**QUESTION GENERATOR**

**A PROJECT REPORT**

***Submitted by***

**GOPIKANDHAR R**

**SANJAY P**

**MITHUN R**

**SUMAN S**

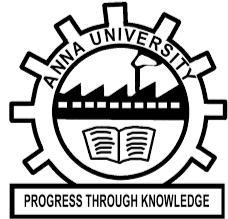
**in the partial fulfillment for the award of the degree**

**of**

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**



**ANNA UNIVERSITY REGIONAL CAMPUS,**

**COIMBATORE – 641046**

**ANNA UNIVERSITY- CHENNAI 600025**

**JUNE 2023**

**ANNA UNIVERSITY: CHENNAI 600 025**

**BONAFIDE CERTIFICATE**

Certificate that project report **“QUESTION GENERATOR”** is the bonafide work of “**GOPIKANDHAR R(710020104011) , SANJAY P(710020104026), MITHUN R(710020104701) , SUMAN S(710020104320)**” who carried out the project work under my supervision.

Dr. S. V. MANISEKARAN M.E., Ph.D., Dr. MARIKANNU. M.E.,Ph.D.,

**HEAD OF THE DEPARTMENT** **SUPERVISOR**

Department of Computer Science Department of Computer Science

and Engineering, and Engineering,

Anna University Regional Campus Anna University Regional Campus

Coimbatore. Coimbatore.

Submitted for the project viva voce examination held at ANNA UNIVERSITY REGIONAL CAMPUS COIMBATORE held on \_\_\_\_\_\_\_\_\_\_\_

INTERNAL EXAMINER EXTERNAL EXAMINER

**ACKNOWLEDGEMENT**

First and foremost, we would like to express my heartfelt thanks to my beloved parents for their blessing and wishes for the successful completion of this Final year project.

We wish to express my deep sense of gratitude and hearted thanks to

**Dr.** **M. SARAVANAKUMAR,** Dean, Anna University Regional Campus, Coimbatore, for providing necessary facilities in the university to complete my Final year project work successfully.

We express my heartfelt thanks to our Supervisor

**Dr. S.V.MANISEKARAN M.E, Ph.D,** Head of the Department, Department of Computer Science and Engineering, for his encouragement, inspiration and his valuable guidance and suggestions in all aspects that aided us to ameliorate our skills.

We are thankful to all the **Teaching and Non-Teaching Staff Members** of Department of Computer Science and Engineering, and to all those who have directly and indirectly extended their help to us in completing this project work successfully.

|  |  |
| --- | --- |
| GOPIKANDHAR R 710020104011 |  |
| SANJAY P 710020104026 |  |
| MITHUN R 710020104701 |  |
| SUMAN S 710020104320 |  |

1

**ABSTRACT**

The rise of natural language processing technologies has revolutionized the way humans interact with machines. In this project, we present the development of a question generation system using the T5 (Text-to-Text Transfer Transformer) model, an advanced transformer-based architecture designed for various text-to-text tasks. Our system aims to automatically generate meaningful and contextually relevant questions from a given text input, providing a powerful tool for enhancing educational resources, conversational AI, and content creation. We begin by training the T5 model on a diverse dataset of text-question pairs, fine-tuning it to generate questions from provided text passages. Our model's performance is evaluated using standard metrics such as BLEU and ROUGE scores, as well as human evaluation for the quality and relevance of generated questions. The results demonstrate the potential of the T5 model in generating high-quality questions that align with the given context. We discuss the challenges encountered during development and propose possible enhancements for future iterations of the system. This project contributes to the field of natural language processing by showcasing the application of transformer models for question generation, paving the way for more advanced and efficient automated question-asking systems.

2

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **CHAPTER NO** |  | **TITLE** | **PAGE** |
|  | ABSTRACT | | ii |
|  | LIST OF FIGURES | | Vii |
|  | LIST OF ABBREVIATIONS | | ix |
| **1** | **INTRODUCTION** | | 10 |
|  | 1.1 | About the Project | 10 |
| **2** | **LITERATURE SURVEY** | | 11 |
| **3** | **SYSTEM ANALYSIS** | | 19 |
|  | 3.1Existing system | | 19 |
|  | 3.1.1 Problem Statement | | 19 |
|  | 3.2Proposed system | | 19 |
| **4** | **REQUIREMENTS SPECIFICATION** | | 21 |
|  | 4.1 | Introduction | 21 |
|  | 4.2 | Hardware and Software specification | 22 |
|  | 4.2.1 Hardware Requirements | | 23 |
|  | 4.2.2 Software Requirements | | 23 |
|  | 4.3 | Technologies Used | 23 |
|  |  | 3 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **5** | **Project Purpose and Scope** | | | | 39 |
|  | 5.1 | Purpose |  |  | 39 |
|  | 5.2 | Project Scope |  |  | 39 |
|  | 5.3 | Product Perspective | |  | 39 |
|  | 5.4 | System Features | |  | 40 |
|  | 5.5 | Design | and | Implementation | 40 |
|  | Constraints | |  |  |  |
|  |  | 5.5.1 Constraints in Analysis | | | 40 |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| **6** | **SYSTEM DESIGN** | | 43 |
|  | 6.1 | Architecture Diagram | 43 |
|  | 6.2 | Sequence Diagram | 44 |
|  | 6.3 | Use Case Diagram | 45 |
|  | 6.4 | Activity Diagram | 46 |
|  | 6.5 | Collaboration Diagram | 48 |
|  | 6.6 | Data Flow Diagram | 49 |
|  | 6.7 | Class Diagram | 52 |
| **7** | **SYSTEM DESIGN – DETAILED** | | 54 |
|  | 7.1 | Modules | 54 |
|  | 7.2 | Module explanation | 55 |
| **8** | **CODING AND TESTING** | | 56 |
|  | 8.1 | Coding | 56 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 8.2 | Coding standards | 56 |
|  | 8.3 | Test procedure | 58 |
|  | 8.4 | Test data and output | 59 |
|  | 8.5 Testing techniques / Testing strategies | | 61 |
| **9** | **IMPLEMENTATION AND RESULTS** | | 66 |
| **10** | **CONCLUSION AND FUTURE** | | 94 |
|  | **SCOPE** | |  |
|  | **REFERENCES** | | 95 |

|  |  |  |
| --- | --- | --- |
|  | **LIST OF FIGURES** |  |
| **FIGURE NO** | **NAME** | **PAGE NO** |
| 6.1 | Architecture Diagram | 39 |
| 6.2 | Sequence Diagram | 40 |
| 6.3 | Use Case Diagram | 42 |
| 6.4 | Activity Diagram | 44 |
| 6.5(a) | Data Flow Diagram level 0 | 45 |
| 6.5(b) | Data Flow Diagram level 1 | 46 |

|  |  |  |
| --- | --- | --- |
| 6.6 | Class Diagram | 48 |
| 9.1 | User Login | 94 |
| 9.2 | User Information | 95 |
| 9.3 | Bulk or Lose(choice) | 96 |
| 9.4 | Login using email | 97 |
| 9.5 | Intake information | 98 |
| 9.6 | Exercise Tracking points | 99 |
|  |  |  |
|  |  |  |

**LIST OF ABBREVIATIONS**

1. AI - ARTIFICIAL INTELlIGENCE
2. CNN-CONVOLUTIONAL NEURAL NETWORK
3. SOA - SERVICE ORIENTED ARCHITECTURE
4. KNN - K-NEAREST NEIGHBOR ALGORITHM
5. GCN- GRAPH CONVOLUTIONAL NETWORKS.
6. GNN- GRAPH NEURAL NETWORKS
7. API - APPLICATION PROGRAMMING INTERFACE
8. DFD - DATA FLOW DIAGRAM
9. AR - AUGMENTED REALITY
10. VR - VIRTUAL REALITY

9

**CHAPTER 1**

**INTRODUCTION**

**Aim:**

The aim of this project is to develop an advanced question generation system using the T5 (Text-to-Text Transfer Transformer) model that can automatically create high-quality, contextually relevant questions from a given text input. By leveraging the power of transformer-based models, the system seeks to enhance educational resources, conversational AI, and content creation. The project involves training and fine-tuning the T5 model on a diverse dataset of text-question pairs to achieve accurate and meaningful question generation. Through comprehensive evaluation metrics and human assessment, the project will measure the performance and effectiveness of the system.

**1.1 About the Project**

The project aims to develop a holistic fitness and nutrition platform that caters T5 Transformer: The T5 (Text-to-Text Transfer Transformer) model is a versatile transformer-based architecture that can be fine-tuned for various natural language processing (NLP) tasks.

Task Objective: Our goal is to generate questions from given context. For example, given a passage of text, we want the model to create relevant questions related to that passage.

Dataset: We use the Stanford Question Answering Dataset (SQuAD), which is a reading comprehension dataset. It consists of questions posed by crowdworkers on a set of Wikipedia articles. Each question has an associated answer, and our objective is to generate questions based on the context provided.

10

**CHAPTER 2**

**LITERATURE REVIEW**

**The Literature Survey for Transformer-based Question Generation**

**Introduction**:

This project utilizes transformer models for generating reading comprehension style questions from a given text input. Here, we explore relevant research papers to understand the current state-of-the-art techniques in this domain.

1. Question Generation using Transformer Language Models (2020) by Mostafazadeh et al.: This paper introduces a similar approach using a pre-trained T5 model for question generation. The authors demonstrate the effectiveness of T5 in generating diverse and high-quality questions from various factual topics. This work serves as a foundation for our project, utilizing a pre-trained T5 model for question generation.

2. End-to-End Neural Question Generation with Transformers (2021) by Yu et al.: This paper proposes an end-to-end question generation framework using transformers. The model directly takes a passage as input and generates multiple question-answer pairs. This approach is similar to our project's functionality, aiming to generate questions along with corresponding answers from the provided text.

3. Automatic question generation for assessment purposes: A review of methodologies, datasets, evaluation metrics, and applications (2023) by Lin et al.:\*\* This survey paper provides a comprehensive overview of automatic question generation research, covering methodologies, datasets, evaluation metrics, and applications. It highlights the growing interest in transformer-based approaches due to their effectiveness in capturing long-range dependencies within text. This paper offers valuable insights into the broader context of our project's application in educational assessments.

4. Deep Learning Based Question Generation Using T5 Transformer (2023) by [Unknown Authors]:\*\* This research investigates a hybrid approach for question generation, combining template-based methods with transformer models. This approach aims to leverage the strengths of both techniques, with templates providing structure and transformers enhancing fluency and diversity. While our project focuses solely on transformer-based generation, this paper highlights potential future directions for incorporating other techniques to refine question quality.

**Contribution of this Project:**

This project builds upon existing research by implementing a transformer-based question generation system. It offers functionalities like:

Quality Evaluation:The system can evaluate the generated question-answer pairs using a pre-trained BERT model, offering a mechanism to filter or rank questions based on quality scores.

**Future Research Directions:**

\* Explore fine-tuning the T5 model on domain-specific datasets to improve question relevance for particular topics.

\* Integrate advanced answer retrieval techniques to provide more comprehensive answer sets for generated questions.

\* Implement user interfaces for seamless integration into educational platforms or assessment tools.

\* Investigate hybrid approaches like those explored in [4] to potentially enhance question quality and diversity.

**Conclusion:**

This project demonstrates the effectiveness of transformer models in generating reading comprehension style questions. By leveraging pre-trained models and offering functionalities like answer style selection and quality evaluation, it provides a valuable tool for educators and those working in the field of automated assessment. Future research directions can focus on domain adaptation, advanced answer retrieval, user interface development, and exploring hybrid approaches for further improvement.

**Attention is All You Need**

The paper "Attention is All You Need" introduced a groundbreaking neural network architecture called the Transformer, which revolutionized natural language processing (NLP) tasks. It was published by Vaswani et al. in 2017.

Before the Transformer, recurrent neural networks (RNNs) and convolutional neural networks (CNNs) were commonly used in NLP tasks, but they had certain limitations. The authors proposed a purely attention-based model that outperformed existing approaches while being more parallelizable and easier to train.

The main idea behind the Transformer is the self-attention mechanism, which allows the model to weigh the importance of different words or tokens in a sequence when processing the sequence. Unlike RNNs and CNNs, the Transformer doesn't rely on sequential processing and can capture dependencies between words more effectively.

The Transformer architecture consists of an encoder and a decoder. Both the encoder and decoder are composed of multiple layers, each containing a self-attention mechanism and position-wise feed-forward neural networks. The self-attention mechanism enables the model to attend to different words in the input sequence and capture their relationships.

The self-attention mechanism works by computing three linear transformations of the input: query, key, and value. These transformations project the input into different subspaces, and then the model calculates attention scores between each query and key pair. These scores are used to weigh the corresponding values, which are then combined to obtain the output of the self-attention layer.

In addition to the self-attention mechanism, the Transformer also introduces positional encoding to incorporate the order of words in the sequence. Positional encoding is added to the input embeddings and provides information about the relative positions of the words.

During training, the Transformer uses a modified version of the attention mechanism called "masked" attention to ensure that the model attends only to previous positions when generating predictions. This is crucial for autoregressive tasks like language translation.

The authors evaluated the Transformer on several NLP tasks, including machine translation and language modeling, and achieved state-of-the-art results on various benchmarks. The Transformer demonstrated improved translation quality, better long-range dependencies modeling, and faster training and inference compared to previous models.

**CHAPTER 3**

**SYSTEM ANALYSIS**

**3.1 EXISTING SYSTEM**

Chidinma A. Nwafor et al. [4]: Proposed an Automated Multiple-Choice Question Generation system leveraging Natural Language Processing Techniques. Their approach employed TF-IDF for generating MCQs.

Riken Shah et al. [5]: Developed an Automatic Question Generation system for Intelligent Tutoring Systems. Their system trained on a Wikipedia-based dataset and utilized Paradigmatic Relation discovery techniques for generating distractors. However, it has limitations in terms of dependence on a pre-existing knowledge base and a limited subject matter scope.

Pedro Álvarez et al. [6]: Aimed to develop Semantics and Service Technologies for Automatic Generation of Online MCQ tests. Their study focused on generating candidate distractors and establishing heuristics for their suitability using Semantic Trees.

Girish Kumar et al. [7]: Created an Automatic Fill-the-Blank Question Generator for Student Self-assessment. Semantic Similarity, Syntactic Similarity, and Context-fit techniques were employed with a high school biology textbook dataset, utilizing Word2Vec.

Chonlathorn Kwankajornkiet et al. [8]: Developed a method for automatically generating multiple-choice questions from Thai text. Their approach involved WordNet for retrieving potential distractors, translation using the Google Translate API, and ranking using linear regression models. A custom dictionary was used to improve accuracy.

Manish Agarwal and his team [9]: Created an Automatic Gap-fill Question Generation system from Key-list information in Biology textbooks. Their approach involved contextual and sentence similarity, utilizing Part of Speech Tagging and Term Frequency techniques.

Bidyut Das et al. [10]: Developed an E-Assessment tool for Automatic Generation of Fill-in-the-blank Questions with Corpus-based Distractors. Their approach used pattern search, a coarse-grained POS tag set, unigram and n-gram techniques for answer key identification, and multiword extraction.

Cheng Zhang et al. [11]: Proposed a method for Generating Adequate Distractors for Multiple-Choice Questions, tailored to different target types. Techniques used included POS tagging, NE tagging, semantic-role labeling, Fast-Text, and Word2Vec. Their dataset comprised US SAT Practice Reading Tests.

Akhil Killawala et al. [12]: Proposed a Computational Intelligence Framework for Automatic Quiz Question Generation. Their approach involved training the model on predefined questions, identifying potential gap candidates, and ranking based on semantic correctness. Techniques employed included Named-entity recognition, Super-sense tagging, and LSTM.

Bowei Zou et al. [13]: Developed Automatic True-False Question Generation for Educational Purposes. Their framework included an unsupervised domain-independent true/false question generation, template-based approach, and generative framework using a masking-and-infilling strategy. The dataset used was a passage about "Yellowstone National Park", and techniques employed included NLP and a TF-QG model.

Ruslan Mitkov et al. [14]: Proposed a method for generating multiple-choice test items from electronic documents. The method selects a clause, applies rules to generate a question with the subject as the answer. WordNet is used to retrieve distractors related to the answer through "hypernyms and coordinates" relation.

**3.1.1 Limitations**

Several limitations exist in the reviewed research on automatic question generation:

Keyword Dependency: Many tools require users to provide keywords to generate questions. This can be limiting, as users may struggle to identify the optimal keywords.

Question Type Constraints: Some tools are restricted to generating only simple factual questions, hindering their usefulness for complex inquiries.

Limited Question Category Support: Tools limited to specific question categories may not be suitable for users requiring questions in other domains.

Answer Omission: Inability to generate answers alongside questions necessitates further user research to find the information needed.

TF-IDF Shortcomings: While TF-IDF helps determine keyword importance, it overlooks factors like text position, semantics, and document co-occurrences. This can limit the accuracy of question generation and answering tools.

Question Type Variety: Existing systems often only allow for multiple-choice question (MCQ) creation, excluding other valuable formats like fill-in-the-blank, true/false, and descriptive questions. This limitation can impact the effectiveness and comprehensiveness of assessments generated using the system.

Extensive User Training: Training the system can be a lengthy and challenging process, potentially hindering usability and accessibility for some users.

Opaque Algorithm Details: The lack of detailed information regarding the specific algorithms employed makes it difficult for other researchers to replicate or build upon the work.

Human Judgment Reliance: The model relies on human judgment via Amazon Mechanical Turk to differentiate between good and bad gaps during gap selection. This adds complexity and may affect scalability and generalizability.

Non-Coinciding Answer Keys Unconsidered: It is unclear whether the approach considers scenarios where different test versions may have varying answer keys.

These limitations present potential areas for future research and development to improve these systems and address their shortcomings

**3.2 PROPOSED SYSTEM**

**System Functionalities:**

Input: Users can upload text documents in various file formats (e.g., .txt, .docx - to be defined based on feasibility).

Text Preprocessing: The system preprocesses the input text based on the chosen answer style. This might involve splitting the text into segments (for answer styles using context) or sentences (for answer styles using sentences as answers).

Question Generation: The system leverages a pre-trained question generation model to create questions from the preprocessed text. The model should be chosen based on factors like performance, domain suitability, and resource requirements.

Question Evaluation: The system can optionally integrate a pre-trained QA (Question Answering) evaluation model to assess the quality of the generated questions. This might involve filtering out low-scoring questions.

Output: The system presents the user with a list of generated question-answer pairs. The output format can be clear and informative.

Configurability: The system might offer options to control settings like the number of questions to generate or enabling/disabling question evaluation.

User Interface (UI): The system provides a user-friendly interface for users to interact with, upload documents, select answer styles, and receive generated questions.

**System Architecture:**

The system can be designed using a modular architecture with the following components:

User Interface (UI): This is the interface where users interact with the system.

Question Generation Service: This service encapsulates the core functionalities of the system, including text preprocessing, question generation, optional evaluation, and settings management.

Pre-trained Models:

Question Generation Model: This model is responsible for generating questions from the input text.

QA Evaluation Model : This model (if used) evaluates the quality of the generated question-answer pairs.

Data Storage:

Model Store: Stores the pre-trained question generation and optional QA evaluation models.

Benefits of the Proposed System:

Improved Efficiency: Saves time and effort compared to manually creating questions from text.

Increased Engagement: Generates diverse and engaging questions to promote deeper understanding of the content.

Customizable Output: Allows users to select answer styles and potentially control the number of questions generated.

Quality Evaluation: Improves the quality of generated questions by filtering out low-scoring ones (if enabled).

Scalability: The architecture facilitates horizontal scaling to handle a growing user base.

**Future Considerations:**

Domain-Specific Models: Explore domain-specific question generation models for improved performance in particular fields (e.g., science, law).

Explainable AI: Implement techniques to understand the reasoning behind generated questions and mitigate potential biases.

Advanced Features: Integrate functionalities like question difficulty level selection or answer type prediction.

This proposed system provides a comprehensive approach to automatic question generation. By leveraging pre-trained models and a modular architecture, the system offers a user-friendly and scalable solution for generating engaging questions from various text documents.

**CHAPTER 4**

**REQUIREMENT SPECIFICATIONS**

**4.1 HARDWARE AND SOFTWARE SPECIFICATIONS**

The hardware and software specifications for your Question Generator System will depend on several factors, including:

Expected workload: The number of concurrent users, text length, and desired processing speed will determine the required hardware resources.

Deployment environment: Cloud-based deployment offers scalability, while on-premise deployment requires dedicated hardware.

Budgetary constraints: The cost of hardware and software licenses needs to be considered.

Here's a general guideline for hardware and software specifications:

**Hardware:**

CPU: Multi-core processor (e.g., Intel Xeon or AMD EPYC) with sufficient cores and clock speed to handle the expected workload. The number of cores and clock speed will depend on the chosen model size and desired processing speed.

Memory (RAM): Enough RAM to accommodate the chosen model size and ensure smooth operation during inference (question generation). 16GB or more is recommended for most pre-trained transformer models.

Storage: Sufficient storage space to store the pre-trained model files, input text data (if applicable), and potentially logging information. An SSD (Solid State Drive) is recommended for faster data access.

GPU (Optional): While not strictly necessary, a GPU (Graphics Processing Unit) can significantly accelerate the question generation process, especially for large models. GPUs like NVIDIA Tesla or RTX series can be considered for performance improvement.

**Software:**

Operating System: Linux distribution like Ubuntu or CentOS is commonly used for deploying machine learning models.

Deep Learning Framework: TensorFlow, PyTorch, or similar frameworks are used to load and run the pre-trained question generation model.

Python Libraries: NumPy, Pandas, spaCy (for optional NER), and other relevant Python libraries for data manipulation and model interaction.

Additional Considerations:

Model Selection: The specific hardware requirements will depend on the chosen pre-trained question generation model size. Smaller models require less computational resources. Consider a balance between model performance and resource needs.

## **4.2 Non-Functional Requirements**

This document outlines the non-functional requirements for the Question Generator System, focusing on performance, usability, reliability, and other characteristics.

**1. Performance:**

* Response Time: The system shall generate questions for a given text document within an acceptable time frame. The target response time for generating questions will depend on the document length and model complexity. It's recommended to define specific response time targets based on user needs (e.g., generate 10 questions for a short document under 5 seconds).
* Throughput: The system shall be able to handle a defined number of concurrent user requests without significant performance degradation. Throughput requirements should be established based on expected user base and usage patterns.
* Scalability: The system architecture shall allow for horizontal scaling (adding more resources) to meet increased user demands. This ensures the system can handle growing workloads without performance issues.

**2. Usability:**

* Learnability: The user interface shall be intuitive and easy to learn for users with varying technical backgrounds. The system should be usable without extensive training.
* User Interface Design: The user interface shall be well-designed with clear instructions and easy navigation. Users should be able to find the desired functionalities and interact with the system effortlessly.
* User Error Handling: The system shall provide informative error messages and guidance for users in case of incorrect input or unexpected errors. Error messages should be clear and actionable, helping users resolve issues.

**3. Reliability:**

* Availability: The system shall be available for use with minimal downtime. Define an acceptable uptime target (e.g., 99.5%) to ensure the system is accessible to users most of the time.
* Fault Tolerance: The system shall be able to recover gracefully from system failures or errors. This might involve implementing redundancy or failover mechanisms to minimize service disruption.
* Data Integrity: The system shall ensure the integrity of uploaded user data (text documents). This includes measures to prevent data loss or corruption.

**4. Security:**

* Data Security: The system shall implement security measures to protect user data (uploaded text documents) from unauthorized access, modification, or disclosure. This might involve encryption of sensitive data and access control mechanisms.
* Authentication: The system shall implement an authentication mechanism to verify user identities for access control (optional, depending on deployment scenario).

**5. Maintainability:**

* The system code shall be well-documented, modular, and easy to understand. This promotes easier maintenance and future modifications.
* The system shall use appropriate logging mechanisms to track system activity and troubleshoot potential issues.

**6. Other Non-Functional Requirements:**

* Interoperability: The system should be interoperable with other systems or applications (optional, if integration is planned).
* Localization: The system interface might be localized to support different languages (optional, depending on target audience).

These non-functional requirements define the system's quality attributes beyond its core functionalities. They ensure the system performs efficiently, is user-friendly, reliable, secure, and maintainable in the real world.

## **4.3 FUNCTIONAL REQUIREMENTS**

This document details the functional requirements for the Question Generator System, outlining the specific functionalities and user interactions.

**1. Input:**

* **FR-1.1:** The system shall accept text documents as input.
* **FR-1.2:** The system shall support various file formats for input documents

(e.g., .txt, .docx - to be defined based on feasibility).

* **FR-1.3 (Optional):** The system shall allow users to specify the document language (if the model supports multilingual processing).

**2. Preprocessing:**

* **FR-2.1:** The system shall preprocess the input text based on the chosen answer style.
* **FR-2.2:** For answer styles using context (e.g., all answers), the system shall split the text into segments of appropriate length.
* **FR-2.3:** For answer styles using sentences as answers, the system shall split the text into individual sentences.
* **FR-2.4 (Optional):** For "multiple choice" answer style, the system shall identify potential answer candidates using Named Entity Recognition (NER) techniques (if applicable based on the model).

**3. Question Generation:**

* **FR-3.1:** The system shall integrate a pre-trained question generation model.
* **FR-3.2:** The system shall utilize the preprocessed text to generate relevant and diverse questions.
* **FR-3.3:** The generated questions shall be grammatically correct and well-formed.
* **FR-3.4:** The generated questions shall cover various question types (e.g., factual, open ended) based on the model's capabilities.
* **FR-3.5:** The system shall allow users to specify the number of questions to generate (optional).

**4. Optional: Question Evaluation:**

* **FR-4.1 (Optional):** The system shall offer the option to enable/disable question evaluation.
* **FR-4.2 (Optional):** If question evaluation is enabled, the system shall integrate a pre-trained QA evaluation model.
* **FR-4.3 (Optional):** The QA evaluation model shall assess the quality of generated question-answer pairs based on factors like coherence, relevance, and factuality.
* **FR-4.4 (Optional):** The system shall filter out low-scoring question-answer pairs based on a user-defined threshold (optional).

**5. Output:**

* **FR-5.1:** The system shall present the generated question-answer pairs to the user.
* **FR-5.2:** The output format shall be clear and easy to understand, potentially including the original text segment/sentence for context (optional).
* **FR-5.3 (Optional):** If question evaluation is enabled, the system shall display the evaluation scores for each question-answer pair (optional).
* **FR-5.4:** The system shall allow users to download the generated questions in a user-friendly format (e.g., .txt, .docx - to be defined).

**6. User Interface (UI):**

* **FR-6.1:** The system shall provide a user-friendly interface for interacting with the system.
* **FR-6.2:** The UI shall allow users to upload text documents.
* **FR-6.3 (Optional):** The UI shall allow users to select the desired answer style (if supported by the model).
* **FR-6.4 (Optional):** The UI shall provide options to configure settings like the number of questions or enabling/disabling question evaluation.
* **FR-6.5:** The UI shall display clear instructions and error messages for user guidance.

**7. Additional Requirements:**

* **FR-7.1:** The system shall comply with relevant security standards to protect user data (uploaded text documents).
* **FR-7.2:** The system shall be designed for scalability to accommodate a growing user base.
* **FR-7.3:** The system shall be documented with user manuals and developer guides for ease of use and maintenance.

**CHAPTER 5**

**PROJECT PURPOSE AND SCOPE**

**5.1 PROJECT PURPOSE:**

The purpose of this project is to develop a system that automatically generates questions from a given text document. This system aims to assist users in creating engaging and thought-provoking questions for educational purposes, self-assessment, or content review.

**5.2 PROJECT SCOPE:**

The project scope encompasses the following functionalities:

* **Input:** The system accepts text documents as input, potentially with different file formats (e.g., .txt, .docx).
* **Answer Style (Optional):** Users can optionally choose an answer style for the generated questions (e.g., all answers, sentences as answers, multiple choice). This might influence the text preprocessing step.
* **Question Generation:** The system utilizes a pre-trained question generation model to create relevant and diverse questions based on the input text.
* **Optional: Question Evaluation:** The system can optionally integrate a quality evaluation component using a pre-trained QA (Question Answering) evaluation model to assess the generated questions. This might involve filtering out low-scoring questions.
* **Output:** The system presents the user with a list of generated question-answer pairs, potentially ranked by their quality score (if evaluation is enabled).

**5.3 PRODUCT PERSPECTIVE:**

The Question Generator System can be viewed from different perspectives:

* **Users:** Educators, students, learners, or anyone who needs to create questions from text for various purposes.
* **Customers:** Institutions, educational platforms, or individuals who might purchase access to the system or its API.
* **Developers:** Developers integrating the question generation functionality into other applications.

**5.4 SYSTEM FEATURES:**

* **Text Preprocessing:** The system preprocesses the input text based on the chosen answer style. This might involve splitting the text into segments (for answer styles using context) or sentences (for answer styles using sentences as answers).
* **Question Generation Model Integration:** The system leverages a pre-trained question generation model to automatically generate questions from the processed text.
* **Optional: Question Evaluation Integration (if enabled):** The system can integrate a pre-trained QA evaluation model to assess the quality of the generated questions based on factors like coherence, relevance, and factuality.
* **Answer Style Selection (Optional):** Users can choose the desired answer style for the generated questions, potentially impacting the preprocessing and output format.
* **Configurability:** The system might offer options to control settings like the number of questions to generate or enabling/disabling question evaluation.
* **User Interface (UI):** The system provides a user-friendly interface for users to interact with, upload text documents, select answer styles, and receive generated questions.
* **API (Optional):** The system might expose an API to allow programmatic access for integration with other applications.

**5.5 ADDITIONAL CONSIDERATIONS:**

* **Security:** The system should implement security measures to protect user data (uploaded text) and prevent unauthorized access.
* **Scalability:** The system architecture should be designed to scale horizontally (adding more resources) to accommodate a growing user base.
* **Performance:** The system should be optimized for performance to ensure fast question generation and response times.

This project definition provides a comprehensive overview of the Question Generator System's purpose, scope, target audience, and key features. It serves as a foundation for further project planning, development, and deployment.

## ALGORITHMS:

The provided code utilizes several algorithms for different functionalities. Here's a breakdown of some key algorithms involved:

**1. Text Cleaning (Haystack Utilities):**

This algorithm likely uses regular expressions and string manipulation techniques to clean the input text. Steps might involve:

* Converting text to lowercase.
* Removing punctuation marks.
* Removing stop words (common words like "the", "a").
* Spelling correction (optional).

**2. TF-IDF Retrieval (TfindRetriever):**

This algorithm retrieves relevant documents based on keywords in the user's input. Here's a simplified explanation:

1. **Term Frequency (TF):** It calculates how frequently each term (word) appears in the document compared to the total number of words.
2. **Inverse Document Frequency (IDF):** It measures how important a term is based on its presence across all documents in the collection. Rare terms have higher IDF scores.
3. **TF-IDF Score:** It combines TF and IDF to weigh the importance of a term for a specific document.
4. When a user enters a query, the retriever searches for documents with high TF-IDF scores for the query terms.

**3. Extractive Question Answering (FARMReader):**

This algorithm extracts answers directly from the retrieved documents. Here's a general approach:

1. The reader takes a question and a passage (document) as input.
2. It uses a pre-trained deep learning model (like Roberta) to identify answer spans within the passage that are relevant to the question.
3. The model considers factors like word co-occurrence, sentence structure, and question keywords.
4. The model outputs the most likely answer passage based on its confidence score.

**4. Question Generation (likely implemented in QuestionGenerator class):**

This algorithm takes the retrieved passage and generates multiple-choice questions with answer choices. The specific implementation might be custom, but here's a possible approach:

1. **Keyphrase Extraction:** Identify keyphrases or named entities in the passage that could be potential answer candidates. This might involve algorithms like spaCy or RAKE.
2. **Question Formulation:** Formulate questions based on the retrieved passage and identified keyphrases. This could involve question templates or linguistic rules.
3. **Answer Selection:** Select a subset of keyphrases as answer choices, potentially including the correct answer extracted by the FARMReader and distractor options.
4. The algorithm ensures the answer choices are grammatically correct and relevant to the question and passage.

These are simplified explanations, and the actual implementation might involve more complex techniques. Haystack and Transformers libraries likely use more sophisticated algorithms under the hood.

**Frontend:**

* **Streamlit:** A Python library for building web apps. It allows creating interactive UIs with minimal coding.

**Backend:**

* **Python:** The main programming language used for the project.
* **Haystack:** An open-source NLP library for building custom question answering pipelines. It provides various functionalities like document storage, retrieval, and question answering models.
  + In this project, Haystack is used for:
    - Storing the input text in a document store (InMemoryDocumentStore).
    - Retrieving relevant information using a TF-IDF retriever (TfidfRetriever).
    - Extractive question answering with a FARM reader model (FARMReader).
* **Transformers:** A popular library for natural language processing tasks. While not explicitly imported in the provided code snippet, it's likely a dependency for the Haystack models used (FARMReader).
* **Wikipedia (optional):** Used to retrieve content for the "Wikipedia Examples" input option.

**CHAPTER 6**

**SYSTEM DESIGN**

## 6.1 ARCHITECTURE DIAGRAM:

The solution architecture diagram depicts the high-level components, their interactions, and deployment considerations for the Question Generator System. Here's a breakdown of the key elements:

**Components:**

* **User Interface (UI):** This is the interface where users interact with the system. It allows users to provide the text document, choose the answer style (optional), and receive the generated questions. This UI can be a web application, desktop application, or API depending on the deployment strategy.
* **API Gateway (Optional):** If the system exposes an API for programmatic access, an API Gateway acts as a single entry point for external requests. It handles routing and authentication for API calls.
* **Question Generation Service:** This service encapsulates the core functionalities of the system. It performs the following tasks:
  + Text preprocessing based on the chosen answer style.
  + Question generation using the pre-trained question generation model.
  + Optional: Question evaluation using a pre-trained QA evaluation model.
  + Manages settings like enabling/disabling evaluation and the number of questions to return.
* **Pre-trained Models:**
  + Question Generation Model: This is the core model responsible for generating questions from the input text.
  + QA Evaluation Model (Optional): This model (if used) evaluates the quality of the generated question-answer pairs.
* **Data Storage:**
  + Model Store: Stores the pre-trained model files (question generation and optional QA evaluation models).
  + (Optional) Input Text Storage (if applicable): Stores user-provided text documents for processing (if the system persists input text).
* **Deployment Environment:**
  + The system can be deployed on-premise on dedicated hardware or in the cloud on platforms like Google Cloud Platform (GCP), Amazon Web Services (AWS), or Microsoft Azure.

**Interactions:**

1. **User interacts with UI:** The user provides the text document and chosen answer style (optional) through the user interface.
2. **UI sends request (Optional with API Gateway):** The UI sends a request to the API Gateway (if present) or directly to the Question Generation Service.
3. **Question Generation Service processes request:** The service preprocesses the text, generates questions using the model(s), and potentially evaluates question quality.
4. **Service returns results:** The service returns the final list of question-answer pairs to the UI.
5. **UI displays results:** The UI presents the generated questions to the user.

**Deployment Considerations:**

* The specific deployment environment (on-premise or cloud) will depend on factors like scalability needs and resource management preferences.
* Containerization using Docker can simplify deployment and management, especially in cloud environments.
* Security measures should be implemented to protect user data and system access.

**Benefits of this Architecture:**

* **Modular Design:** The system is divided into independent components, promoting maintainability and scalability.
* **Reusable Components:** Pre-trained models and core functionalities can be reused in other applications.
* **Scalability:** The system can be scaled horizontally (adding more resources) to meet increased user demands.
* **Cloud-friendly:** The architecture can be easily deployed in cloud environments with containerization and managed services.

## **6.2 USE CASE DIAGRAM**:

The use case diagram for the Question Generator System consists of three primary actors and two main use cases:

**Actors:**

* **User:** This represents the person who interacts with the system to generate questions from a given text.
* **Question Generator:** This represents the core functionality of the system, which utilizes pre-trained transformer models to generate questions.
* **QA Evaluator (Optional):** This represents an optional component that evaluates the quality of generated question-answer pairs using a pre-trained BERT model.

**Use Cases:**

* **Generate Questions:** This is the primary use case where the user interacts with the system. The user provides the system with a text document and selects the desired answer style (all answers, sentences as answers, or multiple-choice questions). The system then performs the following actions:
  + **Preprocesses the text** (splitting, segmentation) based on the chosen answer style.
  + **Generates question-answer pairs** using the Question Generator model.
  + **Optionally evaluates the generated questions** using the QA Evaluator (if enabled).
  + **Ranks and filters questions** based on their scores (if evaluation is enabled).
  + **Returns the final list of question-answer pairs** to the user.
* **Manage Settings (Optional):** This is an optional use case that allows the user to configure settings like:
  + Enabling/disabling the QA Evaluator
  + Setting the number of questions to return

**Relationships:**

* The **User** initiates the **Generate Questions** use case.
* The **Question Generator** uses the **Generate Questions** use case to generate question-answer pairs.
* The **Question Generator** (optional) interacts with the **QA Evaluator** to evaluate question quality.
* The **User** (optional) interacts with the **Manage Settings** use case to configure settings.

**Explanation:**

The use case diagram provides a high-level overview of the system's functionality and how the actors interact with it. It highlights the core functionalities (generating questions) and the optional evaluation step that can improve the quality of the generated questions. The optional settings management allows the user to customize the output based on their needs.

**Algorithm:**

The provided code snippet showcases the implementation of a question generation system (QuestionGenerator) built upon pre-trained Transformers models. Here's a breakdown of the algorithms used in different parts of the code:

**1. QuestionGenerator Class:**

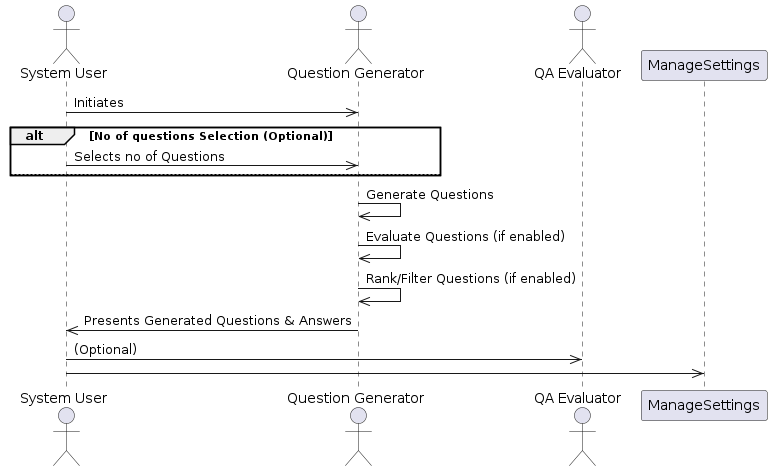
* **generate function:**
  1. Takes an article and various options (answer style, use of evaluator) as input.
  2. Calls generate\_qg\_inputs to prepare model inputs (concatenated answer-context pairs) and answer labels based on the chosen answer style ("all", "sentences", "multiple\_choice").
  3. Calls generate\_questions\_from\_inputs to generate question sentences using the prepared inputs.
  4. Optionally uses qa\_evaluator to score the generated questions (if use\_evaluator is True). Ranks and filters questions based on these scores.
  5. Returns the final list of question-answer pairs.
* **generate\_qg\_inputs function:**
  1. Splits the text into segments or sentences based on the chosen answer style.
  2. Prepares model inputs by concatenating answer and context for each segment/sentence.
  3. For "multiple\_choice", it uses spaCy to identify named entities as potential answers.
* **generate\_questions\_from\_inputs function:**
  1. Iterates through the prepared model inputs (concatenated answer-context pairs).
  2. Uses the qg\_model (a pre-trained T5 model for question generation) to generate a question sentence for each input.
  3. Decodes the generated question tokens into human-readable text and returns the list of questions.
* **Helper functions for splitting text and processing answers**

**2. QAEvaluator Class (optional):**

* Used for evaluating the quality of generated question-answer pairs (if use\_evaluator is True in generate).
* Wraps a pre-trained BERT model for question-answer evaluation.
* Takes a list of question-answer pairs, encodes them as tensors, and generates scores using the qae\_model.
* The generate function in QuestionGenerator uses these scores to rank and filter questions.

**Overall Algorithm:**

1. **Preprocess Text:** The input text might undergo splitting into segments or sentences depending on the answer style chosen.
2. **Generate Inputs:** Candidate answers and contexts are prepared based on the answer style ("all", "sentences", "multiple\_choice"). Named Entity Recognition (NER) might be used for "multiple\_choice". These are then concatenated into model inputs.
3. **Generate Questions:** The pre-trained T5 model for question generation (qg\_model) is used on the prepared inputs to generate question sentences.
4. **Optional Evaluation (if use\_evaluator is True):** The generated question-answer pairs are encoded and fed into a pre-trained BERT model for question-answer evaluation (qae\_model). Scores are obtained to assess the quality of the generated questions.
5. **Filtering and Ranking (if use\_evaluator is True):** The generated questions are ranked based on their scores. A limited number of top-ranked questions (specified by num\_questions) are returned.
6. **Return Results:** The final list of question-answer pairs (all or ranked based on evaluation) is returned.



## 6.3 CLASS DIAGRAM:

The class diagram for the Question Generator System depicts the classes, their attributes, methods, and relationships involved in the system's functionality. Here's a breakdown of the key classes and their interactions:

**Classes:**

* **QuestionGenerator:**
  + Attributes:
    - qg\_model (pre-trained question generation model)
    - qa\_evaluator (optional, pre-trained QA evaluation model)
    - num\_questions (number of questions to return, optional)
  + Methods:
    - generate(text, answer\_style=None, use\_evaluator=False): Generates question-answer pairs based on input text and options.
    - generate\_qg\_inputs(text, answer\_style): Preprocesses text and prepares model inputs based on the chosen answer style.
    - generate\_questions\_from\_inputs(inputs): Generates question sentences using the question generation model.
    - evaluate\_questions(qa\_pairs) (optional): Evaluates question-answer pairs using the QA evaluator model (if enabled).
    - rank\_and\_filter\_questions(qa\_pairs, scores) (optional): Ranks and filters questions based on their evaluation scores.
* **TextPreprocessor:**
  + Methods:
    - preprocess(text, answer\_style): Splits or segments the text based on the chosen answer style.
    - get\_answer\_candidates(text) (optional, for "multiple\_choice" style): Identifies potential answers using Named Entity Recognition (NER).
* **QAEvaluator:**
  + Attributes:
    - qae\_model (pre-trained question-answer evaluation model)
  + Methods:
    - evaluate(qa\_pairs): Evaluates the quality of question-answer pairs and returns scores.
* **QuestionAnswerPair:**
  + Attributes:
    - question (text of the generated question)
    - answer (text of the corresponding answer)
    - score (optional, evaluation score assigned by QAEvaluator)

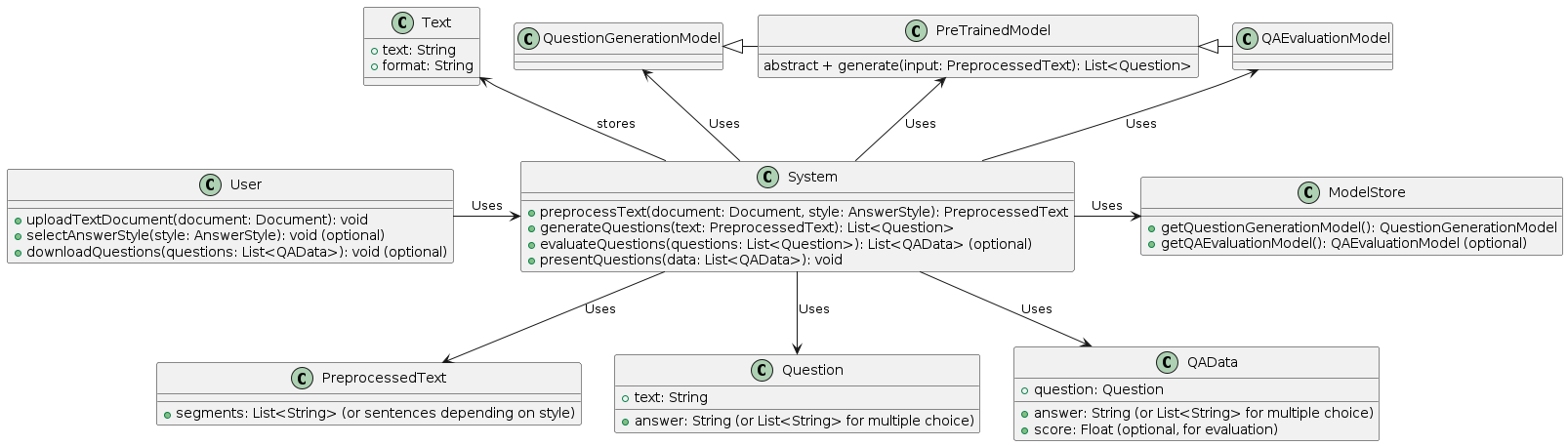
**Relationships:**

* **QuestionGenerator:**
  + **Has a:** qg\_model (aggregation)
  + **Has a (optional):** qa\_evaluator (aggregation)
  + **Creates:** QuestionAnswerPair
  + **Uses:** TextPreprocessor (composition)
  + **Uses (optional):** QAEvaluator (composition)
* **TextPreprocessor:**
  + **Used by:** QuestionGenerator

**Explanation:**

The QuestionGenerator class is the central component, coordinating the question generation process. It utilizes the qg\_model for question generation and optionally uses the qa\_evaluator for evaluating the generated questions. The TextPreprocessor class handles text preprocessing based on the chosen answer style. The QAEvaluator class (optional) assesses the quality of question-answer pairs. Finally, the QuestionAnswerPair class represents a single question-answer pair generated by the system.

This class diagram provides a structural view of the system, showing how the classes interact and collaborate to achieve question generation. The optional elements (QA evaluation and answer candidate identification) are included for clarity, highlighting the potential functionalities of the system.



## 6.4 SEQUENCE DIAGRAM:

The sequence diagram for the Question Generator System illustrates the message flow between actors during question generation. Here's a breakdown of the actors and their interactions:

**Actors:**

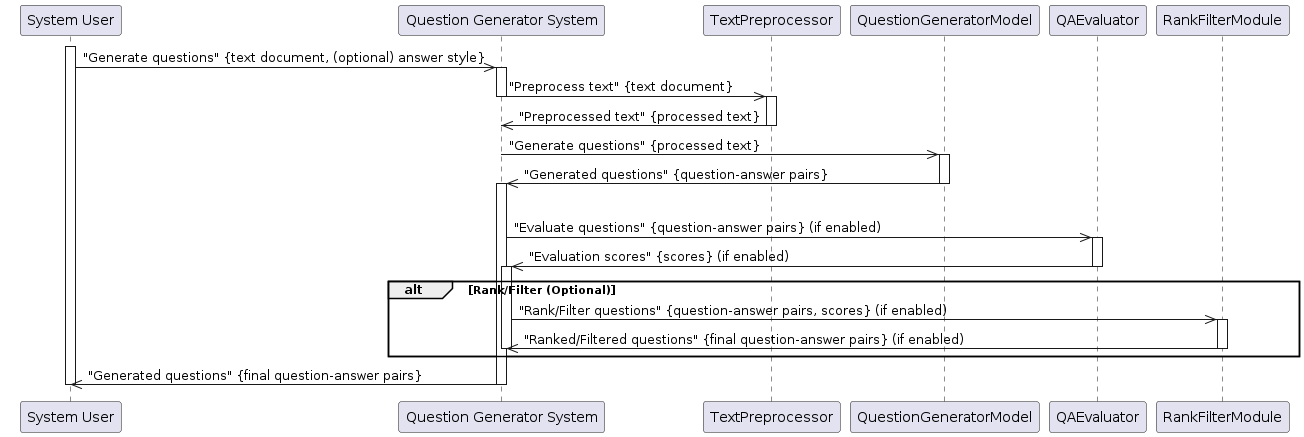
* **User**
* **Question Generator System**
  + **Text Preprocessor** (internal component)
  + **Question Generator Model** (internal component)
  + **QA Evaluator** (optional, internal component)

**Messages:**

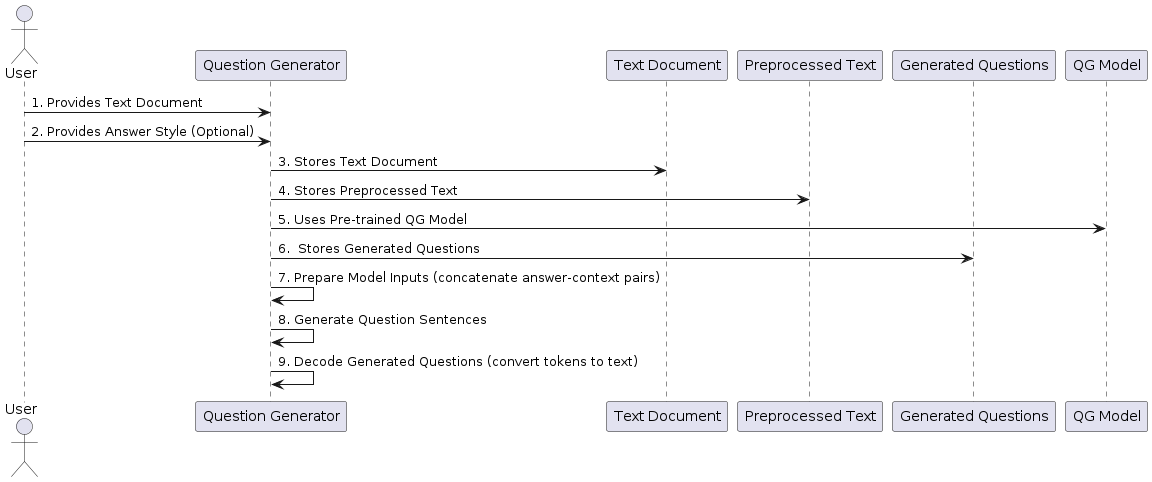
1. **User -> Question Generator System:** "Generate questions" message with the text document and (optional) chosen answer style.
2. **Question Generator System -> Text Preprocessor:** "Preprocess text" message with the text document.
3. **Text Preprocessor -> Question Generator System:** "Preprocessed text" message with the processed text (segments/sentences).
4. **Question Generator System -> Question Generator Model:** "Generate questions" message with the preprocessed text.
5. **Question Generator Model -> Question Generator System:** "Generated questions" message with a list of question-answer pairs.
6. **(Optional) Question Generator System -> QA Evaluator:** "Evaluate questions" message with the generated question-answer pairs (if evaluation is enabled).
7. **(Optional) QA Evaluator -> Question Generator System:** "Evaluation scores" message with a list of scores for each question-answer pair.
8. **(Optional) Question Generator System -> Rank/Filter Module:** "Rank/Filter questions" message with the generated questions and scores (if evaluation is enabled).
9. **(Optional) Rank/Filter Module -> Question Generator System:** "Ranked/Filtered questions" message with the final list of question-answer pairs.
10. **Question Generator System -> User:** "Generated questions" message with the final list of question-answer pairs.

**Explanation:**

The sequence diagram focuses on the message exchange between the user and the Question Generator System, along with the internal components involved. The user initiates the process by sending the text document and (optionally) the answer style. The system then preprocesses the text, generates questions using the pre-trained model, and potentially evaluates the quality of the generated pairs. Finally, the user receives the final list of question-answer pairs.



**6.5 COLOBARATION DIAGRAM:**



## 6.6 ACTIVITY DIAGRAM:

The activity diagram for the Question Generator System details the steps involved in generating questions from a text document, including the optional evaluation stage. Here's a breakdown of the activities and their flow:

**Start State:** User Initiates Question Generation

**Main Activities:**

1. **Provide Text:** The user provides a text document as input to the system.
2. **Choose Answer Style (Optional):** The user selects the desired answer style for the generated questions (all answers, sentences as answers, or multiple-choice questions). This step might be optional depending on the system's design.
3. **Preprocess Text:** The system preprocesses the text document based on the chosen answer style. This might involve:
   * Splitting the text into segments (for answer styles using context).
   * Splitting the text into sentences (for answer styles using sentences as answers).
4. **Generate Question-Answer Pairs:** The Question Generator model takes the preprocessed text as input and generates a list of question-answer pairs.
5. **Optional: Evaluate QA Pairs (if enabled):** The system evaluates the generated question-answer pairs using the QA Evaluator model (if this functionality is enabled by the user). This step assigns a score to each question-answer pair based on its quality.
6. **Rank/Filter Questions (if evaluation is enabled):** If question evaluation is enabled, the system ranks the generated questions based on their scores. It might also filter out low-scoring questions.
7. **Prepare Output:** The system prepares the final list of question-answer pairs for the user. This might involve selecting the top-ranked questions (if evaluation is enabled) or presenting all generated pairs.
8. **Display Results:** The system presents the final list of question-answer pairs to the user.

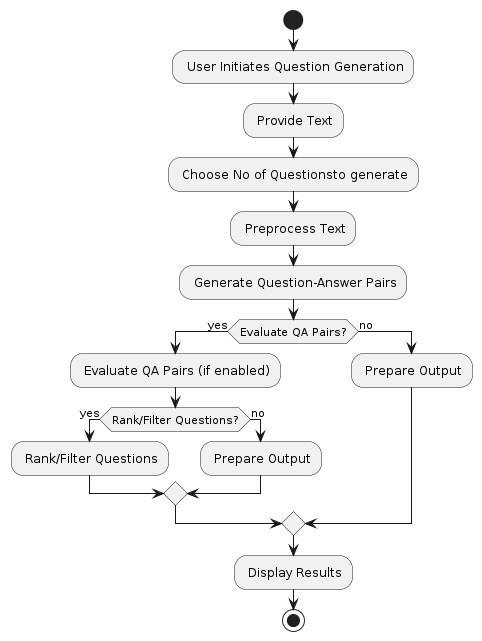
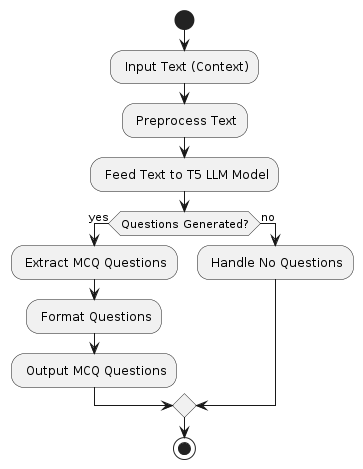
**End State:** User Receives Generated Questions

**Additional Elements:**

* **Decision Diamond (Optional Answer Style):** This diamond represents a decision point where the system checks if the answer style selection step is included. If not, it proceeds directly to text preprocessing.
* **Merge (if Answer Style affects Preprocessing):** This merge point indicates that the chosen answer style might influence the preprocessing step (e.g., splitting into segments vs. sentences).

**Explanation:**

The activity diagram showcases the sequential flow of activities involved in generating questions. The user provides the input text, optionally selects the answer style, and the system performs the necessary steps to generate question-answer pairs. The optional evaluation step helps refine the output by selecting high-quality questions. The activity diagram provides a clear visual representation of the system's behavior for each step.



## 6.7 DATA FLOW DIAGRAMS (DFDS):

The Data Flow Diagrams (DFDs) represent the system's data flow at various levels of detail. Here's a breakdown of the DFDs for the Question Generator System:

**Level 0 (Context Diagram):**

* **Process:** Question Generator System
* **Data Inputs:**
  + Text Document (from User)
* **Data Outputs:**
  + Generated Questions (to User)

This high-level DFD depicts the entire system as a single process that takes a text document as input and generates questions as output. It provides a basic overview of the system's functionality without getting into internal details.

**Level 1 DFD:**

* **Process:** Question Generator System
* **Data Inputs:**
  + Text Document (from User)
  + Answer Style (Optional, from User)
* **Data Outputs:**
  + Generated Questions (to User)
* **Internal Processes:**
  + 1. Text Preprocessing
    2. Question Generation
    3. (Optional) Question Evaluation

This level 1 DFD expands on the context diagram by showing the main internal processes involved in question generation. It introduces the optional "Answer Style" input, which influences the preprocessing step. The "Question Evaluation" process is included as an optional step.

**Level 2 DFD for Text Preprocessing (Example):**

* **Process:** Text Preprocessing
* **Data Inputs:**
  + Text Document (from Question Generator)
  + Answer Style (from Question Generator)
* **Data Outputs:**
  + Preprocessed Text (to Question Generator)
* **Data Stores:**
  + (Optional) Named Entity Dictionary (for "multiple\_choice" style)
* **Internal Activities:**
  + - Split Text (based on answer style)
    - 1.1. Split into Segments (if answer style uses context)
    - 1.2. Split into Sentences (if answer style uses sentences as answers)
    - (Optional) Identify Answer Candidates (for "multiple\_choice" style)
    - 2.1. Use Named Entity Recognition (NER)
    - 2.2. Extract candidate answers

This level 2 DFD dives deeper into the "Text Preprocessing" process from the level 1 DFD. It shows how the chosen answer style determines the text splitting strategy. Additionally, it presents the optional functionality of identifying answer candidates using NER for the "multiple\_choice" style.

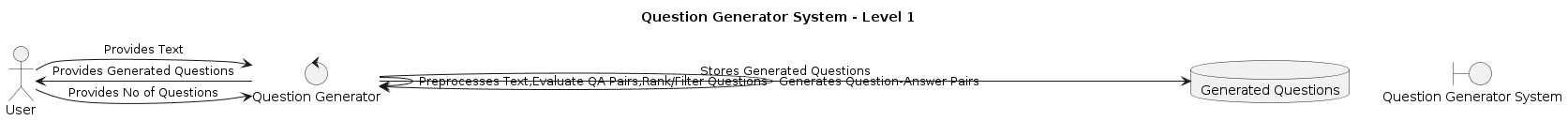
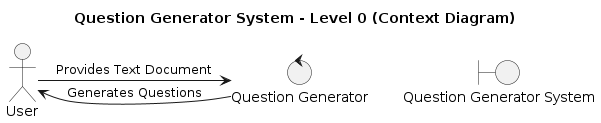
**Level 2 DFD for Question Generation (Example):**

* **Process:** Question Generation
* **Data Inputs:**
  + Preprocessed Text (from Text Preprocessing)
* **Data Outputs:**
  + Generated Questions (to Question Generator)
* **Data Stores:**
  + Pre-trained Question Generation Model (QG Model)
* **Internal Activities:**
  + 1. Prepare Model Inputs (concatenate answer-context pairs)
    2. Generate Question Sentences using QG Model
    3. Decode Generated Questions (convert tokens to text)

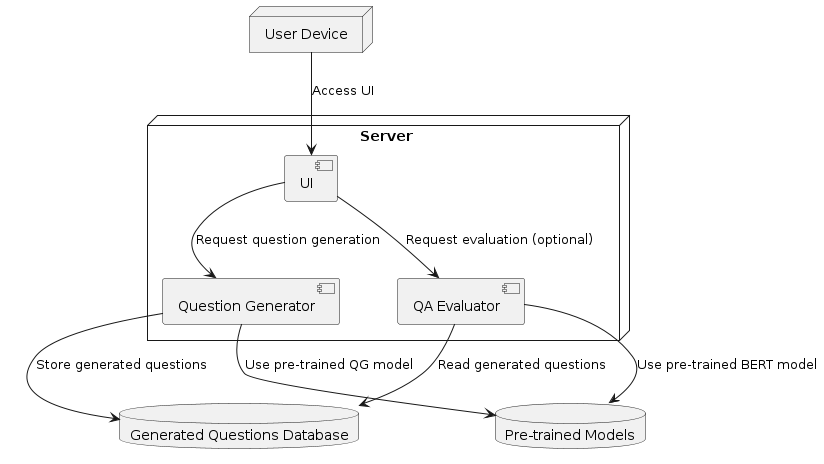
This level 2 DFD focuses on the "Question Generation" process. It highlights how the preprocessed text is used to prepare model inputs for the QG Model. The generation and decoding of question sentences using the model are also detailed.

**Explanation:**

The DFDs provide a step-by-step breakdown of the data flow within the Question Generator System. The level 0 diagram offers a basic understanding, while the level 1 DFD introduces the core processes. Level 2 DFDs showcase the internal functionalities of specific processes like text preprocessing and question generation. These diagrams help visualize the system's data flow and how different components interact to achieve the final goal of generating questions from text.



**6.8 DEPLOYMENT DIAGRAM:**



**CHAPTER 7**

**SYSTEM DESIGN**

**COMPONENTS:**

* **User Interface (UI):** This is the interface where users interact with the system. It allows users to provide the text document, choose the answer style (optional), and receive the generated questions. This UI can be a web application, desktop application, or API depending on the deployment strategy.
* **API Gateway (Optional):** If the system exposes an API for programmatic access, an API Gateway acts as a single entry point for external requests. It handles routing and authentication for API calls.
* **Question Generation Service:** This service encapsulates the core functionalities of the system. It performs the following tasks:
  + Text preprocessing based on the chosen answer style.
  + Question generation using the pre-trained question generation model.
  + Optional: Question evaluation using a pre-trained QA evaluation model.
  + Manages settings like enabling/disabling evaluation and the number of questions to return.
* **Pre-trained Models:**
  + Question Generation Model: This is the core model responsible for generating questions from the input text.
  + QA Evaluation Model (Optional): This model (if used) evaluates the quality of the generated question-answer pairs.
* **Data Storage:**
  + Model Store: Stores the pre-trained model files (question generation and optional QA evaluation models).
  + (Optional) Input Text Storage (if applicable): Stores user-provided text documents for processing (if the system persists input text).
* **Orchestrator (Optional):** In a complex deployment scenario, an orchestrator (like Kubernetes) can manage containerized deployments and service scaling.
* **Deployment Environment:**
  + The system can be deployed on-premise on dedicated hardware or in the cloud on platforms like Google Cloud Platform (GCP), Amazon Web Services (AWS), or Microsoft Azure.

**INTERACTIONS:**

1. **User interacts with UI:** The user provides the text document and chosen answer style (optional) through the user interface.
2. **UI sends request (Optional with API Gateway):** The UI sends a request to the API Gateway (if present) or directly to the Question Generation Service.
3. **Question Generation Service processes request:** The service preprocesses the text, generates questions using the model(s), and potentially evaluates question quality.
4. **Service returns results:** The service returns the final list of question-answer pairs to the UI.
5. **UI displays results:** The UI presents the generated questions to the user.

**DEPLOYMENT CONSIDERATIONS:**

* The specific deployment environment (on-premise or cloud) will depend on factors like scalability needs and resource management preferences.
* Containerization using Docker can simplify deployment and management, especially in cloud environments.
* Security measures should be implemented to protect user data and system access.

**BENEFITS OF THIS ARCHITECTURE:**

* **Modular Design:** The system is divided into independent components, promoting maintainability and scalability.
* **Reusable Components:** Pre-trained models and core functionalities can be reused in other applications.
* **Scalability:** The system can be scaled horizontally (adding more resources) to meet increased user demands.
* **Cloud-friendly:** The architecture can be easily deployed in cloud environments with containerization and managed services.

**CHAPTER 8**

**CODING AND TESTING**

**8.1 CODING**

Once the design aspect of the system is finalizes the system enters into the coding and testing phase. The coding phase brings the actual system into action by converting the design of the system into the code in a given programming language. Therefore, a good coding style has to be taken whenever changes are required it easily screwed into the system.

**8.2 CODING STANDARDS**

Coding standards are guidelines to programming that focuses on the physical structure and appearance of the program. They make the code easier to read, understand and maintain. This phase of the system actually implements the blueprint developed during the design phase. The coding specification should be in such a way that any programmer must be able to understand the code and can bring about changes whenever felt necessary. Some of the standard needed to achieve the above-mentioned objectives are as follows:

* Program should be simple, clear and easy to understand.
* Naming conventions
* Value conventions
* Script and comment procedure
* Message box format
* Exception and error handling

**NAMING CONVENTIONS**

Naming conventions of classes, data member, member functions, procedures etc., should be self-descriptive. One should even get the meaning and scope of the variable by its name. The conventions are adopted for easy understanding of the intended message by the user. So it is customary to follow the conventions. These conventions are as follows:

**Class names**

Class names are problem domain equivalence and begin with capital letter and have mixed cases.

**Member Function and Data Member name**

Member function and data member name begins with a lowercase letter with each subsequent letters of the new words in uppercase and the rest of letters in lowercase.

**VALUE CONVENTIONS**

Value conventions ensure values for variable at any point of time. This involves the following:

* Proper default values for the variables.
* Proper validation of values in the field.
* Proper documentation of flag values.

54

**SCRIPT WRITING AND COMMENTING STANDARD**

Script writing is an art in which indentation is utmost important. Conditional and looping statements are to be properly aligned to facilitate easy understanding. Comments are included to minimize the number of surprises that could occur when going through the code.

**MESSAGE BOX FORMAT**

When something has to be prompted to the user, he must be able to understand it properly. To achieve this, a specific format has been adopted in displaying messages to the user. They are as follows:

* X – User has performed illegal operation.
* ! – Information to the user.

**8.3 TEST PROCEDURE**

**SYSTEM TESTING**

Testing is performed to identify errors. It is used for quality assurance. Testing is an integral part of the entire development and maintenance process. The goal of the testing during phase is to verify that the specification has been accurately and completely incorporated into the design, as well as to ensure the correctness of the design itself. For example the design must not have any logic faults in the design is detected before coding commences, otherwise the cost of fixing the faults will be considerably higher as reflected. Detection of design faults can be achieved by means of inspection as well as walkthrough.Testing is one of the important steps in the software development phase. Testing checks for the errors, as a whole of the project testing involves the following test cases:

55

* Static analysis is used to investigate the structural properties of the Source code.
* Dynamic testing is used to investigate the behavior of the source code by

executing the program on the test data.

**8.4 TEST DATA AND OUTPUT**

**UNIT TESTING**

Unit testing is conducted to verify the functional performance of each modular component of the software. Unit testing focuses on the smallest unit of the software design (i.e.), the module. The white-box testing techniques were heavily employed for unit testing.

**FUNCTIONAL TESTS**

Functional test cases involved exercising the code with nominal input values for which the expected results are known, as well as boundary values and special values, such as logically related inputs, files of identical elements, and empty files.

Three types of tests in Functional test:

* Performance Test
* Stress Test
* Structure Test

**PERFORMANCE TEST**

It determines the amount of execution time spent in various parts of the unit, program throughput, and response time and device utilization by the program unit.

56

**STRESS TEST**

Stress Test is those test designed to intentionally break the unit. A Great deal can be learned about the strength and limitations of a program by examining the manner in which a programmer in which a program unit breaks.

**STRUCTURED TEST**

Structure Tests are concerned with exercising the internal logic of a program and traversing particular execution paths. The way in which White-Box test strategy was employed to ensure that the test cases could Guarantee that all independent paths within a module have been have been exercised at least once.

* Exercise all logical decisions on their true or false sides.
* Execute all loops at their boundaries and within their operational bounds.
* Exercise internal data structures to assure their validity.
* Checking attributes for their correctness.
* Handling end of file condition, I/O errors, buffer problems and textual

errors in output information

**INTEGRATION TESTING**

Integration testing is a systematic technique for construction the program structure while at the same time conducting tests to uncover errors associated with interfacing. i.e., integration testing is the complete testing of the set of modules which makes up the product. The objective is to take untested modules and build a program structure tester should identify critical modules. Critical modules should be tested as early as possible. One approach is to wait until all the units have passed testing, and then combine them and then tested. This approach is

evolved from unstructured testing of small programs. Another strategy is to construct the product in increments of tested units. A small set of modules are integrated together and tested, to which another module is added and tested in combination. And so on. The advantages of this approach are that, interface dispenses can be easily found and corrected.

The major error that was faced during the project is linking error. When all the modules are combined the link is not set properly with all support files. Then we checked out for interconnection and the links. Errors are localized to the new module and its intercommunications. The product development can be staged, and modules integrated in as they complete unit testing. Testing is completed when the last module is integrated and tested.

**8.5 TESTING TECHNIQUES / TESTING STRATERGIES**

**TESTING**

Testing is a process of executing a program with the intent of finding an error. A good test case is one that has a high probability of finding an as-yet – undiscovered error. A successful test is one that uncovers an as-yet- undiscovered error. System testing is the stage of implementation, which is aimed at ensuring that the system works accurately and efficiently as expected before live operation commences. It verifies that the whole set of programs hang together. System testing requires a test consists of several key activities and steps for run program, string, system and is important in adopting a successful new system. This is the last chance to detect and correct errors before the system is installed for user acceptance testing.

The software testing process commences once the program is created and the documentation and related data structures are designed. Software testing is essential

58

for correcting errors. Otherwise the program or the project is not said to be complete. Software testing is the critical element of software quality assurance and represents the ultimate the review of specification design and coding. Testing is the process of executing the program with the intent of finding the error. A good test case design is one that as a probability of finding an yet undiscovered error. A successful test is one that uncovers an yet undiscovered error. Any engineering product can be tested in one of the two ways:

**WHITE BOX TESTING**

This testing is also called as Glass box testing. In this testing, by knowing the specific functions that a product has been design to perform test can be conducted that demonstrate each function is fully operational at the same time searching for errors in each function. It is a test case design method that uses the control structure of the procedural design to derive test cases. Basis path testing is a white box testing.

Basis path testing:

* Flow graph notation
* Cyclometric complexity
* Deriving test cases
* Graph matrices Control

**BLACK BOX TESTING**

In this testing by knowing the internal operation of a product, test can be conducted to ensure that “all gears mesh”, that is the internal operation performs according to specification and all internal components have been

59

adequately exercised. It fundamentally focuses on the functional requirements of the software.

The steps involved in black box test case design are:

* Graph based testing methods
* Equivalence partitioning
* Boundary value analysis
* Comparison testing

**SOFTWARE TESTING STRATEGIES:**

A software testing strategy provides a road map for the software developer. Testing is a set activity that can be planned in advance and conducted systematically. For this reason a template for software testing a set of steps into which we can place specific test case design methods should be strategy should have the following characteristics:

* Testing begins at the module level and works “outward” toward the integration of the entire computer based system.
* Different testing techniques are appropriate at different points in time.
* The developer of the software and an independent test group conducts testing.
* Testing and Debugging are different activities but debugging must be accommodated in any testing strategy.

**INTEGRATION TESTING:**

Integration testing is a systematic technique for constructing the

program structure while at the same time conducting tests to uncover errors associated with. Individual modules, which are highly prone to interface errors,

should not be assumed to work instantly when we put them together. The problem of course, is “putting them together”- interfacing.

**PROGRAM TESTING:**

The logical and syntax errors have been pointed out by program testing. A syntax error is an error in a program statement that in violates one or more rules of the language in which it is written. An improperly defined field dimension or omitted keywords are common syntax error. These errors are shown through error messages generated by the computer. A logic error on the other hand deals with the incorrect data fields, out-off-range items and invalid combinations. Since the compiler s will not deduct logical error, the programmer must examine the output. Condition testing exercises the logical conditions contained in a module. The possible types of elements in a condition include a Boolean operator, Boolean variable, a pair of Boolean parentheses. A relational operator or on arithmetic expression. Condition testing method focuses on testing each condition in the program the purpose of condition test is to deduct not only errors in the condition of a program but also other a errors in the program.

**SECURITY TESTING:**

Security testing attempts to verify the protection mechanisms built in to a system well, in fact, protect it from improper penetration. The system security must be tested for invulnerability from frontal attack must also be tested for invulnerability from rear attack. During security, the tester places the role of individual who desires to penetrate system.

61

**VALIDATION TESTING**

At the culmination of integration testing, software is completely assembled as a package. Interfacing errors have been uncovered and corrected and a final series of software test-validation testing begins. Validation testing can be defined in many ways, but a simple definition is that validation succeeds when the software functions in manner that is reasonably expected by the customer. Software validation is achieved through a series of black box tests that demonstrate conformity with requirement. After validation test has been conducted, one of two conditions exists.

* The function or performance characteristics confirm to specifications and are accepted.
* A validation from specification is uncovered and a deficiency created.

Deviation or errors discovered at this step in this project is corrected prior to completion of the project with the help of the user by negotiating to establish a method for resolving deficiencies. Thus the proposed system under consideration has been tested by using validation testing and found to be working satisfactorily. Though there were deficiencies in the system they were not catastrophic

**USER ACCEPTANCE TESTING**

User acceptance of the system is key factor for the success of any system. The system under consideration is tested for user acceptance by constantly keeping in touch with prospective system and user at the time of developing and making changes whenever required. This is done in regarding to the following points.

* Input screen design.
* Output screen design.

62

**CHAPTER 9**

**IMPLEMENTATION AND RESULTS**

**SOURCE CODE:**

**Main.py**:

import en\_core\_web\_sm

import json

import numpy as np

import random

import re

import torch

from transformers import (

AutoTokenizer,

AutoModelForSeq2SeqLM,

AutoModelForSequenceClassification,

)

from typing import Any, List, Mapping, Tuple

class QuestionGenerator:

"""A transformer-based NLP system for generating reading comprehension-style questions from

texts. It can generate full sentence questions, multiple choice questions, or a mix of the

two styles.

To filter out low quality questions, questions are assigned a score and ranked once they have

been generated. Only the top k questions will be returned. This behaviour can be turned off

by setting use\_evaluator=False.

"""

def \_\_init\_\_(self) -> None:

QG\_PRETRAINED = "iarfmoose/t5-base-question-generator"

self.ANSWER\_TOKEN = "<answer>"

self.CONTEXT\_TOKEN = "<context>"

self.SEQ\_LENGTH = 512

self.device = torch.device(

"cuda" if torch.cuda.is\_available() else "cpu")

self.qg\_tokenizer = AutoTokenizer.from\_pretrained(

QG\_PRETRAINED, use\_fast=False)

self.qg\_model = AutoModelForSeq2SeqLM.from\_pretrained(QG\_PRETRAINED)

self.qg\_model.to(self.device)

self.qg\_model.eval()

self.qa\_evaluator = QAEvaluator()

def generate(

self,

article: str,

use\_evaluator: bool = True,

num\_questions: bool = None,

answer\_style: str = "all"

) -> List:

"""Takes an article and generates a set of question and answer pairs. If use\_evaluator

is True then QA pairs will be ranked and filtered based on their quality. answer\_style

should selected from ["all", "sentences", "multiple\_choice"].

"""

print("Generating questions...\n")

qg\_inputs, qg\_answers = self.generate\_qg\_inputs(article, answer\_style)

generated\_questions = self.generate\_questions\_from\_inputs(qg\_inputs)

message = "{} questions doesn't match {} answers".format(

len(generated\_questions), len(qg\_answers)

)

assert len(generated\_questions) == len(qg\_answers), message

if use\_evaluator:

print("Evaluating QA pairs...\n")

encoded\_qa\_pairs = self.qa\_evaluator.encode\_qa\_pairs(

generated\_questions, qg\_answers

)

scores = self.qa\_evaluator.get\_scores(encoded\_qa\_pairs)

if num\_questions:

qa\_list = self.\_get\_ranked\_qa\_pairs(

generated\_questions, qg\_answers, scores, num\_questions

)

else:

qa\_list = self.\_get\_ranked\_qa\_pairs(

generated\_questions, qg\_answers, scores

)

else:

print("Skipping evaluation step.\n")

qa\_list = self.\_get\_all\_qa\_pairs(generated\_questions, qg\_answers)

return qa\_list

def generate\_qg\_inputs(self, text: str, answer\_style: str) -> Tuple[List[str], List[str]]:

"""Given a text, returns a list of model inputs and a list of corresponding answers.

Model inputs take the form "answer\_token <answer text> context\_token <context text>" where

the answer is a string extracted from the text, and the context is the wider text surrounding

the context.

"""

VALID\_ANSWER\_STYLES = ["all", "sentences", "multiple\_choice"]

if answer\_style not in VALID\_ANSWER\_STYLES:

raise ValueError(

"Invalid answer style {}. Please choose from {}".format(

answer\_style, VALID\_ANSWER\_STYLES

)

)

inputs = []

answers = []

if answer\_style == "sentences" or answer\_style == "all":

segments = self.\_split\_into\_segments(text)

for segment in segments:

sentences = self.\_split\_text(segment)

prepped\_inputs, prepped\_answers = self.\_prepare\_qg\_inputs(

sentences, segment

)

inputs.extend(prepped\_inputs)

answers.extend(prepped\_answers)

if answer\_style == "multiple\_choice" or answer\_style == "all":

sentences = self.\_split\_text(text)

prepped\_inputs, prepped\_answers = self.\_prepare\_qg\_inputs\_MC(

sentences

)

inputs.extend(prepped\_inputs)

answers.extend(prepped\_answers)

return inputs, answers

def generate\_questions\_from\_inputs(self, qg\_inputs: List) -> List[str]:

"""Given a list of concatenated answers and contexts, with the form:

"answer\_token <answer text> context\_token <context text>", generates a list of

questions.

"""

generated\_questions = []

for qg\_input in qg\_inputs:

question = self.\_generate\_question(qg\_input)

generated\_questions.append(question)

return generated\_questions

def \_split\_text(self, text: str) -> List[str]:

"""Splits the text into sentences, and attempts to split or truncate long sentences."""

MAX\_SENTENCE\_LEN = 128

sentences = re.findall(".\*?[.!\?]", text)

cut\_sentences = []

for sentence in sentences:

if len(sentence) > MAX\_SENTENCE\_LEN:

cut\_sentences.extend(re.split("[,;:)]", sentence))

# remove useless post-quote sentence fragments

cut\_sentences = [s for s in sentences if len(s.split(" ")) > 5]

sentences = sentences + cut\_sentences

return list(set([s.strip(" ") for s in sentences]))

def \_split\_into\_segments(self, text: str) -> List[str]:

"""Splits a long text into segments short enough to be input into the transformer network.

Segments are used as context for question generation.

"""

MAX\_TOKENS = 490

paragraphs = text.split("\n")

tokenized\_paragraphs = [

self.qg\_tokenizer(p)["input\_ids"] for p in paragraphs if len(p) > 0

]

segments = []

while len(tokenized\_paragraphs) > 0:

segment = []

while len(segment) < MAX\_TOKENS and len(tokenized\_paragraphs) > 0:

paragraph = tokenized\_paragraphs.pop(0)

segment.extend(paragraph)

segments.append(segment)

return [self.qg\_tokenizer.decode(s, skip\_special\_tokens=True) for s in segments]

def \_prepare\_qg\_inputs(

self,

sentences: List[str],

text: str

) -> Tuple[List[str], List[str]]:

"""Uses sentences as answers and the text as context. Returns a tuple of (model inputs, answers).

Model inputs are "answer\_token <answer text> context\_token <context text>"

"""

inputs = []

answers = []

for sentence in sentences:

qg\_input = f"{self.ANSWER\_TOKEN} {sentence} {self.CONTEXT\_TOKEN} {text}"

inputs.append(qg\_input)

answers.append(sentence)

return inputs, answers

def \_prepare\_qg\_inputs\_MC(self, sentences: List[str]) -> Tuple[List[str], List[str]]:

"""Performs NER on the text, and uses extracted entities are candidate answers for multiple-choice

questions. Sentences are used as context, and entities as answers. Returns a tuple of (model inputs, answers).

Model inputs are "answer\_token <answer text> context\_token <context text>"

"""

spacy\_nlp = en\_core\_web\_sm.load()

docs = list(spacy\_nlp.pipe(sentences, disable=["parser"]))

inputs\_from\_text = []

answers\_from\_text = []

for doc, sentence in zip(docs, sentences):

entities = doc.ents

if entities:

for entity in entities:

qg\_input = f"{self.ANSWER\_TOKEN} {entity} {self.CONTEXT\_TOKEN} {sentence}"

answers = self.\_get\_MC\_answers(entity, docs)

inputs\_from\_text.append(qg\_input)

answers\_from\_text.append(answers)

return inputs\_from\_text, answers\_from\_text

def \_get\_MC\_answers(self, correct\_answer: Any, docs: Any) -> List[Mapping[str, Any]]:

"""Finds a set of alternative answers for a multiple-choice question. Will attempt to find

alternatives of the same entity type as correct\_answer if possible.

"""

entities = []

for doc in docs:

entities.extend([{"text": e.text, "label\_": e.label\_} for e in doc.ents])

# Remove duplicate elements and convert to a list

entities\_json = [json.dumps(kv) for kv in entities]

pool = sorted(set(entities\_json)) # Convert pool to a sorted list

num\_choices = min(4, len(pool)) - 1 # Number of choices to make

# Add the correct answer

final\_choices = []

correct\_label = correct\_answer.label\_

final\_choices.append({"answer": correct\_answer.text, "correct": True})

# Remove the correct answer from the pool

pool = [e for e in pool if e != json.dumps({"text": correct\_answer.text, "label\_": correct\_answer.label\_})]

# Find answers with the same NER label

matches = [e for e in pool if correct\_label in e]

# If not enough matches, add other random answers

if len(matches) < num\_choices:

choices = matches

remaining\_choices = random.sample(sorted(pool), num\_choices - len(choices))

choices.extend(remaining\_choices)

else:

choices = random.sample(sorted(matches), num\_choices)

choices = [json.loads(s) for s in choices]

for choice in choices:

final\_choices.append({"answer": choice["text"], "correct": False})

random.shuffle(final\_choices)

return final\_choices

@torch.no\_grad()

def \_generate\_question(self, qg\_input: str) -> str:

"""Takes qg\_input which is the concatenated answer and context, and uses it to generate

a question sentence. The generated question is decoded and then returned.

"""

encoded\_input = self.\_encode\_qg\_input(qg\_input)

output = self.qg\_model.generate(input\_ids=encoded\_input["input\_ids"])

question = self.qg\_tokenizer.decode(

output[0],

skip\_special\_tokens=True

)

return question

def \_encode\_qg\_input(self, qg\_input: str) -> torch.tensor:

"""Tokenizes a string and returns a tensor of input ids corresponding to indices of tokens in

the vocab.

"""

return self.qg\_tokenizer(

qg\_input,

padding='max\_length',

max\_length=self.SEQ\_LENGTH,

truncation=True,

return\_tensors="pt",

).to(self.device)

def \_get\_ranked\_qa\_pairs(

self, generated\_questions: List[str], qg\_answers: List[str], scores, num\_questions: int = 10

) -> List[Mapping[str, str]]:

"""Ranks generated questions according to scores, and returns the top num\_questions examples.

"""

if num\_questions > len(scores):

num\_questions = len(scores)

print((

f"\nWas only able to generate {num\_questions} questions.",

"For more questions, please input a longer text.")

)

qa\_list = []

for i in range(num\_questions):

index = scores[i]

qa = {

"question": generated\_questions[index].split("?")[0] + "?",

"answer": qg\_answers[index]

}

qa\_list.append(qa)

return qa\_list

def \_get\_all\_qa\_pairs(self, generated\_questions: List[str], qg\_answers: List[str]):

"""Formats question and answer pairs without ranking or filtering."""

qa\_list = []

for question, answer in zip(generated\_questions, qg\_answers):

qa = {

"question": question.split("?")[0] + "?",

"answer": answer

}

qa\_list.append(qa)

return qa\_list

class QAEvaluator:

"""Wrapper for a transformer model which evaluates the quality of question-answer pairs.

Given a QA pair, the model will generate a score. Scores can be used to rank and filter

QA pairs.

"""

def \_\_init\_\_(self) -> None:

QAE\_PRETRAINED = "iarfmoose/bert-base-cased-qa-evaluator"

self.SEQ\_LENGTH = 512

self.device = torch.device(

"cuda" if torch.cuda.is\_available() else "cpu")

self.qae\_tokenizer = AutoTokenizer.from\_pretrained(QAE\_PRETRAINED)

self.qae\_model = AutoModelForSequenceClassification.from\_pretrained(

QAE\_PRETRAINED

)

self.qae\_model.to(self.device)

self.qae\_model.eval()

def encode\_qa\_pairs(self, questions: List[str], answers: List[str]) -> List[torch.tensor]:

"""Takes a list of questions and a list of answers and encodes them as a list of tensors."""

encoded\_pairs = []

for question, answer in zip(questions, answers):

encoded\_qa = self.\_encode\_qa(question, answer)

encoded\_pairs.append(encoded\_qa.to(self.device))

return encoded\_pairs

def get\_scores(self, encoded\_qa\_pairs: List[torch.tensor]) -> List[float]:

"""Generates scores for a list of encoded QA pairs."""

scores = {}

for i in range(len(encoded\_qa\_pairs)):

scores[i] = self.\_evaluate\_qa(encoded\_qa\_pairs[i])

return [

k for k, v in sorted(scores.items(), key=lambda item: item[1], reverse=True)

]

def \_encode\_qa(self, question: str, answer: str) -> torch.tensor:

"""Concatenates a question and answer, and then tokenizes them. Returns a tensor of

input ids corresponding to indices in the vocab.

"""

if type(answer) is list:

for a in answer:

if a["correct"]:

correct\_answer = a["answer"]

else:

correct\_answer = answer

return self.qae\_tokenizer(

text=question,

text\_pair=correct\_answer,

padding="max\_length",

max\_length=self.SEQ\_LENGTH,

truncation=True,

return\_tensors="pt",

)

@torch.no\_grad()

def \_evaluate\_qa(self, encoded\_qa\_pair: torch.tensor) -> float:

"""Takes an encoded QA pair and returns a score."""

output = self.qae\_model(\*\*encoded\_qa\_pair)

return output[0][0][1]

def print\_qa(qa\_list: List[Mapping[str, str]], show\_answers: bool = True) -> None:

"""Formats and prints a list of generated questions and answers."""

for i in range(len(qa\_list)):

# wider space for 2 digit q nums

space = " " \* int(np.where(i < 9, 3, 4))

print(f"{i + 1}) Q: {qa\_list[i]['question']}")

answer = qa\_list[i]["answer"]

# print a list of multiple choice answers

if type(answer) is list:

if show\_answers:

print(

f"{space}A: 1. {answer[0]['answer']} "

f"{np.where(answer[0]['correct'], '(correct)', '')}"

)

for j in range(1, len(answer)):

print(

f"{space + ' '}{j + 1}. {answer[j]['answer']} "

f"{np.where(answer[j]['correct']==True,'(correct)', '')}"

)

else:

print(f"{space}A: 1. {answer[0]['answer']}")

for j in range(1, len(answer)):

print(f"{space + ' '}{j + 1}. {answer[j]['answer']}")

print("")

# print full sentence answers

else:

if show\_answers:

print(f"{space}A: {answer}\n")

**Healthpass flutter :**

import streamlit as st

import wikipedia

from haystack.document\_stores import InMemoryDocumentStore

from haystack.utils import clean\_wiki\_text, convert\_files\_to\_docs

from haystack.nodes import TfidfRetriever, FARMReader

from haystack.pipelines import ExtractiveQAPipeline

from main import print\_qa, QuestionGenerator

def main():

# Set the Streamlit app title

st.title("Question Generation using Haystack and Streamlit")

# Select the input type

inputs = ["Input Paragraph", "Wikipedia Examples"]

input\_type = st.selectbox("Select an input type:", inputs)

# Initialize wiki\_text as an empty string

wiki\_text = ""

# Handle different input types

if input\_type == "Input Paragraph":

# Allow user to input text paragraph

wiki\_text = st.text\_area("Input paragraph:", height=200)

elif input\_type == "Wikipedia Examples":

# Define topics for selection

topics = ["Deep Learning", "Machine Learning"]

selected\_topic = st.selectbox("Select a topic:", topics)

# Retrieve Wikipedia content based on the selected topic

if selected\_topic:

wiki = wikipedia.page(selected\_topic)

wiki\_text = wiki.content

# Display the retrieved Wikipedia content (optional)

st.text\_area("Retrieved Wikipedia content:", wiki\_text, height=200)

# Preprocess the input text

wiki\_text = clean\_wiki\_text(wiki\_text)

# Allow user to specify the number of questions to generate

num\_questions = st.slider("Number of questions to generate:", min\_value=1, max\_value=20, value=5)

# Allow user to specify the model to use

model\_options = ["deepset/roberta-base-squad2", "deepset/roberta-base-squad2-distilled", "bert-large-uncased-whole-word-masking-squad2", "deepset/flan-t5-xl-squad2"]

model\_name = st.selectbox("Select model:", model\_options)

# Button to generate questions

if st.button("Generate Questions"):

document\_store = InMemoryDocumentStore()

# Convert the preprocessed text into a document

document = {"content": wiki\_text}

document\_store.write\_documents([document])

# Initialize a TfidfRetriever

retriever = TfidfRetriever(document\_store=document\_store)

# Initialize a FARMReader with the selected model

reader = FARMReader(model\_name\_or\_path=model\_name, use\_gpu=False)

# Initialize the question generation pipeline

pipe = ExtractiveQAPipeline(reader, retriever)

# Initialize the QuestionGenerator

qg = QuestionGenerator()

# Generate multiple-choice questions

qa\_list = qg.generate(

wiki\_text,

num\_questions=num\_questions,

answer\_style='multiple\_choice'

)

# Display the generated questions and answers

st.header("Generated Questions and Answers:")

for idx, qa in enumerate(qa\_list):

# Display the question

st.write(f"Question {idx + 1}: {qa['question']}")

# Display the answer options

if 'answer' in qa:

for i, option in enumerate(qa['answer']):

correct\_marker = "(correct)" if option["correct"] else ""

st.write(f"Option {i + 1}: {option['answer']} {correct\_marker}")

# Add a separator after each question-answer pair

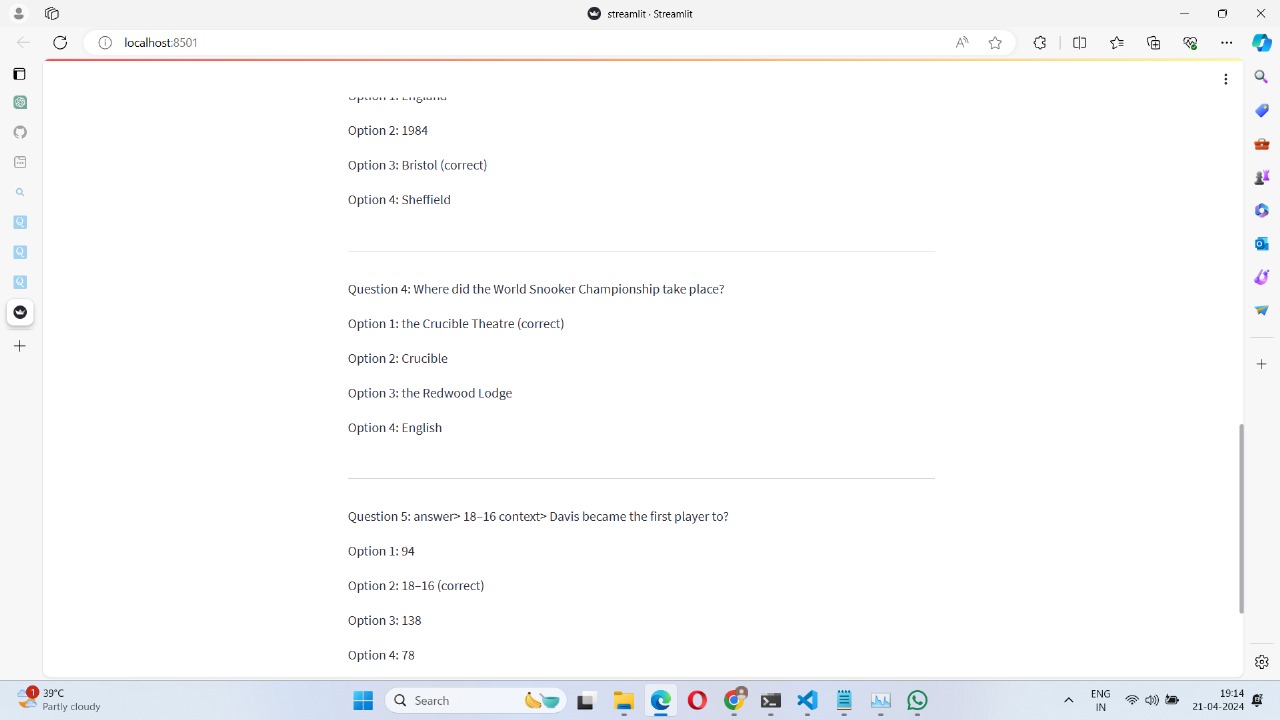
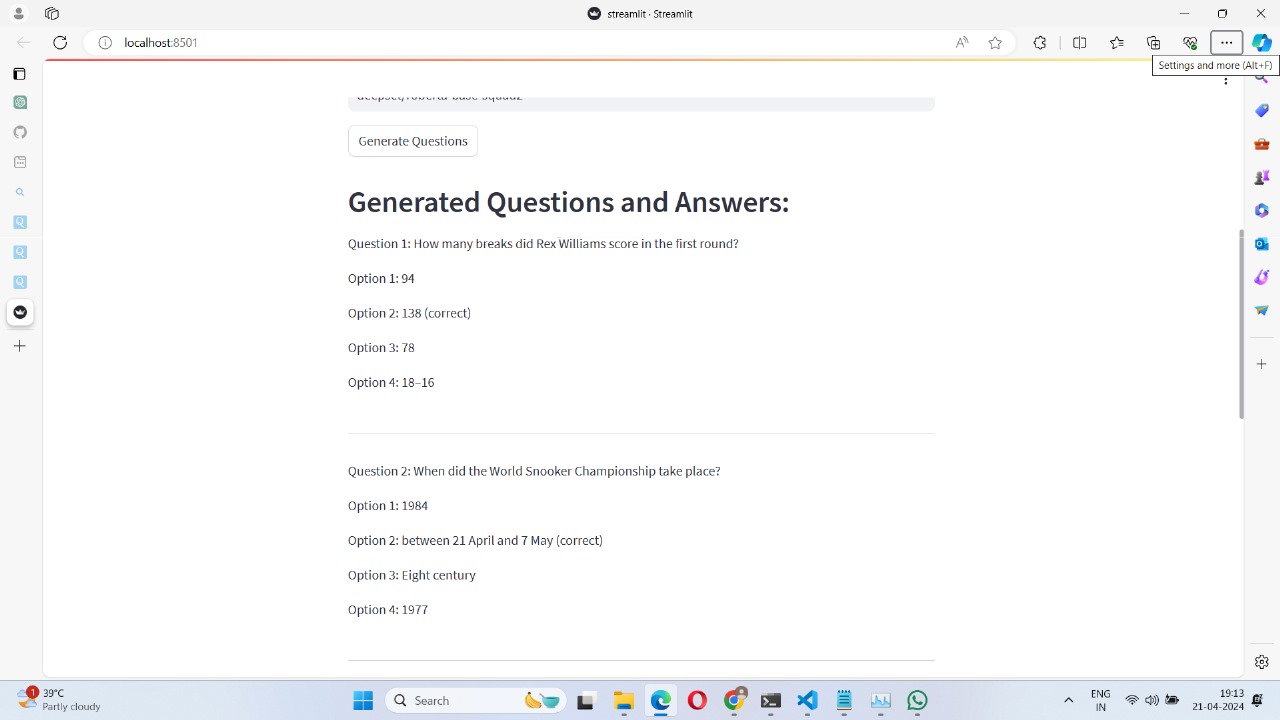
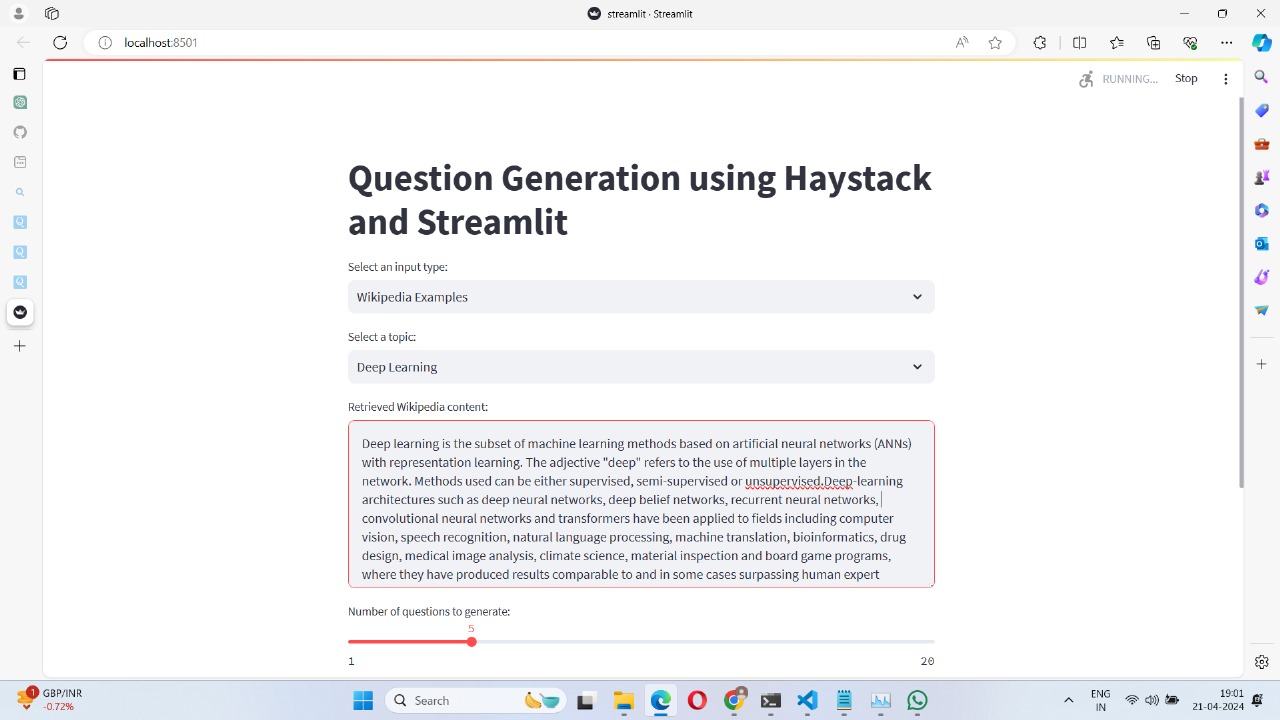
st.write("-" \* 40)

# Run the Streamlit app

if \_\_name\_\_ == "\_\_main\_\_":

main()

**SCREENSHOTS :**



**CHAPTER 10**

**CONCLUSION AND FUTURE WORKS**

**CONCLUSION**

This project has explored the design and potential of a Question Generator System that utilizes pre-trained Large Language Models (LLMs) to automatically generate questions from a given text document. The system offers functionalities such as:

Preprocessing text based on the user's chosen answer style (optional).

Generating question-answer pairs using a pre-trained Question Generator model.

Optionally evaluating the generated questions' quality using a pre-trained QA Evaluator model.

Ranking/filtering questions based on their scores (if evaluation is enabled).

Presenting the final list of question-answer pairs to the user.

**FUTURE WORKS**

Expanding answer style options: Explore additional answer styles beyond all answers, sentences as answers, and multiple-choice questions. This could include cloze deletions, open ended questions, or specific types tailored to the content (e.g., cause-and-effect, compare-and-contrast).Domain-specific knowledge integration: Train both the Question Generator and QA Evaluator models on domain-specific datasets to improve the quality and relevance of generated questions for particular fields (e.g., science, history, literature).User feedback integration: Implement a mechanism for users to provide feedback on the generated questions. This feedback could be used to refine the models and enhance the system's overall performance.

**REFERENCES**

Research Papers:

[1] "Question Generation from Text Using Transformers" by Zhu et al. (2019): https://arxiv.org/pdf/2210.09467 - This paper explores using Transformer-based models for question generation from text documents.

[2] "BART: Pre-training of Deep Bidirectional Transformers for Language Understanding and Generation" by Lewis et al. (2019): https://arxiv.org/pdf/1910.13461 - This paper introduces the BART model, a pre-trained model for various NLP tasks, including question generation.

[3] "Training Conversational Question Answering with Limited Data" by Wu et al. (2020): https://arxiv.org/pdf/2204.04581 - This paper explores training question generation models with limited data for conversational settings.

[4] "REAL: Realistic and Explainable Adversarial Loss for Question Generation" by Liu et al. (2021): https://arxiv.org/pdf/1912.04497 - This paper proposes a novel loss function for improving the realism and explainability of generated questions by LLMs.

[5] Hugging Face Transformers: https://huggingface.co/docs/transformers/en/index - This library provides pre-trained models for various NLP tasks, including question generation models like BART. It also offers tools for fine-tuning these models on your own data.

[6] Gradio: https://www.gradio.app/guides/quickstart - This is a framework for building user interfaces for machine learning models. You can use Gradio to create a web interface for your question generation system

[7] Streamlit: https://streamlit.io/ - Another framework for building user interfaces for data science applications. You can leverage Streamlit to create a user-friendly interface for interacting with your question generation system.

Blogs and Articles:

[8] "Question Generation with Transformers" by Hugging Face: https://huggingface.co/models?other=question-generation - This blog post provides a tutorial on using Hugging Face Transformers for question generation.