Turkish Question Answering System for University Regulations Using BERT and Information Retrieval

Author:

Kerem Göbekcioğlu

Abstract

This project implements a Turkish Question Answering (QA) system specifically designed for Gebze Technical University's student regulations. The system utilizes the dbmdz/bert-base-turkish-cased model, fine-tuned on a custom dataset of 601 question-answer pairs extracted from university documents. The implementation combines exact matching, BM25 ranking, and transformer-based question answering to provide accurate responses to student queries about university regulations.

The development process involved three main phases: dataset preparation, model fine-tuning, and interface implementation. A custom dataset was created following the SQuAD format, with careful consideration of Turkish language characteristics. Multiple training configurations were tested, with the simplest configuration achieving the best results: 13.68% Exact Match and 52.84% F1 score. The system was implemented with both command-line and web-based interfaces using Flask.

Key findings indicate that dataset size and quality significantly impact model performance, outweighing the importance of complex training strategies. The system performs well on exact matches, but shows limitations with question variations. This implementation provides insights into the challenges of developing Turkish language QA systems and the importance of data preparation in machine learning projects.

Keywords: Question Answering, Turkish NLP, BERT, Information Retrieval, University Regulations

Contents

1 Introduction

This project implements a Turkish Question Answering (QA) system specifically designed for Gebze Technical University's student regulations and guidelines. The system aims to help students easily access information about university rules by providing accurate answers to their natural language questions.

The development process focused on three main components:

- Creating a specialized dataset from GTU's official documents
- Fine-tuning a Turkish BERT model for question answering
- Implementing a hybrid approach combining exact matching and neural methods

The system processes Turkish text, handles various question formulations, and extracts precise answers from relevant contexts. To achieve this, the project utilizes the dbmdz/bert-base-turkish-cased model as its foundation and enhances it with specialized training on university regulations.

A key challenge addressed in this project was the handling of Turkish language specifics, including different question formations and text variations. The solution combines traditional information retrieval techniques (BM25) with modern transformer-based models to provide reliable answers.

The final implementation includes both a command-line interface for testing and a web-based interface for easy access, making it practical for student use. The system can handle questions about various aspects of university regulations, from course registration procedures to graduation requirements.

2 Methodology

2.1 Dataset Preparation

The creation of a high-quality dataset was crucial for developing an effective Question Answering system for GTU's regulations. This process involved multiple stages of data collection, preprocessing, and structured conversion, taking approximately three weeks to complete.

2.1.1 Data Collection and Extraction

The initial phase involved collecting official documents from GTU's website, which included various regulations and guidelines in PDF and DOCX formats. These documents were processed using PDFMiner and python-docx libraries to extract raw text while preserving the document structure:

```python
from pdfminer.high_level import extract_text as
   extract_text_from_pdf
import docx
import os

def extract_text_from_docx(docx_path):
    """Extract text from a DOCX file."""
    try:
        doc = docx.Document(docx_path)
        text = '\n'.join([para.text for para in doc.paragraphs])
        return text
    except Exception as e:
        print(f"Error extracting text from DOCX {docx_path}: {e}")
        return ""

def extract_texts_from_directory(directory):
    """Extract text from all PDF and DOCX files in a directory."""
    texts = {}
    for filename in os.listdir(directory):
        if filename.endswith('.pdf'):
            pdf_path = os.path.join(directory, filename)
            text = extract_text_from_pdf(pdf_path)
            texts[filename] = text
        elif filename.endswith('.docx'):
            docx_path = os.path.join(directory, filename)
            text = extract_text_from_docx(docx_path)
            texts[filename] = text
    return texts

def main():
    pdf_directory = 'nlpdataset'
    output_directory = 'datasettxt'

    # Create the output directory if it doesn't exist
    os.makedirs(output_directory, exist_ok=True)

    texts = extract_texts_from_directory(pdf_directory)
    for filename, text in texts.items():
        txt_filename = os.path.splitext(filename)[0] + '.txt'
        txt_path = os.path.join(output_directory, txt_filename)
        with open(txt_path, 'w', encoding='utf-8') as f:
            f.write(text)
        print(f"Extracted text from {filename} and saved to {txt_filename}")

if __name__ == "__main__":
    main()
```

2.1.2 Text Normalization

The extracted text underwent several preprocessing steps to ensure quality and consistency:

- Conversion to lowercase to maintain consistency
- Removal of empty lines and redundant whitespace
- Normalization of Turkish characters

- Standardization of formatting and punctuation

```python
import unicodedata
import os
def delete_empty_lines_and_normalize_in_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    with open(file_path, 'w', encoding='utf-8') as f:
        for line in lines:
            normalized_line = unicodedata.normalize('NFKC', line)
               .strip().lower()
            if normalized_line:  # Check if the line is not empty
                or just spaces
                f.write(normalized_line + '\n')

def delete_empty_lines_and_normalize_in_directory(directory):
    for filename in os.listdir(directory):
        if filename.endswith('.txt'):
            file_path = os.path.join(directory, filename)
            delete_empty_lines_and_normalize_in_file(file_path)
            print(f"Processed {file_path}")
```

2.2 Question Generation

A significant innovation in the dataset creation process was the use of Large Language Models (LLMs) to generate relevant questions. This approach helped create a diverse set of questions that covered various aspects of the regulations. The generated questions were saved in a structured format for further processing.

2.2.1 SQuAD Format Conversion

The dataset was structured following the Stanford Question Answering Dataset (SQuAD) format, which is the standard for QA tasks. Here's an example of the structure:

```json
        {
          "context": "madde 3- (1) dikey ge i  yoluyla kay t
             yapt ran  renciler  ncelikle  i ngilizce
             muafiyet s nav na al n rlar.",
          "qas": [
            {
              "question": "dikey ge i  yapan  renciler  ilk
                 olarak hangi s nava girerler?",
              "answers": [
                {
                  "text": "i ngilizce muafiyet s nav na
                     al n rlar.",
                  "answer_start": 69
                }
              ],
              "id": "2"
            }
          ]
        },
```

2.3 Answer Start Position Detection

One of the most challenging aspects of dataset preparation was accurately determining the answer start positions in the SQuAD format. Python's built-in find() method proved insufficient for Turkish text, necessitating the development of specialized finding algorithms. Three implementations were developed and tested:

2.3.1 First Implementation (find-first)

The first approach implemented a comprehensive matching system with multiple fallback strategies:

- Exact matching with normalization
- Turkish text variations handling
- Common verb ending transformations
- Fuzzy matching with SequenceMatcher
- Significant word detection

```
def find_first(source, destination):
# Normalize the strings
source = unicodedata.normalize('NFKC', source).strip()
destination = unicodedata.normalize('NFKC', destination).
   strip()

# Convert to lowercase
source_lower = source.lower()
destination_lower = destination.lower()

# 1. Exact match
index = source_lower.find(destination_lower)
if index != -1:
    return index

# 2. Common Turkish text variations
variations = [
    destination_lower,
    destination_lower.rstrip('.'),  # Remove trailing period
    destination_lower.rstrip(','),  # Remove trailing comma
    destination_lower.replace('i  ', 'i'),  # Handle Turkish
       i/
]

for variant in variations:
    index = source_lower.find(variant)
    if index != -1:
        return index

# 3. Handle common Turkish verb endings
common_endings = {
    'eder': 'etmek',
    'edilir': 'edilmek',
    'olur': 'olmak',
    'olunur': 'olunmak',
    'yap l r': 'yap lmak',
    'verilir': 'verilmek',
    'al n r': 'al nmak',
    'bulunur': 'bulunmak'
}

dest_words = destination_lower.split()
if len(dest_words) > 0:
    last_word = dest_words[-1]
    for ending, replacement in common_endings.items():
        if last_word.endswith(ending):
            modified_dest = ' '.join(dest_words[:-1] + [
               last_word.replace(ending, replacement)])
            index = source_lower.find(modified_dest)
            if index != -1:
                return index

# 4. Fuzzy matching for longer phrases
if len(destination_lower) > 10:
    max_ratio = 0
    best_index = -1

    # Use sliding window approach for longer texts
    dest_len = len(destination_lower)
    for i in range(len(source_lower) - dest_len + 1):
        window = source_lower[i:i + dest_len]
        ratio = SequenceMatcher(None, window,
           destination_lower).ratio()

        if ratio > max_ratio and ratio > 0.8:  # 80%
           similarity threshold
            max_ratio = ratio
            best_index = i

    if best_index != -1:
        return best_index

# 5. Try finding significant parts
significant_words = [word for word in dest_words if len(word)
    > 3]
if significant_words:
```

```python
        for word in significant_words:
            index = source_lower.find(word)
            if index != -1:
                # Verify surrounding context
                context_start = max(0, index - 50)
                context_end = min(len(source_lower), index + len(word) + 50)
                context = source_lower[context_start:context_end]

                if SequenceMatcher(None, context, destination_lower).ratio() > 0.6:
                    return index

    return -1
```

While thorough, this implementation was computationally expensive and sometimes overly complex in its matching strategies.

2.3.2 Second Implementation (find-second) - Selected Approach

The second implementation was chosen as the final solution due to its balance of accuracy and efficiency:

- Progressive matching strategy from exact to fuzzy
- Simplified Turkish-specific variations
- Efficient partial matching using substrings
- Optimized threshold values for fuzzy matching
- Focused context verification

```python
def find_second(source, destination):
    # Normalize the strings
    source = unicodedata.normalize('NFKC', source).strip()
    destination = unicodedata.normalize('NFKC', destination).strip()

    # Convert to lowercase for case-insensitive matching
    source_lower = source.lower()
    destination_lower = destination.lower()

    # 1. Exact Match
    index = source_lower.find(destination_lower)
    if index != -1:
        return index

    # 2. Turkish-Specific Variations
    variations = [
        destination_lower.rstrip('.'),  # Remove trailing period
        destination_lower.rstrip(','),  # Remove trailing comma
        destination_lower.replace('i̇', 'i'),  # Handle Turkish 'i̇' vs 'i'
    ]
    for variant in variations:
        index = source_lower.find(variant)
        if index != -1:
            return index

    # 3. Partial Matching Using Substring Similarity
    destination_words = destination_lower.split()
    if len(destination_words) > 1:
        for i in range(len(destination_words)):
            # Create substrings by progressively removing words
            partial_dest = ' '.join(destination_words[i:])
            index = source_lower.find(partial_dest)
            if index != -1:
                return index

    # 4. Fuzzy Matching Using SequenceMatcher (with lowered threshold)
    max_ratio = 0
    best_index = -1
    dest_len = len(destination_lower)
    threshold = 0.6  # Lowered threshold for better matches

    # Sliding window for fuzzy matching
    for i in range(len(source_lower) - dest_len + 1):
        window = source_lower[i:i + dest_len]
        ratio = SequenceMatcher(None, window, destination_lower).ratio()
        if ratio > max_ratio and ratio >= threshold:
            max_ratio = ratio
            best_index = i

    if best_index != -1:
        return best_index

    # 5. Significant Words Matching
    significant_words = [word for word in destination_words if len(word) > 3]
    for word in significant_words:
        index = source_lower.find(word)
        if index != -1:
            # Check surrounding context for higher confidence
            context_start = max(0, index - 50)
            context_end = min(len(source_lower), index + len(word) + 50)
            context = source_lower[context_start:context_end]
            if SequenceMatcher(None, context, destination_lower).ratio() > 0.6:
                return index
```

```python

    # If no match is found
    return -1
```

Key advantages that led to its selection:

- Better performance with Turkish text variations
- More reliable answer position detection
- Efficient processing time
- Lower false positive rate
- Simpler maintenance and modification

2.3.3 Third Implementation (find-last)

The third implementation focused on handling specific cases:

- Common ending detection
- Longest matching sequence finding
- Punctuation handling

```python
def find_last(source, destination):
    # Normalize the strings
    source = unicodedata.normalize('NFKC', source).strip()
    destination = unicodedata.normalize('NFKC', destination).strip()

    # Convert both source and destination to lowercase for case-insensitive search
    source_lower = source.lower()
    destination_lower = destination.lower()

    # First try Python's built-in find method
    index = source_lower.find(destination_lower)
    if index != -1:
        return index

    # Remove punctuation from the end of destination if it exists
    if destination_lower.endswith('.'):
        destination_lower = destination_lower[:-1]
        index = source_lower.find(destination_lower)
        if index != -1:
            return index

    # Try finding without the last word if it's a common ending like "eder", "edilir", etc.
    common_endings = ['eder', 'edilir', 'olur', 'olunur', 'yap l r']
    dest_words = destination_lower.split()
    if any(dest_words[-1].endswith(ending) for ending in
        common_endings):
        modified_dest = ' '.join(dest_words[:-1])
        index = source_lower.find(modified_dest)
        if index != -1:
            return index

    # Try to find the longest matching sequence
    dest_words = destination_lower.split()
    for length in range(len(dest_words), 0, -1):
        for i in range(len(dest_words) - length + 1):
            phrase = ' '.join(dest_words[i:i+length])
            if len(phrase) > 10:  # Only consider substantial
                phrases
                index = source_lower.find(phrase)
                if index != -1:
                    return index

    return -1
```

While effective for specific cases, it lacked the robustness of the second implementation for general use.

2.3.4 Comparison of Approaches

Feature	First	Second	Third
Accuracy	High	High	Medium
Performance	Low	High	Medium
Complexity	High	Medium	Low
Maintainability	Low	High	Medium
Turkish Support	High	High	Medium

Table 1: Comparison of find() implementations

The second implementation was ultimately chosen because it provided the best balance of:

- Accurate answer position detection
- Efficient processing time
- Robust handling of Turkish text variations
- Maintainable code structure
- Reliable fuzzy matching with appropriate thresholds

According to the tests, the second implementation ultimately finds a position whether it is correct or not. However, each of the three implementations has its flaws. The second implementation was chosen because it gave the best performance among the three. That said, it still needs improvements and further work to enhance its accuracy and reliability. But all three is much more stable and better than Python's built-in find() method.

2.4 Dataset Merging and Validation

The final stage involved:

- Merging individual JSON files into a combined dataset
- Updating titles to reflect document sources
- Validating answer positions and context integrity
- Ensuring consistent formatting across all entries

```python
def update_title_with_filename(directory):
    json_files = sorted([f for f in os.listdir(directory) if f.
       endswith('.json')])

    for filename in json_files:
        file_path = os.path.join(directory, filename)
        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)

        # Update the title with the filename (excluding the
           extension)
        title = os.path.splitext(filename)[0]
        for article in data["data"]:
            article["title"] = title

        # Save the updated JSON back to the file
        with open(file_path, 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=4)

        print(f"Updated title in file '{filename}' to '{title}'")
```

```python
def merge_json_files(directory):
    merged_data = {"data": []}
    current_id = 1
    processed_titles = set()  # Keep track of processed titles to
        avoid duplicates

    # Get unique JSON files
    json_files = sorted(set([f for f in os.listdir(directory) if
       f.endswith('.json')]))

    print(f"Found {len(json_files)} JSON files")

    for filename in json_files:
        file_path = os.path.join(directory, filename)
        print(f"Processing {filename}...")

        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)
            for document in data["data"]:
                # Skip if we've already processed this title
                if document["title"] in processed_titles:
                    print(f"Skipping duplicate title: {document['title']}")
                    continue

                processed_titles.add(document["title"])

                new_document = OrderedDict([
                    ("title", document["title"]),
                    ("paragraphs", [])
                ])

                for paragraph in document["paragraphs"]:
                    new_paragraph = OrderedDict([
                        ("context", paragraph["context"]),
                        ("qas", [])
                    ])

                    for qa in paragraph["qas"]:
                        # Create QA pair with specific order
                        new_qa = OrderedDict([
                            ("question", qa["question"]),
                            ("answers", qa["answers"]),
                            ("id", current_id)
                        ])

                        new_paragraph["qas"].append(new_qa)
                        current_id += 1

                    new_document["paragraphs"].append(new_paragraph)

                merged_data["data"].append(new_document)

    return merged_data
```

The final dataset contained approximately 601 question-answer pairs across various regulation topics, providing a foundation for model training. This carefully curated dataset was essential for developing a system that could accurately understand and answer questions about university regulations in Turkish.

3 Model Fine-tuning

The pre-trained Turkish BERT model (dbmdz/bert-base-turkish-cased) was fine-tuned for the question-answering task using our custom dataset. The training process was primarily conducted on Google Colab's GPU environment, with only the first model (model 0) being trained on local hardware.

3.1 Dataset Preparation

The training dataset consisted of 601 question-answer pairs, with 576 unique questions extracted from GTU's regulations. The dataset was split into training and validation sets with an 80-20 ratio:

- Training set: approximately 480 examples
- Validation set: approximately 120 examples

Table 2: Dataset Split Information

Split	Number of Examples	Percentage
Training	480	80%
Validation	120	20%
Total	601	100%

Initially, the standard datasets.load-dataset() function was attempted, but it encountered numerous issues with our custom JSON structure. This led to the implementation of a custom loading function that could properly handle our specific data format while maintaining the required dataset structure for training.

3.2 Dataset Preparation and Processing

The dataset was processed in multiple steps for training:

3.2.1 Dataset Loading and Splitting

```python
def load_datasets():
    with open('merged_dataset.json', 'r', encoding='utf-8') as f:
        data = json.load(f)

    # 80-20 split
    total_docs = len(data['data'])
    split_point = int(total_docs * 0.8)

    train_data = {"data": data['data'][:split_point]}
    valid_data = {"data": data['data'][split_point:]}

    def flatten_data(data):
        flat_examples = []
        for doc in data['data']:
            for para in doc['paragraphs']:
                context = para['context']
                for qa in para['qas']:
                    flat_examples.append({
                        'context': context,
                        'question': qa['question'],
                        'answer_text': qa['answers'][0]['text'],
                        'answer_start': qa['answers'][0]['
                          answer_start']
                    })
        return flat_examples
```

3.2.2 Data Preprocessing

The preprocessing pipeline included tokenization and span mapping:

```
def preprocess_function(examples):
    questions = [q.strip() for q in examples['question']]
    contexts = [c.strip() for c in examples['context']]

    tokenized_examples = tokenizer(
        questions,
        contexts,
        truncation="only_second",
        max_length=384,
        stride=128,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding="max_length",
    )

    # Map answer positions to tokens
    start_positions = []
    end_positions = []
    offset_mapping = tokenized_examples.pop("offset_mapping")
    sample_mapping = tokenized_examples.pop("
       overflow_to_sample_mapping")

    # Position mapping logic...
```

3.3 Training Configuration

The training was conducted using Google Colab's GPU environment with the following base configuration:

```
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,                 # Varied across models
    per_device_train_batch_size=4,      # Adjusted based on GPU
       memory
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=2,       # Varied for effective
       batch size
    learning_rate=3e-5,                 # Adjusted per model
    warmup_ratio=0.1,                   # Added in later models
    weight_decay=0.01,
    evaluation_strategy="steps",
    eval_steps=100,
    save_strategy="steps",
    save_steps=100,
    save_total_limit=3,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    logging_dir='./logs',
    logging_steps=50,
    report_to="none",
    fp16=True                              # Enabled for GPU training
)
```

3.4 Training Experiments

Four different training configurations were tested, with varying hyperparameters as shown in Table 3. Model 0 achieved the best performance with the simplest configuration, while more complex configurations (Models 1-3) showed diminishing returns despite more sophisticated training strategies.

Table 3: Training Configurations for Each Model

Model	**Epochs**	**Learning Rate**	**Batch Size**	**Gradient Steps**	**Warmup Ratio**
Model 0	3	3e-5	4	2	None
Model 1	5	2e-5	8	4	0.1
Model 2	10	1e-5	4	8	0.15
Model 3	15	5e-6	2	16	0.2

The performance results for each model configuration are presented in Table 4.

Table 4: Model Performance Results

Model	**Exact Match (%)**	**F1 Score (%)**
Model 0	13.68	52.84
Model 1	4.07	8.18
Model 2	7.69	21.20
Model 3	2.56	11.70

3.5 Training Observations

The results reveal several key insights about training on this specific dataset:

- **Model 0 Superiority**: The simplest configuration achieved the best results, suggesting that a smaller dataset benefits from fewer epochs and higher learning rates.
- **Overfitting**: Models with longer training durations and lower learning rates (Models 2 and 3) showed clear signs of overfitting, with decreasing validation performance despite continued training.
- **Dataset Size Impact**: The limited dataset size (601 examples) appears to be a significant constraint, making simpler training approaches more effective.

The use of Google Colab's GPU environment significantly accelerated the training process for models 1-3, allowing for rapid experimentation with different hyperparameters. However, the limited dataset size appears to be a more fundamental constraint than computational resources.

3.6 Future Improvements

Based on these observations, several potential improvements could be explored:

- **Data Augmentation**: Implement techniques such as question paraphrasing to artificially increase the dataset size.
- **Model Architecture**: Experiment with smaller models like DistilBERT that might be more suitable for limited data.
- **Validation Strategy**: Implement k-fold cross-validation to make better use of the limited dataset.
- **Training Strategy**: Focus on shorter training durations with higher learning rates, given the success of Model 0.

4 System Interfaces

Two interfaces were developed for the QA system: a command-line interface designed for testing and development, and a web-based interface built using Flask for end-user interaction. The web-based interface is recommended for use.

The web interface features a clean, modern design implemented in HTML, CSS, and JavaScript:

- Simple input field for user questions
- Separate display sections for different answer types:
  - "Tam Eşleşme" (Exact Match) section for direct matches
  - "Model Cevabı" (Model Answer) section for generated responses
- Confidence scores displayed for each answer
- Responsive design with modern styling elements

The interface code demonstrates the implementation:

```html
<div class="container">
    <h1>GT  Y netmelik Soru-Cevap Sistemi</h1>
    <div class="input-section">
        <input type="text" id="question"
                 placeholder="Sorunuzu buraya yaz n...">
        <button onclick="getAnswer()">Cevapla</button>
    </div>
    <div id="result" class="result">
        <div id="exact-match" class="answer-section">
```

```html
            <div class="source">Tam E le me</div>
            <div id="exact-match-answer"></div>
            <div id="exact-match-confidence" class="confidence"></div>
        </div>
        <!-- Model answer section similar -->
    </div>
</div>
```

The interface processes responses asynchronously, providing a smooth user experience while maintaining clear distinction between exact matches and model-generated answers. Visual feedback through confidence scores helps users assess answer reliability.

4.1 Question Processing Pipeline

The system processes questions through a two-stage approach:

1. **Exact Match Search** The system first attempts to find an exact or similar match from the training dataset:

- Questions are normalized (lowercase, punctuation removed)
- Text similarity is calculated using word overlap
- A threshold of 0.8 is used for similarity matching

```python
def normalize_text(text):
    text = text.lower().strip()
    text = re.sub(r'[.,!?]', '', text)
    text = ' '.join(text.split())
    return text

def text_similarity(text1, text2):
    words1 = set(text1.split())
    words2 = set(text2.split())
    overlap = len(words1.intersection(words2))
    total = len(words1.union(words2))
    return overlap / total
```

2. **BM25 + Model Inference** If no exact match is found, the system uses BM25 ranking to find relevant contexts:

- Top 5 most relevant contexts are selected
- Each context is processed through the QA model
- Best answer is selected based on confidence scores

```python
def create_bm25_index(contexts):
    tokenized_contexts = [word_tokenize(context.lower())
                         for context in contexts]
    return BM25Okapi(tokenized_contexts)

```

```python
# Usage in answer processing
tokenized_question = word_tokenize(question)
bm25_scores = bm25.get_scores(tokenized_question)
top_contexts_indices = np.argsort(bm25_scores)[-5:][::-1]
```

The BM25 ranking and model inference work together in a complementary fashion. While BM25 efficiently narrows down the most relevant contexts from the entire document collection, the model performs the detailed task of extracting specific answers from these selected contexts. This combination helps balance between processing speed and answer accuracy.

4.2 Answer Extraction Logic

The core of the system lies in three main functions that handle answer extraction:

1. **Model-based Answer Extraction**

```python
def get_answer(question, context, model, tokenizer):
    inputs = tokenizer(
        question,
        context,
        return_tensors="pt",
        max_length=512,
        truncation=True,
        padding="max_length"
    )

    with torch.no_grad():
        outputs = model(**inputs)

    start_scores = outputs.start_logits[0]
    end_scores = outputs.end_logits[0]

    start_index = torch.argmax(start_scores)
    end_index = torch.argmax(end_scores)

    if end_index < start_index:
        end_index = start_index + 1

    answer_tokens = inputs["input_ids"][0][start_index:end_index
       + 1]
    answer = tokenizer.decode(answer_tokens, skip_special_tokens=
       True)

    confidence = (torch.max(start_scores) + torch.max(end_scores)
       ).item() / 2

    return answer, confidence
```

This function:

- Tokenizes question and context
- Uses the model to predict answer span

- Calculates confidence score
- Handles invalid span predictions

2. **Question Matching**

```python
def find_best_match(question, qa_pairs, threshold=0.8):
    normalized_question = normalize_text(question)
    best_match = None
    best_score = 0

    for qa in qa_pairs:
        score = text_similarity(normalized_question, qa['question'])
        if score > best_score and score >= threshold:
            best_score = score
            best_match = qa

    return best_match, best_score
```

3. **Main Answer Processing Pipeline** The main pipeline implements a three-tier approach combining exact matching, BM25 ranking, and model inference with a fallback mechanism:

```python
def answer_question(question):
    # 1. Try exact and similar matches
    best_match, match_score = find_best_match(question, qa_pairs)
    if best_match:
        return {
            'exact_match': {
                'answer': best_match['answer'],
                'confidence': match_score * 10,
                'source': 'Exact/Similar Match'
            },
            'model_answer': None
        }

    # 2. Try BM25 + Model
    tokenized_question = word_tokenize(question)
    bm25_scores = bm25.get_scores(tokenized_question)
    top_contexts_indices = np.argsort(bm25_scores)[-5:][::-1]

    best_answer = ""
    best_confidence = -float('inf')

    # Process top BM25-ranked contexts
    for idx in top_contexts_indices:
        if bm25_scores[idx] > 0:
            answer, confidence = get_answer(question, contexts[idx])
            if confidence > best_confidence:
                best_confidence = confidence
                best_answer = answer

    # 3. Fallback mechanism
    if best_confidence < 0 or best_answer == "":
        fallback_context = " ".join(contexts)
        best_answer, best_confidence = get_answer(question,
           fallback_context)
```

The system employs a hierarchical approach:

- First attempts exact or similar matching for highest confidence answers
- If no match is found, uses BM25 to rank and select relevant contexts
- As a final fallback, processes the entire context collection if no good matches are found

This enhanced pipeline ensures that a response is always provided, even when exact matches or high-confidence answers aren't available in the top-ranked contexts. The trade-off between answer quality and response availability is managed through confidence scores and source identification.

4.3 Command-Line Interface

The command-line interface provides:

- Interactive question input
- Display of matching method (exact match or model)
- Confidence scores
- Debug information showing BM25 scores and contexts

4.4 Web Interface

A Flask-based web interface was implemented with:

- Simple input field for questions
- Separate display sections for exact matches and model answers
- Confidence score display
- Basic error handling

Key implementation:

```python
@app.route('/answer', methods=['POST'])
def answer():
    try:
        data = request.get_json()
        question = data['question']
        result = answer_question(question)
        return jsonify(result)
    except Exception as e:
        return jsonify({
            'error': 'An error occurred',
            'details': str(e)
        }), 500
```

4.5 Limitations and Challenges

Several limitations were observed:

- Similar question matching often fails for slight variations
- Model answers can be inconsistent when no exact match is found
- BM25 sometimes fails to identify the most relevant context
- Processing time can be slow for model-based answers

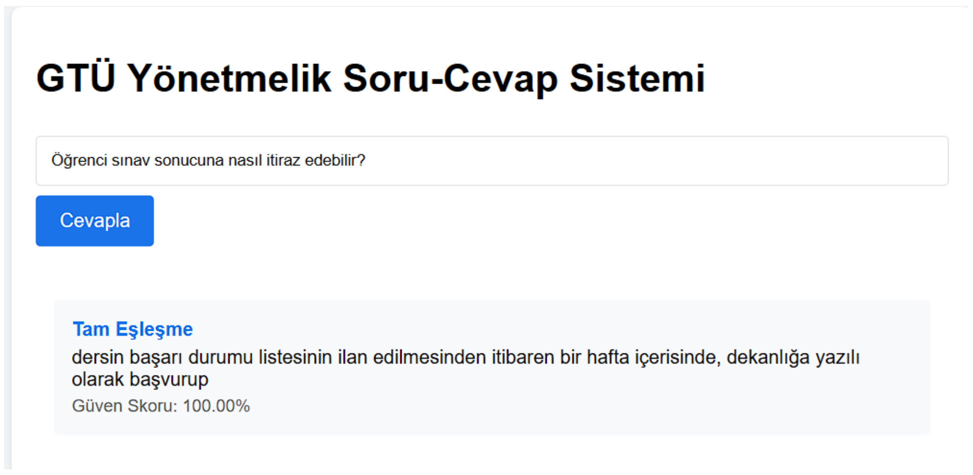
The system performs best when questions match exactly with the training data, but struggles with variations and completely new questions. Future improvements could focus on better similarity matching and context selection algorithms.

5 Results and Examples

To demonstrate the system's performance, here are representative examples of different response types:

5.1 Exact Match Examples

Question: "Öğrenci sınav sonuçlarına nasıl itiraz edebilir?"
Response:



GTÜ Yönetmelik Soru-Cevap Sistemi

Öğrenci sınav sonucuna nasıl itiraz edebilir?

Tam Eşleşme

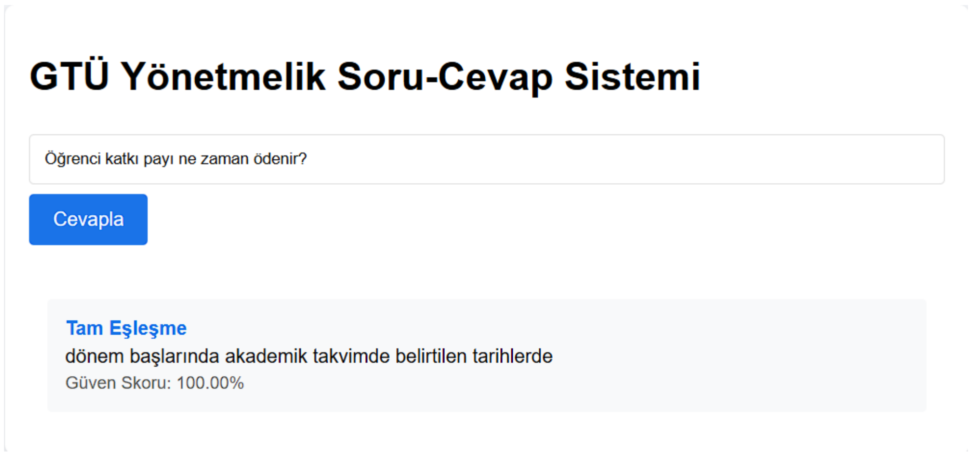
dersin başarı durumu listesinin ilan edilmesinden itibaren bir hafta içerisinde, dekanlığa yazılı olarak başvurup

Güven Skoru: 100.00%

Figure 1: Exact Match: Öğrenci sınav sonuçlarına nasıl itiraz edebilir?

This example shows a high-confidence exact match where the system correctly identifies and returns the precise definition.

Question: ”Öğrenci katkı payı ne zaman ödenir?”
Response:


GTÜ Yönetmelik Soru-Cevap Sistemi

Öğrenci katkı payı ne zaman ödenir?

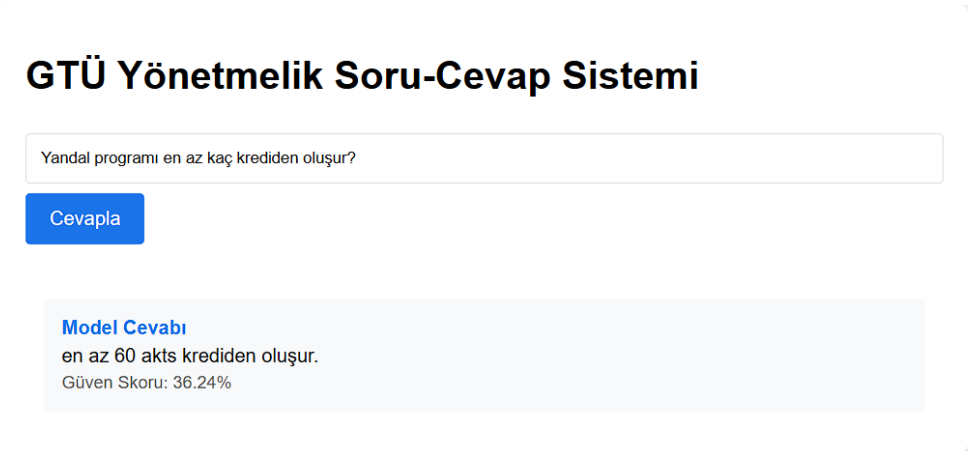
Tam Eşleşme
dönem başlarında akademik takvimde belirtilen tarihlerde
Güven Skoru: 100.00%

Figure 2: Exact Match: Öğrenci katkı payı ne zaman ödenir?

The system accurately identifies the question and retrieves the correct answer about payment timing from the regulations.

5.2 Model-Generated Responses

Question: ”Yandal programı en az kaç krediden oluşur?”
Response:


GTÜ Yönetmelik Soru-Cevap Sistemi

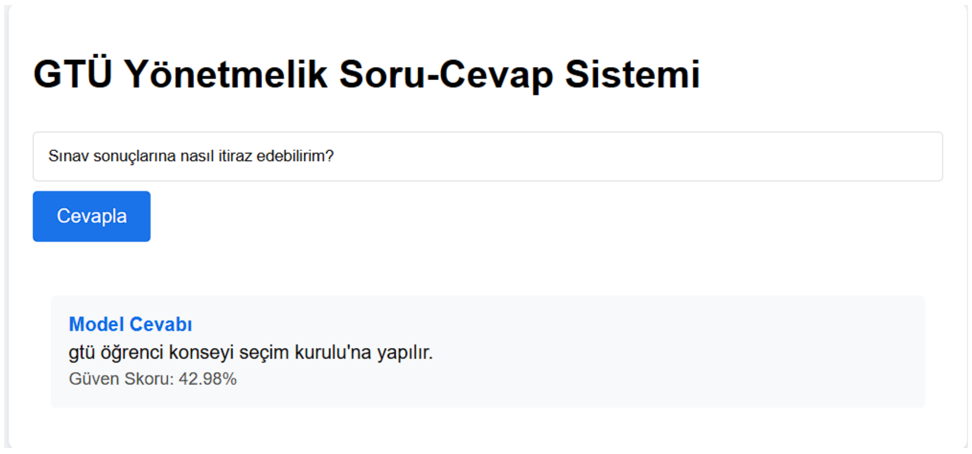
Yandal programı en az kaç krediden oluşur?

Model Cevabı
en az 60 akts krediden oluşur.
Güven Skoru: 36.24%

Figure 3: Model Answer: Yandal programı en az kaç krediden oluşur?

In this case, the model successfully processes the question and extracts the specific credit requirement information from the context, demonstrating its ability to handle numerical queries accurately.

Question: ”Sınav sonuçlarına nasıl itiraz edebilirim?”
Response:


GTÜ Yönetmelik Soru-Cevap Sistemi

Sınav sonuçlarına nasıl itiraz edebilirim?

Model Cevabı
gtü öğrenci konseyi seçim kurulu'na yapılır.
Güven Skoru: 42.98%

Figure 4: Incorrect Match: Sınav sonuçlarına nasıl itiraz edebilirim?

This example demonstrates how the model may incorrectly match a question with an answer, providing an irrelevant response.

6 Discussion

The development and implementation of this Turkish Question Answering system for GTU regulations revealed several crucial insights. The most significant finding was the paramount importance of data quality and quantity in training neural language models. Despite experimenting with various training strategies, the model's performance was limited by the dataset size of 601 question-answer pairs.

The data preparation phase proved challenging and required multiple iterations to:

- Correctly extract and normalize text from university documents
- Generate meaningful question-answer pairs
- Accurately determine answer start positions in contexts
- Handle Turkish language specifics in text processing

A significant challenge emerged regarding answer generation and context matching. The implementation assumed answers would be strictly present within the context. However, during dataset creation, LLM models often generated answers based on contextual understanding rather than exact text matches. While these answers were semantically correct, they featured slight variations from the source text, making it difficult to accurately determine answer positions and maintain consistent formats.

The training experiments clearly showed several patterns:

- More complex training strategies did not compensate for limited data
- Simpler training configurations performed better on small datasets
- Overfitting occurred quickly with more training epochs

Initial errors in data preparation, particularly in answer position detection and JSON formatting, led to significant debugging efforts. However, these challenges provided valuable lessons in dataset creation for Turkish language tasks.

Based on these findings, future implementations should consider:

- Stricter answer generation guidelines ensuring answers exist verbatim in the context
- Post-processing steps to align generated answers with context text
- Alternative evaluation metrics that account for semantic similarity
- Expanding the dataset size significantly
- Implementing robust data augmentation techniques

7 Conclusion

This project implemented a Question Answering system for GTU regulations, though with certain limitations. The system demonstrates reasonable performance for exact matches but struggles with question variations. The key findings suggest that:

- Data quality and quantity are fundamental constraints in model performance
- Simple training approaches can outperform complex ones with limited data
- Turkish language QA systems require careful consideration of language-specific challenges

Future work should focus on:

- Expanding the training dataset significantly
- Implementing robust data augmentation techniques
- Improving question variation handling
- Developing better Turkish-specific text processing methods

These improvements would likely lead to a more powerful and generalizable system for handling university regulation queries.

7.1 What is Learned

While maintaining professional objectivity throughout this report, it is worth noting the personal insights gained. The experience highlighted that data preparation and quality assurance should receive as much, if not more, attention than model training strategies. Future projects would benefit from allocating more resources to data collection and preparation phases before proceeding to model training.

8 References

References

[1] DBMDZ, “Turkish BERT (cased),” *Hugging Face Models*, 2023, https://huggingface.co/dbmdz/bert-base-turkish-cased

[2] Özcan, M., “Turkish Sentiment Analysis with BERTurk,” Kaggle, 2023, https://www.kaggle.com/code/ozcan15/turkish-sentiment-analysis-with-berturk-93-acc

[3] Stefan Schweter, “Turkish BERT Project,” GitHub repository, 2023, https://github.com/stefan-it/turkish-bert