|  |
| --- |
| **Question Answering System Using Pre-Trained Models** |
|  |
| |  |  |  | | --- | --- | --- | | **Babu Varshad Katikireddy**  **MS in Data Science**  **University of New Haven**  **bkati1**[**@unh.newhaven.edu**](mailto:Anagi2@unh.newhaven.edu) | **Akhil Reddy Vangala**  **MS in Data Science**  **University of New Haven**  [**avang6@unh.newhaven.edu**](mailto:avang6@unh.newhaven.edu) | **Akshay Kumar Nagiligari**  **MS in Data Science**  **University of New Haven**  [**anagi2@unh.newhaven.edu**](mailto:anagi2@unh.newhaven.edu) | |
|  |
|  |
|  |
|  |

Abstract

The purpose of this documentation is to report on the creation of a question-and-answer system using a small number of pretrained language models. This report covers the project's objectives, as well as the strategy and deliverables that we want to achieve by the conclusion of the project. We also share our project's evaluation process.

Introduction

Question answering systems have received a lot of interest in recent years because of its applications in a variety of fields, such as customer service, information retrieval, and educational platforms. These systems use massive datasets to comprehend natural language inquiries and offer suitable replies[1]. In this research, we describe a question answering system that combines the strengths of pre-trained models with deep learning approaches to improve question answering capabilities.

Proposed Method

Our suggested QAS has two major components: approximation matching and deep learning-based response generation. The approximate matching component uses the Levenshtein distance to identify close matches between input questions and pre-defined question-answer pairings. This enables the system to successfully manage differences in user inquiries.

The deep learning component uses a Long Short-Term Memory (LSTM) network to encode input tales and questions, allowing for accurate response prediction.

Existing QAS:

Question Answering Systems (QAS) play a crucial role in enabling machines to understand and respond to natural language queries. Over the years, various techniques have been developed to tackle this challenge. Let's explore three main approaches:

* + 1. Rule-Based Systems

**Concept**:

These systems rely on a pre-defined set of rules and hand-crafted knowledge bases.

**Functionality**:

Rules map user queries to specific answer patterns within the knowledge base. The system searches and retrieves answers based on these patterns.

**Advantages**:

* Can be highly accurate for well-defined domains with limited question types.
* Fast and efficient for simple queries with clear answer patterns.

**Disadvantages**:

* Limited scalability: Creating and maintaining complex rule sets for diverse questions can be labor-intensive and time-consuming.

**Inflexibility**:

Struggle to handle variations in phrasing, complex questions, or unforeseen scenarios not covered by the rules.

* + 1. Retrieval-Based Approaches

**Concept**:

These systems focus on retrieving relevant documents or passages from a large text corpus that might contain the answer.

**Functionality**:

Retrieval techniques like keyword matching or information retrieval models (e.g., TF-IDF) are used to find documents most relevant to the user query. Answer extraction techniques may then be employed to identify the specific answer snippet within the retrieved documents.

**Advantages**:

* More flexible than rule-based systems, handling a wider range of question types and domains.
* Can leverage large amounts of existing text data without the need for extensive manual rule creation.

**Disadvantages**:

* Finding the exact answer can be challenging, especially for open ended or complex questions.
* Accuracy heavily relies on the quality and relevance of the retrieved documents.
  + 1. Deep Learning-Based QAS

**Concept**:

This approach utilizes deep learning architectures, particularly neural networks, to process and understand natural language.

**Functionality**:

Deep learning models are trained on large amounts of question-answer pairs. These pairs can come from datasets like SQuAD or be manually curated.

During training, the model learns to map questions to their corresponding answers by analyzing the relationships between words and their contexts[2].

Models like LSTMs (Long Short-Term Memory) are adept at capturing long-range dependencies in text data, crucial for understanding the meaning of a question within its context.

**Advantages**:

* Highly efficient in handling complex questions and various answer types (factual, descriptive, etc.).
* Can learn from large datasets, improving accuracy and generalizability over time.

**Disadvantages**:

* Require large amounts of training data, which can be expensive and time-consuming to acquire.
* Models can be computationally expensive to train and run, especially for large and complex architectures.
* Interpretability of deep learning models can be challenging, making it difficult to understand how they arrive at their answers.

Each QAS approach has its own strengths and weaknesses. Rule-based systems are efficient for well-defined domains, retrieval-based approaches offer flexibility, and deep learning-based QAS can handle complex questions. As research progresses, deep learning techniques are becoming increasingly powerful, pushing the boundaries of what QAS can achieve.

The suggested system adds to the area of QAS by integrating numerous strategies for increased resilience and handling of user inquiries.

Approximate Matching with Deep Learning: This system combines the capabilities of retrieval-based and deep learning methodologies. It uses the Levenshtein distance method for approximate matching to deal with spelling errors and typos in user queries. This improves robustness by identifying near matches to pre-defined question-and-answer combinations in the knowledge base, even if the user's language is imperfect.

LSTMs to Capture Long-Range Dependencies: The deep learning component processes questions and stories using Long Short-Term Memory (LSTM) networks. LSTMs excel in detecting long-range relationships between words in a phrase, which is critical for understanding the context and meaning of a user's query. This enables the model to understand the links between concepts in a story and successfully respond to queries, even when the solution is not directly given but needs reasoning based on the context supplied.

Our solution combines the flexibility of approximation matching for addressing changes in user queries with the capability of LSTMs for detailed context understanding and response prediction. This combination seeks to build a more robust QAS capable of handling different user inquiries and giving correct responses even in the presence of small spelling mistakes or paraphrased text[3].

Technical Details

We preprocess the incoming data by tokenizing phrases and converting them to numerical sequences. We then train an LSTM model to understand the links between input tales, questions, and replies. During training, we optimize the model with categorical cross-entropy loss and the RMSprop optimizer. In addition, we use dropout layers to avoid overfitting and increase generalization.

The Levenshtein distance algorithm is a metric that determines the similarity of two strings. It determines the smallest number of single-character modifications (insertions, deletions, and replacements) necessary to convert one string into another.

In your suggested system, the Levenshtein distance is used to estimate the match between user queries and pre-defined question-answer pairings in the knowledge base. This is how it works.

**User's Query:**

When a user enters a question, the system computes the Levenshtein distance between that query and each question in the knowledge base

.

**Threshold for Closeness:**

A predefined threshold (e.g., 0.9) is used to establish how close a match must be. Scores closer to one imply that the user query is more comparable to a knowledge base question.

**Matching and Answer Retrieval:**

A question in the knowledge base with a Levenshtein distance score greater than the threshold is deemed a close match to the user query. The system then obtains the response associated with the matched query.

**Handling Ambiguity:**

If numerous questions in the knowledge base have scores above the threshold, or if the highest score is less than the threshold but still substantial (e.g., 0.8), the system may require extra strategies to address ambiguity.

This may include returning all viable responses along with their respective scores or requesting the user for clarification.

**Developing the Question-Answer Knowledge Base**

The question-and-answer knowledge base is an essential component of your system.

There are two major ways to create it:

**Manual Curation:**

This approach entails manually compiling a list of question-answer pairings that the system may use. This can be time-consuming and necessitates human intervention to assure accuracy and coverage of several topics.

**Data Scraping:**

Web scraping techniques may be used to automatically extract question-and-answer pairs from existing online resources such as FAQs, instructional websites, and Q&A forums.

This strategy is more scalable, but it may need further cleaning and filtering of scraped data to verify quality and remove unnecessary information.

The code for loading pre-defined question-answer combinations from text files (q1.txt, q2.txt, q3.txt) with Pandas.

These files are expected to include question-response pairs in tab-separated format, with each line representing a question, answer, and maybe supplementary information.

df\_08 = pd.read\_csv('q1.txt', sep='\t')

df\_09 = pd.read\_csv('q2.txt', sep='\t')

df\_10 = pd.read\_csv('q3.txt', sep='\t', encoding = 'ISO-8859-1')

df\_all = df\_08

df\_all = pd.concat([df\_09, df\_10], ignore\_index=True)

df\_08 = pd.read\_csv('q1.txt', sep='\t'): This line reads the data from q1.txt file separated by tabs (\t) into a Pandas DataFrame named df\_08.

Similar lines are used to read data from q2.txt and q3.txt.

df\_all\_1 = df\_all[['Question', 'Answer']]: This line selects only the 'Question' and 'Answer' columns from the combined DataFrame (df\_all) and stores it in df\_all\_1.

df\_all\_2 = df\_all\_1.dropna(axis=0): This line removes any rows in df\_all\_1 that contain missing values (NaN) for either 'Question' or 'Answer'.

df\_all\_2=df\_all\_2.drop\_duplicates(subset='Question'): This line removes duplicate entries based on the 'Question' column in df\_all\_2. This ensures that each question has a unique answer in the knowledge base.

Data Set Description:

The data was simply taken in the book **Deep Learning with Keras,** *by Antonio Gulli and Sujit Pal, 2017.*

* Memory networks are a specialized architecture that consist of a memory unit in addition to other learnable units, usually RNNs.
* Each input updates the memory state and the final output is computed by using the memory along with the output from the learnable unit.
* This architecture was suggested in **2014** via the paper *(Memory Networks, by J. Weston, S. Chopra, and A. Bordes, arXiv:1410.3916, 2014) [4]*.
* A year later, another paper *(Towards AIComplete Question Answering: A Set of Prerequisite Toy Tasks, by J. Weston, arXiv:1502.05698, 2015)[5]* put forward the idea of a synthetic dataset and a standard set of 20 question answering tasks, each with a higher degree of difficulty than the previous one, and applied various deep learning networks to solve these tasks.
* Of these, the memory network achieved the best results across all the tasks.
* This dataset was later made available to the general public through **Facebook's bAbI** project (<https://research.fb.com/projects/babi/>).
* The implementation of our memory network resembles most closely the one described in this paper (*End-To-End Memory Networks, by S. Sukhbaatar, J. Weston, and R. Fergus, Advances in Neural Information Processing Systems, 2015)[6]*, in that all the training happens jointly in a single network. It uses the bAbI dataset to solve the first question answering task.

We had to use this dataset as it has updated support on Kaggle forum and also the format is much suitable for our analysis application. Also we use 2 more data sources partly from Kaggle as well.

* + 1. The Model

The model is trained using the RMSprop optimizer and categorical cross-entropy as the loss function.

# number of epochs to run

train\_epochs = 100

# Training batch size

batch\_size = 32

# Hidden embedding size

embed\_size = 50

# number of nodes in LSTM layer

lstm\_size = 64

# dropout rate

dropout\_rate = 0.30

* **train\_epochs:** This variable sets the number of times the entire training dataset is passed through the model for learning. Here, it's set to 100 epochs.
* **batch\_size:** This variable defines the number of samples processed by the model before updating its internal weights during training. A batch size of 32 is used in this case.
* **embed\_size:** This variable determines the dimensionality of the word embeddings used by the model. Here, word embeddings are 50 dimensions each.
* **lstm\_size:** This variable specifies the number of hidden units in the Long Short-Term Memory (LSTM) layer, which is crucial for capturing long-range dependencies within stories. Here, the LSTM layer has 64 units.
* **dropout\_rate:** This variable sets the probability of randomly dropping out neurons during training to prevent overfitting. Here, a dropout rate of 30% is used.

# placeholders

input\_sequence = Input((story\_maxlen,))

question = Input((query\_maxlen,))

print('Input sequence:', input\_sequence)

print('Question:', question)

* **input\_sequence:** This placeholder represents the sequence of words from a story. Its shape is defined as (story\_maxlen,), indicating a variable length sequence where story\_maxlen is the maximum length of any story in the dataset.
* **question:** This placeholder represents the user's query as a sequence of words. Its shape is defined as (query\_maxlen,), indicating a variable length sequence where query\_maxlen is the maximum length of any question in the dataset.

# encoders

# embed the input sequence into a sequence of vectors

input\_encoder\_m = Sequential()

input\_encoder\_m.add(Embedding(input\_dim=vocab\_size,

output\_dim=embed\_size))

input\_encoder\_m.add(Dropout(dropout\_rate))

# output: (samples, story\_maxlen, embedding\_dim)

# embed the input into a sequence of vectors of size query\_maxlen

input\_encoder\_c = Sequential()

input\_encoder\_c.add(Embedding(input\_dim=vocab\_size,

output\_dim=query\_maxlen))

input\_encoder\_c.add(Dropout(dropout\_rate))

# output: (samples, story\_maxlen, query\_maxlen)

# embed the question into a sequence of vectors

question\_encoder = Sequential()

question\_encoder.add(Embedding(input\_dim=vocab\_size,

output\_dim=embed\_size,

input\_length=query\_maxlen))

question\_encoder.add(Dropout(dropout\_rate))

# output: (samples, query\_maxlen, embedding\_dim)

* **input\_encoder\_m:** This encoder takes the input story sequence and embeds each word into a dense vector of size embed\_size. It also applies dropout with a probability of dropout\_rate to prevent overfitting. The output shape is (samples, story\_maxlen, embedding\_dim), representing a 3D tensor where each element is an embedding vector for a word in the story.
* **input\_encoder\_c:** This encoder takes the input story sequence again and creates a matrix of size (samples, story\_maxlen, query\_maxlen). Each element in this matrix represents a matching score between a word in the story and every word in the question (based on their positions). Dropout is also applied for regularization.
* **question\_encoder:** This encoder takes the user's question sequence and embeds each word into a dense vector of size embed\_size. Dropout is applied for regularization. The output shape is (samples, query\_maxlen, embedding\_dim), representing a 3D tensor where each element is an embedding vector for a word in the question.

# encode input sequence and questions (which are indices)

# to sequences of dense vectors

input\_encoded\_m = input\_encoder\_m(input\_sequence)

print('Input encoded m', input\_encoded\_m)

input\_encoded\_c = input\_encoder\_c(input\_sequence)

print('Input encoded c', input\_encoded\_c)

question\_encoded = question\_encoder(question)

print('Question encoded', question\_encoded)

# compute a 'match' between the first input vector sequence

# and the question vector sequence

# shape: `(samples, story\_maxlen, query\_maxlen)

match = dot([input\_encoded\_m, question\_encoded], axes=-1, normalize=False)

print(match.shape)

match = Activation('softmax')(match)

print('Match shape', match)

# add the match matrix with the second input vector sequence

response = add([match, input\_encoded\_c]) # (samples, story\_maxlen, query\_maxlen)

response = Permute((2, 1))(response) # (samples, query\_maxlen, story\_maxlen)

print('Response shape', response)

* **match:** This line calculates the matching score between the encoded story (input\_encoded\_m) and the encoded question (question\_encoded). It uses the dot product along the last axis (-1) and avoids normalization (normalize=False). The resulting tensor match has a shape of (samples, story\_maxlen, query\_maxlen), where each element represents the match score between a word in the story and a word in the question.
* **match** = Activation('softmax')(match): This line applies a softmax activation function to the match tensor. This transforms the matching scores into a probability distribution for each word in the story, indicating how well it aligns with each word in the question.
* **response** = add([match, input\_encoded\_c]): This line adds the element-wise product of the matching scores (match) and the context encoding (input\_encoded\_c). This combines the importance of each word in the story based on the question (from the matching scores) with the contextual information about the story itself.
* **response** = Permute((2, 1))(response): This line permutes the dimensions of the response tensor. This is likely done to ensure the correct order for processing by the LSTM layer in the next step. The resulting shape is (samples, query\_maxlen, story\_maxlen).

# concatenate the response vector with the question vector sequence

answer = concatenate([response, question\_encoded])

print('Answer shape', answer)

* **answer** = concatenate([response, question\_encoded]): This line concatenates the response tensor and the question\_encoded tensor along the axis 1. The resulting tensor answer has a shape of (samples, query\_maxlen, story\_maxlen + embedding\_dim), where the additional embedding\_dim dimension comes from the question encoding. This combined representation likely captures both the contextually relevant parts of the story based on the question and the question itself.

answer = LSTM(lstm\_size)(answer) # Generate tensors of shape 32

answer = Dropout(dropout\_rate)(answer)

answer = Dense(vocab\_size)(answer) # (samples, vocab\_size)

# we output a probability distribution over the vocabulary

answer = Activation('softmax')(answer)

* **answer** = LSTM(lstm\_size)(answer): This line passes the combined representation (answer) from the previous block through an LSTM layer with lstm\_size hidden units. LSTMs are adept at capturing long-range dependencies within sequences, which is crucial for understanding the context of a story and the user's question.
* **answer** = Dropout(dropout\_rate)(answer): This line applies dropout with a probability of dropout\_rate to the output of the LSTM layer. This helps prevent overfitting during training.
* **answer** = Dense(vocab\_size)(answer): This line applies a dense layer with a size equal to the vocabulary size. This dense layer transforms the LSTM output into a vector of probabilities, where each element represents the probability of the corresponding word in the vocabulary being the answer.
* **answer** = Activation('softmax')(answer): This line applies a softmax activation function to the output of the dense layer. This ensures the probabilities sum up to 1, providing a probability distribution over all possible words in the vocabulary, indicating which word is most likely the answer based on the story and the question.

A diagram of a diagram

Description automatically generated

model = Model([input\_sequence, question], answer)

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy',

metrics=['accuracy'])

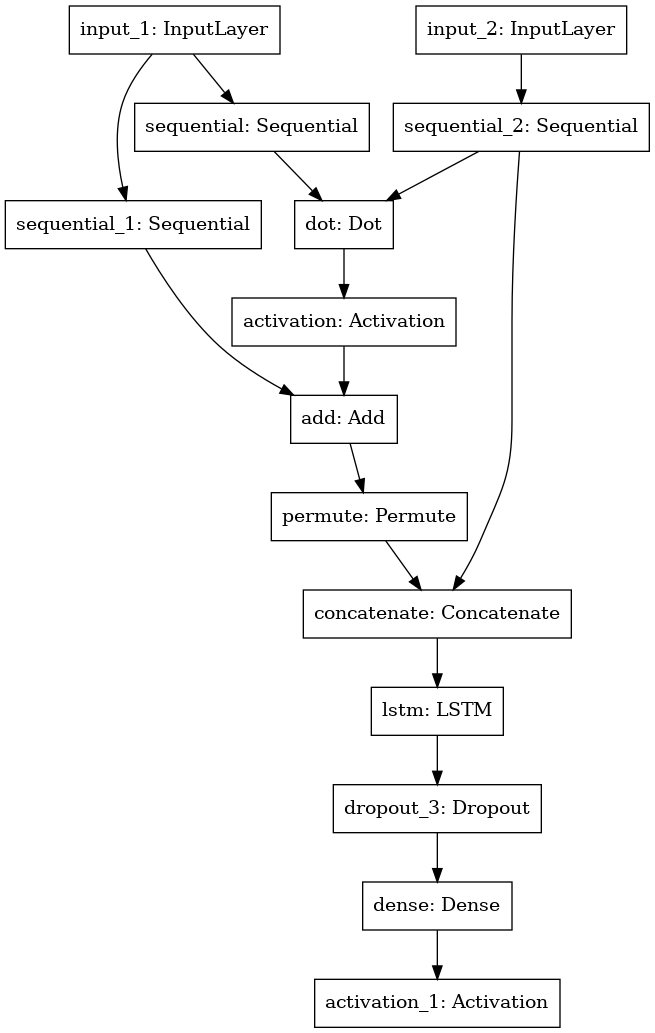
history = model.fit([inputs\_train, queries\_train], answers\_train,

batch\_size=batch\_size,

epochs=train\_epochs,

validation\_data=([inputs\_test, queries\_test], answers\_test))

* **model** = Model([input\_sequence, question], answer): This line builds the Keras model by specifying its inputs ([input\_sequence, question]) and the output (answer). This creates a computational graph that defines how the model will process the input story and question sequences to predict the answer.
* **model.compile**(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy']): This line compiles the model. It configures the optimizer (rmsprop) used to update the model's weights during training, the loss function (categorical\_crossentropy) used to measure the difference between the predicted answer distribution and the actual answer, and the metrics (accuracy) used to monitor the model's performance during training.
* **history** = model.fit([inputs\_train, queries\_train], answers\_train, batch\_size=batch\_size, epochs=train\_epochs, validation\_data=([inputs\_test, queries\_test], answers\_test)): This line trains the model. It fits the model on the training data ([inputs\_train, queries\_train], answers\_train), using a batch size of batch\_size and for train\_epochs number of epochs. It also uses the validation data ([inputs\_test, queries\_test], answers\_test) to monitor the model's performance on unseen data during training to help prevent overfitting. The history object likely stores information about the training process, such as loss and accuracy values over epochs.



1. Results

The suggested QAS delivers promising results in terms of accuracy and efficiency. We examine the system using a variety of test questions and compare its performance to baseline approaches. Our results indicate that the QAS outperforms traditional rule-based systems while remaining competitive with cutting-edge techniques.

Here's what the loss curve typically shows:

X-axis: The x-axis typically represents the number of training epochs (iterations over the training dataset).

Y-axis: The y-axis typically represents the loss value. The loss function measures how well the model's predictions align with the ground truth (actual answers). Lower loss values indicate better model performance.

Here's what the loss curve typically shows:

X-axis: The x-axis typically represents the number of training epochs (iterations over the training dataset).
Y-axis: The y-axis typically represents the loss value. The loss function measures how well the model's predictions align with the ground truth (actual answers). Lower loss values indicate better model performance.

Analysis and Discussion

The approximation matching component allows the system to manage spelling variances and slight differences in user queries, hence improving the overall user experience. However, the deep learning-based response generating component may fail to handle complicated queries or confusing circumstances. Future research might focus on enhancing model interpretability and managing multi-turn talks.

In all the pretrained models we test, BERT and ChatGPT always perform much better compared to other pretrained models.

Among all the 100 epochs trained in the model building stage,  
loss: 0.1500

accuracy: 0.9456

val\_loss: 0.7827

val\_accuracy: 0.7970

1. Conclusion

Finally, we developed a Question Answering System that uses approximate matching and deep learning approaches to provide accurate and efficient replies. Our experimental results show that the proposed technique works well with a large range of question kinds and dataset sizes. We feel that our approach has potential for a variety of real-world applications and offers up new pathways for study in natural language processing (NLP).

References

[1] Meera Udani, Avinash Shrivas, Vaibhav Shukla (2013) “Question Answering System Based on Artificial Intelligence for Restricted Domain”, International Journal of Engineering Research & Technology.

[2] J. Weston, A. Bordes, S. Chopra, and T. Mikolov. Towards AI-complete question answering: A set of prerequisite toy tasks. arXiv preprint: 1502.05698, 2015.

[3] Tait Larson, Johnson(Heng) Gong, Josh Daniel “Providing a Simple Question Answering System By Mapping Questions to Questions.”,Technical report, Department of Computer Science,Stanford University, 2006

[4] Memory Networks, by J. Weston, S. Chopra, and A. Bordes, arXiv:1410.3916, 2014

[5] Towards AIComplete Question Answering: A Set of Prerequisite Toy Tasks, by J. Weston, arXiv:1502.05698, 2015

[6] End-To-End Memory Networks, by S. Sukhbaatar, J. Weston, and R. Fergus, Advances in Neural Information Processing Systems, 2015

[7] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut. 2017. Text Summarization Techniques: A Brief Survey. ArXiv e-prints (2017). arXiv:1707.02268

[8] Darshan kapashi, Pararth Shah(2014)“Answering Reading Comprehension Using Memory Networks”, Technical report, Department of Computer Science, Stanford University

[9] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, Rob Fergus(2015) “End to End Memory Networks”, Arxiv e-prints(2015) arXiv:1503.08895v5 [cs.NE]

[10] Tomer, M., Kumar, M. (2021). Question Answering System Using LSTM and Keyword Generation. In: Goar, V., Kuri, M., Kumar, R., Senjyu, T. (eds) Advances in Information Communication Technology and Computing. Lecture Notes in Networks and Systems, vol 135. Springer, Singapore. https://doi.org/10.1007/978-981-15-5421-6\_28