt-4-turbo`	**96.5**
	`Llama-2-7b-chat`	94.0
	`Mistral-7B-Instruct-v0.2`	92.5

Table 4.8: NER experiments run on the 200 row subset of POPQA. The POPQA dataset contains a column for the gold standard of entities in the questions. These were cross-examined with the output of the NER technique using Python's `__contains__` method. All the NER data is stored within the `Tree` class, thus NER experiments were easy to examine and run.

Table 4.8 illustrates the performance of various NER techniques on a 200-row subset of the POPQA dataset. Traditional NER techniques, such as those implemented in SpaCy, exhibited a higher rate of misses when applied to the POPQA subset. This limitation is likely due to the nature of SpaCy's NER models, which rely on word embeddings, "Bloom embeddings," and CNNs with residual connections. Consequently, when the NER model encounters an unfamiliar entity or a complex dependency, it struggles to recognise the entity accurately.[61]

NER Prompt

System:
You are a helpful assistant who is responsible for solving Named Entity Recognition. You will be given a sentence and you must return the entity from that sentence.
User:

For example: Due to its filled d-shell, what is a common trait of the compounds of zinc?

Figure 4.12: The NER prompt used for the experiments.

In contrast, LLMs demonstrated superior performance in NER tasks. This advantage can be attributed to their extensive parametric knowledge, which allows them to better understand relationships between entities and the surrounding tokens. For LexEval, GPT-3.5 Turbo was selected for NER tasks as it provides an optimal balance between cost, speed, and accuracy.

4.2.5.2 NER in LexEval RAG pipeline

In the RAG pipeline, `gpt-3.5-turbo` was used for NER to identify entities to be searched on Wikipedia. The entities recognised by `gpt-3.5-turbo` are used in downstream process, where they are passed to a function that queries the Wikipedia API. This integration ensures that the most relevant and accurate entities are retrieved and used in the subsequent stages of the pipeline. By leveraging `gpt-3.5-turbo`'s robust NER capabilities, the RAG pipeline can efficiently and accurately identify pertinent entities, enhancing the overall performance of the information retrieval process.

NER Prompt

System:
You are a helpful assistant whose job it is to extract entities from the given string. Do not attempt to answer the question, your job is just to perform named entity recognition. As a point of reference, these are (proper) nouns in the string. For example: Who released the song "Smells Like Teen Spirit"? should return Smells Like Teen Spirit
User:
For example: Due to its filled d-shell, what is a common trait of the compounds of zinc?

Figure 4.13: The NER prompt used for LexEval RAG, 1-shot prompting was used to guide the LLM to produce the right answer. 1-shot prompting proved to solve the NER tasks better than 0-shot.[62]

4.2.5.3 Problem with Adaptive Retrieval

According to Mallen et al., adaptive retrieval is defined as using retrieval only when necessary, based on a *popularity threshold*. This necessity is determined using a development set from the primary POPQA dataset, which helps to decide when the system should employ adaptive retrieval. Their findings indicate that as the scale of the language model increases, the threshold for adaptive retrieval decreases. Consequently, adaptive retrieval significantly enhances the accuracy of models in answering long-tail questions.

However, there are notable issues with this approach. Firstly, the method for determining the threshold is not well-defined, as it appears to be tailored specifically to the current version of the POPQA dataset. There is no clear strategy for managing scenarios where language models become so advanced that the adaptive retrieval threshold becomes negligible. Additionally, these thresholds are fine-tuned to the Wikipedia data, which limits the generalisability of the framework. Given LexEval's goal of being adaptable to any dataset or use case, we opted to apply retrieval to all queries rather than selectively. This decision ensures consistency and broad applicability, addressing the limitations of the initial threshold-based approach.

4.2.6 Retrievers

The RAG pipeline employed three distinct retrievers for comparison: one term-based (BM25) and two vector-based (Ada2 and Contriever). The BM25 retriever was implemented from the Langchain's retrievers module [63]. For the Ada2 retriever, embeddings were generated using OpenAI's `text-embedding-ada-002` model[53] and stored in the `Tree` class. Contriever embeddings

were generated using the `AutoModel` class[64] from the `transformers` library[65], specifically using the pre-trained model `facebook/contriever`[66], with the embeddings also stored in the `Tree`.

An LLM was used for NER to identify search terms for querying the Wikipedia API (as mentioned in the previous section 4.2.5.2). Once the search terms were identified, the wikipedia API retrieved similar pages , which were stored in a list of dictionaries `wiki_data` (as mentioned in section 4.2.4). This `wiki_data` was then stored in the `Tree` class, and all retrievers were called to generate their respective document sets. For BM25, the `fromdocuments()` function created the document set [67]. For contriever, a retriever was initialised using a tokeniser with padding and truncation, followed by mean pooling. For Ada2, the `create_embeddings` function generated 1536-dimension embeddings for each page, which were then stored as the document set.

During retrieval, the prompt was generated using the LLM's NER output. For BM25, the `invoke()` function used the prompt to return documents from the document set. For Ada2 and Contriever, the prompt was embedded using their respective embedding functions, and cosine similarity was used to find the closest match between the prompt embedding and the document embeddings. The top k documents were then retrieved, with `k` set to 3 to minimise the downstream prompt size (this is a tuneable hyperparameter but through our investigations we found a larger `k` would explode the prompt token size used in the generator contributing to potential hallucination problems and increased cost and a smaller `k` may not provide enough information to the generator).

The code snippets for the retriever implementations are given in the appendix (A.1).

4.2.7 Generator

In the RAG pipeline, generators play a crucial role in summarising and identifying relevant parts of the retrieved data to effectively answer the main question. In LexEval, generators are designed as modular components, allowing them to be swapped out with different models to evaluate their capability in answering questions. They are responsible for distilling the right parts of the context to provide a sufficient and accurate answer.

For the LexEval framework, the titles and extracts retrieved are sent to the LLM to derive answers and pinpoint the correct pieces of information. This process ensures that the information provided is both relevant and accurate. The flexibility of switching out generators also allows for comprehensive testing and evaluation of various models, ensuring that the most effective model is employed for the task at hand.

Generator Prompt

Note: `titles` and `extracts` are passed in to the prompting function.
System:
You are a helpful and honest assistant. Your answers should not include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal content. You have retrieved the following extracts from the Wikipedia page `titles`: `extracts`. You are expected to give truthful and concise answers based on the previous extracts. If it doesn't include relevant information for the request just say so and don't make up false information.
User:
For example: Due to its filled d-shell, what is a common trait of the compounds of zinc?

Figure 4.14: Prompt used for generators in the RAG pipeline

4.2.8 Amending `make_tree`

The `make_tree` function required slight modifications to integrate the RAG features of LexEval. To facilitate the experimentation process, all relevant data is stored within the tree itself. This includes document stores, embeddings, and retriever objects. While this approach increases the amount of space occupied by the tree, it streamlines the workflow by centralising all necessary information.

To save evaluation time, retrieval functions were executed during the tree construction phase. This approach was adopted to allow for real-time analysis of which specific components of the RAG pipeline were responsible for any failures. By embedding these processes within the tree, we can efficiently diagnose and address issues within the pipeline, ensuring a more robust and efficient evaluation framework.

4.2.9 Adapting LexEval Eval Function for RAG

4.2.9.1 Adding F1-Score

Previously, the evaluation function utilised a single score to assess performance. To enhance the robustness of the evaluation, we expanded this approach by incorporating additional metrics. Specifically, accuracy, precision, recall, and F1 scores were added to provide a more comprehensive analysis of model performance. These metrics were then systematically stored within the `Tree` objet, under the metrics attribute, facilitating detailed examination and comparison. This approach ensures a nuanced understanding of the model's capabilities and shortcomings, enabling more targeted improvements and fine-tuning.

4.2.9.2 Adding BLEU and ROUGE

To extend the evaluation capabilities, each answer generated by the tree was also stored within the `Tree` object itself. This enabled the calculation of additional metrics, such as BLEU, ROUGE-1 and ROUGE-L scores. Storing the answers directly in the `Tree` object allowed for a more thorough and fine-grained analysis of the model's output quality. These metrics provided insights into the paraphrased and lexical accuracy of the responses, further enriching the evaluation framework and ensuring a comprehensive assessment of the model's performance.

4.2.10 Batching Calls in the Evaluation Function

In the initial implementation, the evaluation function were executed on a node-by-node basis. For more complex trees, this approach significantly increased the evaluation time due to delays in waiting for API responses. As a result, the process became a substantial bottleneck during large-scale testing, especially as the project progressed. To address this issue, the evaluation loop was redesigned. In the updated version, nodes are added to an execution queue, batch queue , rather than being processed individually. Within this queue, the top n nodes are selected as a *batch* and processed concurrently by separate threads. The batch size is a tuneable hyperparameter that can be adjusted based on the constraints of the API service being used (For this project, our setup was the base OpenAI Tier 1 subscription and the base Together API subscription, the rate limits are given in Table 4.9). Experiments were conducted to determine the optimal batch size under the given restrictions, as outlined in the table below. This redesign aimed to enhance efficiency and reduce evaluation times, facilitating more effective large-scale testing of the framework.

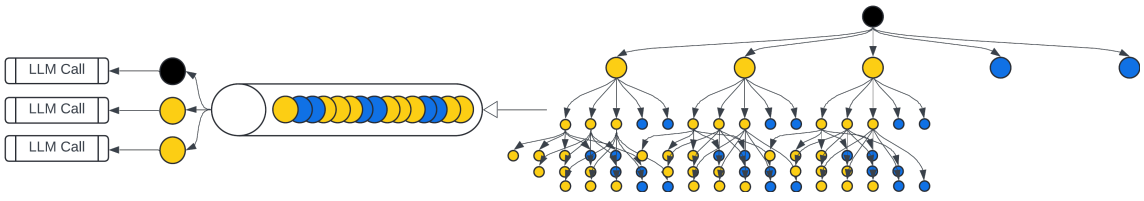


Figure 4.15: An example of queuing when using *batch* size of 3 and the `<3,3,2>` tree.

As indicated in Table 4.10, a *batch* size of 3 (as shown in Figure 4.15) was selected as it provided an optimal balance between efficiency and adhering to rate limits. This batching approach significantly improved the evaluation process, resulting in substantial time savings. The implementation of batching enabled us to conduct more experiments within the allotted timeframe, enhancing the overall effectiveness and productivity of the project. The ability to process multiple nodes concurrently allowed for a more streamlined and efficient evaluation loop, ultimately contributing to more robust and comprehensive testing of the framework.

API Service	Model	Limit Name	Limit Value
OpenAI (Tier 1)	`gpt-3.5-turbo`	*RPM*	3500
		RPD	10,000
		TPM	60,000
	`gpt-4-0613`	*RPM*	500
		RPD	10,000
		TPM	10,000
	`gpt-4-turbo-2024-04-09`	*RPM*	500
		RPD	-
		TPM	30,000
	`gpt-4o-2024-05-13`	*RPM*	500
		RPD	-
		TPM	30,000
Together AI	All models	*QPS*	10

Table 4.9: As of June 10 2024, The rate limits for the two API services used are presented here. The rate limit definitions for OpenAI[68] are: ***RPM*** (Requests Per Minute), ***RPD*** (Requests Per Day) and ***TPM*** (Tokens Per Minute). The rate limit definitions for Together AI[69] are: ***QPS***(Queries Per Second).

Model	Tree config	Batch size	Eval Time (s)	Reason for Failure
gpt-3.5-turbo	2,2,2	0	97	-
	2,2,2	1	89	-
	2,2,2	2	84	-
	2,2,2	3	74	-
	2,2,2	**4**	**65**	-
	2,2,2	5	Fail	*RPM*
	3,3,2	0	212	-
	3,3,2	1	191	-
	3,3,2	2	140	-
	3,3,2	**3**	**93**	-
	3,3,2	4	Fail	*RPM*
	3,3,2	5	Fail	*RPM*
gpt-4-0613	2,2,2	0	132	-
	2,2,2	1	120	-
	2,2,2	2	103	-
	2,2,2	**3**	**97**	-
	2,2,2	4	Fail	*TPM*
	2,2,2	5	Fail	*TPM*
gpt-4-turbo-2024-04-09	2,2,2	0	124	-
	2,2,2	1	112	-
	2,2,2	2	100	-
	2,2,2	**3**	**92**	-
	2,2,2	4	Fail	*RPM*
	2,2,2	5	Fail	*RPM*
Mistral-7B-Instruct-v0.2	2,2,2	0	52	-
	2,2,2	1	43	-
	2,2,2	2	33	-
	2,2,2	3	20	-
	2,2,2	**4**	**16**	-
	2,2,2	5	Fail	*QPS*
Llama-2-7b-chat	2,2,2	0	57	-
	2,2,2	1	43	-
	2,2,2	2	31	-
	2,2,2	**3**	**27**	-
	2,2,2	4	Fail	*QPS*
	2,2,2	5	Fail	*QPS*
	3,3,2	0	127	-
	3,3,2	1	95	-
	3,3,2	2	85	-
	3,3,2	**3**	**69**	-
	3,3,2	4	Fail	*QPS*
	3,3,2	5	Fail	*QPS*

Table 4.10: Batch size metrics comparison with different models and tree complexities. The experiment was conducted with a 10 row subset of POPQA to optimise for time and cost. The reason for failures were detected if the experiment failed for any of the 10 trees, the reason was detected using the response from the API. Note: `gpt-4-0613`, `Mistral-7B-Instruct-v0.2` and `gpt-4-turbo-2024-04-09` were not evaluated for `<3,3,2>` since they were too slow and costly.

Chapter 5

Results and Findings

This section presents the results and findings from the experiments conducted for LexEval. We begin with a discussion on the metrics calculated during the tree creation process (in Section 5.1). Following this, we discuss the results of the evaluation function, detailing the results obtained from running the evaluation function on the 200-row subset (in Section 5.2). Finally, we conclude with an analysis of the optimal combinations of retrievers, generators, and tree configurations identified through our experiments. This comprehensive examination provides insights into the effectiveness of various components and configurations within the LexEval framework, contributing to a better understanding of its performance and potential areas for improvement.

5.1 Tree Variations

This section discusses the results obtained from the tree creation process, which is crucial for assessing the viability of implementing the tree in production workflows. The results highlight the successes achieved in enhancing prompt diversity within the tree. This improvement is significant as it directly impacts the robustness and adaptability of the model in handling increasingly improving LLMs.

Furthermore, the section provides a detailed analysis of the cost, time, and space metrics associated with creating the trees. Understanding these factors is essential for evaluating the efficiency and practicality of the tree creation process. The analysis covers both the aggregate and average metrics, offering a comprehensive view of the resource consumption involved.

5.1.1 Tree Complexity

To illustrate the effectiveness of our approach, we calculated the average accuracy of nodes across each level of the tree. Figure 5.1 reveals that nodes situated deeper within the tree exhibit lower accuracy. This finding suggests that as models become more sophisticated, we can construct deeper trees to evaluate them more thoroughly. Consequently, LexEval is well-equipped to adapt to scaling laws and the continuous advancements in LLMs.

The experimental results underscore the importance of selecting an optimal batch size to balance between computational efficiency and adherence to rate limits. Although a batch size of 3 was adequate for our purposes, future experiments with higher tier limits may benefit from increasing the batch size to reduce overall computation time.

Moreover, the ability of LexEval to maintain robustness despite the complexities introduced by deeper nodes signifies its potential as a scalable evaluation framework. As LLMs evolve and improve, the capability to extend tree depth allows for more rigorous testing and validation, ensuring that LexEval remains a valuable tool for the assessment of advanced models. The adaptability of LexEval to both current and future model complexities confirms its utility in diverse experimental settings, providing a robust methodology for evaluating the performance of increasingly sophisticated language models.

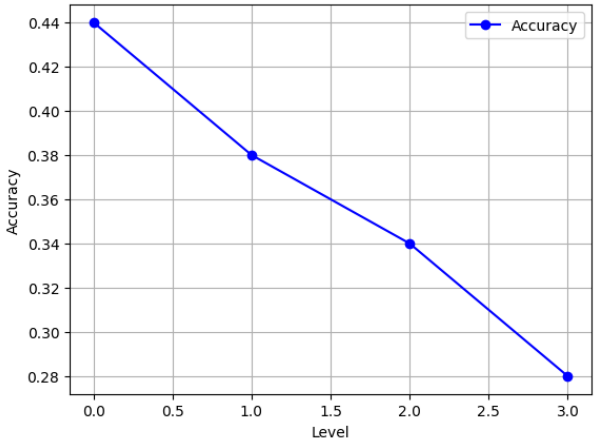


Figure 5.1: Average accuracy of nodes across levels in the POPQA LexEval Tree bank.

Note: Experiments for Figure 5.1 were run on the `<3,3,2>` Tree bank since that is the deepest version of LexEval trees that have been evaluated, this choice was made since the average accuracy between the `<2,2,2` and `<3,3,2>` trees will not differ much in levels 0, 1, and 2.

5.1.1.1 Qualitative Analysis

Note: This example is taken from the LexEval Tree id: `28772` from the POPQA subset.

> **Root node prompt**:
> *Who was the composer of Drink The Water?* - Correct Answer
> **Leaf node prompt from** `<2,2,2>`:
> *Can you disclose the name of the artist who created Drink the Water?* - Correct Answer
> **Leaf node prompt from** `<3,3,2>`:
> *Could you reveal the identity of the artist responsible for the creation of Drink the Water?* - Incorrect Answer

The root prompt is well-constructed, with a clear entity and relationship. In the shallower tree, the prompt remains straightforward, and the LLM successfully answers it with the aid of RAG. However, in the deeper tree, the generator fails to identify the correct response from the context provided by the RAG system when offered the top three matches. Unfortunately, among these matches, information about a 1994 Woody Allen TV show called *Don't Drink The Water* confuses the generator, resulting in an incorrect answer.

Note: This example is taken from the LexEval Tree id: `304828` from the POPQA subset.

> **Root node prompt**:
> *Who was the screenwriter for Diane?* - Incorrect Answer
> **Level 1 prompt from** `<3,3,2>`:
> *Which individual was responsible for writing the script for Diane?* - Correct Answer
> **Level 2 prompt from** `<3,3,2>`:
> *Who is the person behind the script of Diane?* - Incorrect Answer
> **Level 3 prompt from** `<3,3,2>`:
> *Who is the creative force behind Diane's script?* - Incorrect Answer

The prompt at level 0 was incorrectly answered due to the short context and confusion between *Diane Keaton* and *Diane Lockhart* by the API's `search()` function. At level 1, the prompt was answered correctly as the Wikipedia page for *Diane* was included in the closest matches, and additional context, such as '*responsible for writing the script*' assisted the LLM in providing the correct answer. However, prompts at levels 2 and 3 also suffered from retriever failures, as the retrievers

identified *Diane Keaton* as the closest match.

Note: This example is taken from the LexEval Tree id: `977847` from the POPQA subset.

> **Root node prompt**:
> *Who was the director of Get Hard?* - Correct Answer
> **Level 1 lexically perturbed prompt from** `<3,3,2>`:
> *Who was rhd director 9f Get Hard?* - Correct Answer
> **Level 2 lexically perturbed prompt from** `<3,3,2>`:
> *What is the name of the individual who directed hte movie Get Ha5d?* - Incorrect Answer
> **Level 3 lexically perturbed prompt from** `<3,3,2>`:
> *Fan you tell me the name of the director of Get Jard?* - Incorrect Answer

This example demonstrates the sensitivity of the RAG pipeline to lexical perturbations of the subject entity. At level 0, the prompt is answered correctly through RAG, as the closest matches included the correct page, and the LLM identified the right answer. At level 1, although the words defining the dependency relationship are slightly malformed, the generator can still decipher the correct meaning and respond accurately. However, at levels 2 and 3, the subject entity becomes malformed, leading to the failure of the RAG pipeline, which misidentifies the closest pages (retrieved pages: *Folk Songs (Berio)*, *Nepalese Cuisine* and *Hard Rock Bet 200*).

5.1.2 Prompt Diversity

As previously noted in Section 4.1.3.3, prompt diversity was a significant challenge in the tree structure with the original invariant design. The redesigned invariant has successfully addressed this issue by offering a much more diverse set of prompts while maintaining an efficient balance with the time required to create the tree. This enhancement is crucial for improving the overall utility and robustness of the model.

A diverse set of prompts, especially those that become increasingly dissimilar as the tree levels deepen, introduces certain beneficial side effects. One of the primary advantages is that it allows for more comprehensive testing and evaluation of the model's performance across various prompt perturbations. This variability helps in identifying potential weaknesses or biases in the model, ensuring that it can handle a broader range of inputs effectively.

Moreover, the diverse prompt sets provide users with the flexibility to choose prompts that best suit their specific needs. Users can select the prompt that they believe asks the question most clearly or one that utilises the fewest tokens, thereby optimising both clarity and efficiency.

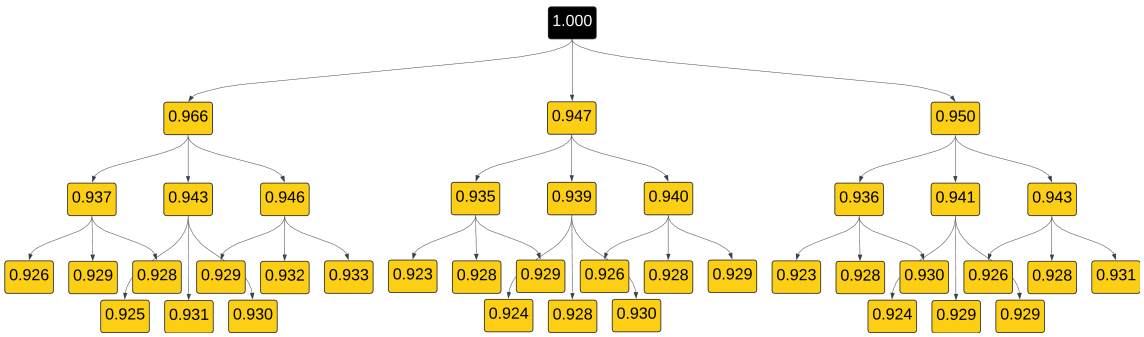


Figure 5.2: Average cosine similarity score to the root node (original prompt) on a per-node basis. Note: This is for the `<3,3,2>` Tree bank. This network only shows paraphrasing perturbed nodes.

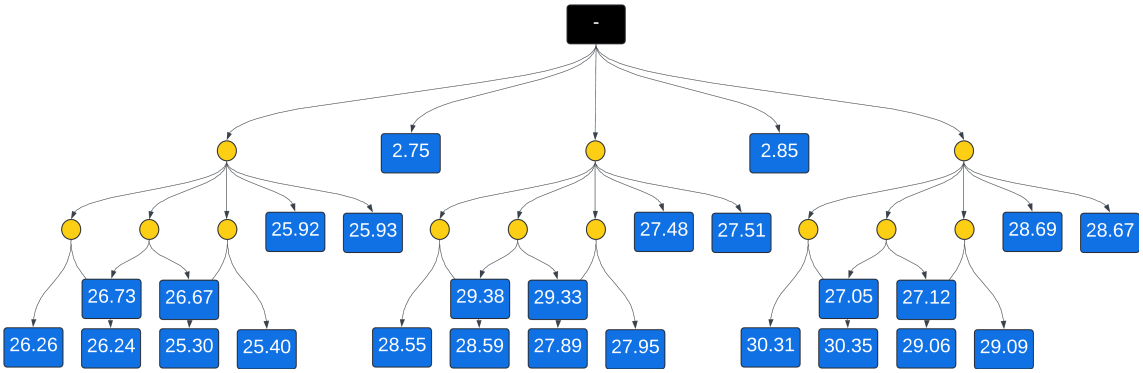


Figure 5.3: Average Levenshtein Distance to the root node (original prompt) on a per-node basis. Note: This is for the <3,3,2> Tree bank. This network only shows values for lexically perturbed nodes.

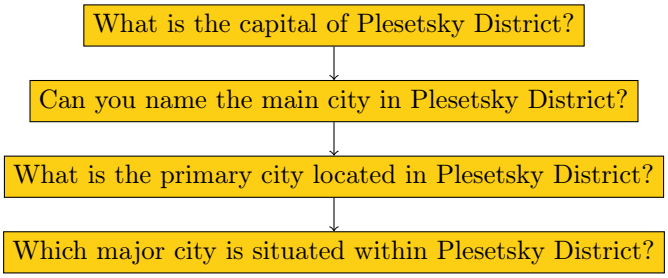


Figure 5.4: A path from the LexEval tree for SQuAD2.0 (LexEval id: 1904656)

5.1.2.1 Qualitative Analysis

Figure 5.4 shows an example of a path in a LexEval tree, this example displays the effects of repeated perturbations of the root prompt. We can witness the change in the orderings of the relationships of the question to the main entity along with a direct instruction to the LLM in the node at level 1. Fortunately, the LLM managed to answer all questions correctly with the aid of RAG (retrieved page was the Plesetsky District from Wikipedia).

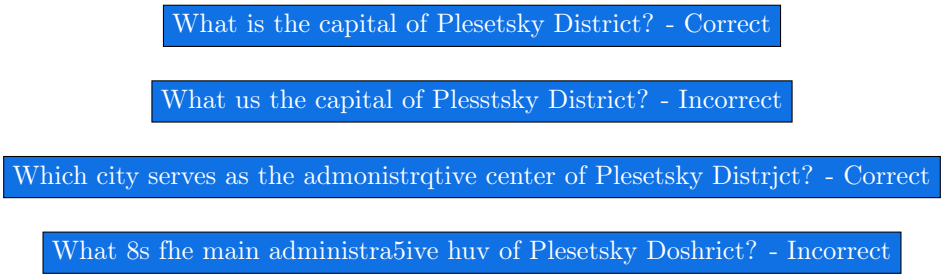


Figure 5.5: Lexically perturbed prompts from the LexEval tree for SQuAD2.0 (LexEval id: 1904656) shown in order of their existence in the tree.

Figure 5.5 highlights the important of well-formed entities since there have been numerous times when malformed entities have caused a failure in the RAG pipeline. This tells us that the retrievers are fairly sensitive to search terms and can be percieved as a major point of failure in the pipeline.

5.1.3 Cost Effectiveness

As demonstrated in Table 5.1, the average cost of creating the LexEval trees increases with the addition of more nodes. This is directly attributable to the increased computational requirements associated with more complex tree structures. For this experiment, a *batch* size of 3 was used.

While a larger batch size could potentially enhance efficiency, we were constrained by tier limits that necessitated careful batching to avoid rate limit errors. Consequently, the total duration of the experiments posed a significant challenge during the experimentation phase, as the sequential execution of experiments was time-consuming.

	<2,1,1>	<2,2,1>	<2,2,2>	<3,1,1>	<3,2,1>	<3,3,2>
Avg. Creation Time (s)	68.9s	72.4s	75.3s	90.3s	157.5s	290.4s
Total Creation Time	3h50m	4h2m	4h11m	5h3m	8h45m	16h8m
Average Storage	224KB	234KB	263KB	403KB	764KB	1.1MB
Total Storage	44.8MB	48.6MB	52.6MB	80.6MB	152.8MB	220MB
Avg. Creation Cost ($)	0.0009	0.0012	0.0014	0.0023	0.0044	0.0050
Total Creation Cost ($)	0.1821	0.2403	0.2853	0.4614	0.8834	1.0020

Table 5.1: Key LexEval metrics for different tree configurations. The format for configuration is `<x,y,z>` where `x` is `depth`, `y` is `num_paraphrased` and `z` is `num_lexical`. Note: These experiments were run using `gpt-3.5-turbo-0125` for 200 randomly sampled rows from POPQA.

5.1.3.1 Creation

Table 5.1 indicates that the space required for more complex trees increases due to the addition of more nodes. At each node, the storage is primarily dominated by the embeddings, which include both the embeddings for the prompt and the document store for each RAG technique. However, despite this increase in complexity, the storage requirements remain relatively minimal, with the most complex tree (`<3,3,2>`) occupying only 1.1MB of space.

Additionally, the average cost of tree creation rises with the increasing number of nodes due to more API requests. This is a direct consequence of the expanded tree structure, which demands more computational resources and interactions with external APIs.

Running LexEval on the 200-row subset of the dataset provides valuable insights into the total time, space, and cost required to apply the methodology to the entire dataset. By extrapolating from the subset data, we can approximate the resources needed for the full dataset, allowing for better planning and resource allocation (shown in Table 5.2).

	<2,1,1>	<2,2,1>	<2,2,2>	<3,1,1>	<3,2,1>	<3,3,2>
Total Creation Time	272h56m	286h46m	298h16m	357h42m	623h53m	1150h18m
Total Storage (GB)	3.19	3.34	3.75	5.75	10.89	15.69
Total Creation Cost ($)	12.83	17.11	19.96	32.80	62.74	71.30

Table 5.2: Extrapolated metrics for running LexEval on the entire POPQA dataset. This would be a 1-time cost.

Overall, these findings highlight the scalability of LexEval, demonstrating that even with increased complexity, the storage requirements remain manageable. The cost implications, while increasing with complexity, provide a clear understanding of the resources needed for comprehensive evaluations. This analysis underscores the importance of balancing tree complexity with resource constraints to achieve efficient and effective evaluations across the entire dataset.

5.1.3.2 Evaluation

Table 5.3 demonstrates that increasing the complexity of the tree results in a corresponding increase in the average time required to evaluate the tree. Among the models evaluated, `gpt-3.5-turbo-0125` was the least expensive, while `gpt-4-turbo-2024-04-09` was the most costly. This discrepancy is primarily due to OpenAI's asymmetric pricing model, where input tokens are cheaper than output tokens, leading to higher costs associated with generating output (as shown by Table 2.3).

In terms of evaluation speed, `gpt-3.5-turbo-0125` emerged as the fastest model, closely followed by `gemma-2b-it`. It is important to note that models from Together AI, such as `gemma-2b-it`, can be stored locally, significantly enhancing inference speed through local computation. Conversely, the slowest model was `Mistral-7B-Instruct-v0.2`, with a total evaluation time approaching 8 hours.

While there is potential for further optimisation of the evaluation framework, most improvements would necessitate either subscribing to a higher rate limit or storing models locally to reduce dependence on external API calls. Due to cost and time constraints, `gpt-4o-2024-05-13` and `gpt-4-0613` were not evaluated for the <3,3,2> tree configuration, as they were deemed too expensive and slow for the scope of this project.

Model Name	Metric	<2,2,2>	<3,3,2>
gpt-3.5-turbo-0125	Avg. Time (s)	19.07	36.04
	Total Time	1h5m	2h01m
	Avg. Cost ($)	0.0008	0.0041
	Total Cost ($)	0.1528	0.8199
gpt-4-turbo-2024-04-09	Avg. Time (s)	53.11	92.11
	Total Time	2h57m	5h07m
	Avg. Cost ($)	0.0099	0.0409
	Total Cost ($)	1.9954	8.1820
Llama-2-7b-chat	Avg. Time (s)	68.01	121.00
	Total Time	3h47m	6h44m
	Avg. Cost ($)	0.0008	0.0219
	Total Cost ($)	0.1653	4.3799
Mistral-7B-Instruct-v0.2	Avg. Time (s)	53.47	-
	Total Time	2h58m	-
	Avg. Cost ($)	0.0018	-
	Total Cost ($)	0.3505	-
gpt-4o-2024-05-13	Avg. Time (s)	61.15	-
	Total Time	3h23m	-
	Avg. Cost ($)	0.0194	-
	Total Cost ($)	3.8875	-
gemma-2b-it	Avg. Time (s)	21.98	49.01
	Total Time	1h13m	2h43m
	Avg. Cost ($)	0.0001	0.0018
	Total Cost ($)	0.0025	0.3566
gemma-7b-it	Avg. Time (s)	70.11	114.53
	Total Time	3h53m	6h22m
	Avg. Cost ($)	0.0005	0.0238
	Total Cost ($)	0.1087	4.7641

Table 5.3: Key LexEval evaluations metrics for two different tree configurations. The format for configuration is <x,y,z> where x is `depth`, y is `num_paraphrased` and z is `num_lexical`.

5.2 RAG Evaluations

This section evaluates the RAG pipeline from start to finish, focusing on both its successes and limitations. We discuss the performance of various retrievers within the RAG pipeline, highlighting their strengths and weaknesses. This analysis will provide insights into which retrievers are most effective for different types of queries and scenarios.

Additionally, we will examine the role of generators in the pipeline, assessing their impact on the quality and accuracy of the final answers. The effectiveness of generators is crucial, as they are responsible for synthesising and summarising information to produce coherent and relevant

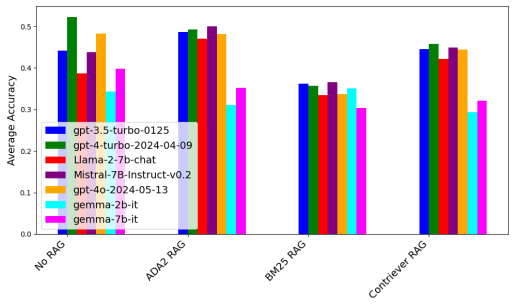
responses. By evaluating the generators' performance, we can better understand their contribution to the overall success of the RAG pipeline and identify areas for further improvement.

5.2.1 RAG Retriever Evaluation

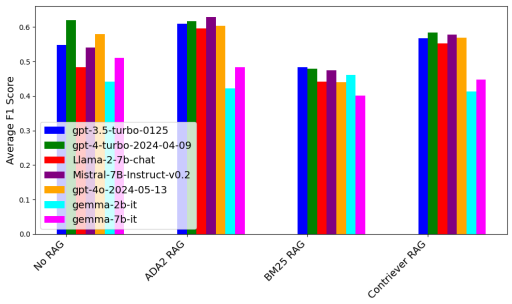
Figure 5.6 illustrates the average accuracy and F1 scores for 7 LLMs grouped by their respective retrievers. The results indicate that using no RAG yields decent but inconsistent outcomes. When employing BM25 RAG, the results are consistently lower compared to any other retriever type.

Contriever's performance is comparable to that of ADA2, albeit slightly lower. This disparity can be attributed to the differences in their embedding spaces. ADA2 benefits from a larger embedding space due to the higher dimensionality of its vectors ($n = 1536$). This greater dimensionality allows for more precise cosine similarity scores, which, in turn, facilitates more accurate retrievals. In contrast, Contriever operates with a smaller vector space ($n = 768$), resulting in somewhat less accurate retrievals.

Despite this, Contriever still outperforms BM25 because BM25 is a term-based retriever. Term-based retrievers, like BM25, struggle with variations in word order or syntax, making them highly susceptible to failures when perturbations occur. In essence, the robustness of ADA2's larger vector space provides a significant advantage in maintaining accuracy and consistency in retrieval tasks, outperforming both Contriever and BM25.

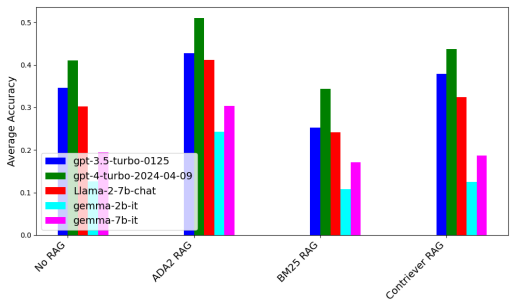


(a) Average accuracy for 7 LLMs across retriever types.

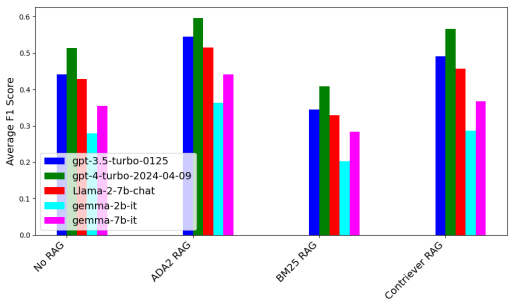


(b) Average F1-scores for 7 LLMs across retriever types.

Figure 5.6: Average accuracy and F1-scores for the LexEval POPQA `<2,2,2>` Tree Bank classified by retriever types.



(a) Average accuracy for 7 LLMs across retriever types.



(b) Average F1-scores for 7 LLMs across retriever types.

Figure 5.7: Average accuracy and F1-scores for the LexEval POPQA `<3,3,2>` Tree Bank classified by retriever types. Note: `gpt-4o-2024-05-13` and `Mistral-7B-Instruct-v0.2` have been omitted due to their high cost.

When examining the results for `<3,3,2>` (Figure 5.7a), the results become more inconsistent without RAG. `gpt-4-turbo-2024-04-09` exhibits outstanding performance with ADA2 RAG, whereas the other retriever types follow similar trends observed with the metrics for the smaller tree.

In evaluating the performance characteristics of various models, it is essential that a good model demonstrates high accuracy across all POPQA popularity ranges. Our analysis (shown in Figure 5.8) revealed that the ADA2 RAG model consistently exhibits higher baseline accuracy compared to other retriever methods across all popularity ranges. This suggests that ADA2 RAG maintains a robust performance irrespective of the popularity of the data.

Furthermore, our observations indicate that the Contriever RAG method displays a higher rate of improvement, as shown by its slightly steeper accuracy curve. This steeper gradient implies that Contriever RAG is capable of better adapting and improving as the data's popularity increases. Despite this, ADA2 RAG remains the superior choice due to its consistently higher baseline performance.

In contrast, the BM25 RAG model, while demonstrating a similar gradient to ADA2 RAG, generally achieves lower accuracy. This indicates that while BM25 RAG can improve in a manner comparable to ADA2 RAG, it does not reach the same level of performance across the popularity spectrum.

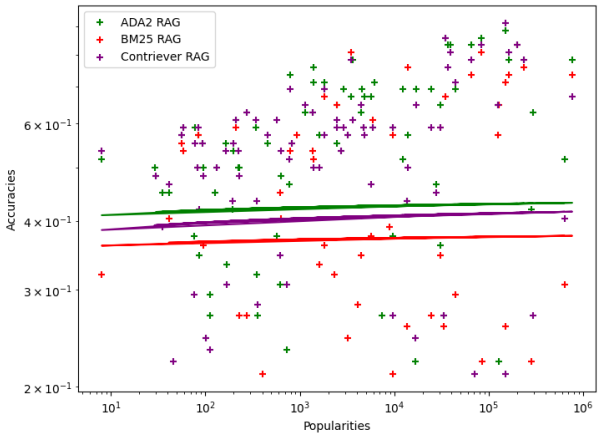


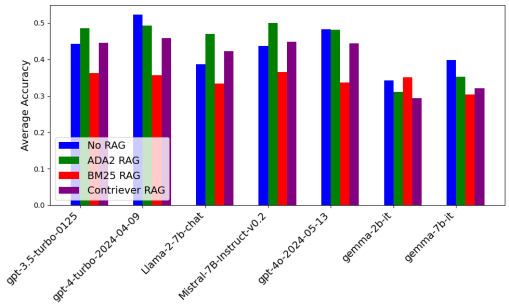
Figure 5.8: Average accuracy of nodes across levels in the POPQA LexEval Tree bank.

5.2.2 RAG Generator Evaluation

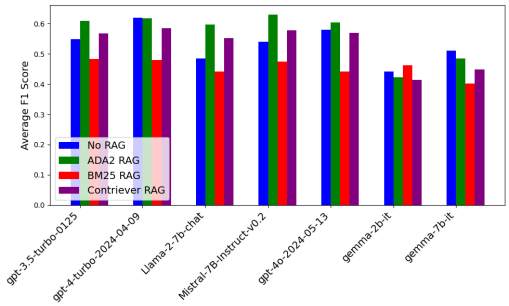
From Figure 5.10, we observe the average performance of various LLMs. For the simpler `<2,2,2>` tree configuration, `gpt-4-turbo-2024-04-09` demonstrates the highest performance without RAG. However, when the tree complexity increases to `<3,3,2>` (shown in Figure 5.10), the performance without RAG significantly dips, whereas the performance with ADA2 RAG improves substantially. This indicates that `gpt-4-turbo-2024-04-09` is more prone to errors with long-tail entities when questions are perturbed, but RAG aids in retrieving the correct information, allowing GPT-4 Turbo to effectively summarise and select the relevant information from the RAG context.

Additionally, the `gemma` models show significantly lower performance compared to other models. This may be attributed to the exclusion of long-tail entities in their pre-training data. Even with RAG, Gemma models struggled to retrieve the correct information from the given contexts.

Furthermore, among the 7B parameter models, `Mistral-7B-Instruct-v0.2` exhibits the best performance without RAG. When combined with ADA2 RAG, it achieves one of the highest accuracies. This is promising because `Mistral-7B-Instruct-v0.2`, being a smaller model, can be run locally and more quickly. When paired with ADA2 RAG, its performance rivals that of much larger models like `gpt-3.5-turbo-0125`. This demonstrates the potential of smaller, locally executable models to perform efficiently when supplemented with robust retrieval mechanisms like ADA2 RAG.

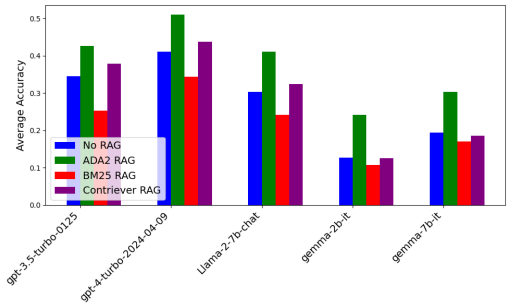


(a) Average accuracy for retriever types across 7 LLMs.

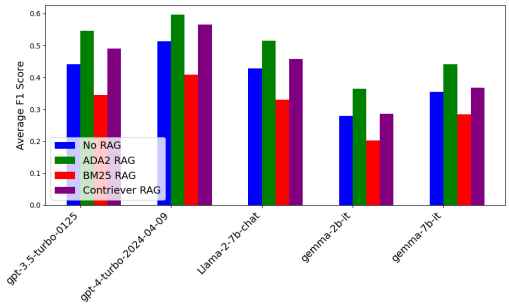


(b) Average F1-scores for retriever types across 7 LLMs.

Figure 5.9: Average accuracy and F1-scores for the LexEval POPQA <2,2,2> Tree Bank classified by LLMs.



(a) Average accuracy for retriever types across 7 LLMs.



(b) Average F1-scores for retriever types across 7 LLMs.

Figure 5.10: Average accuracy and F1-scores for the LexEval POPQA <3,3,2> Tree Bank classified by LLMs. Note: `gpt-4o-2024-05-13` and `Mistral-7B-Instruct-v0.2` have been omitted due to their high cost.

5.2.3 Discussion on the quality of responses

LexEval also provides metrics assessing the quality of model-generated responses. BLEU scores, traditionally used in machine translation, are applied here to evaluate the precision of generated answers compared to reference answers. This metric is particularly useful when the user specifies a specific output format, such as JSON, YAML, or simply the answer.

Figures 5.11 and 5.12 display BLEU and ROUGE metrics for two tree banks. Notably, the maximum ROUGE scores for the simpler trees are slightly higher, indicating that the precision of responses tends to degrade in deeper trees. Similarly, the BLEU scores for deeper trees show a higher frequency of near-zero scores compared to shallower trees. This trend underscores the challenges models face in maintaining precision as the complexity of the task increases.

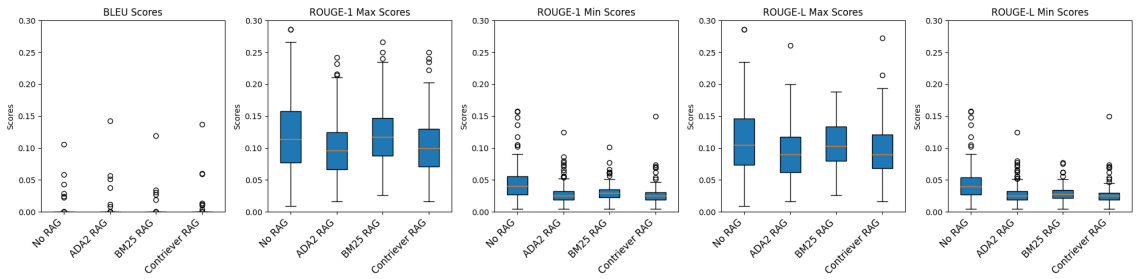


Figure 5.11: Box plots for BLEU, ROUGE-1 and ROUGE-L Scores for the <2,2,2> trees for `gpt-3.5-turbo-0125`.

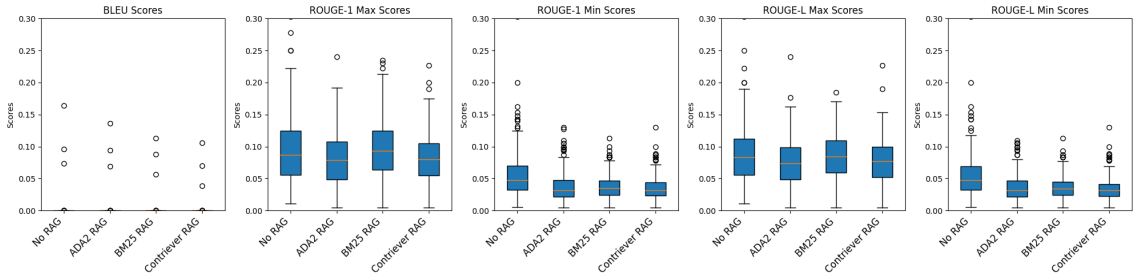


Figure 5.12: Box plots for BLEU, ROUGE-1 and ROUGE-L Scores for the <3,3,2> trees for gpt-3.5-turbo-0125.

5.2.4 Weakest Link in RAG Pipeline

Table 5.4 outlines the points of failure within the RAG pipeline. Note: NER values remain constant as gpt-3.5-turbo-0125 was used for NER across the board. The data reveals a significant number of API failures within the pipeline; however, these are beyond the scope of this evaluation. Implementing a more reliable API could potentially mitigate these failures.

The most substantial failures are attributed to the retrievers, with BM25 performing the worst and ADA2 the best. This confirms previous observations regarding the relative effectiveness of different retrievers. Among the generators, Mistral 7B stands out as the most reliable for shallower trees, demonstrating the fewest failures.

When comparing performance across tree complexities, generators show a slight decline in effectiveness for deeper trees. This trend highlights the increased difficulty in maintaining high performance levels as the complexity of the tree structure grows. Overall, the analysis underscores the importance of selecting optimal components within the RAG pipeline to minimise failures and enhance performance.

Tree config	Model	Cause of Failure (out of 200)					
		NER	API	Retriever			Generator
				BM25	ADA2	Contriever	
<2,2,2>	gpt-3.5-turbo-0125	9	29	78	44	57	19
	gpt-4-turbo	9	32	83	47	52	11
	Mistral-7B-Instruct	9	27	67	39	56	6
	gemma-2b-it	9	29	60	53	53	27
<3,3,2>	gpt-3.5-turbo-0125	9	30	83	50	61	24
	gpt-4-turbo	9	35	79	45	66	18
	gemma-2b-it	9	31	75	41	60	21

Table 5.4: The causes of failures at each stage of the RAG pipeline. Each cell shows the number of times the RAG pipeline failed at that particular stage. Note: Mistral-7B-Instruct has been omitted from <3,3,2> due to cost and time constraints.

5.2.4.1 Example from <3,3,2>

Note: This example is taken from the LexEval Tree id: 402939 from the POPQA subset.

Root node prompt:
Who was the composer of Chances Are?
Leaf node prompt:
Who is tbe creator of Chahcex Are?

The example above illustrates that lexical perturbation of the prompt at the leaf node resulted in a malformed entity. Consequently, the LLM NER failed to recognise the malformed entity, causing a breakdown in the NER pipeline. This failure prevented the pipeline from identifying the closest match in the document store, resulting in no answer to the original prompt.

Chapter 6

Conclusions and Future Work

In this project, we have developed a scalable evaluation framework for LLMs. Our framework addresses variations in human prompt construction and accounts for potential misspellings or typos. Initially, we aimed to create this framework for the SQuAD dataset (4.1) but soon realised that this approach was insufficient, prompting a shift to the POPQA dataset (4.2).

The choice to focus on long-tail entities in POPQA was strategic, allowing us to effectively test the parametric knowledge of LLMs. This dataset's diverse and less common entities provided a robust challenge for evaluating the models' retrieval and generation capabilities. Additionally, integrating RAG with LexEval offered a valuable avenue for exploration and research. This combination allowed us to better evaluate the models' performance by leveraging external knowledge sources and understanding how effectively the models could incorporate retrieved information into their responses.

Our framework not only tested various LLMs but also assessed different retriever techniques, providing comprehensive insights into the efficacy of each method (4.2.3). By comparing the performance of different retrievers, we identified strengths and weaknesses, which informed our understanding of the best practices for optimising LLM performance in diverse scenarios. This multi-faceted evaluation approach ensures that our framework is robust, versatile, and applicable to a wide range of LLMs and retrieval techniques, making it an excellent starting point for future research in the field of LLM evaluations.

After extensive investigations, we determined that the most common failures in the RAG pipeline were attributable to the retriever. The selection of retrievers is crucial for the success of long-tail entity question answering (we recommend ADA2 based on experiments in Section 5.2.1. Additionally, our qualitative experiments revealed that lexically perturbing the entities being searched for significantly reduces performance. Among models evaluated without RAG, `gpt-4-turbo-2024-04-09` demonstrated the best performance. For an optimal balance between cost and accuracy, `Mistral-7B-Instruct-v0.2` combined with ADA2 RAG provided adequate performance while being considerably more cost-effective. These findings underscore the importance of careful retriever selection and highlight viable model options for various performance and budgetary requirements.

6.0.1 Future Work

There are several avenues for future work stemming from this project. First, the perturbation strategy used during tree creation could be refined to enhance its precision. This could involve more targeted perturbations, such as selectively altering the relationship or entity, or implementing multi-step paraphrasing techniques. Additionally, exploring different models for tree creation would be valuable, though time and cost constraints limited this during our project. Abstractive summarisation techniques could also be employed to improve prompt perturbation at each level, resulting in more diverse and challenging prompts.

Certain implementation details could be optimised to reduce tree creation and evaluation time, such as developing an extendable version of the tree that can be scaled down from larger configu-

rations. This would allow for more flexible and efficient evaluations.

In terms of evaluation, incorporating a more fine-grained evaluator could provide deeper insights. Metrics such as BERTScore and windowed highest similarity scores could offer a more nuanced understanding of model performance. Additionally, enhancing the evaluation function to account for these detailed metrics would improve the robustness and accuracy of the assessment.

Moreover, a laboratory with higher tier limits and greater resources—both computational and budgetary—could significantly extend the current TreeBank. This would enable the evaluation of the entire dataset with varying tree complexities. An interesting extension of this work would involve testing the framework's robustness across different datasets, providing a broader validation of its effectiveness and applicability.

Overall, these enhancements and expansions would not only refine the current framework but also extend its utility and applicability, making it a more comprehensive tool for evaluating LLMs.

Bibliography

[1] Pranav Rajpurkar, Robin Jia, and Percy Liang. "Know What You Don't Know: Unanswerable Questions for SQuAD". In: arXiv:1806.03822 (June 2018). arXiv:1806.03822 [cs]. DOI: 10.48550/arXiv.1806.03822. URL: http://arxiv.org/abs/1806.03822.

[2] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". en. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0.

[3] Hao Li et al. "Visualizing the Loss Landscape of Neural Nets". In: arXiv:1712.09913 (Nov. 2018). arXiv:1712.09913 [cs, stat]. DOI: 10.48550/arXiv.1712.09913. URL: http://arxiv.org/abs/1712.09913.

[4] Vinay Williams et al. "Development of PPTNet a Neural Network for the Rapid Prototyping of Pulsed Plasma Thrusters". In: Sept. 2019.

[5] URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[6] Tomas Mikolov et al. "Recurrent neural network based language model". In: vol. 2. Jan. 2010, pp. 1045–1048.

[7] URL: https://matplotlib.org/stable/plot_types/index.html.

[8] Rahuljha. *LSTM Gradients*. en. June 2020. URL: https://towardsdatascience.com/lstm-gradients-b3996e6a0296.

[9] Ashish Vaswani et al. "Attention Is All You Need". In: arXiv:1706.03762 (Aug. 2023). arXiv:1706.03762 [cs]. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762.

[10] URL: https://www.d2l.ai/chapter_attention-mechanisms-and-transformers/transformer.html.

[11] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: arXiv:1810.04805 (May 2019). arXiv:1810.04805 [cs]. DOI: 10.48550/arXiv.1810.04805. URL: http://arxiv.org/abs/1810.04805.

[12] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: arXiv:1910.01108 (Feb. 2020). arXiv:1910.01108 [cs]. DOI: 10.48550/arXiv.1910.01108. URL: http://arxiv.org/abs/1910.01108.

[13] URL: https://huggingface.co/blog/bert-101.

[14] URL: https://huggingface.co/blog/bert-101.

[15] June 2021. URL: https://huggingface.co/datasets/squad.

[16] Alec Radford et al. "Improving Language Understanding by Generative Pre-Training". In: (2018). URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[17] Priya Shree. *The Journey of Open AI GPT models*. en. Nov. 2020. URL: https://medium.com/walmartglobaltech/the-journey-of-open-ai-gpt-models-32d95b7b7fb2.

[18] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[19] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: arXiv:2005.14165 (July 2020). arXiv:2005.14165 [cs]. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165.

[20] OpenAI et al. "GPT-4 Technical Report". In: arXiv:2303.08774 (Dec. 2023). arXiv:2303.08774 [cs]. DOI: 10.48550/arXiv.2303.08774. URL: http://arxiv.org/abs/2303.08774.

[21] Albert Q. Jiang et al. "Mistral 7B". In: arXiv:2310.06825 (Oct. 2023). arXiv:2310.06825 [cs]. DOI: 10.48550/arXiv.2310.06825. URL: http://arxiv.org/abs/2310.06825.

[22] URL: https://ai.meta.com/blog/meta-llama-3/.

[23] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. en. Dec. 2023. URL: https://arxiv.org/abs/2312.11805v3.

[24] URL: https://openai.com/index/hello-gpt-4o/.

[25] Zhuosheng Zhang et al. "Automatic Chain of Thought Prompting in Large Language Models". In: arXiv:2210.03493 (Oct. 2022). arXiv:2210.03493 [cs]. DOI: 10.48550/arXiv.2210.03493. URL: http://arxiv.org/abs/2210.03493.

[26] Shunyu Yao et al. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models". In: arXiv:2305.10601 (Dec. 2023). arXiv:2305.10601 [cs]. DOI: 10.48550/arXiv.2305.10601. URL: http://arxiv.org/abs/2305.10601.

[27] Rahul Singh et al. "Model Robustness with Text Classification: Semantic-preserving adversarial attacks". In: arXiv:2008.05536 (Aug. 2020). arXiv:2008.05536 [cs]. DOI: 10.48550/arXiv.2008.05536. URL: http://arxiv.org/abs/2008.05536.

[28] John X. Morris et al. "TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP". In: arXiv:2005.05909 (Oct. 2020). arXiv:2005.05909 [cs]. DOI: 10.48550/arXiv.2005.05909. URL: http://arxiv.org/abs/2005.05909.

[29] Sebastian Gehrmann et al. "The GEM Benchmark: Natural Language Generation, its Evaluation and Metrics". In: arXiv:2102.01672 (Apr. 2021). arXiv:2102.01672 [cs]. DOI: 10.48550/arXiv.2102.01672. URL: http://arxiv.org/abs/2102.01672.

[30] Milad Moradi and Matthias Samwald. "Evaluating the Robustness of Neural Language Models to Input Perturbations". In: arXiv:2108.12237 (Aug. 2021). arXiv:2108.12237 [cs]. DOI: 10.48550/arXiv.2108.12237. URL: http://arxiv.org/abs/2108.12237.

[31] Jason Wei and Kai Zou. "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 6382–6388. DOI: 10.18653/v1/D19-1670. URL: https://aclanthology.org/D19-1670.

[32] Steffen Eger et al. "Text Processing Like Humans Do: Visually Attacking and Shielding NLP Systems". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 1634–1647. DOI: 10.18653/v1/N19-1165. URL: https://aclanthology.org/N19-1165.

[33] Meng Lu. "FastAttacker: Semantic Perturbation Functions via Three Classifications". en. In: *Journal of Information Security* 14.2 (Feb. 2023), pp. 181–194. DOI: 10.4236/jis.2023.142011. URL: https://www.scirp.org/journal/paperinformation.aspx?paperid=124640.

[34] Alex Mallen et al. "When Not to Trust Language Models: Investigating Effectiveness of Parametric and Non-Parametric Memories". In: arXiv:2212.10511 (July 2023). arXiv:2212.10511 [cs]. DOI: 10.48550/arXiv.2212.10511. URL: http://arxiv.org/abs/2212.10511.

[35] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: arXiv:2005.11401 (Apr. 2021). arXiv:2005.11401 [cs]. DOI: 10.48550/arXiv.2005.11401. URL: http://arxiv.org/abs/2005.11401.

[36] Stephen Robertson and Hugo Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond". In: *Foundations and Trends in Information Retrieval* 3 (Jan. 2009), pp. 333–389. DOI: 10.1561/1500000019.

[37] en. URL: https://spacy.io/models.

[38] en. URL: https://www.anthropic.com/news/claude-3-family.

[39] Dan Hendrycks et al. “Measuring Massive Multitask Language Understanding”. In: arXiv:2009.03300 (Jan. 2021). arXiv:2009.03300 [cs]. DOI: 10.48550/arXiv.2009.03300. URL: http://arxiv.org/abs/2009.03300.

[40] en. URL: https://paperswithcode.com/sota/multi-task-language-understanding-on-mmlu.

[41] Alex Wang et al. “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding”. In: arXiv:1804.07461 (Feb. 2019). arXiv:1804.07461 [cs]. DOI: 10.48550/arXiv.1804.07461. URL: http://arxiv.org/abs/1804.07461.

[42] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. “Neural Network Acceptability Judgments”. In: arXiv:1805.12471 (Oct. 2019). arXiv:1805.12471 [cs]. DOI: 10.48550/arXiv.1805.12471. URL: http://arxiv.org/abs/1805.12471.

[43] Richard Socher et al. “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Ed. by David Yarowsky et al. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1631–1642. URL: https://aclanthology.org/D13-1170.

[44] Adina Williams, Nikita Nangia, and Samuel R. Bowman. “A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference”. In: arXiv:1704.05426 (Feb. 2018). arXiv:1704.05426 [cs]. DOI: 10.48550/arXiv.1704.05426. URL: http://arxiv.org/abs/1704.05426.

[45] Alex Wang et al. *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*. en. May 2019. URL: https://arxiv.org/abs/1905.00537v3.

[46] Yann Dubois et al. “Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators”. In: arXiv:2404.04475 (Apr. 2024). arXiv:2404.04475 [cs, stat]. DOI: 10.48550/arXiv.2404.04475. URL: http://arxiv.org/abs/2404.04475.

[47] Xiang Li, Yunshi Lan, and Chao Yang. “TreeEval: Benchmark-Free Evaluation of Large Language Models through Tree Planning”. In: arXiv:2402.13125 (Feb. 2024). arXiv:2402.13125 [cs]. DOI: 10.48550/arXiv.2402.13125. URL: http://arxiv.org/abs/2402.13125.

[48] Vinu Sankar Sadasivan et al. “Fast Adversarial Attacks on Language Models In One GPU Minute”. In: arXiv:2402.15570 (Feb. 2024). arXiv:2402.15570 [cs]. DOI: 10.48550/arXiv.2402.15570. URL: http://arxiv.org/abs/2402.15570.

[49] Xiang Gao et al. “SPUQ: Perturbation-Based Uncertainty Quantification for Large Language Models”. In: arXiv:2403.02509 (Mar. 2024). arXiv:2403.02509 [cs]. DOI: 10.48550/arXiv.2403.02509. URL: http://arxiv.org/abs/2403.02509.

[50] URL: https://huggingface.co/docs/api-inference/en/index.

[51] URL: https://huggingface.co/datasets/rajpurkar/squad_v2/viewer.

[52] URL: https://pandas.pydata.org/docs/.

[53] URL: https://platform.openai.com/docs/guides/embeddings.

[54] Yihe Deng et al. “Rephrase and Respond: Let Large Language Models Ask Better Questions for Themselves”. In: arXiv:2311.04205 (Apr. 2024). arXiv:2311.04205 [cs]. DOI: 10.48550/arXiv.2311.04205. URL: http://arxiv.org/abs/2311.04205.

[55] URL: https://github.com/alexyorke/butter-fingers.

[56] Jared Kaplan et al. “Scaling Laws for Neural Language Models”. In: arXiv:2001.08361 (Jan. 2020). arXiv:2001.08361 [cs, stat]. DOI: 10.48550/arXiv.2001.08361. URL: http://arxiv.org/abs/2001.08361.

[57] en. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[58] Google LLC. *rouge-score: Pure python implementation of ROUGE-1.5.5*. Python. URL: https://github.com/google-research/google-research/tree/master/rouge.

[59] Jonathan Goldsmith. *wikipedia: Wikipedia API for Python*. Python. URL: https://github.com/goldsmith/Wikipedia.

[60] URL: https://wikipedia.readthedocs.io/en/latest/.

[61] Shuhe Wang et al. “GPT-NER: Named Entity Recognition via Large Language Models”. In: arXiv:2304.10428 (Oct. 2023). arXiv:2304.10428 [cs]. DOI: 10.48550/arXiv.2304.10428. URL: http://arxiv.org/abs/2304.10428.

[62] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: arXiv:2005.14165 (July 2020). arXiv:2005.14165 [cs]. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165.

[63] en. URL: https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/.

[64] URL: https://huggingface.co/transformers/v3.0.2/model_doc/auto.html.

[65] URL: https://huggingface.co/docs/transformers/en/index.

[66] URL: https://huggingface.co/facebook/contriever.

[67] URL: https://api.python.langchain.com/en/latest/retrievers/langchain_community.retrievers.bm25.BM25Retriever.html.

[68] en. URL: https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-one.

[69] en. URL: https://docs.together.ai/docs/rate-limits.

Appendix A

A.1 Retriever Implementations

A.1.1 BM25 Implementation

Creating Documents and retriever:

```
retriever = BM25Retriever.from_documents(
    [
        Document(page_content=json.dumps(data))
        for data in wiki_data
    ]
)
```

Retrieval Function:

```
def retrieve_bm25(retriever, prompt: str):
    return retriever.invoke(prompt)
```

A.1.2 Contriever Implementation

Creating Documents and retriever:
This implementation is taken from the recommended implementation from HuggingFace, `padding` and `truncation` are set to `True` since the documents are not of equal size. The mean pooling is an implementation detail from HuggingFace's implementation.

```
def mean_pooling(token_embeddings, mask):
token_embeddings = token_embeddings.masked_fill(
    ~mask[..., None].bool(), 0.0
)
sentence_embeddings = (
    token_embeddings.sum(dim=1) / mask.sum(dim=1)[..., None]
)
return sentence_embeddings

inputs = tokenizer(
    [wiki_data],
    padding=True,
    truncation=True,
    return_tensors="pt",
)

embeddings = model(**inputs)
return mean_pooling(embeddings[0], inputs["attention_mask"])
```

Retrieval Function:

```
def find_closest_contriever_match(wiki_data: List[Dict], prompt: str):
if not wiki_data:
    return None
prompt_embedding = contriever_retriever(prompt)
closest_match = None
max_similarity = 0
contriever_db = create_contriever_db(wiki_data)
for i in range(len(contriever_db)):
    if contriever_db[i][1] is None:
        similarity_score = 0
    else:
        similarity_score = prompt_embedding @ contriever_db[i][1].T
    if similarity_score > max_similarity:
        max_similarity = similarity_score
        closest_match = contriever_db[i][2]
return closest_match
```

A.1.3 ADA2 Implementation

Creating Documents and retriever:

```
texts = [data["summary"] for data in wiki_data]
res = []
for text in texts:
    res.append(embedder.encode(text))
```

Retrieval Function:
The retrieval function has been omitted since it is functionally the same as the retrieval function for Contriever.