Spring 4-12-2024

Comprehensive Question and Answer Generation with LLaMA 2

Matous Hybl
matoush@southern.edu

COMPREHENSIVE QUESTION AND ANSWER GENERATION WITH LLAMA 2

by

Ac Hýbl

A THESIS

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Germán H. Alférez, Ph.D.

Collegedale, Tennessee

April, 2024

Comprehensive Question and Answer Generation with LLaMA-2

Approved by:

Germán Harvey Alférez

Professor Germán H. Alférez, Ph.D., Adviser

Roberto Ordóñez

Professor Robert Ordóñez, MSc.

Brent Hamstra

Professor Brent Hamstra, Ph.D.

Date Approved 4/12/2024

COMPREHENSIVE QUESTION AND ANSWER GENERATION WITH LLAMA 2

Ac Hýbl, M.S.

Southern Adventist University, 2024

Adviser: Germán H. Alférez, Ph.D.

Since the introduction of transformers, large language models have proven capable in many natural language processing fields. However, existing systems still face challenges in generating high-quality extractive questions. Base models and public chatbots fall short if the question source or quantity are critical. Our contribution is a question and answer generator for generating comprehensive, extractive questions and answers. This approach includes fine-tuning a LLaMA 2 base model for answer extraction (AE) and question generation (QG). We evaluate the resulting system using common automated metrics and a manual evaluation. We find that our system is comparable to the latest research and meets our objectives.

ACKNOWLEDGMENTS

I want to thank Lisa Myaing, Michael Babienco, Ted Ashton, Beth deFluiter, Sharon Crews, and Ki Song for providing the of the data for this thesis. Without your years of diligent work, whether authoring or gathering questions, this thesis would not have been possible.

Also I extend my gratitude towards those that participated in the evaluation of the question generator: Gabriella Grundy, Ryan Ramirez, Lisa Myaing, Michael Babienco, Beth deFluiter, and Ted Ashton.

Contents

Acronyms

AAQG	Answer-aware Question Generation
AE	Answer Extraction
AQG	Automatic Question Generation
CNN	Convolutional Neural Network
GAN	General Adversarial Network
GPT	Generative Pre-trained Transformer
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
LM	Language Model
LoRA	Low-rank Adaptation
LSTM	Long Short-term Memory
NE	Named Entity
NER	Named Entity Recognition
NLG	Natural Language Generation
NLP	Natural Language Processing
NLU	Natural Language Understanding
NQG	Neural Question Generation
PBE	Pathfinder Bible Experience
PEFT	Parameter-efficient Fine-tuning
POS	Part of Speech
PTM	Pre-trained Model
QA	Question Answering
QAG	Question and Answer Generation
QG	Question Generation
RNN	Recurrent Neural Network
SRL	Semantic Role Labeling
Seq2Seq	Sequence to Sequence
T2T	Text-to-Text
T5	Text-to-Text Transfer Transformer

Chapter 1

Introduction

This chapter introduces the motivation and goals for this thesis research. First, the problem statement and motivation are presented, describing the need for automatic question and answer generation for specific contexts. Next, the proposed solution is outlined, detailing plans for a system for question and answer generation. This chapter concludes with the objectives, limitations, delimitations, and organization of this project.

1.1 Problem Statement and Motivation

In recent years, large language models (LLMs) like OpenAI's ChatGPT have demonstrated unprecedented natural language generation capabilities [1]. They can mimic creativity, carry conversations, and tutor students in a very personalized manner. However, their publicly available forms still struggle with comprehensively generating quality questions for specific contexts.

Past research often focuses on generating questions to help teachers educate or test their students [2, 3, 4, 5]. Other research aims to improve machine question

answering or generation capabilities [6, 7, 8]. While some research works produce quality results, projects undertaken to help teachers in classrooms usually generate only a few representative questions from a larger context. Even projects that produce databases of questions favor asking many questions about large amounts of data over generating comprehensive questions for a single context.[1]

To better articulate the problem, consider a youth group near Southern Adventist University that memorizes books from the Bible and then answers questions about the studied text in a tournament. To date, the leaders of the program, including the author of this thesis, have written thousands of precise, extractive questions and answers to test the participants' knowledge. Given the considerable amount of time and labor involved in authoring these questions, automatic question and answer generation (QAG) has obvious benefits.

Historically, many research works have performed basic question generation (QG) by simply identifying parts of speech and sentence patterns with rule-based algorithms [9, 10, 11, 12, 13, 14, 15, 4, 5]. Later works recognized that producing somewhat creative, high-quality questions required more sophisticated approaches involving neural networks [6, 7, 3, 2, 16, 17, 8, 18, 19, 20, 21, 22, 23, 24].

In 2023, large language models (LLMs) like OpenAI's ChatGPT[2] demonstrated new natural language generation capabilities. However, their publicly available forms still struggle to generate questions for our case study. Fortunately, some of these transformer-based models can be further trained for specific tasks such as QAG.

Therefore the motivation for this thesis was to automate QAG using LLMs via fine-tuning. Our resulting models are capable of partially automating the

[1] For example, SQuAD offers only 1.16 questions per context.

[2] https://chat.openai.com/

hundreds of hours of tedious work spent writing the large quantities of questions needed to deeply examine knowledge of specific texts.

1.2 Solution

Our solution comprises a system that can be leveraged to generate a full database of questions and answers covering all sections of the source. Our approach explores fine-tuning an open-source pre-trained LLM released by Meta and Microsoft, Large Language Model Meta AI 2 (LLaMA 2)[3] for QAG.

The question generator must me comprehensive. In other words, the system must be able to produce many questions given just a few sentences of context. The model should also be able to ask questions from various perspectives. For example, if one question asks, "Who performed this action?" then the model should also pose the reverse question, "What action did this character perform?"

There are two main methods for achieving comprehensive coverage. We will attempt to use a combination of answer extraction (AE) and question generation (QG), called the "pipeline" method [22]. This methods produces a robust set of questions and answers that humans can review and use for practice or examination.

1.3 Objectives

This project's objectives are to:

- Fine-tune LLaMA 2 on comprehensive, extractive QAG data.
- Generate two or more questions for each input (an input may range from a sentence to a short paragraph).

[3] https://about.fb.com/news/2023/07/llama-2/

- Achieve an average question acceptability score of $\frac{4}{5}$ or more according to human judges involved in the Bible memorization tournament. In this thesis, acceptability measures how useful model output is for the Bible memorization tournament.
- Reduce the time necessary for question authoring. This will be calculated mathematically using the original time taken to author questions and average acceptability scores.
- Make the model publicly available.

1.4 Limitations

Due to time and monetary constraints, this project has several limitations:

- The quality of the generated questions is constrained by the training data.
- Because the best GPUs available for this project are limited by 16GB of RAM, this project does not experiment with larger LLaMA 2 models such as the 34B and 70B parameter models.
- Biases and limitations inherent in the model architecture and training data may lead to problematic or unfair outputs.
- The public question generator may be limited by computing resources and cost.

1.5 Delimitations

This project is scoped and delimited in the following ways:

- The QAG system has been trained to generate extractive questions only. Deductive or abstract questions will not be generated.
- The QG models's output includes individual questions and answers only so that the model does not have to learn to map one context to many questions. This may make the overall generation somewhat slower.
- Manual evaluation was performed by a small set of human judges.
- Ethical implications of automating question generation were not examined.

1.6 Organization

This document is organized as follows:

- Chapter 2 presents a theoretical framework of the concepts for this thesis.
- Chapter 3 outlines our methodology for fine-tuning base models and creating a full QAG system.
- Chapter 4 details the plans for project evaluation.
- Chapter 5 reports the results of the evaluation.
- Chapter 6, the conclusion, summarizes project objectives, the proposed solution, and future work.

Chapter 2

Background

This chapter presents the theoretical framework and background research utilized in this thesis. First, the theoretical framework explains the particular concepts used and how they are connected. The background research section examines the progression of QG methodologies in past literature. It also explores the QAG capabilities of popular LLM chatbots and other tools.

2.1 Theoretical Framework

This section provides a brief explanation of the key concepts used in this thesis. Figure 2.1 summarizes the various concepts and their relationships.

2.1.1 Natural Language Processing

QAG belongs to a broad field of problems involving machines and human language. These problems can be divided into several categories.

K. R. Chowdhary defines one category, natural language processing (NLP), as "a collection of computational techniques for automatic analysis and representa-

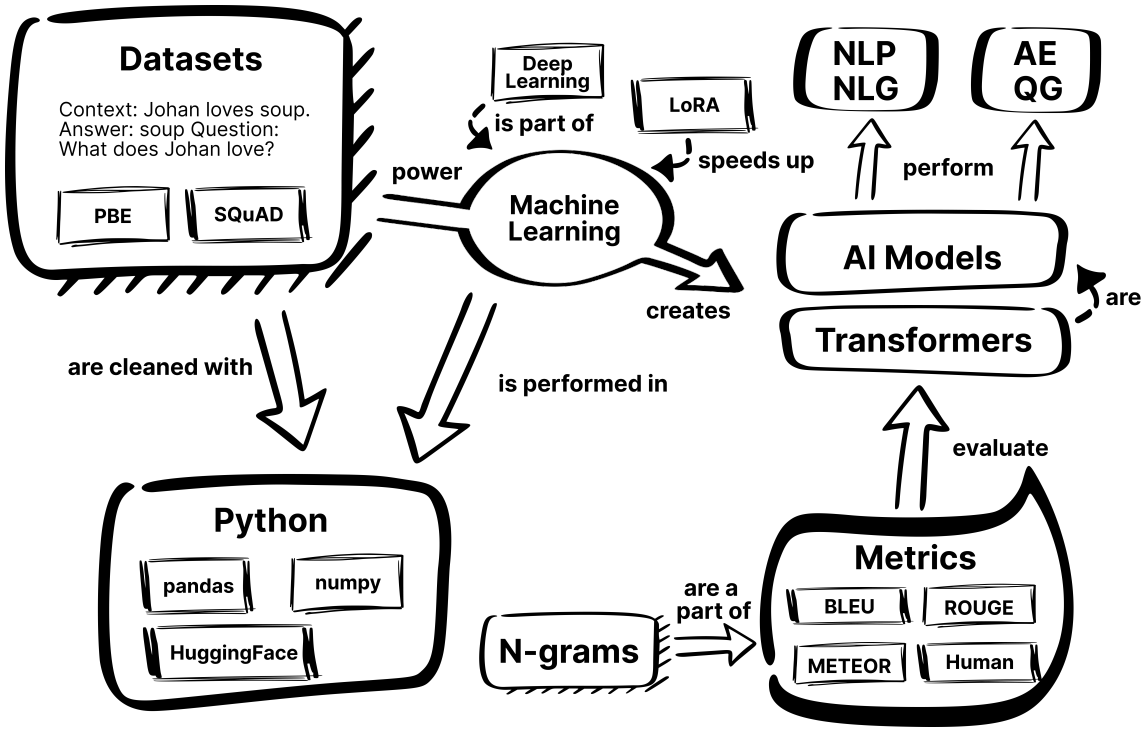

Figure 2.1: Concept map

tion of human languages, motivated by theory" [25]. NLP ranges from simpler functionality such as spell checking to more challenging problems like the retrieval of important information. Some NLP problems can be further categorized under natural language understanding (NLU). NLU is a more advanced form of NLP in which machines understand language using background information much like humans do. NLU requires semantic information, abstract concepts, and various modules.

Chowdhary separates natural language generation (NLG) from NLP, categorizing both as branches of computational linguistics. While NLP is focused on language analysis, NLG involves machines writing human language rather than just reading and understanding it.

While these categories have different goals, NLG is interconnected with NLP. To generate text, a machine must first achieve a level of NLU. Because QAG

requires a machine to generate natural language to achieve the desired output, it is considered part of NLG.

2.1.2 AI Models

Stanford University's Human-Centered Artificial Intelligence (HAI) institute cites the original definition of artificial intelligence (AI), "the science and engineering of making intelligent machines," coined by John McCarthy in 1955.[1] This definition relies on the definition of intelligence, which the HAI defines as the ability to learn. Hence, it follows that *artificial* intelligence is identified by a machine's ability to learn.

In *Machine Learning: An Artificial Intelligence Approach* [26], Michalski et al. introduce the necessity of this type of learning by explaining that some tasks are simply too difficult to "laboriously program" into a computer. In other words, AI allows computers to perform tasks that humans perform but cannot fully articulate algorithmically. Rather than explicitly programming billions of decisions with conditional statements (*if* this *then* do that), machine learning allows a computer to "learn by example" and program these "decisions" automatically.

2.1.3 Machine Learning

Machine learning (ML) is a broad term for the process by which a computer constructs an AI model to perform a task. Arthur Samuel, who coined the term [27], defined "machine learning" as the "field of study that gives computers the ability to learn without being explicitly programmed." Machine learning can take

[1] hai.stanford.edu

many different forms such as logistic regression, Naive Bayes, or clustering. In this thesis, we focus on deep learning.

2.1.4 Deep Learning

Michael Nielsen provides an excellent introduction to deep learning and neural networks in *Neural Networks and Deep Learning* [28]. Deep learning is a subset of machine learning used to train neural networks. The fundamental unit of such a network is often called a node, neuron, or perceptron (see Figure 2.2).

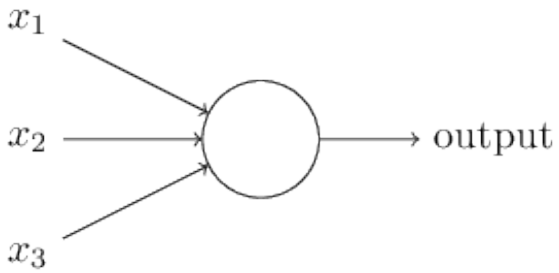

Figure 2.2: A perceptron with three inputs x_1, x_2, and x_3 [28]

An artificial neuron receives input, performs a mathematical (usually linear) transformation, and then returns some output, usually 0, 1, or some rational number in between. By combining many of these neurons in sequence, output to input, and in parallel, computer scientists construct neural networks (see Figure 2.3).

The importance, or weight, of the connection between two perceptrons is the "trainable" part of the network. Deep learning uses backpropagation to adjust these connections by constantly taking partial derivatives between a provided example and the current state of each neuron. The neuron weights must be repeatedly tuned until the model can perform well with problems it has never seen before. Thus, to

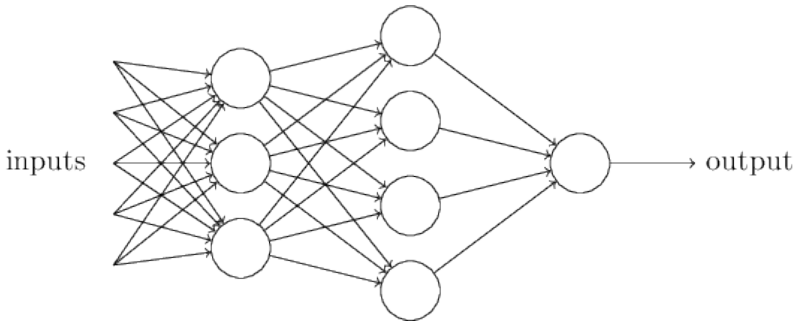

Figure 2.3: Connecting many perceptrons forms an artificial neural network [28]

train a neural network using deep learning, we must possess many examples to present to the learning model. These examples are referred to as data.

2.1.5 Data

For a model to learn based on empirical evidence, it must be shown many examples representative of the target task. The success of this learning process relies on the format and quality of these examples. For this thesis, we plan to utilize two separate but similar datasets.

SQuAD

Many published QG projects based in machine and deep learning [15, 3, 2, 16, 17, 18, 19, 20, 21, 29, 23] have used the Stanford Question Answering Dataset (SQuAD) since its publication in 2016 [30]. Due to its size and quality, we initially used SQuAD to train our QAG model. As the name indicates, SQuAD was originally developed for question answering (QA). Thus, in this thesis, we used an inverted version of the dataset on HuggingFace[2] that is formatted for QAG.

[2] https://huggingface.co/datasets/lmqg/qg_squad

PBE

A case study for this thesis is a memorization-based youth program involving an extractive question tournament. This program is part of the Seventh-day Adventist church's youth group called "Pathfinders". Because the tournament is focused on Bible memorization, the program is called Pathfinder Bible Experience (PBE). Each year, the international PBE administrators designate one or more books of the Bible for study. In preparation for each tournament level, participants train themselves using thousands of human-written questions that cover virtually all details of the material studied.

To fine-tune a model for this comprehensive, extractive domain, the model must be trained on high-quality, domain-specific examples. Thus, after training on SQuAD, we plan to further fine-tune the model on past questions obtained from this bible memorization tournament.

These questions have been collected from current or past leaders in PBE. These individuals comprise Lisa Myaing, Michael Babienco, Beth deFluiter, Sharon Crews, and Ki Song. Sharon Crews's data is primarily provided by Ted Ashton. After cleaning and deduplication, the entire PBE dataset contained 60,609 examples, where each example contains a reference, question, answer, answer point value, source, and question category flags. Figure 2.4 shows a cleaned example (hidden columns are not used in this thesis).

2.1.6 Transformers

Most QG systems before the 2010s were rule-based, systematically transforming a context from a declarative sentence to a question. The introduction of transformers revolutionized much NLG research including the field of question generation [31].

book	chapter	verse	endVerse	question
1 Peter	1	4	4	What kind of inheritance has God our Father given us?
...	...	...	...	...

answer	points	source	bigPoints	...
(1) incorruptible (2) undefiled (3) does not fade away (4) reserved...	4	Beth deFlutier	TRUE	...
...	...	...	...	...

Figure 2.4: PBE dataset sample

Because many more thorough explanations can be found online and in published research, we will only briefly explain transformers at a high level here. As the title of Vaswani et al.'s "Attention is All You Need" paper suggests, rather than using long short-term memory (LSTM), transformers leverage attention mechanisms to allow a model to selectively retain pieces of important information from past input and output sequences. This provides the model with a selective but theoretically infinite memory. Self-attention allows words near each other to adjust each other's meanings. Moreover, unlike Recurrent Neural Networks (RNNs) which generate words in sequence by feeding output back into the model, transformers utilize positional encoding, allowing for faster inference and training. This architecture not only performs well in NLG tasks, but is also parallelizable, making transformer training on vast datasets much easier.

2.1.7 PEFT with LoRA

While transformers enable faster training on enormous datasets, base models have recently grown so large that they require expensive computing resources and weeks of training time. Generally, although fine-tuned models are versions of a base model, they require repeated distribution of the entire model. This wastes a lot of space.

Several solutions have been proposed to reduce the training costs and distribution size of LLMs. Adapters are popular because they can greatly reduce the amount of parameters that need training by introducing layers into the model that are trained instead of the rest of the model. However, the additional layers also slow down the model during inference.

Hu et al. [32] propose low-rank adaptation (LoRA), a type of parameter-efficient fine-tuning (PEFT). LoRA's implementation relies on the basic linear algebra shown in Figure 2.5. X is an input matrix of size $1 \times d$ and h is a the output of the same order. Rather than training all the weights W that are in the base model's weight matrix of order $d \times d$, LoRA trains matrices A and B of sizes $d \times r$ and $r \times d$, respectively, where r is referred to as the "rank" of the LoRA matrix. If $r \ll d$, LoRA can train much less weights than W while still producing a properly-sized output of $1 \times d$ because $M_{1\times d} \cdot M_{d\times r} \cdot M_{r\times d} = M_{1\times d}$. B's output matrix is then summed with the base model's during training to produce h.

Training layers separately and then combining them allows LoRA to store only the difference in weights and indirectly focus on the most appropriate base weights for the task domain. Using these mechanisms, LoRA can reduce the amount of parameters that need fine-tuning by a factor of $10,000$. Furthermore, LoRA does not incur a performance cost in the final model because the LoRA weights are

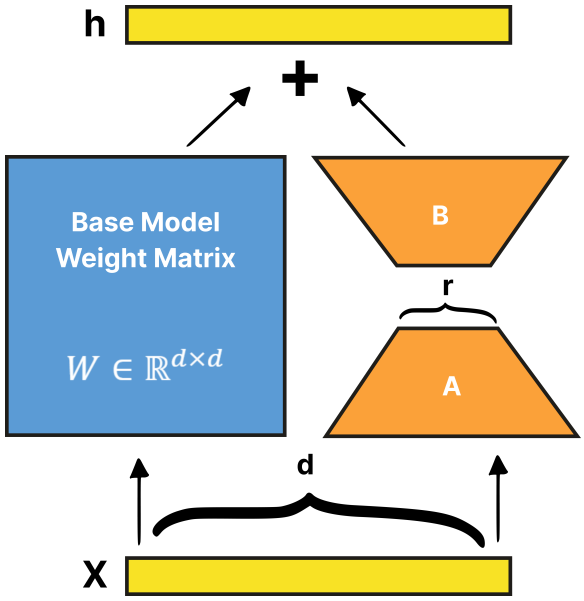

Figure 2.5: Low-Rank Adaptation

combined with the base W for inference. Due to our hardware restrictions and LoRA's success in fine-tuning, we will use LoRA during fine-tuning.

2.1.8 Python and Libraries

Due to the various state-of-the-art algorithms and resources included in LLM fine-tuning, research works commonly use Python for configuring their training experiments [6, 5, 29, 23, 22, 24]. Python simplifies the implementation of these algorithms and the access to these resources through its vast amount of useful AI and data management libraries. For AI tasks, we plan to utilize a family of libraries released by HuggingFace[3] centered around the transformers library [33]. To clean and format data we will use the Python libraries `pandas` and `numpy`.

[3] https://huggingface.co/

2.1.9 N-grams

In the field of natural language processing (NLP), many systems logically operate with n-grams rather than words. The n in n-grams simply refers to the number of consecutive grams a system pays attention to. A 1-gram is referred to as a unigram, 2-gram as a bigram, etc. A gram can be understood as a single unit of text, usually a word, excluding punctuation and whitespace.[4] For example, the sentence "I am very tired." can be split into four unigrams, ["I", "am", "very", "tired."], or into three bigrams, ["I am", "am very", "very tired"]. These units of text are especially useful when discussing evaluation methods of QG systems.

2.1.10 N-gram Precision and Recall

Precision and recall are two foundational metrics in machine translation (MT). Evaluating NLG often consists of comparing the generated text to provided references. N-gram precision is defined as the number of n-gram matches between the generated (candidate) text and the reference (target) divided by the total n-grams generated in the candidate.

$$p = \frac{n\text{–}gram\ matches}{n\text{–}grams\ in\ candidate} \tag{2.1}$$

N-gram recall differs from precision simply by dividing the number of n-grams in the reference.

$$r = \frac{n\text{–}gram\ matches}{n\text{–}grams\ in\ reference} \tag{2.2}$$

[4] analyticsvidhya.com

These two metrics can be modified and combined to produce several NLG evaluation methods.

2.1.11 Automatic MT Evaluation Metrics

While n-gram precision and recall report useful measures of how close a candidate is to a reference, it remains unclear which metric to use for ranking models. Should the metrics be combined? Is one metric more important than others? Is word comparison sufficient for capturing proximity in meaning?

To solve these conundrums, several researchers in the early 2000s proposed three increasingly sophisticated evaluation metrics. Papineni et al. [34] proposed the Bilingual Evaluation Understudy (BLEU) metric which focused on precision and comparing a candidate to several references. On the other hand, Lin's Recall-Oriented Understudy for Gisting Evaluation (ROUGE) gives much more weight to recall than precision [35]. We use a version of this metric called ROUGE-L which operates on the longest common subsequence (LCS). Finally, the Metric for Evaluation of Translation with Explicit Ordering (METEOR) claims to improve upon BLEU by combining precision and recall, further incentivizing correct word order, and considering word stems and semantic roles.

2.1.12 QAG Generation Methods

There are two primary methods to produce a comprehensive set of questions and answers for a given context [29]. The pipeline method comprises answer extraction (AE) and question generation (QG). This method of QAG relies on the model producing several potential answers which are then individually added to the context and fed back into a QG model to produce a question for each answer (see

the top of Figure 2.6). "End-to-end" QAG, on the other hand, requires the model to produce many questions and answers simultaneously as shown in the bottom of Figure 2.6.

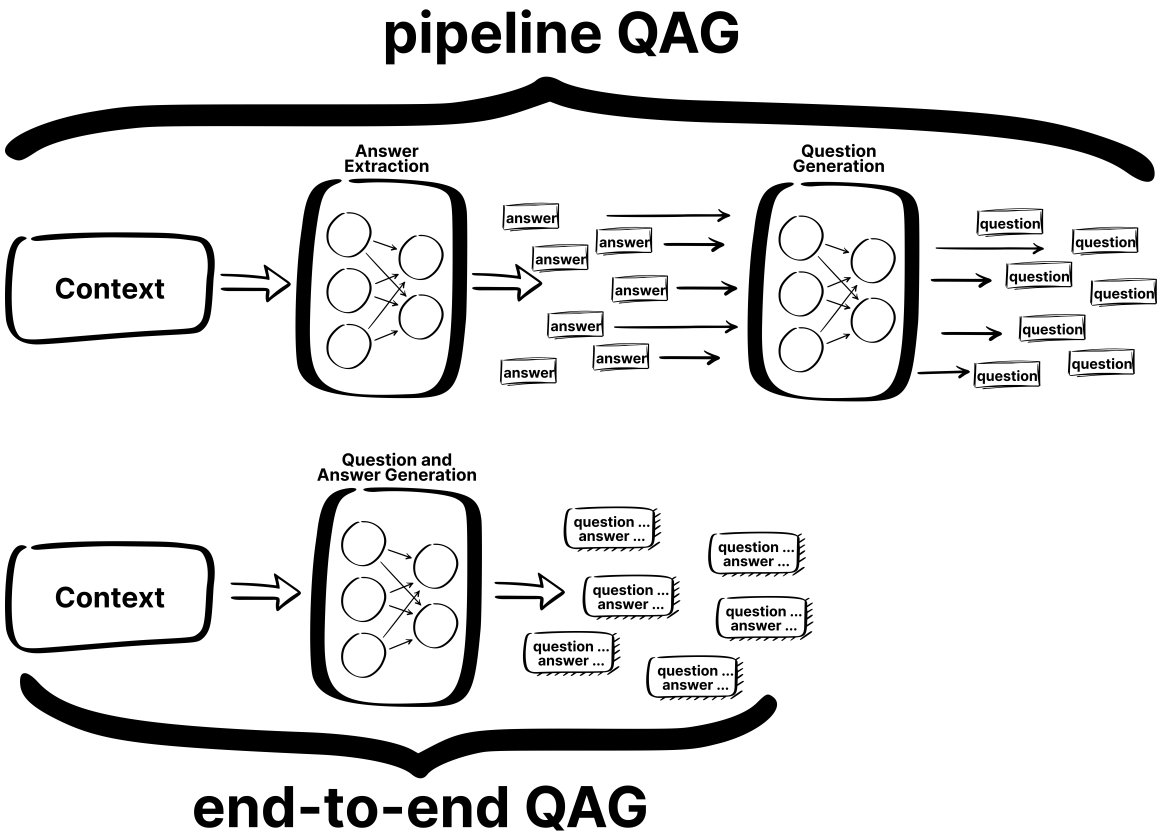

Figure 2.6: Pipeline vs. end-to-end QAG

Much past research assumes answer-aware QG [6, 7, 2, 16, 8, 18], and several complete QAG systems have been successful via AE and QG in recent research [20, 23, 22]. Lopez et al. also report that generating only one question at a time achieves slightly higher performance than generating several questions at once [18]. Thus, we chose to approach QAG with the pipeline method as well so that the model can focus on producing a single question at a time. For the rest of this thesis, unless noted otherwise, understand that when discussing our QAG system we are referring to an AE and QG system.

2.2 State of the Art

Computer scientists have been attempting to automate question-asking for half a century [36]. John H. Wolfe developed one of these early systems [9]. In 1976, Wolfe published AUTOQUEST, an Eliza-like chatbot that leverages English syntax to ask questions about paragraphs of text presented to the user. While this early implementation of QG foreshadows some of the research conducted within the past decade, modern computation and research have significantly advanced this sub-field of NLG.

Notably, with the introduction of transformers in 2017 [31], NLP systems have become more common and successful. Though much effort has been directed towards creating knowledgeable, responsive, intriguing chatbots, transformers already perform well in many NLG tasks and can be applied to the QG task [16].

In this section, we first consider similar research and then present some of the currently available tools for question generation available online. During this discussion, we point out the strengths and weaknesses of recent approaches and how our proposed project differs from others in method and goal.

2.2.1 Published Research

Question generation has attracted significant research featuring diverse goals, question types, and generation methods. Last and Danon [37] identify three primary goals for automatic question generation (AQG): automatically generated assessments, making students recall information as they learn, and generating synthetic questions for training machine learning systems. The current body of research accurately reflects this trichotomy as many papers claim educational purposes as a primary application for their work [10, 11, 2, 3, 14, 19, 4, 5, 21, 23],

while several others appeal to improving machine question asking and information gathering capabilities [6, 7, 8, 38, 24].

Which question type is generated provides another distinguishing characteristic to each research work. If two QG systems use similar methods but produce altogether different types of questions, they are not interchangeable. Although Lehnert [39] proposes dividing questions into twelve categories, Mulla and Gharpure [40] instead propose a four-way categorization including factual questions, questions covering multiple sentence, yes/no questions, and deep understanding questions. Most papers we found recognized some form of these categories, specifically distinguishing between factual questions and deep understanding questions using various names such as one-hop vs. multi-hop [41], objective vs. subjective [23], or factoid vs. high-level [37]. In this thesis, we ignore multiple-choice questions and yes or no questions, choosing instead to focus on two broad categories, namely extractive and deductive questions.

Extractive questions are factual, objective questions for which answers can be extracted directly from the source text. The deductive category represents high-level, multi-hop questions that may require the answerer to put together information from various sources or deduce the correct answer through logical reasoning. In other words, the answers to abstract questions cannot be directly found in the source text. This thesis concerns itself with producing extractive questions, not deductive questions.

The available body of QG research can be organized by the method used to generate questions. All early, but even some recent, works rely on rule-based approaches. These can be divided into syntax-based, template-based, and semantic-based approaches [37]. Since 2017, several machine learning approaches for QG have emerged and have also proven successful. Most of these novel approaches

focus on neural question generation using recurrent neural networks (RNNs) and long short-term memory (LSTM) networks or transformers[31]. We explain each method using several examples below. Note that due to many methods lacking common, or any, evaluation metrics, project evaluation is addressed in a separate subsection.

Syntax-Based Methods

Although syntax-based methods can be traced back fifty years, there is no need to cover their entire history. Michael Heilman's 2011 dissertation on factual question generation [11] provides an adequate first milestone as this QG system is widely cited and used as a benchmark for QG system performance [14, 13, 3, 2, 8]. Heilman's approach is an effective representative of syntax-based QG. His pipeline can be summarized as follows. First, the author determines that he does not need to generate questions from every sentence, just the key sentences. Thus, the system selects certain sentences based on their position, length, entities, and other features. The resulting set of sentences is processed into a simplified factual format wherein pronouns are resolved, conjunctions and modifiers are organized, and words are tagged with their appropriate part of speech. Second, these simplified facts are transformed into questions using algorithmic rules such as inverting the fact's verbs and placing question words at the start of the sentence. Finally, because questions are over-generated in the previous step, they are now ranked by a regression model and only the best questions are retained, yielding a 52% question acceptance rate.

This method of "transforming" the source sentence into a question is so typical of syntax/rule-based QG systems that it is often called a transformation system [37]. Syntax-based systems are characterized by their use of sentence parsing, part

of speech (POS) tagging, named entity recognition (NER), and word stemming to simplify the problem before turning a fact into a question.

Pabitha et al. [12] approach the problem using common syntax-based techniques such as NER, noun filtering, and stemming. However, they also enhance their QG system by leveraging Naive Bayes to first determine an answer for which a question may then be generated. Methods that perform some sort of AE have the added benefit of providing not only questions but also the correct answers to them. This is particularly important to projects that attempt to produce databases of questions and answers [6], such as ours.

Khullar et al. [14] put a different spin on the typical syntax-based method by extracting the question word (who, what, etc.) from the source text itself using relative pronouns and adverbs. For instance, in the source sentence "Johan eats soup which is dairy-free," the word "which" is identified to produce the question "Which soup does Johan eat?" This makes question-type misclassification virtually impossible. Although this strategy outperforms Heilman's in particular instances, these types of questions are severely limiting, only accounting for 20% of the sentences in Khullar et al.'s own example documents.

Syntax-based systems are not limited to older research. In 2021, Panchal et al. [4] and Kumar et al. [5] both employed syntax-based systems. Like Heilman's, the studies begin by preprocessing the source text and ranking sentences. After ranking sentences with NLTK's TextRank, they use NER to find entities such as people to replace with question words such as "who." Panchal et al. also categorize some sentences to be processed with discourse algorithms which mainly perform transformation, yielding a 49% question acceptance rate overall. Kumar et al., on the other hand, focus on producing multiple choice questions (MCQs) using POS tagging and a custom-trained named entity recognizer. Unfortunately, this

question and answer generation (QAG) system produces unnecessarily lengthy answers (according to examples) and struggles to handle sentences containing multiple verbs.

Despite their partial success, the primary shortcoming of syntax-based approaches is easy to identify. Considering the source text syntax but not its meaning forsakes the primary purpose of language. These systems cannot produce all types of meaningful questions by considering rigid sentence structures.

Template and Semantic-Based Methods

Template-based QG methods differ from syntax-based systems by explicitly outlining common sentence structures and question types that can be based on those structures. These methods also often employ semantic role labeling to include the text's general meaning in the decision-making process. Wolfe's AUTOQUEST [9] provides a good example of pattern-matching strategies for asking questions. For example, AUTOQUEST identifies sentence patterns such as "S_1 so that S_2" and then constructs a question like "Why S_1?" with the expected answer then being S_2.

Of course, modern template-based methods are much more advanced than the syntactic pattern-matching available in 1976. Like Heilman [11], Liu et al. [10], first employ several syntactical and preprocessing tools such as NER and sentence parsing with Tregex and the Stanford Parser to produce questions from a scientific article. However, this project further employs a semantic word network named SentiWordNet [42] to detect opinions based on verbs. If the syntactic and semantic information for a sentence matches predefined templates within a template database, the corresponding questions are generated by using parts of the source sentence.

Mazidi and Tarau [13] take the pattern-matching and semantic labeling method

even further. By leveraging natural language understanding (NLU), their system first seeks to understand what the source text communicates. That understanding enables the categorization of the sentence. Surprisingly, only eight main sentence types are found to account for 96% of the sentences in their test sets. The system creates sentence objects based on the information gathered so far and then matches these objects against tens of templates to determine which questions should be asked. Using Amazon Mechanical Turk[5] for human evaluation, Mazidi and Tarau achieve a 55% question acceptability rate, slightly better than Heilman's [11].

While a more recent study by Keklik et al. [15] successfully uses NER, SRL, and templates to generate questions comparable to some model-based approaches [3], labeling words with roles is insufficient. Language is meant to be understood cohesively. Rule-based methods only roughly approximate the human question-asking process. The following subsection presents methods that leverage increasingly sophisticated models of language to generate questions.

Question Generation with Models

While neural networks had been employed in question answering prior to this [43], the first example we found of neural question generation (NQG) was published in 2016. In their research, Serban et al. [6] attempt to generate a database of questions for machine learning. They use a knowledge base (KB) named Freebase to extract facts - triplets containing a subject, relationship, and the related object - which are then used for QG. These facts are embedded in a 200-dimensional space configured for learning new entities and relationships. Using the SimpleQuestions dataset, embeddings are trained and then decoded using an RNN.

Although this system is used to generate thirty million questions and answers,

[5] https://www.mturk.com/

this vast database is not used nearly as often as the handwritten Stanford Question Answering Dataset (SQuAD) [30]. Published only two months earlier, SQuAD contains 300 times less questions and answers but has quickly become a standard in the field. This contrasts the question quality between the datasets and encourages this thesis to utilize SQuAD.

Du, Shao, and Cardie [3], who put SQuAD to good use, are often cited as pioneers in NQG. Because SQuAD is a question answering (QA) dataset, Du et al. invert the dataset by locating the question context using the provided answer locations. This flexibility makes SQuAD very popular in QAG projects. Like Serban et al. [6], Du et al. use a decoder to generate questions. However, they also use an encoder to process the source text. This encoder-decoder architecture practically functions by summarizing an input sequence into a hidden representation that is then translated back into a text sequence by the decoder. Though this project also uses attention mechanisms like the ones popularized by Vaswani et al. [31], its neural networks are still based on RNNs and LSTMs. After training against SQuAD, this model understandably outperforms Heilman in a SQuAD-based evaluation metric and achieves state of the art performance according to automated metrics (discussed later).

Less than a year later, the introduction of transformers revolutionized much NLG research including the field of question generation [31]. Explained in the theoretical framework (Section 2.1) above, transformers consider both individual word meanings and sequence meanings in inference. Due to the parallel computing methods, training transformers is also much easier than comparable RNNs. Due to these advantages, most approaches for QG after this publication use transformers.

Zhou et al. [2] published "Neural question generation from text: A preliminary study" one month later. Zhou et al. use an attention-based decoder for answer-

aware QG. Answer-aware question generation describes a model that uses answers marked within their context sentence or paragraph to generate corresponding questions. Like Du et al. [3], this study inverted SQuAD. Zhou et al. marked the desired answers using the BIO scheme where "B" denotes the beginning of an answer, "I" a continuation, and "O" the end. This method was also joined with other features, such as a copy-mechanism to help the model handle rare words. The resulting model produced more relevant questions than Heilman's [11].

Kriangchaivech and Wangperawong [16] soon combined the power of transformers with old mechanisms such as POS tagging and NER to help their Bidirectional Encoder Representations from Transformers (BERT) model learn more effectively. Intriguingly, they introduced additional context to their model, providing their model with the sentences surrounding SQuAD's designated answer sentence. Though this system generated questions with a word error rate (WER) of 9.66 on average, even questions that were significantly different form the target were fluent and relevant.

The following year, Liu et al. [8] combined transformers with an approach resembling Serban et al.'s [6] fact-guided knowledge base. Their system uses an algorithm called the "information extractor" to gather answers, choose clues (context information), and determine question style (ACS). This is then fed into a fine-tuned generative pretrained transformer model (GPT-2 small) for question generation. This AE and QG system aims to convert a one-to-many problem of generating multiple questions from a single context into a one-to-one problem where each generation produces a different question based on the ACS information provided. Not only does this approach solve the common issue that models do not generate enough questions, but it also avoids the issue where automated metrics penalize model outputs for putting questions in the "wrong" order [22].

The introduction of the Text-to-Text Transfer Transformer (T5) [44] in 2020 transformed the NLG playing field again. A year later Grover et al. [19] trained T5 small on SQuAD to produce answer-agnostic questions using nothing but the model. Nguyen et al. [21] soon showed that T5 could be fine-tuned for topical QG rather than extractive or even deductive QG.

Zhang et al. [20] use similar fine-tuning methods to Nguyen et al., but they also present a more thorough QG system called Transformer with Preprocessing and Postprocessing Pipelines (TP3). During preprocessing, their system performs AE using NLP tools including SRL and POS analysis and filters out unsuitable sentences and answers. The context sentences and answers are then fed to a T5 model fine-tuned on SQuAD and similar datasets. After filtering questions in the postprocessing step, 92% of questions are accepted.

While AE using NLP tools worked in the TP3 method, Goyal et al. [23] instead uses the multi-tasking capabilities of T5 to perform AE and QG with the same fine-tuned model. Using HuggingFace's Transformers library [33], this project fine-tunes T5 by prepending contexts with "extract answers" and highlighting the sentences to produce answers from. Likewise, the model is then trained with the command "generate questions" and shown answer-aware context sentences. The model is then used to extract answers and generate relatively high quality questions.

Ushio et al. [29] contribute perhaps the best-explained example of QAG with transformers. Their research compares the performances of various BART (another pretrained model) and T5 model versions. They use the HuggingFace Transformers library [33] to fine-tune these models. First they test several ranges of hyperparameters comprising learning rate, label smoothing, and batch size. After identifying the best configuration, the models are trained and tested with

automatic metrics. T5 large performs the best with its small version close behind. Intriguingly, Ushio et al. also test training answer-agnostic models. This variation of T5 large actually performs worse than the answer-aware T5 small model, lending more credibility to answer-aware QG. Paragraph-level generation models also outperform those that only operate with a single sentence.

In a subsequent paper [22] the same authors publish a Python library aiming to simplify the unnecessary complexity of most QG pipelines. They are not the first to raise this concern [18]. Unless packaged and made publicly available, most rule-based methods possess too many moving parts - preprocessing, ranking, NER, POS tagging, parsing, SRL, template matching, postprocessing - for non-experts to use. Even many model-based methodologies are very complex [6, 8, 24]. `lmqg`, language model question generation, enables inference in one line of code, provides a framework for training custom models, tests models against combined popular metrics, and integrates with the open-source HuggingFace platform. Furthermore, to improve on a system like Goyal et al.'s [23], `lmqg` allows for the training of separate AE and QG models. All the models are available on HuggingFace and model inference can be tested at `https://autoqg.net/`. Unfortunately, because `lmqg` only supports models such as T5, we cannot use the library in this thesis to simplify our pipeline.

Evaluation Methods

Evaluating QG systems is notoriously difficult and imperfect. For example, during their evaluation, Kriangchaivech and Wangperawong [16] found that the questions that were ranked the farthest away from their target outputs were still valid, grammatical, and relevant. This is common and expected in the QG field. Because the target output can be expressed in many different ways through natural language,

it is difficult to automatically determine the model's level of success.

Due to the flexibility of QG solutions and the nature of this NLG problem, comparing systems is difficult as there are no universal metrics. Various automatic metrics are used across the literature including word error rate (WER) [16], compression/omission ratios [12], and F1 score [5]. These automated metrics for machine translation (MT) are often closely related to or based on precision and recall. Fortunately, the metrics presented in the Theoretical Framework (Section 2.1), BLEU, ROUGE-L, and METEOR, can be found for approximately half of the studies presented.

Papineni et al. [34] first proposed BLEU for quick evaluation of machine translation between human languages. Modeled after WER, BLEU is a precision-based metric that compares a model's output, the candidate translation, to one or many reference translations by matching up groups of words called n-grams. For example, unigram matching (BLEU-1) counts the number of words in the candidate that also appear in the reference and divides by the total number of words in the candidate. This allows BLEU to not only measure how close the machine's output is to the target sequence, but also allows for the output to deviate from the exact target. This is important as the model trains and incrementally achieves higher scores as well as in the case when the model produces a different but equally valid translation from the reference. - Besides testing for basic precision, BLEU limits the matches in the candidate to the word occurrences in the reference and includes a decaying exponential brevity penalty that penalizes the model for producing translations much shorter than the reference. At the unigram level however, the metric could return a perfect score for a translation that has all the right words in a nonsensical order. This is why many researchers choose to compare longer strings of words such as two (bigrams) or four. The most common metric in QG

recently has been BLEU-4 [15, 3, 27, 17, 8, 18, 29, 23].

Recall-Oriented Understudy for Gisting Evaluation (ROUGE), as its name suggests, gives much more weight to recall than precision [35]. Originally developed for automated evaluation of summaries, ROUGE sums matching n-grams, over multiple references if available, and then divides by the average reference length. In this way rather than assessing whether all of the candidate is similar to some reference, ROUGE gives a score focused on how much of the reference is contained in the candidate. In the QG field, a variant of ROUGE called ROUGE-L is used which is not based on matching a predefined n-gram length, but rather on finding the longest common subsequence (LCS) between the candidate and reference.

In 2005, Banerjee et al. [45] proposed a third automated metric, METEOR, to improve Papineni's popular BLEU. Rather than matching all n-grams at once based on the exact words, METEOR matches in several stages with varying modules such as a stemmer and word synonym module. Thus, words with the same meanings or semantic role can be matched and included in grams. Multiple matches across modules are prioritized based on a formula that incentives correct semantic order. Like ROUGE, METEOR focuses on recall when calculating the final score. METEOR also includes a chunk penalty which minimizes the number of n-grams the candidate must be split into for matching between candidate and reference.

While each new metric claims to outperform its predecessors, the researcher must choose between the metric's accuracy and the added computational complexity required to compute it. Zhang et al. [46] proposed a metric based on an encoder model called BERT, Bidirectional Encoder Representations from Transformers, for NLG tasks that require the extra complexity. BERTScore takes word meaning into account when matching a candidate to a reference by generating

context-dependent embedding representations of each gram. Then the matching is performed on the embedding space using cosine similarity.

Ji et al. [47] propose a different approach entirely. Due to the one-to-many nature of the QG problem [22], even human-written questions may not match the target reference. Rather than penalizing the mismatch like BLEU, ROUGE, and METEOR would, QAScore disregards any output references and instead focuses on the answerability of the question. QAScore is based solely on a QA model's ability to produce the proper answer given the generated question and context.

While these more complex metrics show promise, BLEU-4, ROUGE-L, and METEOR are by far the most commonly reported metrics in the QG field. Thus, Table 2.1 only reports these metrics. Note, however, that it is difficult to objectively determine which model is "best" as these metrics fundamentally depend on what references the QG system is compared *to*. The aforementioned training dataset SQuAD is also commonly utilized as a benchmark for these metrics. However, not all papers report these details.

2.2.2 Existing Tools

Although question generators have yet to be perfected in academic research, various online tools and businesses already claim to offer question generation services. Popular transformer-based chatbots are also capable of many NLP tasks [16], and must be considered as potential solutions to reach our objective:

[6]Empty cells signify that we were unable to find a particular evaluation metric for that system.

[7]If a system displays two citations, the first specifies the original publication and the second the source of the system's evaluation metrics.

[8]Input-Output Type: QG means only questions were generated (or that answers were not directly made available). AAQG stands for Answer-Aware Question Generation where answers have to be provided to the system before generating questions. QAG means both questions and answers were generated by the system. AE QG means that the system extracted answers, then generated questions, and made both available.

System			Evaluation [6]		
Method	Author[7]	I/O[8]	B4	R-L	MTR
Syntax	Heilman et al. [11, 15]	QG	11.18	30.98	15.95
	Pabitha et al. [12]	AAQG	—	—	—
	Khullar et al. [14]	QG	—	—	—
	Panchal et al. [4]	QG	—	—	—
	Kumar et al. [5]	QAG?	—	—	—
Template	Wolfe et al. [9]	QG	—	—	—
	Liu et al. [10]	QG	—	—	—
	Mazidi et al. [13]	QG	—	—	—
	Keklik et al. [15]	QG	10.61	40.38	25.01
RNN	Serban et al. [6]	AAQG	—	—	35.38
Seq2Seq	Du et al. [3]	QG	12.28	39.75	16.62
	Zhou et al. [2, 8]	AAQG	13.51	41.60	18.18
	Kim et al. [17]	QAG?	16.20	43.96	19.92
Transformer	Kriangchaivechet al. [16]	AAQG	—	—	—
	Liu et al. [8]	AAQG	22.05	53.25	25.11
	Lopez et al. [18]	QG	8.27	44.38	21.20
	Grover et al. [19]	QG	—	—	—
	Zhang et al. [20]	AE QG	—	50.99	48.98
	Nguyen et al. [21]	QG	—	—	—
	Ushio et al. [29]	AAQG	27.21	54.13	27.70
	Goyal et al. [23]	AE QG	18.87	40.64	25.24
	Hwang et al. [24]	QG	—	—	—

Table 2.1: System Category and Evaluation: Unless otherwise noted, evaluations are self-reported. Only the best system's performance from each publication is shown. The best system is defined as the proposed system with the largest number of highest automatic evaluation scores. Note that only the three common scores, BLEU-4 (B4), ROUGE-L (R-L), and METEOR (MTR), are shown here.

automatic, comprehensive, extractive QAG. Thus, the below sections summarize QAG tests performed with some popular LLM chatbots and the best tools for QG available online.

LLM Chatbots

We focus our attention on four popular free chatbots representing distinct model families and parent organizations: Bard by Google, HuggingChat by HuggingFace,

ChatGPT by OpenAI, and Claude by Anthropic. We provided a standardized prompt (see A.1 in the appendix) to each model which requested the model to generate seven extractive questions and answers based on a short context. To test the models with the material used in our proposed case study, we provided the verses from the New King James version of the Bible as contexts (shown in Text A.2). The prompt was engineered to maximize the quality of the questions generated. Each model was tested in multiple conversations.

Bard produced less than desirable outcomes (see Text 2.1). It struggled to generate relevant extractive questions based on the actual contexts provided. Trained to incorporate broader knowledge into its responses, Bard referenced other verses with answers outside the provided context. Bard also asked subjective questions without clear answers. The questions that were generated were also formatted inconsistently and surrounded by the chatbot's commentary.

Sure, here are 7 questions and answers based on the source text you gave me:
What happened in the days when the judges ruled?
A famine came to pass in the land. (verse 1)
. . .
What was the man's name?
Elimelech. (Not explicitly stated in the text, but this is the name given to the man in later verses.) . . .
Here are some additional questions that you could ask:
. . .
How did Naomi end up returning to Bethlehem?

Text 2.1: Examples of Bard's responses ordered from best to worst

HuggingChat fared slightly better than Bard. Text 2.2 shows how the LLaMA 2-based model successfully focused on the provided context and generated less boilerplate text. However, sometimes the provided answers were incorrect or too long.

ChatGPT also struggled with answer length and uniquely sometimes generated

Q: Which city did they travel to?
A: He dwelt in the country of Moab.
...
Q: According to John 1:1 who or what was in the beginning?
A: The Word

Text 2.2: Examples of HuggingChat's responses

questions that it could not find the answers to, replying instead with "Answer not found in source text." ChatGPT also struggled to author only extractive questions, favoring more abstract, subjective questions.

Claude performed the best among the public LLM chatbots. This system generated consistently formatted questions that even included the desired multi-point numbering which no other model produced. However, some of Claude's questions were still deductive or subjective rather than extractive.

Also, during our last systematized experiment, Claude stopped the test short as shown in Figure 2.7 due to its rate limit of fifty messages per three hours [48]. Most public models place limits on users to protect their systems from abuse. Unfortunately, this too makes bulk QG difficult for public chatbots.

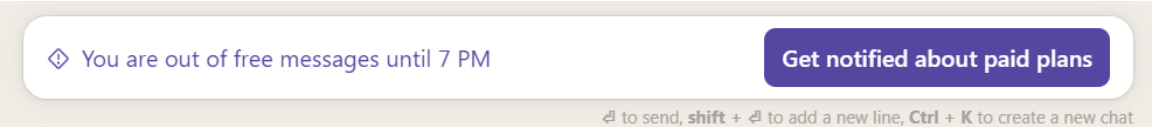

Figure 2.7: Claude's rate-limiting message

These experiments suggest that some of these chatbots' capabilities hinder their effectiveness in asking the particular types of questions needed. This is to be expected as the primary purpose of chatbots is to converse with humans and provide conversational feedback. In a way, they are a vastly more advanced version of Wolfe's conversational question generation chatbot [9]. The background knowledge and reasoning capabilities they possess are excellent tools for many

NLG tasks, including conversational abstract question generation, but they cannot be used effectively to generate a database of extractive questions.

Question Generation Websites

We also tested many online tools for question generation. While websites like Hyperwrite[9], Quizbot[10], Scalenut[11], QueTab[12], and ToolsDay[13] may adequately serve certain purposes, they do not produce the output required for this project. Hyperwrite, Quizbot, and Scalenut for example, do not generate extractive questions, but rather focus on topical or deductive questions. Other tools such as QueTab and ToolsDay generate the right types of questions but do not generate any answers for them.

On the other hand, even more relevant tools such as OpExams[14], LogicBalls[15] and QuizGecko[16] suffered from long answers. When presented with the first nine verses from the book of Joshua, the OpExams system generated several valid and relevant questions (see Text 2.3). However, most of the questions were too general, requiring longer explanations based on the text rather than short excerpts from it. QuestGen[17] generated much better extractive answers. However, this system requires a subscription to use and still gets some answers wrong.

AutoQG[18] is another promising tool created by Ushio et al. [22]. AutoQG provides several models and QAG architectures. Although the small T5 model

[9] https://www.hyperwriteai.com/aitools/question-generator
[10] https://quizbot.ai/ai-templates
[11] https://www.scalenut.com/tools/engaging-questions
[12] https://www.quetab.com/ai/question/generator
[13] https://toolsaday.com/writing/questions-generator
[14] https://opexams.com/free-questions-generator/
[15] https://logicballs.com/tools/question-and-answer-generator
[16] https://quizgecko.com/create
[17] https://dashboard.questgen.ai/
[18] https://www.autoqg.net/

What did the LORD say about the observance of the law in verse 7? The LORD commanded Joshua to observe and do according to all the law which Moses commanded him, not turning from it to the right or to the left, so that he may prosper wherever he goes.

Text 2.3: OpExams generated relevant questions with long, non-extractive answers

generates inadequate and incoherent questions, its large counterpart fared much better, suggesting that model size can be important for ensuring quality. Tests with T5 small also indicated that the AutoQG models were not as accustomed to the type of context we provided it, old English. For example, given the first verse of Joshua, T5 small misinterpreted "Judah" as the name of an individual rather than the location of an Israelite tribe (see Text 2.4). This further supports a project using custom data to train a QAG model for generating our database.

question: Who went to dwell in the country of Moab?, answer: Judah

Text 2.4: Sample AutoQG output

While existing online QG tools have varying capabilities, none fully meet the needs of this project. This supports developing a custom model trained on domain-specific data to produce high-quality, extractive questions and answers.

2.3 Discussion

Although the State of the Art (Section 2.2) includes many methods for QG, none sufficed for the goals listed in Chapter 1. For example, we desired to generate both questions and answers from study materials. Among the model-based studies we found and presented, only three [8, 20, 23] generated answers along with questions. While other studies for answer-aware QG may have advanced the state of the art by generating deeper or greater quantities of questions, there are fewer practical uses for question generators that necessitate answers to be pre-selected by hand.

Rather than creating a few representative questions to test whether a student has read their study materials or not, we aimed to be able to generate a comprehensive database of specific questions to assess source text memorization and subsequent recall. The closest project to this we found was Serban et al.'s [6] generation of a vast question and answer dataset. However, the goal of this generation was to train machines whereas our goal was for human use.

Many studies also only produced more deductive than extractive questions [10, 15, 21]. Public chatbots also struggled to generate the requested question type, and they struggled to exclusively use information in the provided source text.

Because all of the available QG systems fell short of a comprehensive, extractive question and answer generator, we continued to develop our own system.

Chapter 3

Methodology

Because no former research or current tools met our requirements, we undertook the process of fine-tuning a large language model to cater specifically to QAG in our domain.

Due to their recent success, we chose to leverage transformer-based language models to produce a system for comprehensive, extractive QAG. All the Python scripts associated with this project can be found on the qag GitHub repository. The development of this repository and the overall project unfolded through these steps:

1. Selecting a base Language Model (LM)
2. Aggregating various data sources
3. Processing data for fine-tuning
4. Programming the fine-tuning scripts
5. Experimentally choosing the ideal training configuration
6. Identifying optimal hyperparameters

In the pursuit of each of these steps, we drew insights from the examples provided by past QAG systems and other relevant research. Our approach in each step is expanded in the following sections.

3.1 Selection of a base LM

We utilized LLaMA 2 [49] as our base model due to its accessibility and capabilities. Although an increasing number of corporations have been investing in LLM research (IBM Watsonx, Microsoft Copilot, Snapchat's My AI, Poe AI), many organizations safeguard their models due to their commercial value. This leaves only a handful of options for open-source base model families. Of these, three LLM families stand out: old "GPT" models such as the GPT-NeoX [50], Google's T5 [44], and Meta's models such as LLaMA 2 [49]. According to Touvron et al., the LLaMA 2 model family outperforms other associated open-source models (such as MPT, Falcon, and Vicuna) in both pure text-completion and conversational output. Furthermore, LLaMA 2 is comparable or better than closed-source models such as GPT 3.5 and PaLM.

Although base model capability is imperative, none of this capability matters if the model's infrastructure is too complex to set up and train. Due to LLaMA 2's connection to Meta, we expected it to provide better platform integration options and documentation than most other open-source LLMs. This support is critical when fine-tuning a model for novel tasks.

LLaMA 2 provides a family of base models consisting of text-completion models at different sizes - 7, 13, 34, and 70 billion parameters - and fine-tuned chat models for each size. For simplicity, henceforth we will refer to the text-completion models as just "text" models. Due to hardware limitations, only the 7b and 13b text and

chat base models were tested on answer extraction prior to training (five or more trials). As shown in Table 3.1, there was little difference in execution time for the two sizes, especially for the faster chat models.

	7b	13b
text	3.92	4.93
chat	1.80	2.12

Table 3.1: Average base model execution times in seconds

Although larger, the 13b models did not provide improved model output. If we intended to retrain all the weights during fine-tuning as is done to produce the LLaMA-2-Chat models, then perhaps training more parameters could have improved model performance. However, because we actually aimed to train a LoRA layer that would be magnitudes smaller than the base model, we proceeded with the 7b models. If the need arose for more fine-grained tuning of more weights, the LoRA rank could be increased instead of the base model size.

Both text-completion models struggled to coherently continue the prompt as desired. For example, when presented with the prompt in Text 3.1, the 7b-text model would produce some phrases from the context, some random phrases, guesses at the verse reference, and many incoherent unicode characters. The 13b-text model performed similarly. Sometimes, rather than continuing a natural output, the 13b-text model would ignore the prior text and begin a German lesson about the pronunciation of the letter "ä". Furthermore, both text models were incapable of stopping their output naturally. The output was often truncated mid-word.

The LLaMA chat models, however, followed instructions much more accurately and naturally completed their output. Intriguingly, this positive behavior only occurred while *n*ot formatting the chat model input as in the prompt in Text 3.2,

<s> ### Here is a context verse. ### Verse: <context> ### Here are seven nouns and noun phrases in a comma-separated list that appear in this Bible verse:

Text 3.1: The most successful prompt for text-completion models. "<context> " is replaced by actual contexts at runtime.

as the meta-llama repository recommends. If the prompt was formatted in that instructional manner, the model would try to discuss the context and questions rather than continually generate many of them.

<s> [INST] <<SYS>>\n{Extract any entities, actions, key phrases, or lists that appear in the following prompt. Separate each unit by a hashtag (#). Only return words, phrases, and lists that are in the prompt provided.}\n <</SYS>>\n\n{<context>} [/INST]

Text 3.2: One of the prompts tested that followed chat-specific formatting. The system prompt belongs in the first set of curly braces and the first user input, in this case a context, in the second set.

After considering all of the above preliminary results, we chose to primarily fine-tune the 7b-chat model and to ignore the chat-specific formatting. Unless stated otherwise, henceforth any base model referred to is the LLaMA-2-7b-Chat model.

3.2 Data aggregation

Due to SQuAD's prevalence in QG [2, 3, 16, 17, 18, 19, 20, 21, 29, 23], we first attempted to use this dataset for fine-tuning our model. We used Ushio et al.'s [22] inverted SQuAD dataset.[1], whose format is shown in Figure 3.1 The eight columns include smaller and larger contexts, questions, and answers. `<hl>` tokens are used to indicate the portion of context the model should focus on.

[1] https://huggingface.co/datasets/lmqg/qg_squad

answer	paragraph_question	question	sentence
Johan the Orangutan	question: Who ate soup?, context: Researchers were...	Who ate soup?	Johan the Orangutan ate soup that day.
...	...	...	...

paragraph	sentence_answer	paragraph_answer	paragraph_sentence
Researchers were excited! Johan the...	<hl> Johan the Orangutan <hl> ate soup that day.	Researchers were excited! <hl> Johan...	Researchers were excited! <hl> Johan the...
...	...	...	...

Figure 3.1: Ushio et al.'s inverted SQuAD

Surprisingly, fine-tuning on this dataset did not improve the model's performance for AE. The most probable reason for this was that the SQuAD was not sufficient for AE specifically. A basic QAG training dataset consists of three columns: context, question, and answer. These columns are prepared for training a model for QG, but more processing is required to train for AE. To present the model with proper examples of extracting relevant phrases and words from a context, a QAG dataset must be grouped by contexts, and the answers must be aggregated into a single string. This makes possible training examples where the model is shown the context and then is trained to extract several potential answers. Unfortunately, very few of the context paragraphs in the SQuAD dataset can be effectively aggregated in this way. The resulting dataset contains one or two extracted answers per training example. For Ushio's specific dataset, there were only 1.16 answers per context. Effectively, the model trained to produce only one potential answer in its output. Because SQuAD's answers are all contained in the context, the model learned to simply repeat the context in order to minimize loss.

After encountering this obstacle, we studied Touvron et al.'s data collection process for fine-tuning the LLaMA 2 family of models for conversation [49].

Touvron et al. fine-tune their base models using a method called supervised fine-tuning (SFT). The ”supervised” in SFT simply refers to training a model on ”labeled” examples - examples where both the input and the desired model output are written by humans. Touvron et al.’s description of SFT emphasizes obtaining higher quality data rather than a greater quantity of data. For example, rather than utilizing significant amounts of third-party SFT data, Touvron et al. collected their own, high-quality training examples.

Following this guidance, we chose to discard the SQuAD dataset and instead focus on collecting high-quality data specific to our domain. Toward this end, we leveraged personal connections from past PBE tournaments to gather a hand-written question and answer PBE dataset. By sourcing data directly from tournament participants, we ensure high-quality, authentic examples for fine-tuning. We received the data in various formats. 75% of our examples were gathered from various Microsoft Excel spreadsheets and personal PBE Quizlet sets. A single semicolon-separated database dump accounted for 18.5% of the data, and about 6% was provided in comma-separated values (CSV) format. The remaining half percent came from OCR applied to images within PDFs provided by the North American Division (NAD) PBE administrators. All together, Lisa Myaing, Michael Babienco, Beth deFluiter, Sharon Crews, and Ki Song provided 60,609 unique examples for training. Although these examples came from many different sources, due to their strict adherence to PBE tournament standards, they each provide questions, answers, and references to contexts that can be used for training.

Adhering to the same format does not, however, imply that all data sources provided examples of the same quality. Some question authors might labor for weeks coming up with the best questions possible, while others may prioritize quickly writing questions over question relevance and logical sense. We even found

that some data sources were poisoned with absurd data. For example, we had to manually scour the semicolon-separated database dump to remove questions and answers such as the ones shown in Text 3.3.

```
“Question”;“Answer”
“Papa is now a...”;“Donut”
“Who am i”;“Ur mom lol”
“Chimmy-chungas”;“Yeeeeee”
```

Text 3.3: A few of the poisoned rows that had to be manually removed

Due to this disparity between data sources, a quality score was assigned to each source so that questions can be filtered by this column as necessary to optimize model output during training. 20 contexts were chosen at random from each data source and manually assessed with regard to the following criteria:

1. Question and answer count per context: Higher counts are critical for AE training.

2. Question and answer formatting: Do examples include unicode or non-English characters?

3. Answer grouping quality: Are lists presented as multi-point examples? Are there multiple concepts needlessly grouped together into multi-point questions?

4. Context for end-users: Do questions include enough of their source context to help the user recall the correct passage?

Based on these criteria, Lisa Myaing’s and Sharon Crews’s examples received the highest quality ratings of $\frac{9}{10}$. Surprisingly, some of the official NAD examples received the lowest rating of $\frac{3}{10}$ because they had very few questions per context

and were littered with random unicode characters due to the OCR that had to be performed on the provided PDFs.

Before all the datasets were aggregated, however, each needed to contain the same columns. Babienco's database dump used four columns to specify where in the Bible to find the question's context: `book`, `chapter`, `verse`, and `endVerse`. This database also contained a column that specified the number of points each question was worth: `points`. All other sources included this information in the question column as shown in Text 3.4. This column not only included the question, but also a reference for the context and the point value. Thus, we wrote a Python cleaning script, pbeClean.py, to extract this information into separate columns to match Babienco's dataset.

According to 1 Peter 1:23, how have you been born? 3 points

Text 3.4: Important information is included as semi-structured plain text.

The script, shown in Listing 3.1, primarily utilizes regular expressions and built-in string manipulation functions in Python. First the two datasets, both named after their primary data sources, are loaded. `lsb` (Lisa, Sharon, Beth) contains the columns `refQuestion`, `answer`, `categories`, `source`, and `quality`. The `bab` (Babienco) dataset, on the other hand, contains `question`, `answer`, `points`, `book`, `chapter`, `verse`, `endVerse`, and `source`. For brevity, we use the `Dataframe` named "`df`" to represent operations performed on both datasets, and we condense some expressions or lists into descriptors surrounded by angle brackets (< and >). The script performs these basic data cleaning steps before aggregation:

1. Trimming whitespace and removing consecutive spaces (line 8).

2. Basic question and answer deduplication (line 10).

```
1  import pandas as pd, numpy as np, re, csv
2  # load data and fill in Babienco's source and quality columns
3  lsb = pd.read_excel('<excelDataSource>')
4  babMain = pd.read_csv('<databaseDumpDataSource>', sep=';')
5  babCSV = pd.read_csv('<csvFileSource>')
6  bab = pd.concat([babMain, babCSV])
7  bab['source'] = 'Babienco'; bab['quality'] = 8;
8
9  # 1. trim whitespace, rm consecutive spaces
10 df = df.replace(r'^ +| +$', r'', regex=True).replace(r'\s+', ' ', regex=True)
11 # 2. basic deduplicate on reference, question, and answer
12 df = df.drop_duplicates(subset=['<question>', '<answer>', '<refCols>'])
13 # 3. filter to remove FITB && T/F
14 fitbTF = r'_|fitb|fill in the blanks|t\/f|true or false'
15 df = df[df['<question>'].str.contains(fitbTF, regex=True, flags=re.I) == False]
16 df = df[df['<answer>'].str.contains(r'^(true|false)', regex=True, flags=re.I) == False]
17 # 4. drop rows with missing references, questions, or answers
18 df = df.dropna(subset=['<question>', '<answer>', '<refCols>'])
19 # 5. extract point values "2 points", "2-pts." etc.
20 ptsRe = r'\s*\(?(?P<points>\d+)\s*-?(?:point|pt|pt)s?\.?\)?\s*'
21 lsb['refQuestionCategories'] = lsb['refQuestion'] + lsb['categories'].astype(str)
22 lsb['points'] = lsb['refQuestionCategories'].str.extract(ptsRe, flags=re.I)
23 lsb['points'] = lsb['points'].replace(np.nan, 1).astype(np.int64)
24 lsb['refQuestion'] = lsb['refQuestion'].str.replace(ptsRe, '', flags=re.I, regex=True)
25 lsb['categories'] = lsb['categories'].str.replace(ptsRe, '', flags=re.I, regex=True)
26 # 6. pull categories out into their own columns
27 cats = ['2To3', 'bigPoints', 'people', 'places', 'names', 'numbers']
28 for cat in cats: df[cat] = df['categories'].str.contains(cat.lower()) == True
29 # 7. extract reference "according to..."
30 refRe = r'\s*According to (?P<book>(?:\d\s)?[a-zA-Z]+)\s(?P<chapter>\d+):(?P<verse>\d+)
       (?:[-,]?[ ]?(?P<endVerse>\d+))?,?\s*'
31 newCols = lsb['refQuestion'].str.extract(refRe, flags=re.I)
32 lsb['question'] = lsb['refQuestion'].str.replace(refRe, '', flags=re.I, regex=True)
33 lsb = pd.concat([lsb, newCols], axis=1)
34 # 8. combine datasets
35 cols = ['<refCols>', '<question>', '<answer>', '<commonCols>', '<categoryCols>']
36 df = df[cols]
37 data = pd.concat([lsb, bab])
```

Listing 3.1: Simplified data aggregation steps from pbeClean.py

3. Removal of unwanted question types: fill in the blank questions and true or false questions (line 12).

4. Dropping rows with missing references, questions, or answers (line 16).

5. Extraction of point values from various columns (line 18).

6. Extraction of extra categorical data into separate columns (line 25).

7. Reference extraction from the column containing mixed information (line 30).

Finally, the `lsb` and `bab` `Dataframes` are joined in the eigth step using a the `concat` function. The rest of the script performs data cleaning.

3.3 Data processing

The most important part of training better models as the project progressed was improving the data the model was trained on. Before and during model training, we continuously updated the remainder of the `pbeClean.py` data-cleaning script. For brevity and clarity, we explain all the cleaning steps and improvements as they appear in the cleaning script in Listing 3.2.[2] The cleaning steps, continued from Listing 3.1, are as follows:

9. Convert columns to their proper data types and fill in the `endVerse` column (line 38).

10. Filter out rows with impossible references (line 42).

11. Standardize point formatting in multi-point questions so that the model only learns one notation: `'(#)'`. First we attempt to parse the answers based on a format assuming points are numbered within the answer (line 44). If this does not work, we attempt to split the answer on commas or other punctuation the authors of the example use (not shown).

12. Occasionally, data authors capitalized words such as "NOT" for emphasis. For the sake of data consistency, we remove this capitalization (line 62).

13. Remove simple instructions about how to answer questions (line 65).

[2]Note that this code is only representative of the full script found at https://github.com/MatousAc/qag/blob/main/scripts/pbeClean.py

```
# 9. change column type as necessary
data['endVerse'] = data['endVerse'].fillna(data['verse'])
data['chapter'] = data['chapter'].astype(np.int64)
data['answer'] = data['answer'].astype(str)
# 10. drop rows where endVerse was parsed as larger than start verse
data = data[data['endVerse'] >= data['verse']]
# 11. format all numbered answers the same: (#)
def formatAnswers(answer: str, allegedPoints: int):
  def processAnswer(match): # local function for processing each number
    number = next(group for group in match.groups() if group is not None)
    return f'({number}) '

  numRe = r'(?:(?:\((\d+)\))|(?:(\d+)\.)|(?:(\d+)\)))\s*'
  # substitute different formats w/ correct format using above function
  answer = re.sub(numRe, processAnswer, answer)
  pointCount = countPoints(answer)

def countPoints(answer: str): # a simple point counting function
  numRe = r'\((\d+)\)'
  return max(len(re.findall(numRe, answer)), 1)

# apply the functions to the dataset
data['answer'] = data.apply(lambda r: formatAnswers(r['answer'], r['points']), axis=1)
data['points'] = data['answer'].apply(countPoints)
# 12. uncapitalize unnecessarily capitalized words
capsRe = r'\b([A-Z]{2,})\b'
data['col'] = data['col'].str.replace(capsRe, lambda m: m.groups()[0].lower(), regex=True)
# 13. remove all formats and misspellings of the Be Specific specifier
beRe = r'\s*\(?be\sspe((cific)|(ific)|(cfic)|(cifc))\.?\)?\.?\s*'
data['question'] = data['question'].str.replace(beRe, r'', flags = re.I, regex=True)
# 14. remove any rows with a point-value greater than 10
data = data[data['points'] <= 10]
# 15. return lost quotes
data['question'] = data['question'].str.replace('<badChar>', "'")
# 16. remove "Do not confuse with verse #", 'Note:' and "Different from ..."
dncRe = r'(?:\(?(?:(?:do not confuse)|(?:note:?)).*\)?)|\((?:different).*\)\.?'
data['question'] = data['question'].str.replace(dncRe, r'', regex=True, flags=re.I)
# 17. replace special/double characters
data['answer'] = data['answer'].str.replace(<regexForAllCurlyQuotes>, r'"', regex=True)
# 18. remove surrounding quotes and periods
data['answer'] = data['answer'].str.replace(r'^"(.+)"$', r'\1', regex=True)
# 19 remove anything within parentheses or brackets that is not just a point value
data['question'] = data['question'].str.replace(parenRe, r'', regex=True)
# 20. final trimming, stripping, and replacements
data['question'] = data['question'].str.strip().str.replace(' ?', '?')
# 21. replace "None" with "none" so that these answers are properly loaded next time
data['answer'] = data['answer'].str.replace(r'^None$', r'^none$', regex = True)
# 22. final deduplication based on reference, question, and answer
data = data.drop_duplicates(subset=['book', 'chapter', 'verse', 'question', 'answer'])
# 23. save to a csv
data.to_csv(<dataDest>, index=False)
```

Listing 3.2: Simplified data cleaning steps from pbeClean.py

14. Filter out questions worth more than ten points, as anything greater is excessive (line 68).

15. Replace badly-encoded smart quotes with regular ASCII quotes (line 70).

16. Data authors sometimes included hints in their questions for end users. These generally informed users to "not confuse one verse with another". The model tended to overproduce these hints incorrectly, so we removed them all (line 72).

17. Here we replace non-ASCII characters, primarily curly quotes, with their ASCII representations (line 75). Due to issues with rendering these characters in LaTeX, they are not included. We also fix doubled up quotes and parentheses here.

18. Some authors excessively quoted all their answers or placed periods at the end of answers. This step removes these issues from the dataset (line 77).

19. This step removes unnecessary information in parentheses and brackets (line 79). The regex is not included as it is full of quote characters that are not properly highlighted by `listings`.

20. After all the above steps are performed, questions and answers may have spaces or punctuation littering their beginning or end. Thus, we trim and strip the columns (line 81).

21. This step replaces answers that are literally "None" with "none" so that `pandas` will load them as strings in the future (line 83).

22. Finally, we deduplicate the dataset using a subset of its columns (line 85). This step removes only approximately 0.3% of our data.

23. We save the dataset as a CSV file (line 87). At this point, the dataset is cleaned and saved in the format shown in Figure 3.2.

book	chapter	verse	endVerse	question
1 Peter	1	4	4	What kind of inheritance has God our Father given us?
...	...	...	...	...

answer	points	source	bigPoints	...
(1) incorruptible (2) undefiled (3) does not fade away (4) reserved...	4	Beth deFlutier	TRUE	...
...	...	...	...	...

Figure 3.2: The data format after initial cleaning

From this point on, all data processing, model training, and QAG inference is handled by the family of classes in the `src/` folder. This allows various aspects of input/output cleaning methods and training parameters to be easily changed in a central location and connected thorough inheritance or dependency. The class hierarchy is shown in Figure 3.3.

The `ConfigBase` class is a base class for most of the classes in the `src/` folder. It sets up common configurations, utility f(x)s, and member variables using the `qag.ini` configuration file. This configuration file follows the basic section, key, value format demonstrated in Listing 3.3. This configuration file is then parsed by the `configparser` package allowing our program to read the individual values using python dictionary syntax: `value = configParser['section']['key']`. The configuration file controls model and data paths, model prompt templates, training hyperparameters, evaluation metrics, and more.

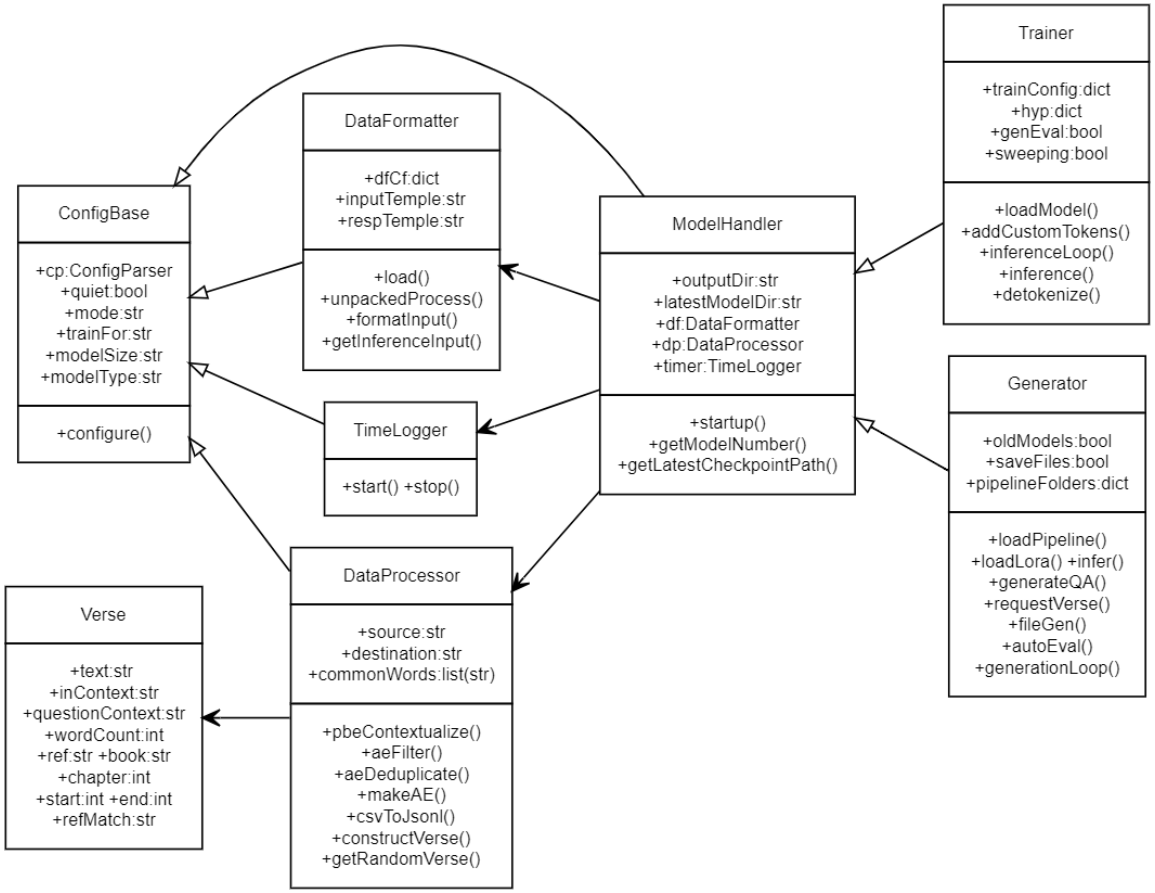

Figure 3.3: UML class diagram for the QAG repository

```
[section]
key: value
```

Listing 3.3: The configuration file format

Because we wish to format the PBE dataset similarly to Ushio et al.'s inverted SQuAD (Figure 3.1), we had to convert Bible verse references into the actual contexts. The only difference between our format and Ushio et al.'s is that we include several extra categorical columns and we do not highlight answers within their contexts using `<hl>` tokens. Highlighting answers for multiple, potentially overlapping phrases does not work well when answers are joined during training.

We perform this data transformation from references to contexts using the

`DataProcessor`'s `pbeContextualize` function. This function first internally calls the `constructVerse` function and uses the returned verse to fill columns in a `DataFrame` before calling `data.to_csv(self.destination)` to save the data. The internal function, `constructVerse`, is more noteworthy because it interfaces with a `DataFrame` based on a CSV file containing the entire Bible. On line 2 of Listing 3.4, the function first passes a verse reference through the `*args` parameter and then lets the `Verse` constructor handle storing basic verse information such as the book, chapter, start verse, end verse, and plaintext reference. If the verse has a verse before or after it, these verses' numbers are calculated. Next the a `DataFrame` for just the current chapter is extracted from the Bible object and a function for locating a single verse is defined. Next context verses and the primary verse text are fetched and combined.

```
1  def constructVerse(self, *args) -> Verse:
2    v = Verse(*args)
3    # get context verse numbers if applicable
4    previousNum = v.start - 1 if v.start > 1 else None
5    followingNum = v.end + 1 if notLast(v.end)
6    # get verse text
7    chap = self.bible.loc[(self.bible['book'] == v.book)
8      & (self.bible['chapterNumber'] == v.chapter)]
9    def getVrs(num: int): return chap.loc[chap['verseNumber'] == num, 'verse'].values[0]
10
11   v.previous = getVrs(previousNum) if previousNum else ''
12   v.following = getVrs(followingNum) if followingNum else ''
13   targetVerses = []
14   v.text = ' '.join([getVrs(n) for n in range(v.start, v.end + 1)])
15   v.inContext = f'{v.previous} {v.text} {v.following}'.strip()
16   v.wordCount = len(v.text.split())
17   return v
```

Listing 3.4: Verse construction

Running the `pbeContextualize` function finally transforms the data into the

desired format shown in Figure 3.4. Once the data has been transformed into this standardized format, it can be easily accessed and reformatted. For example, because HuggingFace requires training data as JSON Lines, we convert file formats from CSV to JSONL by running `python dataProcessor -csvToJsonl`. This produces the final data format used for training our QG model.

answer	paragraph_question	question	sentence
sanctification of the Spirit	question: These elect receive what from the Spirit? conte...	These elect receive what from the Spirit?	elect according to the foreknowledge of God...
...	...	...	...

paragraph	paragraph_sentence	points	source	quality
Peter, an apostle of... elect according to the...	Peter, an apostle of... <hl> elect according to the foreknowledge...	1	Beth deFlutier	9
...	...	...	...	...

Figure 3.4: PBE dataset example

For answer extraction, we had to train the base model to extract several potential answers from a single context - thereby addressing QAG's one-to-many problem [22]. The format in Figure 3.4 does not directly contain these types of training examples. It must first be aggregated so that all the answers from a particular context are concatenated with a special token in one string. This data transformation is also performed using the `DataProcessor` via the `makeAE` function (see Listing 3.5). First we use `pandas` to filter the data by the quality currently specified in the configuration file. Then we aggregate the answers and deduplicate them so that the model learns to produce diverse answers rather than repeated ones.

While the Listing 3.5 may appear trivial, it internally uses the answer deduplication function, `aeDeduplicate` which is much more complex. Only a simplified

```
def makeAE(self):
  data = pd.read_csv(self.source)
  qualityThreshold = int(self.cp['dataFormatter']['qualityThreshold'])
  data = data[data['quality'] > qualityThreshold]
  # group answers by verse and remove near duplicates
  grouped = data.groupby('sentence').agg({
    'answer': lambda x: ' <sep> '.join(self.aeDeduplicate(x)),
    'quality': 'mean'
  }).reset_index()
  dataset = Dataset.from_pandas(grouped) # HuggingFace format
  dataset.to_json(self.destination) # JSONL
```

Listing 3.5: Aggregating the dataset for AE training

version is shown in Listing 3.6 for brevity. The primary loop starting on line 22 collects unique answers in the `uniqueElems` set by looping over each answer and comparing it to any answers already present in the unique set.

The local function `compare` is where most of the logic lives. `compare` determines whether two answers are different enough to keep both of them, and if not, which answer to retain. It returns -1 for keeping the left answer, 0 for keeping both, and 1 for keeping the right answer. First, `compare` checks for identical answers and then for answers containing the same set of cleaned words. Next, it checks set length, assuming answers are not similar if their length difference is more than 50% of the smaller answer and more than 3 words. Finally, we calculate the sets' intersection and compare its size to a threshold value based on answer length. Answers are considered similar if they intersect "threshold" percent of the time or do not contain enough unique words for smaller answers. This way, we can dynamically specify a similarity percentage to deduplicate answers at. We prefer retaining the slightly longer answer because longer answers tend to include articles useful for the question generator and longer answers may be split up into multiple points.

As shown in Listing 3.5, the deduplicated answers are then joined using a separator and the `DataFrame` then written to a file as JSON lines. This yields the

```python
def aeDeduplicate(self, answers):
  answers = list(answers)
  def compare(text1, text2, threshold = 0.7):
    if text1 == text2: return -1 # identical
    # split texts into words
    words1 = set(text1.split())
    words2 = set(text2.split())
    longerText = -1 if len1 >= len2 else 1 # prefer left
    if words1 == words2: return longerText # if set identical
    # check length
    maxLen = max(len1, len2); minLen = min(len1, len2)
    if abs(len1 - len2) > max(int(minLen * 0.5), 3): return 0 # not similar

    # set similarity check
    if maxLen <= 3:
      # different enough if each has a word the other doesn't have
      if len(words1.symmetric_difference(words2)) >= 2: return 0
    if maxLen <= 5: maxNumSimilarWords = max(maxLen - 2, 1)
    else: maxNumSimilarWords = int(threshold * maxLen)
    if len(words1.intersection(words2)) <= maxNumSimilarWords: return 0 # not similar
    else: return longerText

  # collect the best unique answers
  uniqueElems = set()
  for currAns in answers:
    addFlag = True
    for uniqueElem in uniqueElems:
      decision = compare(uniqueElem, currAns)
      if decision == 0: continue # keep uniqueElem, compare current further
      if decision == 1: # replacement
        uniqueElems.remove(uniqueElem)
      elif decision == -1: # skip
        addFlag = False
      break
    # only add if different from all
    if addFlag: uniqueElems.add(currAns)
  return list(uniqueElems)
```

Listing 3.6: Simplified procedure for answer deduplication

data format shown in Figure 3.5 and sets up the final data required for fine-tuning.

sentence	answer	count
and into an inheritance that can never perish, spoil . . .	(1) perish (2) spoil (3) fade <sep> an inheritance <sep> in heaven	3
...	...	...

Figure 3.5: Answer extraction example format

3.4 Fine-tuning scripts

LLMs are complex. Simply running inference requires significant computing resources. In fact, LLM training typically requires expensive graphical processing units (GPUs), Compute Unified Device Architecture (CUDA) configuration, tensor computation libraries such as `torch`, adapter integration, training loops, metric computation, tokenization, input padding, attention masking, binary data formatting, and more. Rather than attempting to recreate the state of the art, we will use the HuggingFace ecosystem of libraries which simplifies training configuration and execution.

3.4.1 Training Configuration

HuggingFace `transformers`, published in 2020 [33], allows for easy model and tokenizer loading and configuration. For example, Listing 3.7 shows how easy it is to load a four-bit quantized model and its tokenizer within our `loadModel` function. Line 2 assigns a configuration object specifying the data type of the model. Lines 3 through 5 specify that the model should be loaded in four bits, the quantization (compression) type is "nf4", and the base model weights should be treated as PyTorch float16 values during computation. Lines 8 through 12 initialize a model object using a path to the model[3] and the `bnbConfig` created on lines 2 through 6. `device_map="auto"` allows HuggingFace `transformers` to automatically determine what hardware to execute model computation on. Finally, line 14 initializes the model tokenizer using the `tokenizer.json` file in the model folder, and line 16 returns the model to the caller (important for hyperparameter

[3]Note that the `pretrained_model_name_or_path` parameter can also accept a model name from HuggingFace which it will download at runtime. To avoid this each time we train a model, we store copies of the LLaMA 2 base models locally.

```
def loadModel(self):
  bnbConfig = BitsAndBytesConfig(
    load_in_4bit = True,
    bnb_4bit_quant_type = "nf4",
    bnb_4bit_compute_dtype = torch.float16
  )

  baseModel = AutoModelForCausalLM.from_pretrained(
    pretrained_model_name_or_path='<path>',
    quantization_config=self.bnbConfig,
    device_map='auto'
  )

  self.tokenizer = AutoTokenizer.from_pretrained(self.paths['base'])
  return baseModel
```

Listing 3.7: Basic configuration for a Causal_LM model.

sweeping).

Training arguments and hyperparameters must also be configured before training a model even if they have not yet been systematically optimized. To set most of this configuration, the `transformers` package provides the `TrainingArguments` class which is used to configure learning rate, epoch number, GPU configuration, evaluation metric scales, model checkpointing, and logging.

For example, on line 11 of Listing 3.8, the configuration specifies where to save the model during and after training. We set the output folder to the `Trainer`'s auto-incrementing `outputDir` member. If we receive new data and wish to rerun training, the `ModelHandler` class will automatically determine the next directory to save the model to such as "7b-chatAE05", "7b-chatAE06", and so forth.

The class can be instantiated with various different configuration options and then is passed into the trainer. Most of our configuration is read from our configuration file `qag.ini` and follows the pattern of retrieval and configuration shown in Listing 3.8 between lines 3 and 13.[4] Any settings that are not retrieved directly from the configuration file (such as `self.hyp['weightDecay']`) are set from the `hp` object which contains values from the configuration file that are

[4]For the full `configureTraining` configuration see `trainer.py` in our public repository.

```
1  def configureTraining(self, hp):
2    # configure wandb naming
3    os.environ['WANDB_PROJECT'] = 'sweep' if self.sweeping else self.trainFor
4    self.trainingArgs = Seq2SeqTrainingArguments(
5      # tunable hyperparameters
6      learning_rate = hp['learningRate'],
7      weight_decay = float(self.hyp['weightDecay']),
8      report_to = 'none' if self.sweeping or self.mode == 'test' else 'wandb',
9      run_name = os.path.split(self.outputDir)[1], # name after output folder
10     # output settings
11     output_dir = self.outputDir,
12     eval_steps = stepSize, # more settings ...
13   )
14
15   self.loraConfig = LoraConfig(
16     lora_alpha = int(hp['loraAlpha']),
17     lora_dropout = float(hp['loraDropout']),
18     r = int(hp['r']),
19     # enable more lora layers
20     bias = hp['bias'],
21     target_modules = [f'{l}_proj' for l in hp['loraLayers']]
22   )
```

Listing 3.8: Basic configuration for a Causal_LM model.

optionally overwritten by current sweep variables.

Notice the Weights & Biases integration settings on lines 3 and 8. Weights & Biases[5] (wandb) is a leading AI development platform that stores and visualizes project metrics such as model performance and configuration. It provides a Python package named `wandb` that enables further integration such as automatic logging during training and hyperparameter sweeping across multiple threads and multiple devices at once. HuggingFace seamlessly integrates with wandb for model evaluation logging, system information logging, metric visualization, and hyperparameter sweeping.

Line 8 specifies that this integration will be used when we are not testing. `self.sweeping` is also technically excluded in the block below because starting a sweep automatically reports to wandb and specifying it again doubles up reporting processes and breaks the integration. Line 9 sets the run name to the model's output directory, such as the "7b-chatAE05" mentioned above. Finally, line 3 sets

[5] https://wandb.ai

a project name for the current run. We set this to the `trainFor` member, which can be `AE` or `QG`, so that our fine-tuning runs are automatically categorized into two types of fine-tuning experiments. Sweeps are also categorized into their own project.

Because we plan to only train LoRA [32] for our base model, we had to configure various LoRA-specific settings. Again, we use the HuggingFace ecosystem because it seamlessly implements LoRA in the `peft` library and integrates it with the transformer reinforcement learning (`trl`) model-training library. On lines 16 through 21 we specify the α scaling factor, adapter dropout rate, rank r (size) of the adapter, bias type, and which LoRA layers to train. This is then wrapped in a `LoraConfig` object so it can easily be passed to the HuggingFace trainer.

The final step to perform before training is to load and format the training examples with the `DataFormatter`. This occurs in three main `DataFormatter` functions: `load`, `unpackedProcessing`, and `formatInput`. When the `DataFormatter` is created, `load` creates `DataFrames` containing the appropriate data based on the `trainFor` configuration using HuggingFace `datasets`' `load_dataset` function. Next, the examples are filtered by the quality specified in `qag.ini`. Using the method `dataset.train_test_split`, the data is then split into training and evaluation sets that become members of the `DataFormatter` object.

The `unpackedProcessing` function, shown in Listing 3.9, is simple. It loops through the provided rows and creates a list of strings that are used as training examples. Because the `examples` parameter provided by the HuggingFace trainer is a dictionary of lists, we iterate using an index and then let the internal function locate the correct data row using this index.

Finally, the `formatInput` function (shown in Listing 3.10), transforms each data row into a string that the model will train to predict. HuggingFace calls the

```
def unpackedProcessing(self, examples) -> list:
  output_texts = []
  for i in range(len(examples["answer"])):
    text = self.formatInput(examples, i)
    output_texts.append(text)
  return output_texts
```

Listing 3.9: unpackedProcessing is used by the trainer to set up each plaintext training example.

```
def formatInput(self, example, i = 0,  formatFor: str = None) -> str:
  if formatFor == None: formatFor = self.trainFor # default
  if isinstance(example['answer'], list): # for unpacked processing
    context = example['sentence'][i]
    answer = example['answer'][i]
    if formatFor == 'QG': question = example['question'][i]
  else: # for packed processing or generation
      context = example['sentence']
      answer = example['answer']
      if formatFor == 'QG': question = example['question']
  # construct example
  template = self.dfCf[f'inputTemple{formatFor}']
  template = template.replace('<context>', context)
  template = template.replace('<answer>', answer)
  if formatFor == 'QG': template = template.replace('<question>', question)
  return template.strip()
```

Listing 3.10: formatInput places the context, answers, and questions into the appropriate input template matching the formatFor parameter

words that go right before the model's output the response template. In our case, that would be the something like `\#\#\# Generate Question:`. We will also call this the response template. Likewise, we will call the entire format of the input the input template. This input template is taken from the configuration file and determines which prompt the `DataFormatter` should use to prepare the training data. The `DataFormatter` takes this input template and interpolates the actual values for the context, answer, and question.

For example, the input template in Text 3.5 was used early on in AE fine-tuning. `formatInput` replaces the `<context>` placeholder with the row's `sentence` column, the `<answer>` placeholder with that row's `answer`, and if fine-tuning for QG, the `<question>` placeholder with the actual `question`. Note that `formatInput` also

accepts a dictionary containing keys and values provided by the caller. This allows this function to be used for packed processing and generation. Packed processing involves placing several training examples into a single string separated by end-of-sequence (EOS) tokens so that several training examples can be executed at once and training is sped up. While we enable this method of training, we never needed to utilize it as training proceeded in a timely manner.

<s> ### Extract any nouns, noun phrases, actions, key phrases, and lists that appear in the following context and return them separated by <sep> . ### Verse: <context> ### Nouns, noun phrases, actions, key phrases, and lists: <answer>

Text 3.5: An early input template for AE training

3.4.2 Training

With the training arguments set, the LoRA configuration defined, and the data loaded, the model can begin training. Because LLaMA 2 is trained using Reinforcement Learning with Human Feedback (RLHF), we will use HuggingFace's trainer tailored for Supervised Fine-tuning: the `SFTTrainer`. Listing 3.11 shows the basic construction and use of this trainer, taken from our own `train` function.

We provide this trainer with the objects and functions prepared above: the base model, tokenizer, training configurations, data, and data formatting function. However, there are also some new parameters. The data collator is responsible for setting the attention values for each token in the full training example. The `DataCollatorForCompletionOnlyLM` object masks all tokens before and including the `self.respTemple`, or the response template. When a token is "masked" its attention value is set to -1 so that the the model does not pay attention to optimizing for it while backpropagating error. Thus, rather than teaching the model about the context data provided in the prompt, the fine-tuning process only

```
1  collator = DataCollatorForCompletionOnlyLM(self.df.respTemple, tokenizer=self.tokenizer)
2
3  trainer = SFTTrainer(
4    model = baseModel,
5    train_dataset = self.df.trainDataset,
6    eval_dataset = self.df.evalDataset,
7    peft_config = self.loraConfig,
8    formatting_func = self.df.unpackedProcessing,
9    max_seq_length = int(self.trainConfig['maxSeqLength']),
10   tokenizer = self.tokenizer,
11   args = self.trainingArgs,
12   packing = self.trainConfig['packing'] == 'True',
13   data_collator = collator,
14   compute_metrics = self.nlgMetrics if self.genEval else None,
15   preprocess_logits_for_metrics = self.preprocessLogits if self.genEval else None,
16 )
17
18 trainer.train()
```

Listing 3.11: Basic creation and use of the SFTTrainer inside our train function

teaches the model to transform the context into the desired output. In other words, the model only internalizes the process of transforming the input context to the output (extracted answers or generated questions), but it does not remember any of the original context.

Other training configurations allow us to control the maximum output sequence length and what metric function to report during training. We control whether to use default or custom metrics using the `generativeEval` key in the configuration file.

Once the trainer has been configured, we begin training on line 18. The `SFTTrainer` automatically detects available GPUs and distributes model training across them using PyTorch and CUDA. During model fine-tining, the `SFTTrainer` saves model checkpoints and tracks the best version of the model throughout training. At the end, the trainer saves the best adapter checkpoints so that that they can be loaded for inference with the `Generator`.

3.4.3 Inference

We handle the final process of QAG with our `Generator` class. Fortunately, where most pipeline systems would need to handle two models simultaneously, one for AE and one for QG, our method allows us to load a single base model and two relatively lightweight adapters. We load the base model with the same logic as in the trainer (Listing 3.7). Adapters are loaded and labeled with the model's `load_adapter` method in Listing 3.12.

```
model.load_adapter(adapterLocation, adapter_name='<AE|QG>')
```

Listing 3.12: A simple Python expression for loading adapters

Listing 3.13 shows the `Generator`'s `infer` function which handles generation and extraction of output given the prompt and pipeline stage/adapter name. To load an adapter, we simply use the `set_adapter` function on line 3. `torch.nograd` simply means that gradients for backpropagation will not be computed during this inference. This eliminates a useless step and slightly speeds up model generation. Otherwise, the inference procedure is as expected: input tokenization, model generation, and detokenization.

To facilitate the the "pipeline" process, the `Generator` defines a `generateQA` function that takes any `Verse` object and returns a `DataFrame` of questions and answers. Using the `DataFormatter`'s `formatInput` function, `generateQA` first formats a model input for answer extraction using `Verse` objects. The "answer" passed into the `formatInput` is an empty string because we want the model to generate these answers. Next, the generator leverages its `infer` function to receive output, it splits the answers apart, and finally filters and cleans them. We have seen the answer deduplication function before in Listing 3.6, but the function `aeFilter` is new. This function simply filters out answers with the following disqualifying

```
1  def infer(self, inferenceInput: str, pipelineType: str):
2    self.timer.start()
3    self.model.set_adapter(pipelineType)
4    modelInput = self.tokenizer(inferenceInput, return_tensors='pt').to('cuda')
5    self.model.eval()
6    with torch.no_grad():
7      tokens = self.model.generate(**modelInput, max_new_tokens=100)[0]
8      output = self.tokenizer.decode(tokens, skip_special_tokens=True)
9      self.timer.stop() # the model's job is done @ this point
10     # only return what was generated
11     response = output.split(self.cp['dataFormatter'][f'respTemple{pipelineType}'])[1]
12     return response
```

Listing 3.13: The generator's infer function

characteristics:

- a word count over 25
- more than 13 words and only 1 point
- multiple points where an answer for one point is much longer than the answer to another point
- multiple points that are identical or nearly identical (sometimes the model would repeat itself between points)

Once the `Generator` obtains a cleaned, filtered, deduplicated list of answers, it performs QG for each answer (Line 10 of Listing 3.14). The procedure is very similar to AE inference except that rather than using the model's full output, we just use the text generated up to the first question mark to capture the first question generated. This question is sometimes the only question generated, and if not, is always the most relevant question to the provided answer. The final question is constructed using the generated text, and the point number. The accumulated `DataFrame` of questions and their answers is then returned so that it can be displayed, saved to a file, or sent through an API.

```python
def generateQA(self, verse: Verse) -> pd.DataFrame:
  qa = pd.DataFrame(columns=['question', 'answer'])
  # AE
  aeInput = self.df.formatInput({'sentence': verse.text, 'answer': ''}, formatFor = 'AE')
  # split answers and remove the last one
  answers = self.infer(aeInput, 'AE').split('<sep>')[:-1]
  answers = [a.strip() for a in answers] # clean whitespace
  answers = self.dp.aeFilter(answers)
  answers = self.dp.aeDeduplicate(answers)
  # QG
  for answer in answers:
    qgInput = self.cp['dataFormatter']['inputTempleQG']
    qgInput = self.df.formatInput({
      'sentence': verse.questionContext, 'answer': answer, 'question': verse.ref + ','
    }, formatFor = 'QG')
    question = self.infer(qgInput, 'QG')
    question = question.split('?')[0] # only the first question is relevant
    ptNum = self.countPoints(answer) # count points and prepend
    question = question.strip()
    question = f'({ptNum}pt{"s" if ptNum > 1 else ""}) {question}?'
    qa.loc[len(qa)] = [question, answer]
  return qa
```

Listing 3.14: The Generator's primary QAG function

3.5 Training Experiments

Collecting data and building all the above classes and functions enables model training and sets up a basic QAG system. However, many specific questions about training and inference remain unanswered or entirely unasked. What is the best way to prompt the model? How many directions does the model need? Does the model perform better with more or less context? Even with the overall method determined through research, the best QAG system can only be achieved through many training experiments and by supplementing even the best model's shortcomings with custom algorithms.

We began training both AE and QG models with very specific prompts in order to try to instruct the model to perform the desired task. For example, our first AE model, **7b-chatAE00**, was trained to respond to the prompt in Text 3.6. Prior to training, the LLaMA-2-Chat base model could sometimes extract a few nouns, but the formatting and variety fell far short of the necessary performance

for the desired use case. With the **00** adapter, the model produced a much more consistently-formatted list containing important people, places, actions, phrases, and even some multi-point questions with the correct formatting. The primary issue with **7b-chatAE00** was that it tended to duplicate the answers it produced and could not naturally terminate its output, running out of the maximum tokens it was allowed to generate instead.

<s> ### Extract any nouns, noun phrases, actions, key phrases, and lists that appear in the following context and return them separated by <sep> . ### Verse: <context> ### Nouns, noun phrases, actions, key phrases, and lists: <answer>

Text 3.6: An early input template for AE training

To mitigate our first model's answer duplication problem, we lowered the similarity threshold in the `DataProcessor.aeDeduplicate` function and refined the algorithm to result in fewer, more unique answers in the training data. As expected, the resulting **7b-chatAE01** model directly reflected the change the training data and produced less duplicate answers.

The first question generation models, on the other hand, suffered from producing unnecessary words and characters after their question. The unprocessed output in Text 3.7, shows model **7b-chatQG00** attempting to optimize for loss by overproducing "hints" that were occasionally present in the training data. To fix this issue with the model, we had to further clean the data. We primarily focused on removing the notes authors left beside questions using `pbeClean.py`, shown in step 16 of Listing 3.2.

Training **7b-chatQG01** on the cleaner data removed the hints in the format "(Do not confuse with <verse> .)", but still left stray punctuation marks behind the question. Moreover, in the place of the hints, the model tried to continue the context by either continuing or repeating the context.

Given the following context verse and answer, write a question for the answer. ### Verse: by so much more Jesus has become a surety of a better covenant. ### Answer: A surety of a better covenant ### Question: How has Jesus become?.) (Do not confuse with 7:22a.) (Do not confuse with 7:23.) (Do not confuse with 7:24.) (Do not confuse with 7:25.) (Do not confuse with 7:26.) (Do not confuse with 7:27.) (Do not confuse with 7:28.)

Text 3.7: Early output from **7b-chatQG00**

Next we trained both models, **7b-chatAE02** and **7b-chatQG02**, with custom tokens added to the tokenizer to see whether tokens with newly-defined meanings would improve performance. Past QG projects have used various separator tokens including `B`, `I`, and `O` for "begin", "continue", and "end" [2], `<sep>` [19], `*` [16], `<answer>` and `<context>` [20], and `<hl>` [29]. Some of these special tokens are used to mark output within its context, while others separate the concatenated input and output.

Lopez et al. [18] report the most controlled special token experiments in QG using GPT-2. They use three different separation token schemes: special tokens (`[SEP]`), normal English words, and numbers. Their automatic evaluation indicates that numbers and special tokens have an advantage over plain text for eventual performance. However, because LLaMA 2 does not recognize any of these tokens by default, we introduce our own special tokens: a padding token, the separator used for separating answers in the AE model, and a highlight token in case we wanted to more closely match Ushio et al.'s dataset from Figure 3.1 in the future.

To add custom tokens we use our `Trainer`'s `addCustomTokens` function shown in Listing 3.15. Line 8 updates the model's "token embeddings". In other words, the base model is configured to produce a list of 32,000 probabilities, one for each token in the tokenizer's vocabulary. Thus, when we expand the tokenizer, we must also prompt the model to expand its output by three additional probabilities to

```
def addCustomTokens(self, model: AutoModelForCausalLM):
  specialTokens = {
    "pad_token":"<pad>",
    "sep_token":"<sep>",
    "sep_token":"<hl>"
  }
  numAddedToks = self.tokenizer.add_special_tokens(specialTokens)
  return model.resize_token_embeddings(len(self.tokenizer))
```

Listing 3.15: Adding custom tokens to the tokenizer and then resizing the model

account for the three additional tokens. Finally, all that is left is to turn on custom tokens in the configuration file and re-run training.

Unfortunately, rather than improving model output, including custom tokens greatly reduced the AE model's capabilities. The answer extraction model failed to extract even one valid answer or separator. Instead it returned the continuous output in Text 3.8. The QG output did not improve in any noticeable way either.

Extract any nouns, noun phrases, actions, key phrases, and lists that appear in the following context and return them separated by <sep> . ### Verse: But whoever has this world's goods, and sees his brother in need, and shuts up his heart from him, how does the love of God abide in him? ### Potential answers: His brother in need (the brother in need is his neighbor) and shuts up his heart from him (he does not help him) how does the love of God abide in him? (it

Text 3.8: Adding custom tokens in **7b-chatAE02** resulted in poor performance

Whether the poor performance was a result of adding custom tokens or of not properly configuring model training or inference with custom tokens, due to time limitations and related research, we decided to move forward without utilizing custom tokens.

As we trained the various early models, we attempted to improve their output by providing more instructions in their prompt. However, we saw no improvement in model performance as we improved the prompts. We hypothesized that the exact prompt content was not as important as the prompt's consistency from training to inference. That is, as long as we used the same prompt during inference

as we provided during training, the model would perform the same transformation from context to output at inference as it was trained to perform during fine-tuning.

Thus, we trained our next models with less specific prompts. The input template in Text 3.9 was used for **7b-chatAE03**. Reducing the length and specificity of the prompt did not impact the AE or QG models' outputs. In fact, the automated evaluation loss, visualized in Figure 3.6, significantly decreased between the **02** and **03** AE models. The template in Text 3.10 was used for **7b-chatQG03** and did not noticeably affect model output or loss. Due to their performance and simplicity, these less instructional prompts were used for the remainder of the project.

<s> ### Extract potential answers to questions and return them separated by <sep> . ### Verse: <context> ### Potential answers: <answer>

Text 3.9: An input template for AE with less instructions for the model

<s> ### Write a question for the context and answer. ### Verse: <context> ### Answer: <answer> ### Question: According to <question>

Text 3.10: A generic input template for QG

While prompt specificity had little impact on the question generation models, training data impacted QG model loss more than AE loss. Thus far, model training was terminated after 1000 training steps. While this was enough steps to cycle through the nearly 5000 AE examples, it did not suffice for the 60,000 QG examples. Thus, the next models were trained on a full epoch of data rather than for a specified number of steps. The additional data and reduced evaluation loss by 0.15 between the **03** and **04** QG models (see Figure 3.7).

Despite the positive results above, the eval loss metric is not ideal for evaluating question generation. Calculating the loss between a sequence of produced tokens fails to capture the importance of grammatical correctness, logical sense, word order, and related word proximity. Therefore, while training our next iteration of

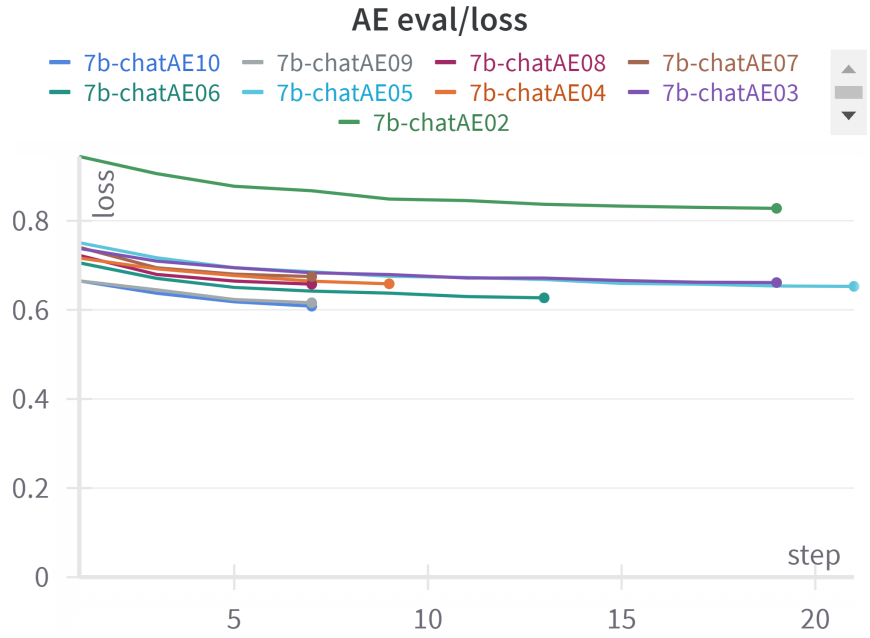

Figure 3.6: Evaluation loss for most of the AE models trained in this project. Lower values aim to indicate better models.

models, we supplied custom evaluation metrics to the `SFTTrainer` which reported BLEU, ROUGE-L, and METEOR scores.

To calculate these machine translation (MT) metrics, we used HuggingFace's `evaluate` library. Listing 3.16 shows the two functions required to implement custom metrics while training. The `preprocessLogits` function deals with reducing the logits generated by the model into tokens. For each token that the model generates, the `logits` object contains an array of non-normalized probabilities the size of the current tokenizer vocabulary. Each probability in this enormous list represents the likelihood that the next token in the generated sequence should be the token with the id that is the index of the probability value within the list. Because higher values represent higher probabilities, we just pick the index of the highest value in

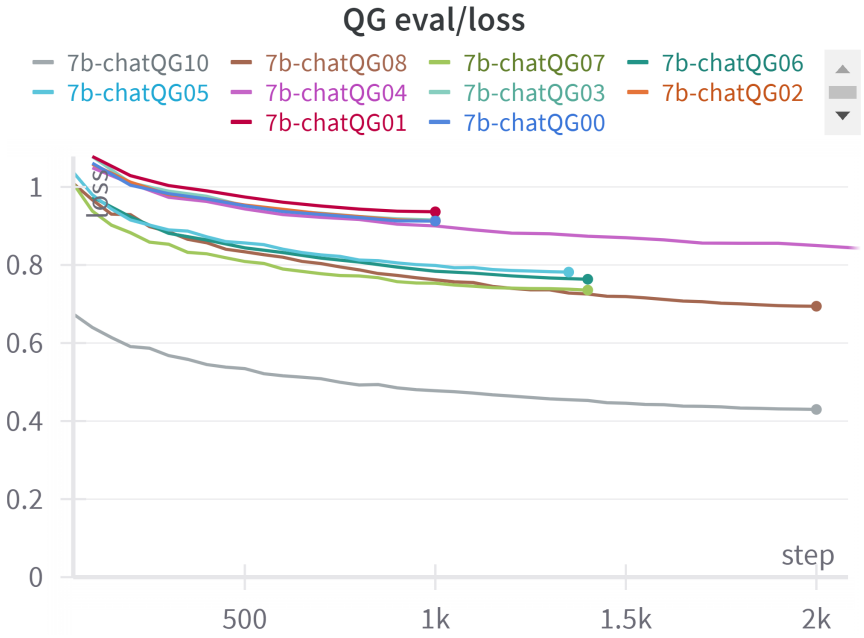

Figure 3.7: Evaluation loss for all QG models

the list to get the most probable token id.[6] While longer, the `nlgMetrics` function is actually simpler to describe. It receives a tuple of predictions and references (called labels) from the model and decodes them using the tokenizer so that the MT metrics can process the plaintext predictions and references. On line 9 we ignore the -100 tokens, as those are the tokens from the prompt and don't need to be compared to the references. Next, each metric is loaded with `evaluate.load`, calculated according to the HuggingFace evaluate documentation, and stored into the `result` dictionary. Finally, we report the results scaled up as values between 0 and 100 as is common in QAG research.

These two functions are then passed to the `Trainer`'s constructor through the `preprocess_logits_for_metrics` and `compute_metrics` parameters, respectively.

[6]The `dim = -1` parameter specifies to only reduce the deepest arrays, as the logits object is actually a triply-nested list of many sequences each containing a list of tokens that is itself a list of probabilities. In machine learning, these last two layers are often referred to as `Tensors`.

```
def preprocessLogits(self, logits: torch.Tensor, labels: torch.Tensor) ->  torch.Tensor:
  return logits.argmax(dim = -1)

def nlgMetrics(self, evalPred):
  preds, labels = evalPred
  preds = np.where(preds != -100, preds, self.tokenizer.pad_token_id)
  preds = self.tokenizer.batch_decode(preds, skip_special_tokens=True)
  labels = np.where(labels != -100, labels, self.tokenizer.pad_token_id)
  labels = self.tokenizer.batch_decode(labels, skip_special_tokens=True)

  rouge = evaluate.load('rouge')
  bleu = evaluate.load('bleu')
  meteor = evaluate.load('meteor')
  result = {
    'rougeL': rouge.compute(predictions=preds, references=labels,
              use_stemmer=True, use_aggregator=True, rouge_types=['rougeL'])['rougeL'],
    # get fourth precision which reports bleu-4
    'bleu': bleu.compute(predictions=preds, references=labels)['precisions'][3],
    'meteor': meteor.compute(predictions=preds, references=labels)['meteor']
  }
  result = {key: value * 100 for key, value in result.items()} # scale between 0-100
  return result
```

Listing 3.16: Adding custom tokens to the tokenizer and then resizing the model

Figure 3.8 shows the BLEU, ROUGE-L, and METEOR scores calculated during two QG models' training. Unfortunately, the figure further casts doubt on the usefulness of automated metrics for QG model evaluation. While the metric scores vary across model, they do not improve between the beginning and end of training. In fact, the BLEU score for **7b-chatAE06** decreases from start to finish even though it was one of the best AE models this project produced.

This seems to suggest that the base model is equally fit to author questions as a fully trained one, and that the performance difference between the two models in Figure 3.8 is due to other settings entirely, such as additional context. However, as discussed above this is not the case at all. Fine-tuning greatly improved model output even with the first **00** models. Instead, these metric values indicate the shortcomings of their own calculation. As is often the case with automated evaluation of generated text, even evaluation metrics tailored to text generation are poor indicators of model performance in a specific domain. For completeness, we report the machine translation metrics for several following models in Figure 3.9.

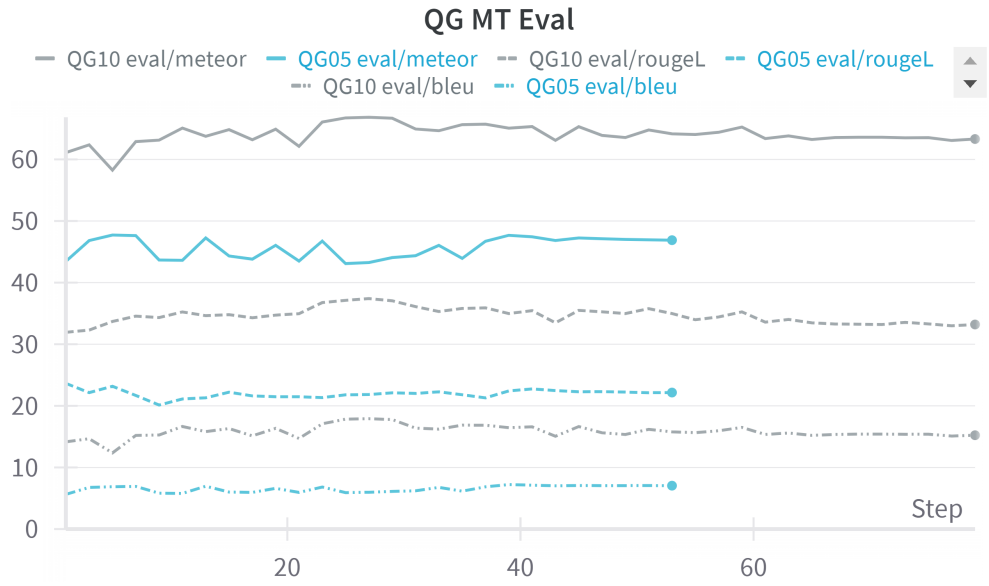

Figure 3.8: Automatic MT evaluation for two QG models over their training

Because the metrics did not change significantly over the course of training, we show only the final values for the metrics and leave further evaluation discussion for later chapters. Moving forward, decisions about which model performed "better" during the experiment stage were not made based on these metrics but rather on manual inspection.

By the sixth and seventh iterations of model training, the input data had been thoroughly cleaned with the perfected `pbeClean.py` script and model performance had generally stabilized. Rather than trying to improve model performance overall, we focused on solving specific issues with the models. For example, testing the **7b-chatAE06** and **7b-chatQG05** models showed that they underperformed when their random contexts were especially short. Text 3.11 shows two such handpicked contexts and the questions that were problematic. In the first example, the QG model attempts to provide some context in the question itself (something the training data encourages), but because the context is so short, the model assumes

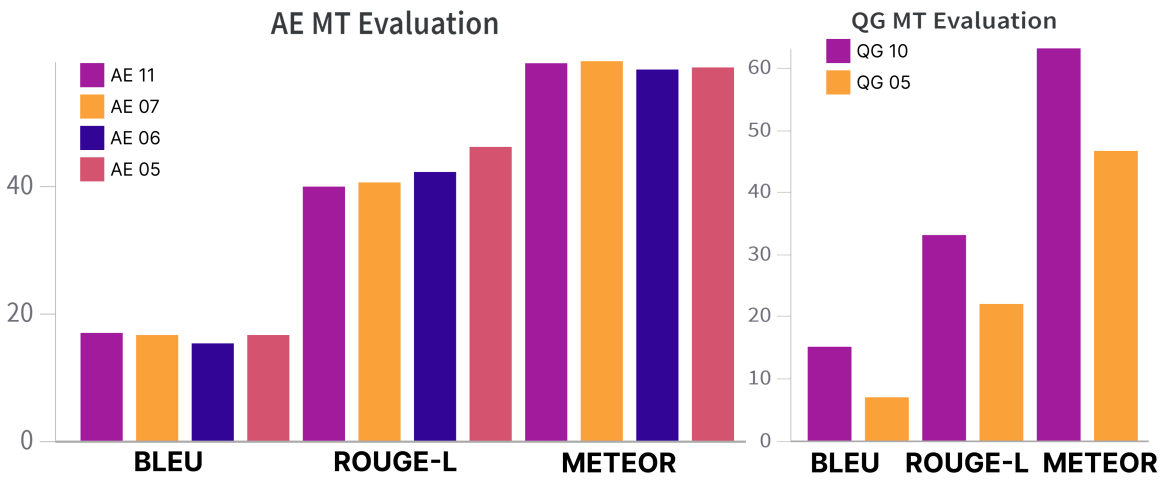

Figure 3.9: Final values for MT metrics across several later models

an incorrect fact about the context. Rather than Abdon, Jephthah ruled before Ibzan. Likewise, the second context assumes Jesus is speaking when in fact the author is Paul.

Reference: Judges 12:8
After him, Ibzan of Bethlehem judged Israel.
Question: According to Judges 12:8, who judged Israel after Abdon of Ephraim?
Answer: Ibzan of Bethlehem
Question: According to Judges 12:8, who judged Israel after Abdon?
Answer: Ibzan

Reference: Philemon 1:12
I am sending him back. You therefore receive him, that is, my own heart,
Question: According to Philemon 1:12, what was Jesus telling Peter to do?
Answer: Receive him, that is, my own heart
Question: According to Philemon 1:12, what does Jesus tell the Samaritan etc...

Text 3.11: Poor **7b-chatQG05** model output due to very short contexts

In an attempt to solve this issue, we trained one of the next models, **7b-chatQG07**, using the `paragraph_sentence` column from the dataset rather than the `sentence` column used thus far (see Figure 3.4).[7] After fine-tuning, we tested

[7]The `paragraph_sentence` column contains the verse prior and after the target verse, unless

this model on the same verses as we had tested **7b-chatQG05**. As shown in Text 3.12, providing the question generator with additional context allowed it to use real, rather than fabricated, facts in the question body.

Reference: Judges 12:8
After him, Ibzan of Bethlehem judged Israel.
Question: According to Judges 12:8, who judged Israel after Jephthah?
Answer: Ibzan of Bethlehem
Question: According to Judges 12:8, who was the next judge of Israel after Jephthah?
Answer: Ibzan

Reference: Philemon 1:12
I am sending him back. You therefore receive him, that is, my own heart,
Question: According to Philemon 1:12, what does Paul say about Onesimus?
Answer: I am sending him back
Question: According to Philemon 1:12, what does Paul say about Onesimus?
Answer: He is being sent back

Reference: Joshua 18:14
Then the border extended around the west side to the south, from the hill that lies before Beth Horon southward; and it ended at Kirjath Baal (which is Kirjath Jearim), a city of the children of Judah. This was the west side.
Question: According to Joshua 18:14, the border of the tribe of Ephraim extended around the west side to the south, from the hill that lies before what place?
Answer: (1) Beth Horon (2) Kirjath Baal
Question: According to Joshua 18:14, the border of the tribe of Ephraim etc...

Text 3.12: Additional context improved QG for very short contexts but reduced factual correctness for longer contexts

However, the model also began to fabricate false facts about larger contexts. For example, in Text 3.12 the context from Joshua makes no mention of a tribe of Ephraim. In fact, such a concept does not appear in that whole chapter at all. **7b-chatQG07** overall seemed more creative which is not a desirable attribute for its target application.

the verse is the first or last in a chapter.

Fortunately, we found a compromise that incorporated the best of both methods. Adding additional context (excluding highlight tokens) to the **7b-chatQG05** model for contexts shorter than 15 words produced the same improved results as **7b-chatQG07**. Therefore, future models were trained without the additional contexts to keep them focused on the context to question transformation, but during inference, contexts shorter than 15 words were expanded for QG.[8]

While these changes improved the model with respect to certain types of contexts, it still struggled not to assume incorrect information in certain cases. There are sections of the Bible which appear nearly identical. For example, the author Paul wrote many letters to churches, and most of these letters began with some variation on "Paul, an apostle of Jesus Christ by the will of God, etc...". Because the verse reference in the form of "According to Colossians 1:1, ..." was only prepended to the question after generation, the question generator sometimes assumed the wrong book and asked questions about the Galatians based on a context from Colossians. Futhermore, much of the rest of Paul's writing does not contain enough context in a single verse to let the model differentiate between books. This was especially an issue when other, less prolific authors sounded like Paul. For instance, Text 3.13 clearly shows the **7b-chatQG08**[9] model mistaking John's writing for Paul's. This issue was also observed within prophetic writings, and proverbs and psalms.

Introducing the verse reference into the training examples ran the risk of the model learning to rely on its knowledge of the provided reference. Nevertheless, our next experiment included verse references in the training data. In order to

[8] A fortunate effect of our pipelined QAG method is that while we may show additional verses to the QG model, the question and answer will still focus on the target verse because the AE model is only shown the target verse.

[9] This version was identical to the **07** version except that it was trained on a nearly 7000 more examples that was received later in the project.

Reference: 3 John 1:14
but I hope to see you shortly, and we shall speak face to face. Peace to you. Our friends greet you. Greet the friends by name.
Question: According to 3 John 1:14, what does Paul wish on the believers?
Answer: Peace
Question: According to 3 John 1:14, when does Paul hope to see the believers?
Answer: shortly
etc...

Text 3.13: The QG model could mistake one book for another written in a similar style of writing

retain consistent formatting, we did not modify `pbeClean.py` (Listing 3.2), which removes the references from the questions. Rather, we prepended the references in the `DataProcessor`'s `pbeContextualize` function. We then fine-tuned a new QG adapter with this data and adjusted the inference.

Fortunately, given the same context but this time with the reference, the **7b-chatQG10**[10] model no longer made the mistake of assuming the incorrect author or context. Most importantly, including the context's reference in the prompt did not cause the model to primarily rely on its exterior knowledge. Evidently, the training data had performed its purpose well. Although in Text 3.14 the QG model does use its broader knowledge to recall the letter's recipient (Gaius) from much earlier in the chapter, it does so perfectly. While this behavior is not necessary, it is helpful when done correctly and sparingly.

Given the recent improvements in QG models and the time limit for this thesis, this model was the final one to be trained and was used in the evaluation of this thesis. AE models did not improve much after the **06** version. Some experiments were performed with AE data that was filtered using the `DataProcessor`'s `aeFilter` function. While these models were more careful about producing long and multi-

[10]Due to programming errors, the data used to train the **7b-chatQG09** model was accidentally missing the references during training, so this model was ignored.

Reference: 3 John 1:14
but I hope to see you shortly, and we shall speak face to face. Peace to you. Our friends greet you. Greet the friends by name.
Question: According to 3 John 1:14, how are the friends to be greeted?
Answer: By name
Question: According to 3 John 1:14, what does John say to the beloved Gaius?
Answer: (1) Peace to you (2) Our friends greet you (3) Greet the friends by name etc...

Text 3.14: Introducing context references completely solved the issue with the model assuming the incorrect context

point answers, they underproduced multi-point questions and sometimes lacked diversity. Because question volume was more important, very little was changed in the training of the last model except that it was also trained on the latest data received later in the project. The final models used in the evaluation of this thesis were called **7b-chatAE11** and **7b-chatQG10**.

3.6 Hyperparameter Optimization

Achieving the best model often involves many experiments with different hyperparameters. For example, Goyal et al.'s QAG system [23] is trained at a stable learning rate of $1x10^{-4}$ for 18 epochs. Zhang et al.'s [20] model is instead trained with a decaying learning rate which begins at $1.905x10^{-3}$ and is only trained for 3 epochs. However, both works produce systems that score high (41 and 51 in ROUGE-L against SQuAD, respectively) in automatic metrics. Thus, even for the same base model, optimal training strategies may differ from project to project.

One common approach to selecting hyperparameters for a particular project is to perform an automated hyperparameter sweep [20, 29]. We rely on HuggingFace's integration with wandb to perform hyperparameter sweeps.

Starting a sweep with the Python `wandb` package is quite simple. There are

two main components: a sweep configuration and running sweep agents with the provided `sweepID`. Listing 3.17 shows the sweep configuration from the `sweep.yml` file in our repository.

Lines 1 to 3 determine which project and team wandb should report the logging information to. The `method` key determines how the sweep agent will choose the next hyperparameters for training a new model. The `grid` method systematically trains a model with every possible combination of hyperparameter options. The other options for sweep method are `random` and `bayesian`. A random selection of hyperparameters provides no advantage when we can specify which parameters to choose from. A Bayesian sweep infers what hyperparameters should be attempted based on results from previous choices. While a Bayesian sweep sounds the most logical, it is fundamentally difficult to parallelize and scales poorly to many hyperparameters. Because we had the option of parallelizing our sweeps, and we have many hyperparameters with few, discreet options, we chose to sweep with the grid method.

The `metric` key refers to the name of the metric that our training script reports which should be used as the ranking of which group of hyperparameters produces the best model. For our training, we use the loss on evaluation data, the same metric shown in Figures 3.6 and 3.7.

Several of the hyperparameters such as learning rate, LoRA dropout, and bias were explained previously. The new hyperparameters are explained below:

- The rank `r` determines the size of the matrices in the adapter.
- α is a scaling factor that controls how important the fine-tuning is compared to the base model's current weights. Higher values amplify both the good and

```yaml
project: sweep
entity: matousac
name: sweep
method: grid
iterations: 36
metric:
  name: eval_loss
  goal: minimize
parameters:
  learningRate:
    values: [0.0001, 0.01, 0.1]
  r:
    values: [32, 64, 128]
  loraAlpha:
    values: [32, 64, 128, 256]
  loraDropout:
    values: [0.01, 0.1]
  bias:
    values: ['none', 'lora_only', 'all']
  quality:
    values: [5, 7, 8, 9]
  loraLayers:
    values: ['qv', 'qvk', 'qvko']
```

Listing 3.17: Adding custom tokens to the tokenizer and then resizing the model

the bad characteristics of fine-tuning data. LoRA α and r are fundamentally connected because the actual weight scaling value s is calculated as $s = \frac{\alpha}{r}$.

- `quality` refers to the data quality threshold for training. At 5 any data ranked at quality 5 and above is included in training. Thus, as the quality threshold increases, fewer, higher-quality training examples are included in training.

- Finally, the `loraLayers` parameter controls which layers of the adapter are fine-tuned. The characters "q", "v", "k", and "o" stand for the query, value, key, and object projection layers, respectively.

Note that we do not test different values for the "epoch" training argument. This is because our research showed that training for multiple epochs did not significantly improve performance. In fact, the publishers of the LLaMA 2 family of models report that training for multiple epochs leads to over-fitting, impairing performance [49].

Starting a wandb hyperparameter sweep is simple. As Listing 3.18 shows, the `Trainer`'s `sweep` function simply sets an internal flag, reads the configuration from the `yaml` file, and calls `wandb.sweep`. This provides a `sweepID` which can then be passed to one or more agents that start threads executing the function passed in. Because we have a custom training function, we supply the `Trainer`'s `train` function. It is also useful to specify the number of iterations the agent should perform so the sweep will complete in a predetermined amount of steps.

```
1 def sweep(self):
2   self.sweeping = True
3   config = yaml.safe_load(Path(f'sweep.yml').read_text())
4   sweepID = wandb.sweep(config, project = config['project'])
5   wandb.agent(sweepId, self.train, count = config['iterations'])
```

Listing 3.18: Reading a sweep configuration and beginning a sweep with wandb

Because a grid sweep tests every combination of hyperparameters provided, the amount of time a sweep takes grows multiplicatively. This becomes an issue when training each QG model already takes approximately eighteen hours. With the current configuration in Listing 3.17 the estimated runtime would be over five years. The first step to reduce sweep time is to limit the number of steps a model can be trained for. We determined to limit the model training to 200 steps so that the data and current configuration would have enough influence on the model to give a good estimate of evaluation loss. This reduces training time to 28 days.

While a great improvement, even this is unrealistic. Luckily, not all hyper-

parameters need to be part of the same sweep. For example, the data quality and LoRA layers trained have little effect on the optimal learning rate and the LoRA α. Because adding runs from two sets of options yields far fewer runs than multiplying all the options, we divided the hyperparameters into two sweeps. The first four parameters, learning rate, r, α, and dropout were part for the first sweep while bias, quality and layers were part of the second. This reduced the sweep time to four and three days respectively (the second sweep was allowed up to 300 training steps).

The first sweep's results are visualized in Figure 3.10. The best models have the lowest loss values and are colored the darkest blue. It is immediately obvious that the best models have low learning rates. Furthermore, it may initially appear that the LoRA α and r have little effect on model performance. Dark blue models *do* pass through each option for the two parameters. However, when observing the exact loss values, the models with alpha values larger than the rank were slightly more successful. Overall, the best model achieved an evaluation loss of 0.8356 and had the following settings:

- LoRA dropout: 0.01
- learning rate: 0.0001
- α: 256
- r: 64

Figure 3.11 visualizes the second sweep's results. The models achieved much more varied loss values than the first sweep's models. Models trained with more layers achieved lower loss values. On the other hand, LoRA bias made no difference whatsoever to model loss. Finally, data quality made a significant impact. Higher

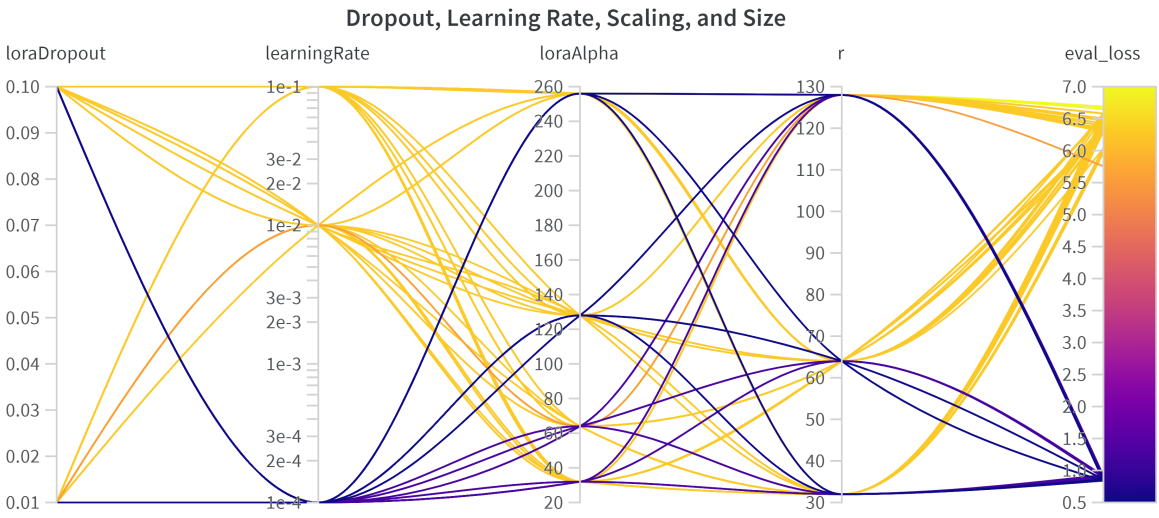

Figure 3.10: Hyperparameter sweep of LoRA dropout, learning rate, LoRA alpha scaling factor, and LoRA rank *r*

quality data improves model performance. However, it is difficult to judge whether this is only because the model has better training examples or also due to the reduction of data variety.

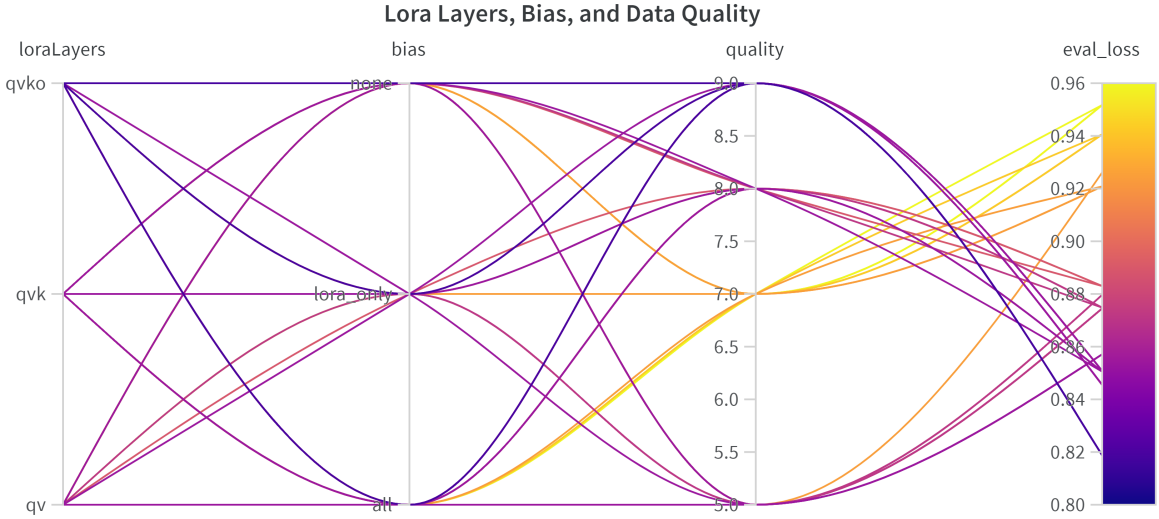

Figure 3.11: Hyperparameter sweep of LoRA layer settings, LoRA bias, and data quality

The best settings from both hyperparameter sweeps were used for fine-tuning the last the AE qnd QG adapters, **7b-chatAE11** and **7b-chatQG10**.

Chapter 4

Evaluation Plan

As explained in the State of the Art (Section 2.2), QAG models elude formulaic and definitive evaluation by nature. This is primarily because automated metrics cannot adequately take into account the inherent creativity of NLG. Moreover, a single metric cannot be quickly adapted to varying model goals. Human evaluation, on the other hand, is subjective and time-consuming. Nevertheless, we utilize the measures of model success that have prevailed in recent QG research.

4.1 Automatic Evaluation

We have evaluated our final AE and QG adapters using the three automatic evaluation metrics most commonly seen in QG: BLEU-4, ROUGE-L, and METEOR. Because these metrics each require references, we have reserved 1,749 rows (3%) of our collected data for evaluation purposes.

We do not set specific automated metric "targets" here because these metrics are not necessarily good measures of model success. For instance, as shown in Table 2.1, the highest BLEU-4 metric reports only 27.21% similarity between candidate

and reference despite the presence of several capable QG models in the table.

Because we wish to evaluate the QAG system as a whole, we will compare a complete list of questions and answers produced by the generator to references. This list looks just like the outputs shown in Text 3.14 for example. While collecting the generator's output in this form involves trivial string manipulation, the references must also be aggregated for the comparison. For this processing, we use another `DataProcessor` function, `aggQAByContext` (Listing 4.1). This function uses the same procedure as `pbeContextualize` to get verse texts, but then uses the verse reference columns to aggregate multiple questions and answers into a single value. We use the newline as a delimiter.

```
def aggQAByContext(self):
  df = pd.read_csv(self.source)
  # contextualize DataFrame but keep reference columns
  df['qa'] = 'Question: According to ' + df['question'] + '\nAnswer: ' + df['answer']
  df['count'] = 1
  grouped = df.groupby(['book', 'chapter', 'verse', 'endVerse']).agg({
    'qa': lambda x: '\n'.join(x),
    'count': 'sum'
  }).reset_index()
  # save file ...
```

Listing 4.1: This function aggregates questions and answers by context

For the actual calculation, we used HuggingFace's `evaluate` Python library. The `Generator`'s `autoEval` function (Listing 4.2) handles the automatic evaluation. First, it divides the dataset into five partitions within a `DatasetDict` based on the number of questions and answers contained in each example. We do this because the reference data we had saved had far fewer questions per context than the system tended to generate. While the system usually generated between 5 and 10 questions, the aggregated reference data often only had 1 or 2 questions

```
def autoEval(self):
  data = load_dataset(self.paths['data'])['train']
  # split on counts
  dsDict = DatasetDict()
  for i in range(1,5):
    dsDict[str(i)] = data.filter(lambda row: row['count'] == i)
  dsDict['5+'] = data.filter(lambda row: row['count'] >= 5)

  for name, dataset in dsDict.items():
    verses = [self.constructVerse(row) for row in dataset]
    labels = dataset['qa']
    preds = []
    for i, v in enumerate(verses):
      df = self.generateQA(v)
      df['qa'] = 'Question: ' + df['question'] + '\nAnswer: ' + df['answer']
      preds.append(df['qa'].str.cat(sep = '\n'))
    metrics = self.calculateMTMetrics(preds = preds, labels = labels)
    # log metrics ...
```

Listing 4.2: autoEval() collects references, generates predictions, and calculates the MT metrics for five groups of datasets

per context. We hypothesized that this comparison would improve metric scores for metrics focusing on recall (ROUGE-L) because there is a high chance that the few questions in the reference will be contained within the many questions in the prediction. Inversely, this should decrease the scores for metrics that focus on precision (BLEU-4) because a long prediction will contain many things that the short reference will not contain. To see this difference in scores across reference counts, we calculated separate metric scores for references with 1, 2, 3, 4, and 5 or more questions and answers.

The rest of the `autoEval` function handles generation using `generateQA` and then calculates the metrics using `calculateMTMetrics`. The code for this function is virtually identical to 3.16, which was used to automatically evaluate models as they were training. Finally, the results are logged to a file that we will discuss in the next chapter.

4.2 Manual Evaluation

We also perform a human evaluation. Unfortunately, manual evaluation metrics are even less standardized in QG than automatic ones. The most common metrics include some measure of overall question acceptability [9, 4, 10, 13, 20] and grammaticality [14, 10, 15, 3, 8, 29]. Other metrics are often included to measure a project's specific goals [10, 21].

Thus, for our manual evaluation, we measured overall question acceptability and grammaticality. Grammaticality reports the average score for a question and answer pair's grammatical structure. Acceptability measures the average score for how useful the output is for the Bible memorization tournament case study. These metrics are judged on a five-point scale by domain experts (PBE leaders and participants) with no direct involvement in this thesis. The five-point scale is chosen due to its prevalence in other QG literature [14, 15, 3, 29].

In order to help standardize the scale, we described what each rating for grammaticality and acceptability meant in Texts 4.1 and 4.2. However, we also left room for personal interpretation because we wanted evaluators to think about whether the generated questions were useful to them personally.

Grammaticality: Does the question/answer have good grammatical structure, and does it make sense? Rate from 1-5. Here is basically what each rating would mean.
1: The question/answer makes no sense and includes non-English characters.
2: The question/answer makes no sense but is all in English.
3: The question/answer has mostly grammatically correct phrases, but is disconnected and it is difficult to understand what it should be asking.
4: The question/answer is very close to coherent. I would only need to change a word or two or add a punctuation mark somewhere.
5: The question/answer is grammatically correct and easy to understand.

Text 4.1: The grammaticality scale as described to evaluators

Each evaluator was given a CSV file of approximately 60 questions and answers.

Acceptability: How useful is the question and answer as a PBE practice question? Also rated from 1-5.
1: The question makes little to no sense. The answer is not correct. I can't use this.
2: The question makes some sense but gets certain facts or connections wrong. The answer is not correct. I can't use this.
3: I understand what the question is asking, and the answer is at least partially correct. However, I'd rather discard this question than take the time to try to fix it.
4: The answer is correct for the given question, and I would only need to make minor adjustments to the pair before using it in PBE. Minor changes might include changing a few words, removing a part of the answer, or changing the number of points the question is worth. Overall, I'd rather fix this question than discard it.
5: The question and answer are ready to use as they are. I would make only very minor changes (adding a missing quote character for example) or no changes at all before using this question for practice with PBE participants.

Text 4.2: The acceptability scale as described to evaluators

```
def fileGen(self):
  for fileNum in range(11):
    cols = ['reference', 'additionalContext', 'verse', 'question', 'answer']
    qa = pd.DataFrame(columns = cols)
    while len(qa) < 60:
      verse = self.dp.getRandomVerse()
      currQA = self.gen(verse = verse)
      currQA['reference'] = verse.ref
      currQA['additionalContext'] = verse.inContext
      currQA['verse'] = verse.text
      currQA = currQA[cols]
      qa = pd.concat([qa, currQA])
    qa.to_csv(<filePath>, index=False)
```

Listing 4.3: fileGen() generates questions and answers based on random contexts

To generate these files, we used the `Generator`'s `fileGen` function. We initially had 11 individuals willing to evaluate, so we generated 11 files. The question generation loop simply selects a random verse from the Bible and then generates a list of questions and answers that is appended to a `DataFrame` for the current file. Once the `DataFrame` had over 60 questions, the results were written to a file. Each file contains different random questions in order to maximize the diversity of the contexts tested.

These ratings are then used to report average scores in the next chapter. We also

asked each evaluator to report the time spent evaluating their questions. These reports were then used to calculate the time saved by the QAG system.

To estimate the time saved by our system when compared to manual question authoring we define Equation 4.1. The amount of time taken to write questions by hand is represented by $T_{original}$. This is the baseline for T_{saved} because if the QAG system is thorough enough, it should have the potential to save all human effort spent question writing.

However, because some basic filtering is required to remove inadequate generations, we subtract $T_{filtering}$ which represents the human time taken to filter out useful from non-useful questions. The "per question" times of $T_{original}$ and $T_{filtering}$ are measured empirically.

Furthermore, we take into account the amount of time taken to fix nearly acceptable questions - that is, questions ranked $\frac{4}{5}$ acceptable. This value is obtained by multiplying the percentage of questions ranked as nearly acceptable ($Q_{nearly\ acceptable\ \%}$) by the time taken to fix them, assumed to be a third of the average question authoring time. We believe this estimate to be quite conservative because the majority of writing the question has already been performed for questions ranked as 4. Changing the point values for an answer or adding/removing a few words is much easier than identifying information in the context worthy of asking a question about, and then typing out a question targeting that knowledge while still including enough context. This time estimate is removed from the potential time saved. Notice that all less-acceptable questions (questions ranked lower than 4 in acceptability) are not considered useful for saving time.

If the model fails to generate enough questions, a human may have to write some by hand, further reducing the time saved. We take this into account by subtracting the term $T_{original} \times Q_{human}$ which represents the time a human may

have to spend authoring questions by hand if the generator does not produce enough questions on its own. Q_{human} refers to the number of questions written by a human. Note that Q_{human} can be zero if for each context the generator produces as many questions as or more questions than the average human author.

$$T_{saved} = T_{original} - T_{filtering} - \left(Q_{nearly\ acceptable\ \%} \times \frac{1}{3} T_{original} \right) - T_{original} \times Q_{human} \tag{4.1}$$

In our results, we estimate the total time for writing questions $T_{original}$ for the next year of PBE based on the book of Romans. Also, notice that we do not incorporate the time it takes the generator to generate the questions as this takes no human effort.

Chapter 5

Results

In this chapter, we present the comprehensive results of our evaluation of the performance of our QAG system. We begin by discussing the outcomes of our automatic evaluation, in which we compute the most common machine translation metrics for the QAG field. We discuss our results and compare them with existing literature. Subsequently, we delve into the manual evaluation phase, where human evaluators assessed the grammaticality and acceptability of generated questions. This phase of the evaluation allows us to estimate the time our system saves end users.

5.1 Automatic Evaluation

We used the `Generator`'s `autoEval` function to calculate the automatic machine translation metrics BLEU-4, ROUGE-L, and METEOR. The results are shown in Table 5.1 and visualized in Figure 5.1.

As expected, the score for BLEU-4 steadily increased with the number of questions and answers in the reference. This is due to the importance BLEU places

QA Count	Sample Count	BLEU-4	ROUGE-L	METEOR
1	550	5.8584	25.3562	50.9099
2	176	10.8152	34.9804	60.5151
3	122	15.0181	41.8569	65.3382
4	67	18.8245	45.9443	66.3968
5+	37	23.5294	47.4699	66.2978

Table 5.1: Automated metric results

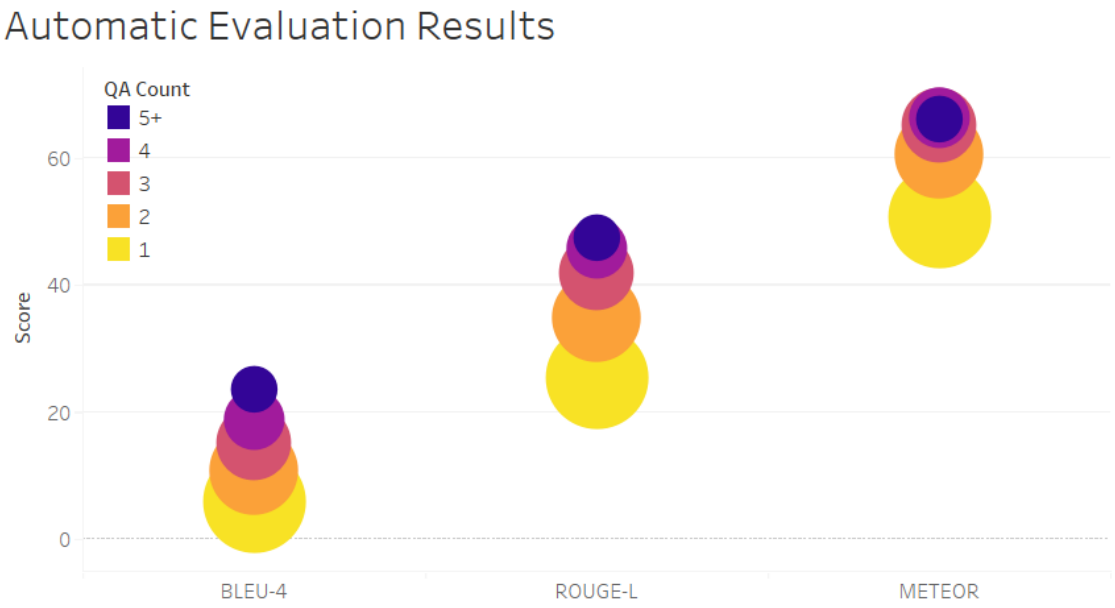

Figure 5.1: MT metric results as a Score vs. Metric graph. Color represents the different reference datasets separated by their question and answer count. Size represents dataset size.

on precision. Because the model generated many questions and answers, reference datasets with only one question and answer per context naturally had less text towards which the model could produce "precise" questions and answers.

When there were more questions and answers in the reference, the system's output quadrupled its BLEU-4 score. Because the references were human-written and diverse, this increase in score suggests that the model is also able to produce diverse questions. Fortunately, as we have seen in previous examples, the system

is able to produce diverse questions. This is primarily due to the way the pipeline QAG method translates the one-to-many problem of QAG into a one-to-one QG problem.

The ROUGE-L score also increased significantly with the question and answer count. This may be due to there being more n-grams to recall from the reference. METEOR also improved, though not as much as BLEU-4 or ROUGE-L.

Comparing our results to previous QAG research is not simple even though we have calculated all three most popular metrics. Although our BLEU-4 and ROUGE-L scores are comparable with, and our METEOR scores significantly surpass, the scores reported in Table 2.1, it would be naive to claim that our research has produced superior models. Not only did we train models for a slightly different type of QAG, but we also trained and evaluated the system using a different dataset than most other works, which commonly use SQuAD.

Nevertheless, we can draw some conclusions based on the intriguing difference in our BLEU-4, ROUGE-L and METEOR scores. While our BLEU-4 and ROUGE-L scores are not higher than the best reported to date, our METEOR score is nearly 20 points higher than Zhang et al.'s leading score of 49 [20]. Given that METEOR attempts to match word meanings and semantics, this suggests that our model is quite experienced in producing the same questions as a human would write though perhaps not with precisely the same exact words. This difference between metric scores is expected as the questions in the evaluation dataset were generally *not* written by the same authors as those in the training dataset. Thus, although the model could be said to have a different "writing style" from the references, the questions and answers produced largely matched those desired.

5.2 Manual Evaluation

Of the 11 evaluation files sent to evaluators, 5 were completed and returned.[1] The statistics reported below are based on all evaluations joined together with equal weight. These statistics revolve around the two metrics explained in Section 4.2: grammaticality and acceptability. In total, 341 questions were manually evaluated.

The average grammaticality score among these questions was 4.5, or 90%. This compares quite well to Ushio et al.'s 2.3 acceptability score [29] and Liu et al.'s 40% [8] "well formed" score. We must recognize that these ratings are fundamentally subjective, but the ratings seem sufficient for our purposes. Although the model does not always produce sentences with perfect grammatical structure, we never expected perfection.

Though not as high, our acceptability score also showed promise. At an average acceptability of 4.0 (80%), our system scored much higher than the best rule-based QAG systems. Among model-based QAG systems, only Zhang et al.' report an acceptability score. Their system yielded acceptable questions 92% of the time [20]. Liu et al. report a similar relevancy score [8] of 93.7%. Ushio et al. report a much lower answerability score of 2.8. We were unable to find any other manual evaluation metrics similar to acceptability. In light of those we do have, our system seems to rank near the better end of QAG systems, though we certainly have room for improvement.

One intriguing finding with respect to our manual metrics was that they varied considerably across different context types. We made this discovery by dividing the randomly selected contexts in our evaluation into five categories (see Appendix B.1 for the classifications). Then, we computed the separate acceptability and

[1] These files are available in the `data/evaluation/manual` folder on our GitHub repository.

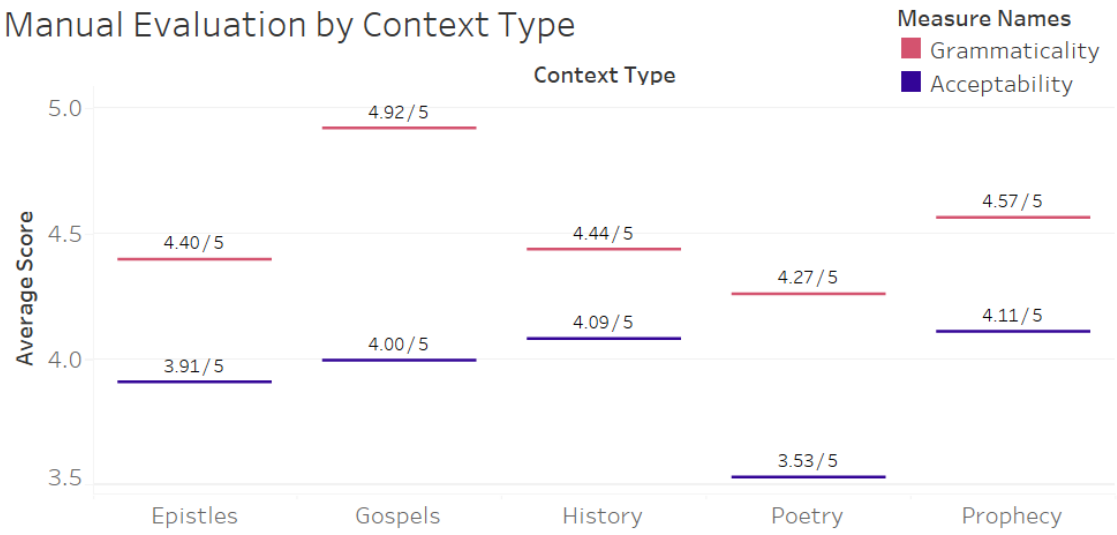

Figure 5.2: Acceptability and grammaticality per context type

grammaticality scores shown in Figure 5.2.

In Figure 5.2, there are two primary outliers. The grammaticality rating for gospels is nearly perfect. On the other hand, it appears the model struggles to generate acceptable questions taken from poetic contexts. This makes sense as there were no examples from these contexts in the models' training data. Another surprise is that the model generates the most acceptable questions for prophetic contexts. Normally, these contexts would be considered more difficult to write questions for.

As discussed in our evaluation plan, using the acceptability scores for the fully and nearly acceptable questions, we calculated the time saved by our method. The next year of the PBE tournament case study includes the books of Romans, 1 Corinthians and 2 Corinthians. Combined, these books contain 1,127 verses. Thus, we used this number to estimate $T_{original}$ from Equation 4.1.

To estimate $T_{original}$ for contexts that have not yet had questions written for them, we estimated the average human question authoring time. This was done

using question authoring experiences in the Old and New Testaments from two separate authors who provided data for this study. Authoring questions from more "straightforward" passages in the New Testament, such as the book of Luke tended to proceed at a rate of around to 40 questions per hour. The rate for more complex portions of books such as Daniel proceeded at a slower 25 questions per hour. Based on the relative amount of material processed overall, the question authors estimate 30 questions written per hour, or 2 minutes per question.

Our highest quality data came from Lisa Myaing. For 4,000 verses, there were over 15,200 questions and answers written, yielding approximately 4 questions per context. This forecasts approximately 4,250 questions for the contexts in Romans and Corinthians. At 2 minutes per question, $T_{original}$ represents approximately 141 hours of human labor.

Equation 4.1 also requires the percentage of questions and answers given specific acceptability ratings. In addition to the percentage of predictions given a rating of 4 and 5, we report percentages for the other ratings in Figure 5.3. Combined, questions were ranked at or above $\frac{4}{5}$ 74.2% of the time.

Because the QAG system produced 6.6 questions per context in our manual evaluation and 74.2% of those are good enough to be used, we can expect 4.9 useful questions per context. This is significantly higher than the 4 questions per context present in our data. Thus, we conclude that few to no questions will have to be written by humans and assign Q_{human} a value of 0.

Finally, to estimate $T_{filtering}$ we requested our evaluators to report how long it took them to evaluate the questions they received. For the 341 questions evaluated, a total of 190 minutes were spent evaluating them. This yields an average of 0.55 minutes per question. Scaled up by the 4,250 questions estimated to cover the example's context, $T_{filtering}$ represents approximately 39 hours.

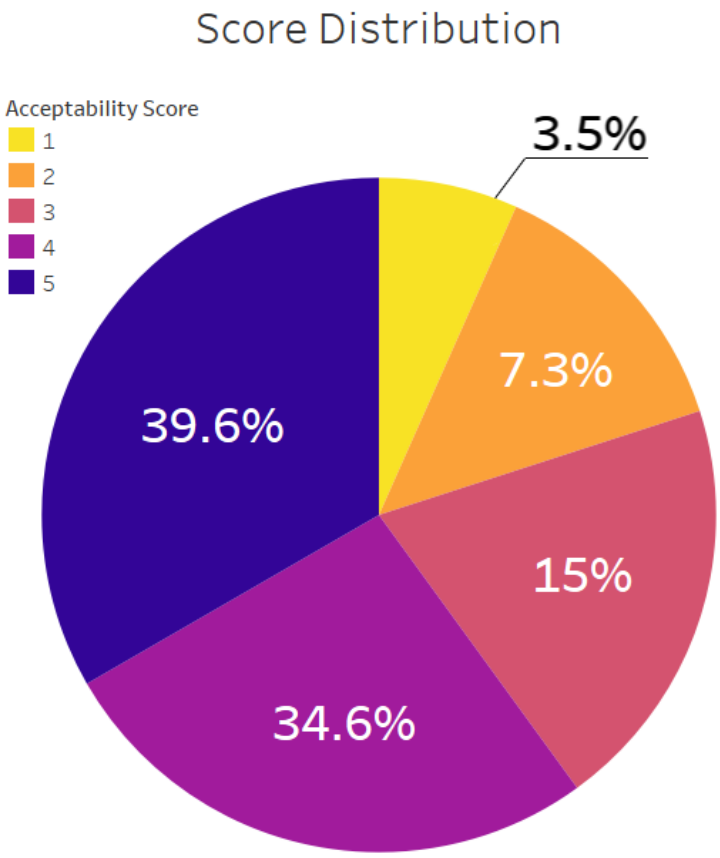

Figure 5.3: The manual evaluation score distribution

With all the input variables for Equation 4.1 assigned to values, we can calculate T_{saved}. As shown in Equation 5.1, the time saved for authoring questions is estimated at 85 hours for just one tournament for one question author. This accounts for 61% of the original time.

$$T_{saved} = 140 - 39 - \left(34.6\% \times \frac{1}{3} \times 140\right) - 140 \times 0 \approx 85 \tag{5.1}$$

Moreover, $T_{filtering}$ is perhaps an unreasonably high estimate. During the evaluation, the evaluators had to take the time to determine both grammaticality and acceptability on a 5-point scale. Some evaluators even took the time to explain why they rated certain questions lower than a 5 which is time already taken into account by $Q_{nearly\ acceptable\ \%} \times \frac{1}{3} T_{original}$. Normally, a user of our system would only need to choose whether to keep or discard a question, and occasionally would fix an imperfect question or answer. If we divide $T_{filtering}$ in half, T_{saved} becomes 104 hours, or 74.5% of the original question authoring time.

Chapter 6

Conclusion

6.1 Summary and Objectives

Transformers and large language models have shown promise for natural language processing tasks, but still face challenges in extractive question generation. To address this limitation, in this thesis we fine-tuned LLaMA-2-Chat-7b for the production of high-quality, human-oriented questions and answers.

Several concepts critical to QAG were presented, the most foundational being NLP, AI, ML, and Transformers. Also, several research works and tools for QG were explored. However, none satisfied the goals of this thesis, so we used the latest ideas and technology to construct a QAG system. Various resources were leveraged to achieve this goal, including the open-source LLaMA 2 base models, HuggingFace's LLM-oriented libraries available in Python, and public and private training data.

After training our models and constructing a simple QAG pipeline, we evaluated the system as a whole using the best standardized metrics available: BLEU-4, ROUGE-L, and METEOR. Furthermore, we performed a manual evaluation to

measure the model output grammaticality and acceptability. Not only are both sets of statistics promising, but future users that have seen the QAG system's output have largely praised the system. One simply stated "I'm impressed". Another user wrote: "The questions are good. ... [they will] save a question writer a significant amount of time!"

To determine this project's success overall, we refer to the objectives set in the introduction. We have successfully:

- Fine-tuned LLaMA 2 on QAG data
- Generated far more than two questions for each context
- Achieved an average question acceptability score of $\frac{4}{5}$
- Reduced the time necessary for question authoring
- Begun the process of making the model publicly available

Out of five objectives, four have been meet or surpassed. The fifth and final objective is still in progress because tournament administrators have taken an interest in the system and wish to use it as part of an official Question Generator. Given these results, we believe we have fulfilled the primary goal of this thesis.

6.2 Future Work

There are a myriad of ways the work done for this thesis could be expanded and refined. The data could be further cleaned by hand which could only serve to improve the QAG system. Larger base models could be fine-tuned or larger adapters could be developed. During our experiments, we did not change the maximum sequence length at all. Potentially decreasing it for the QG model could

increase model's speed without limiting its output, half of which is already being discarded. Despite the many different approaches that may slightly improve model performance, we have found two major pieces of future work.

One issue with our AE and QG pipeline method of QAG is that the answers cannot be easily adjusted to naturally answer the question. For example, consider that the QG model is provided the answer "writing his thesis" and it generates the question "Why didn't Ac get much sleep last night?". Although the question and answer have high potential to match, the answer should really read "because he was writing his thesis".

One way to mitigate this type of question and answer mismatch is to require the QG model to repeat the answer after the question and allow its knowledge of natural language and grammar to potentially improve the answer. Another approach would be to try end-to-end QAG. Training a model for end-to-end QAG is the most promising aspect of future work, as it would simply require a reformatting of data and slightly more generalized code in the `Generator`, but it has the potential to solve any issues introduced by pipelined QAG. The primary drawback to watch out for in end-to-end QAG would be lack of diversity and number of questions.

The other improvement to this thesis would be to test our system against the SQuAD dataset to facilitate a slightly different comparison of automated metrics to past research works. If we trained an end-to-end system using SQuAD, we could make much more direct comparisons between our scores and those in past research. This cannot properly be done with our current methodology as the pipelined system can not train capable AE models on SQuAD data. This type of comparison would more directly compare the capabilities of the base models with respect to QAG.

Given our current success, we will continue to improve our data and attempt to train better models for our use case. We hope to soon save even more time writing extractive, comprehensive questions and answers.

Appendix A

Supplementary Texts

In this conversation I want you to generate questions and answers based on the source text I give you. The questions should be extractive only. That is, the answer to any question must be directly found in the source text. Following is an example of how to format your response based on my input.

- Begin Example
Input:
The LORD spoke to Joshua the son of Nun, Moses' assistant, saying: "Moses My servant is dead. Now therefore, arise, go over this Jordan, you and all this people, to the land which I am giving to them–the children of Israel."
Response:
Q: Who is the LORD's servant and is dead?
A: Moses

Q: Who is the son of Nun?
A: Joshua

Q: Who is Joshua the son of?
A: Nun

Q: Who is Moses's assistant?
A: Joshua the son of Nun

Text A.1a: The standardized prompt given to models

Q: Where does the LORD tell ”you and all this people” to go? (2 pts)
A: 1. over this Jordan 2. to the land which I am giving to them

Q: What two things does the LORD tell Joshua to do? (2pts)
A: 1. arise 2. go over this Jordan
- End example

Try to ask questions in different ways. Ask not only ”What did this character do?” but also ”what character did this action?” Generate at least 7 questions for each source text.
The questions should use the same words as are in the source text so that the person answering would more easily remember the context surrounding the answer. Include some context from the source text in the question.
Also, if the answer to a question is any sort of a list, make the question multi-point and number the answers accordingly.
Notice that the answers to questions are short excerpts directly taken from the source text. They usually won't be full sentences. If one word answers the question, one word is all you need to put in your response. Make sure the answer is not too long.

Your first source text is:
Now it came to pass, in the days when the judges ruled, that there was a famine in the land. And a certain man of Bethlehem, Judah, went to dwell in the country of Moab, he and his wife and his two sons.

Text A.1b: The standardized prompt given to models continued

Next source text:
In the beginning was the Word, and the Word was with God, and the Word was God.

Text A.2a: The contexts provided to the models - John 1:1

Next source text:
He was in the beginning with God.

Text A.3a: John 1:2

Next source text:
All things were made through Him, and without Him nothing was made that was made.

Text A.4a: John 1:3

Next source text:
Blessed be the God and Father of our Lord Jesus Christ, who has blessed us with every spiritual blessing in the heavenly places in Christ,

Text A.4b: Ephesians 1:1

Appendix B

Additional Figures

Figure B.1: The division of contexts into five categories obtained from biblerr.com

Bibliography

[1] A. Gokul, "LLMs and AI: Understanding Its Reach and Impact," 2023, available at: https://doi.org/10.20944/preprints202305.0195.v1. 1.1

[2] Q. Zhou, N. Yang, F. Wei, C. Tan, H. Bao, and M. Zhou, "Neural Question Generation from Text: A Preliminary Study," in *Natural Language Processing and Chinese Computing: 6th CCF International Conference, NLPCC 2017, Dalian, China, November 8–12, 2017, Proceedings 6*. Springer, 2018, pp. 662–671, available at: https://doi.org/10.1007/978-3-319-73618-1_56. 1.1, 2.1.5, 2.1.12, 2.2.1, 2.2.1, 2.2.1, 3.2, 3.5

[3] X. Du, J. Shao, and C. Cardie, "Learning to Ask: Neural Question Generation for Reading Comprehension," *arXiv preprint arXiv:1705.00106*, 2017, available at: https://doi.org/10.48550/arXiv.1705.00106. 1.1, 2.1.5, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 3.2, 4.2

[4] P. Panchal, J. Thakkar, V. Pillai, and S. Patil, "Automatic Question Generation and Evaluation," *Journal of University of Shanghai for Science and Technology*, vol. 23, pp. 751–761, 2021, available at: https://www.doi.org/10.51201/JUSST/21/05203. 1.1, 2.2.1, 2.2.1, 4.2

[5] A. Kumar, D. Singh, A. Kharadi, and M. Kumari, "Automation of Question-Answer Generation," in *2021 Fourth International Conference on Computational*

Intelligence and Communication Technologies (CCICT), 2021, pp. 175–180, available at: https://doi.org/10.1109/CCICT53244.2021.00043. 1.1, 2.1.8, 2.2.1, 2.2.1, 2.2.1

[6] I. V. Serban, A. García-Durán, C. Gulcehre, S. Ahn, S. Chandar, A. Courville, and Y. Bengio, "Generating Factoid Questions With Recurrent Neural Networks: The 30M Factoid Question-Answer Corpus," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 588–598, available at: http://dx.doi.org/10.18653/v1/P16-1056. [Online]. Available: https://aclanthology.org/P16-1056 1.1, 2.1.8, 2.1.12, 2.2.1, 2.2.1, 2.2.1, 2.3

[7] X. Yuan, T. Wang, C. Gulcehre, A. Sordoni, P. Bachman, S. Zhang, S. Subramanian, and A. Trischler, "Machine Comprehension by Text-to-Text Neural Question Generation," in *Proceedings of the 2nd Workshop on Representation Learning for NLP*. Vancouver, Canada: Association for Computational Linguistics, Aug. 2017, pp. 15–25, available at: http://dx.doi.org/10.18653/v1/W17-2603. [Online]. Available: https://aclanthology.org/W17-2603 1.1, 2.1.12, 2.2.1

[8] B. Liu, H. Wei, D. Niu, H. Chen, and Y. He, "Asking Questions the Human Way: Scalable Question-Answer Generation from Text Corpus," in *Proceedings of The Web Conference 2020*, 2020, pp. 2032–2043, available at: https://doi.org/10.1145/3366423.3380270. 1.1, 2.1.12, 2.2.1, 2.2.1, 2.2.1, 2.2.1, 2.3, 4.2, 5.2

[9] J. H. Wolfe, "Automatic Question Generation from Text - An Aid to Independent Study," in *Proceedings of the ACM SIGCSE-SIGCUE technical sym-*

posium on Computer science and education, 1976, pp. 104–112, available at: https://www.doi.org/10.1145/800107.803459. 1.1, 2.2, 2.2.1, 2.2.2, 4.2

[10] M. Liu, R. A. Calvo, and V. Rus, "Automatic Question Generation for Literature Review Writing Support," in *Intelligent Tutoring Systems: 10th International Conference, ITS 2010, Pittsburgh, PA, USA, June 14-18, 2010, Proceedings, Part I 10*. Springer, 2010, pp. 45–54, available at: https://doi.org/10.1007/978-3-642-13388-6_9. 1.1, 2.2.1, 2.2.1, 2.3, 4.2

[11] M. Heilman, "Automatic Factual Question Generation from Text," Ph.D. dissertation, Carnegie Mellon University, 2011, available at: https://search.proquest.com/openview/aa3e8452038f2d7b68792e0809b9d56b/1?pq-origsite=gscholar&cbl=18750. 1.1, 2.2.1, 2.2.1, 2.2.1, 2.2.1

[12] P. Pabitha, M. Mohana, S. Suganthi, and B. Sivanandhini, "Automatic Question Generation System," in *2014 International Conference on Recent Trends in Information Technology*. IEEE, 2014, pp. 1–5, available at: https://doi.org/10.1109/ICRTIT.2014.6996216. 1.1, 2.2.1, 2.2.1

[13] K. Mazidi and P. Tarau, "Automatic Question Generation: From NLU to NLG," in *Intelligent Tutoring Systems: 13th International Conference, ITS 2016, Zagreb, Croatia, June 7-10, 2016. Proceedings 13*. Springer, 2016, pp. 23–33, available at: https://doi.org/10.1007/978-3-319-39583-8_3. 1.1, 2.2.1, 2.2.1, 4.2

[14] P. Khullar, K. Rachna, M. Hase, and M. Shrivastava, "Automatic Question Generation using Relative Pronouns and Adverbs," in *Proceedings of ACL 2018, Student Research Workshop*, 2018, pp. 153–158, available at: https://www.doi.org/10.18653/V1/P18-3022. 1.1, 2.2.1, 2.2.1, 4.2

[15] O. Keklik, T. Tuglular, and S. Tekir, "Rule-Based Automatic Question Generation Using Semantic Role Labeling," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 7, pp. 1362–1373, 2019, available at: https://doi.org/10.1587/transinf.2018EDP7199. 1.1, 2.1.5, 2.2.1, 2.2.1, 2.3, 4.2

[16] K. Kriangchaivech and A. Wangperawong, "Question Generation by Transformers," *arXiv preprint arXiv:1909.05017*, 2019, available at: https://doi.org/10.48550/arXiv.1909.05017. 1.1, 2.1.5, 2.1.12, 2.2, 2.2.1, 2.2.1, 2.2.2, 3.2, 3.5

[17] Y. Kim, H. Lee, J. Shin, and K. Jung, "Improving Neural Question Generation Using Answer Separation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 6602–6609, available at: https://doi.org/10.1609/aaai.v33i01.33016602. 1.1, 2.1.5, 2.2.1, 3.2

[18] L. E. Lopez, D. K. Cruz, J. C. B. Cruz, and C. Cheng, "Simplifying Paragraph-Level Question Generation via Transformer Language Models," in *PRICAI 2021: Trends in Artificial Intelligence*, D. N. Pham, T. Theeramunkong, G. Governatori, and F. Liu, Eds. Cham: Springer International Publishing, 2021, pp. 323–334, available at: https://doi.org/10.1007/978-3-030-89363-7_25. 1.1, 2.1.5, 2.1.12, 2.2.1, 2.2.1, 3.2, 3.5

[19] K. Grover, K. Kaur, K. Tiwari, Rupali, and P. Kumar, "Deep Learning Based Question Generation Using T5 Transformer," in *Advanced Computing: 10th International Conference, IACC 2020, Panaji, Goa, India, December 5–6, 2020, Revised Selected Papers, Part I 10*. Springer, 2021, pp. 243–255, available at: https://www.doi.org/10.1007/978-981-16-0401-0_18. 1.1, 2.1.5, 2.2.1, 2.2.1, 3.2, 3.5

[20] C. Zhang, H. Zhang, Y. Sun, and J. Wang, "Downstream Transformer Generation of Question-Answer Pairs with Preprocessing and Postprocessing Pipelines," in *Proceedings of the 22nd ACM Symposium on Document Engineering*, 2022, pp. 1–8, available at: https://doi.org/10.1145/3558100.3563846. 1.1, 2.1.5, 2.1.12, 2.2.1, 2.3, 3.2, 3.5, 3.6, 4.2, 5.1, 5.2

[21] H. A. Nguyen, S. Bhat, S. Moore, N. Bier, and J. Stamper, "Towards Generalized Methods for Automatic Question Generation in Educational Domains," in *European conference on technology enhanced learning*. Springer, 2022, pp. 272–284, available at: https://doi.org/10.1007/978-3-031-16290-9_20. 1.1, 2.1.5, 2.2.1, 2.2.1, 2.3, 3.2, 4.2

[22] A. Ushio, F. Alva-Manchego, and J. Camacho-Collados, "A Practical Toolkit for Multilingual Question and Answer Generation," *arXiv preprint arXiv:2305.17416*, 2023, available at: https://doi.org/10.48550/arXiv.2305.17416. 1.1, 1.2, 2.1.8, 2.1.12, 2.2.1, 2.2.1, 2.2.2, 3.2, 3.3

[23] R. Goyal, P. Kumar, and V. Singh, "Automated Question and Answer Generation from Texts using Text-to-Text Transformers," *Arabian Journal for Science and Engineering*, pp. 1–15, 2023, available at: https://doi.org/10.1007/s13369-023-07840-7. 1.1, 2.1.5, 2.1.8, 2.1.12, 2.2.1, 2.2.1, 2.2.1, 2.3, 3.2, 3.6

[24] M.-H. Hwang, J. Shin, H. Seo, J.-S. Im, H. Cho, and C.-K. Lee, "Ensemble-NQG-T5: Ensemble Neural Question Generation Model Based on Text-to-Text Transfer Transformer," *Applied Sciences*, vol. 13, no. 2, p. 903, 2023, available at: https://doi.org/10.3390/app13020903. 1.1, 2.1.8, 2.2.1, 2.2.1

[25] K. Chowdhary and K. Chowdhary, "Natural Language Processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020, available at: https://doi.org/10.1007/978-81-322-3972-7_19. 2.1.1

[26] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach (Volume I)*. Elsevier, 2014, vol. 1, available at: https://books.google.com/books?hl=en&id=Aw2jBQAAQBAJ. 2.1.2

[27] Z.-H. Zhou, *Machine Learning*. Springer Nature, 2021, available at: https://books.google.com/books?hl=en&lr=&id=ctM-EAAAQBAJ&oi=fnd&pg=PR6&dq=machine+learning&ots=oZRmW8Xr_u&sig=AEotOqjqV_3MF4Gp1cPrW_m3hUE. 2.1.3, 2.2.1

[28] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination press San Francisco, CA, USA, 2015, vol. 25, available at: https://www.ise.ncsu.edu/fuzzy-neural/wp-content/uploads/sites/9/2022/08/neuralnetworksanddeeplearning.pdf. 2.1.4, 2.2, 2.3

[29] A. Ushio, F. Alva-Manchego, and J. Camacho-Collados, "Generative Language Models for Paragraph-Level Question Generation," *arXiv preprint arXiv:2210.03992*, 2022, available at: https://www.doi.org/10.48550/arXiv.2210.03992. 2.1.5, 2.1.8, 2.1.12, 2.2.1, 2.2.1, 3.2, 3.5, 3.6, 4.2, 5.2

[30] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," *arXiv preprint arXiv:1606.05250*, 2016, available at: https://doi.org/10.48550/arXiv.1606.05250. 2.1.5, 2.2.1

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach,

R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017, available at: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 2.1.6, 2.2, 2.2.1, 2.2.1

[32] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," *arXiv preprint arXiv:2106.09685*, 2021, available at: https://doi.org/10.48550/arXiv.2106.09685. 2.1.7, 3.4.1

[33] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45, available at: http://dx.doi.org/10.18653/v1/2020.emnlp-demos.6. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6 2.1.8, 2.2.1, 3.4.1

[34] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318, available at: https://www.doi.org/10.3115/1073083.1073135. 2.1.11, 2.2.1

[35] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81, available at: https://aclanthology.org/W04-1013. 2.1.11, 2.2.1

[36] P. J. Finn, "A Question Writing Algorithm," Ph.D. dissertation, University of Chicago, Department of Education, 1973, not available. 2.2

[37] M. Last and G. Danon, "Automatic question generation," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 10, no. 6, p. e1382, 2020, avaliable at: https://doi.org/10.1002/widm.1382. 2.2.1, 2.2.1

[38] S. Narayan, G. Simoes, J. Ma, H. Craighead, and R. Mcdonald, "QURIOUS: Question Generation Pretraining for Text Generation," *arXiv preprint arXiv:2004.11026*, 2020, available at: https://doi.org/10.48550/arXiv.2004.11026. 2.2.1

[39] W. G. Lehnert, "A conceptual theory of question answering," in *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, 1977, pp. 158–164, available at: https://dl.acm.org/doi/abs/10.5555/1624435.1624467. 2.2.1

[40] N. Mulla and P. Gharpure, "Automatic question generation: a review of methodologies, datasets, evaluation metrics, and applications," *Progress in Artificial Intelligence*, vol. 12, no. 1, pp. 1–32, 2023, available at: https://doi.org/10.1007/s13748-023-00295-9. 2.2.1

[41] V. Mavi, A. Jangra, and A. Jatowt, "A Survey on Multi-hop Question Answering and Generation," *arXiv preprint arXiv:2204.09140*, 2022, available at: https://doi.org/10.48550/arXiv.2204.09140. 2.2.1

[42] F. Sebastiani and A. Esuli, "SentiWordNet: A Publicly Available Lexical Resource for Opinion Mining," in *Proceedings of the 5th international conference on language resources and evaluation*. European Language Re-

sources Association (ELRA) Genoa, Italy, 2006, pp. 417–422, available at: http://www.lrec-conf.org/proceedings/lrec2006/pdf/384_pdf.pdf. 2.2.1

[43] J. Yin, X. Jiang, Z. Lu, L. Shang, H. Li, and X. Li, "Neural Generative Question answering," *arXiv preprint arXiv:1512.01337*, 2015, available at: https://doi.org/10.48550/arXiv.1512.01337. 2.2.1

[44] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020, available at: https://doi.org/10.48550/arXiv.1910.10683. 2.2.1, 3.1

[45] S. Banerjee and A. Lavie, "METEOR: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72, available at: https://aclanthology.org/W05-0909.pdf. 2.2.1

[46] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTscore: Evaluating Text Generation with BERT," *arXiv preprint arXiv:1904.09675*, 2019, available at: https://doi.org/10.48550/arXiv.1904.09675. 2.2.1

[47] T. Ji, C. Lyu, G. Jones, L. Zhou, and Y. Graham, "QAScore—An Unsupervised Unreferenced Metric for the Question Generation Evaluation," *Entropy*, vol. 24, no. 11, p. 1514, 2022, available at: https://doi.org/10.3390/e24111514. 2.2.1

[48] Anthropic, "About Claude Pro usage," August 2023, available at: https://support.anthropic.com/en/articles/8324991-about-claude-pro-usage. 2.2.2

[49] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv preprint arXiv:2307.09288*, 2023, available at: https://doi.org/10.48550/arXiv.2307.09288. 3.1, 3.2, 3.6

[50] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "GPT-NeoX-20B: An Open-Source Autoregressive Language Model," *arXiv preprint arXiv:2204.06745*, 2022, available at: https://doi.org/10.48550/arXiv.2204.06745. 3.1