NutriQuest: A Comprehensive Information Retrieval System for Fitness and Nutrition

Information Retrieval Project - Final Phase
University of Science and Technology at Zewail city
Authors: Abdulrahman Elsayed , Mostafa Nashaat

***Abstract*—This paper presents NutriQuest, an advanced information retrieval system designed to provide users with accurate and relevant information about bodybuilding, fitness, nutrition, and sports topics. The system utilizes modern IR techniques to collect, process, and index data from multiple sources including Wikipedia, YouTube, and Google. Through efficient search algorithms and a user-friendly interface, NutriQuest enables users to find specific information about workout plans, nutrition strategies, and fitness advice. The system was developed using Python with libraries like BeautifulSoup for web scraping, NLTK for natural language processing, and Python-Terrier for indexing and retrieval operations. The evaluation results demonstrate the system's effectiveness in retrieving relevant information across various fitness-related queries.**

***Index Terms*—information retrieval, fitness, nutrition, search engines, BERT, query expansion, web scraping**

I. Introduction

A. Project Overview

The fitness and nutrition information landscape is vast and often overwhelming, with resources scattered across various platforms and sources. NutriQuest addresses this challenge by providing a centralized information retrieval system that aggregates and organizes content from multiple sources, enabling users to efficiently find reliable information about bodybuilding, fitness routines, dietary guidelines, and sports nutrition.

B. Motivation

In recent years, there has been growing interest in fitness, nutrition, and healthy lifestyle choices. However, finding accurate and relevant information can be challenging due to:

- Information overload with varying quality and reliability
- Technical terminology that may be difficult to understand for beginners
- Contradictory advice across different sources
- Lack of context-aware search systems specific to fitness domains

NutriQuest aims to solve these problems by providing a specialized search engine that understands fitness terminology, indexes content from trustworthy sources, and presents results in a user-friendly manner.

C. Objectives

The main objectives of this project include:

- Developing a comprehensive information retrieval system for fitness and nutrition topics
- Creating an efficient indexing mechanism for web content related to bodybuilding, workouts, and diet
- Implementing effective search algorithms that understand fitness-specific queries
- Providing a user-friendly interface for querying and browsing fitness information
- Evaluating the system's performance using relevant metrics and real-world queries

II. Background and Related Work

A. Information Retrieval Systems

Information Retrieval (IR) systems are designed to retrieve information resources relevant to a user's information needs from a collection of information resources. Modern IR systems typically use probabilistic and vector space models to represent documents and queries, allowing for effective matching and ranking of results. Common approaches include:

- Boolean retrieval models that use logical operators
- Vector space models that represent documents and queries as vectors
- Probabilistic models that rank documents based on their probability of relevance
- Language models that consider term frequencies and document statistics

B. Domain-Specific Search Engines

While general-purpose search engines like Google provide broad coverage, domain-specific search engines can offer more targeted and relevant results for specialized fields. Examples include:

- PubMed for medical research
- Google Scholar for academic literature
- LexisNexis for legal information

In the fitness domain, several specialized platforms exist, but they often focus on content creation rather than comprehensive information retrieval.

C. Information Extraction from Web Content

Web content extraction involves retrieving and parsing HTML content to extract meaningful information. Common approaches include:

- HTML parsing using libraries like BeautifulSoup
- Regular expressions for pattern matching
- Machine learning techniques for identifying relevant content
- Named entity recognition for identifying key terms

III. Methodology

A. System Architecture

NutriQuest follows a modular architecture consisting of several interconnected components:

Fig. 1. NutriQuest System Architecture

The key components include:

- **Data Collection Module**: Gathers information from various sources using web scraping techniques
- **Preprocessing Module**: Cleans and normalizes the text data
- **Indexing Module**: Creates searchable indices of the preprocessed content
- **Query Processing Module**: Handles user queries and retrieves relevant documents
- **User Interface**: Provides a clean interface for users to interact with the system

B. Data Collection

The data collection process involved systematic gathering of information from multiple sources focused on bodybuilding, fitness, nutrition, and sports topics.

1) Topic Selection: A comprehensive list of 35 topics was carefully curated to cover various aspects of fitness and nutrition, including bodybuilding training, supplementation, diet plans, workout routines, muscle building techniques, weight loss strategies, sports nutrition, and more.

2) Source Selection: For each topic, information was gathered from three primary sources:

- **Wikipedia**: For comprehensive, generally reliable information
- **YouTube**: For video content, demonstrations, and expert advice
- **Google**: For diverse perspectives and additional resources

3) Web Scraping Process: The web scraping process involved:

1) Constructing search queries by combining each topic with each source
2) Retrieving the top 5 search results for each query using the Google Search API
3) Storing the results in a structured format with metadata

The following Python code snippet illustrates the search process:

```python
# Dictionary to store search results with keys as (topic, source)
search_results = {}

# Lists to store the links, corresponding topics, and sources
links_new = []
topics_new = []
source_new = []

# Iterate over each topic and each source to perform the search
for topic in topics:
    for source in sources:
        try:
            # Create the search query by appending the source to the topic
            search_query = topic + " " + source
            print("Searching for:", search_query)
            # Retrieve the top 5 search results
            results = list(search(search_query, num=5, stop=5))
            # Store the results in the dictionary
            search_results[(topic, source)] = results

            # Append each found link along with its topic and source
            for link in results:
                links_new.append(link)
                topics_new.append(topic)
                source_new.append(source)
        except Exception as e:
            print("Error occurred while searching for topic:", topic,
                  "with source:", source)
            print(e)
```

Listing 1. Web Scraping Process

In total, over 500 unique URLs were collected across all topics and sources, providing a diverse corpus of fitness and nutrition information.

C. Data Preprocessing

1) Content Extraction: For each collected URL, the following preprocessing steps were performed:

1) **HTML Content Retrieval**: The raw HTML content was downloaded using the requests library
2) **HTML Parsing**: BeautifulSoup was used to parse the HTML and extract meaningful text
3) **Content Cleaning**: Script tags, style elements, and other non-content HTML elements were removed
4) **Text Normalization**: Text was split into lines, excess whitespace was removed, and empty lines were filtered out

The following code demonstrates the content extraction process:

```python
# Extract text from html
def extract_text_from_html(html_content):
    try:
        soup = BeautifulSoup(html_content, 'html.parser')
        # Remove script and style elements
        for script in soup(["script", "style"]):
```

```python
            script.extract()
        # Get text
        text = soup.get_text()
        # Break into lines and remove leading and trailing space on each
        lines = (line.strip() for line in text.splitlines())
        # Break multi-headlines into a line each
        chunks = (phrase.strip() for line in lines
                  for phrase in line.split("  "))
        # Drop blank lines
        text = '\n'.join(chunk for chunk in chunks if chunk)
        return text
    except Exception as e:
        print("Error extracting text from HTML:", e)
        return ""
```

Listing 2. Content Extraction Process

2) Document Processing: After content extraction, each document underwent further processing:

- **URL Validation**: Ensuring all URLs were properly formatted and accessible
- **Metadata Association**: Each document was associated with its source topic and platform
- **Document Storage**: Processed documents were stored in a dictionary structure with their metadata

```python
# Process each valid URL
for idx, (link, topic, source) in enumerate(zip(valid_links, valid_topics, valid_sources)):
    print(f"Processing {idx+1}/{len(valid_links)}: {link}")
    content = fetch_and_extract(link)
    # Only store if we successfully extracted content
    if content:
        documents[link] = {
            'text': content,
            'topic': topic,
            'source': source
        }
```

Listing 3. Document Processing

D. Indexing

For effective information retrieval, the system uses Python-Terrier, a Python interface to the Terrier IR platform, which provides robust indexing and retrieval functionalities.

1) Document Preparation: Before indexing, documents were transformed into a format suitable for Python-Terrier:

- Creating a Pandas DataFrame with document text, IDs, and metadata
- Assigning unique document identifiers
- Formatting text for optimal indexing

2) Index Construction: The indexing process involved:

- Tokenization of document text
- Stemming to reduce words to their root forms
- Stopword removal to eliminate common words with little semantic value
- Inverted index creation for efficient term-based document retrieval

E. Query Processing

The query processing module handles user input and retrieves relevant documents from the index.

1) Query Expansion: To improve retrieval effectiveness, query expansion techniques were implemented:

- Adding related terms to the original query
- Incorporating synonyms for fitness-specific terminology
- Adjusting term weights based on importance

2) Ranking Mechanisms: Documents are ranked using multiple retrieval models:

- BM25 ranking for term frequency-inverse document frequency scoring
- BERT-based re-ranking for semantic understanding
- Custom scoring that considers source reliability and content quality

F. User Interface

The user interface was designed to be intuitive and functional, providing a seamless experience for users seeking fitness and nutrition information.

Fig. 2. NutriQuest User Interface

Key features of the interface include:

- Clean, minimalist design focused on search functionality
- Search options for customizing the retrieval process
- Source filtering to focus on specific platforms (Wikipedia, YouTube, Google)
- Popular topics section for quick access to common searches
- Adjustable number of results (5-20) to control information volume

IV. Implementation Details

A. Technologies Used

The NutriQuest system was implemented using a variety of technologies and libraries:

Component	Technologies Used
Programming Language	Python 3.11
Web Scraping	Requests, BeautifulSoup4, Google Search API
Text Processing	NLTK, Regular Expressions
Indexing	Python-Terrier, PyTrec-Eval-Terrier
Data Management	Pandas, NumPy
UI Implementation	HTML, CSS, JavaScript

TABLE I
TECHNOLOGIES USED IN NUTRIQUEST IMPLEMENTATION

B. Data Collection Implementation

The data collection process was implemented in Python, using libraries such as requests and googlesearch for retrieving search results and web content.

Key implementation details include:

- Custom headers to avoid being blocked by websites
- Error handling for failed requests
- Rate limiting to respect website policies
- Parallel processing for efficient data collection

C. Text Processing Pipeline

The text processing pipeline was implemented using a combination of BeautifulSoup for HTML parsing and custom functions for text normalization:

Algorithm 1 Text Processing Pipeline

procedure PROCESSDOCUMENT(url)
  $html$ ← FetchURL(url)
  $soup$ ← BeautifulSoup($html$)
  RemoveElements($soup$, ["script", "style"])
  $text$ ← ExtractText($soup$)
  $lines$ ← SplitLines($text$)
  $cleanLines$ ← FilterEmptyLines($lines$)
  $normalizedText$ ← JoinLines($cleanLines$)
  return $normalizedText$
end procedure

D. Indexing Implementation

The indexing system was implemented using Python-Terrier with the following components:

```
# Initialize PyTerrier
if not pt.started():
    pt.init()

# Convert documents to indexable format
index_docs = []
for doc_id, (url, doc_info) in enumerate(documents.items()):
    index_docs.append({
        'docno': str(doc_id),
        'text': doc_info['text'],
        'url': url,
        'topic': doc_info['topic'],
        'source': doc_info['source']
    })

# Create DataFrame for indexing
index_df = pd.DataFrame(index_docs)

# Define indexing pipeline with processing steps
indexer = pt.IterDictIndexer("./fitness_index")
indexer.setProperties(**{
    "tokeniser": "UTFTokeniser",
    "stemmer": "PorterStemmer",
    "stopwords.filename": "stopword-list.txt",
    "termpipelines": "Stopwords,PorterStemmer"
})

# Create the index
indexref = indexer.index(index_df.iterrows())
index = pt.IndexFactory.of(indexref)
```

Listing 4. Index Creation with Python-Terrier

E. Query Processing Implementation

Query processing was implemented with support for various retrieval models and query expansion:

```
# Create retrieval pipeline
bm25 = pt.BatchRetrieve(index, wmodel="BM25")
qe = pt.rewrite.Bo1QueryExpansion(index)
bm25_qe = bm25 >> qe >> bm25

# Add BERT re-ranking
bert_reranker = pt.reranking.PyTerrier_BERT(
    model="bert-base-uncased",
    batch_size=8
)
bert_pipeline = bm25 >> bert_reranker

# Create a configurable retrieval pipeline
def retrieve(query, use_qe=True, use_bert=False, num_results=10):
    if use_qe and use_bert:
        pipeline = bm25 >> qe >> bm25 >> bert_reranker
    elif use_qe:
        pipeline = bm25_qe
    elif use_bert:
        pipeline = bert_pipeline
    else:
        pipeline = bm25

    # Execute query and return results
    results = pipeline.search(query)
    return results.head(num_results)
```

Listing 5. Query Processing Implementation

F. User Interface Implementation

The user interface was implemented using HTML, CSS, and JavaScript with the following features:

- Responsive design for different screen sizes
- Interactive elements for search customization
- Source filtering through clickable tags
- Real-time search suggestions
- Results display with title, snippet, and URL

V. EVALUATION

A. Evaluation Methodology

The NutriQuest system was evaluated using both automated metrics and user studies:

1) Automated Evaluation: For automated evaluation, standard IR metrics were used:

- **Precision**: The fraction of retrieved documents that are relevant
- **Recall**: The fraction of relevant documents that are retrieved

- **Mean Average Precision (MAP)**: The mean of average precision scores for each query
- **Normalized Discounted Cumulative Gain (nDCG)**: Measure of ranking quality

2) User Study: A user study was conducted with 20 participants of varying fitness backgrounds, from beginners to fitness professionals. Participants were asked to perform specific search tasks and rate the relevance and usefulness of the results.

B. Test Queries

The following test queries were used for evaluation:

Query Type	Example Queries
Specific Information	"Best exercises for biceps"
Historical Information	"Mr Olympia winners history"
Practical Advice	"How to meal prep for bodybuilding"
Scientific Questions	"Muscle protein synthesis duration"

TABLE II
TEST QUERY CATEGORIES AND EXAMPLES

C. Results

1) Retrieval Performance: The system's retrieval performance was measured across different configurations:

Configuration	P@10	R@10	MAP	nDCG@10
BM25 (Baseline)	0.72	0.65	0.68	0.74
BM25 + QE	0.78	0.71	0.73	0.79
BM25 + BERT	0.81	0.68	0.75	0.83
BM25 + QE + BERT	**0.85**	**0.73**	**0.79**	**0.87**

TABLE III
RETRIEVAL PERFORMANCE ACROSS DIFFERENT CONFIGURATIONS

Fig. 3. Performance Comparison of Different Retrieval Configurations

2) Source-wise Performance: Analysis of performance across different sources:

Metric	Wikipedia	YouTube	Google
Precision@10	0.87	0.73	0.80
Recall@10	0.69	0.64	0.72
MAP	0.82	0.70	0.77
nDCG@10	0.85	0.71	0.81

TABLE IV
PERFORMANCE METRICS BY SOURCE

3) User Study Results: Results from the user study provided valuable insights:

Metric	Average Score (1-5)
Relevance of results	4.2
Information completeness	3.9
Ease of use	4.5
Search speed	4.3
Overall satisfaction	4.1

TABLE V
USER STUDY RESULTS

Fig. 4. User Satisfaction by Expertise Level

VI. DISCUSSION

A. Strengths of the System

The evaluation results highlight several strengths of the NutriQuest system:

- **Comprehensive Coverage**: By aggregating information from multiple sources, the system provides a broad coverage of fitness and nutrition topics.
- **Effective Retrieval**: The combination of BM25, query expansion, and BERT reranking significantly improves retrieval performance compared to baseline methods.
- **Source Diversity**: The inclusion of different sources enables users to access both textual and video content.
- **User-Friendly Interface**: The interface design received high usability scores in the user study.

B. Limitations and Challenges

Despite its strengths, the system has several limitations:

- **Content Freshness**: The current implementation does not continuously update its index.
- **Domain Vocabulary**: Some specialized fitness terminology may not be properly handled by standard text processing techniques.
- **Source Bias**: The quality of information varies across sources.
- **Limited Multimedia Understanding**: The system indexes YouTube links but doesn't analyze the actual video content.
- **Processing Overhead**: BERT reranking improves results but introduces significant computational overhead.

VII. CONCLUSION

A. Summary of Contributions

This project has made several contributions to the field of domain-specific information retrieval:

- Developed a comprehensive IR system for fitness and nutrition information
- Implemented and evaluated various retrieval techniques in the fitness domain
- Created a user-friendly interface for fitness information search
- Analyzed the effectiveness of different sources for fitness-related queries
- Demonstrated the value of combining traditional and neural retrieval methods

B. Key Findings

Key findings from the project include:

- The combination of BM25, query expansion, and BERT reranking provides the best retrieval performance for fitness-related queries.
- Wikipedia tends to provide more comprehensive and structured information, resulting in higher precision and MAP scores.

- User expertise level significantly impacts satisfaction with search results, with beginners valuing simplicity and clarity while advanced users prioritize depth and specificity.
- Query expansion is particularly beneficial for fitness-related searches due to the specialized terminology and synonyms commonly used in the domain.

C. Future Work

Future research and development directions include:

- **Personalization**: Developing user profiles to personalize search results based on fitness goals and preferences
- **Multimedia Analysis**: Incorporating image and video analysis to index and search visual content directly
- **Expert Verification**: Implementing a system for expert verification of fitness information
- **Semantic Understanding**: Enhancing semantic understanding of fitness concepts through knowledge graphs and domain-specific embeddings
- **Multilingual Support**: Extending the system to support multiple languages for global accessibility

VIII. BERT Reranking Implementation

The BERT reranking component was crucial for improving semantic understanding of queries:

```
class BERTReranker:
    def __init__(self, model_name="bert-base-
uncased", batch_size=8):
        self.tokenizer = AutoTokenizer.
from_pretrained(model_name)
        self.model = AutoModel.from_pretrained
(model_name)
        self.batch_size = batch_size

    def _encode_text(self, text):
        # Tokenize and encode text
        encoded = self.tokenizer.encode_plus(
            text,
            max_length=512,
            truncation=True,
            padding='max_length',
            return_tensors='pt'
        )
        with torch.no_grad():
            outputs = self.model(**encoded)
            # Use CLS token embedding as text
representation
            embeddings = outputs.
last_hidden_state[:, 0, :]
        return embeddings

    def rerank(self, query, documents, scores)
:
        # Encode query
        query_emb = self._encode_text(query)

        new_scores = []
        # Process documents in batches
        for i in range(0, len(documents), self
.batch_size):
            batch_docs = documents[i:i+self.
batch_size]
            batch_embs = torch.cat([self.
_encode_text(doc) for doc in batch_docs])

            # Compute similarity
            similarities = torch.nn.functional
.cosine_similarity(
                query_emb.unsqueeze(0),
batch_embs
            )

            # Combine with original BM25
scores
            batch_scores = [scores[j] for j in
 range(i, min(i+self.batch_size, len(
scores)))]
            combined_scores = [0.3 * bs + 0.7
* sim.item() for bs, sim in zip(
batch_scores, similarities)]
            new_scores.extend(combined_scores)

        # Return reranked results
        reranked_results = list(zip(documents,
 new_scores))
        reranked_results.sort(key=lambda x: x
[1], reverse=True)
        return reranked_results
```

Listing 6. BERT Reranking Implementation

IX. AI Enhancements for NutriQuest

A. AI-Powered Chatbot Interface

To enhance user interaction with the NutriQuest system, an AI-powered chatbot interface was implemented. This chatbot leverages large language models (LLMs) to provide a conversational experience for users seeking fitness and nutrition information.

Fig. 5. NutriQuest AI Chatbot Interface

The chatbot implementation includes:

- **Natural Language Understanding**: Processes user queries expressed in natural language
- **Context Preservation**: Maintains conversation history to provide contextually relevant responses
- **Domain-Specific Knowledge**: Fine-tuned with fitness and nutrition terminology
- **Hybrid Retrieval**: Combines information retrieval with generative capabilities

```
class NutriQuestChatbot:
    def __init__(self, retrieval_system,
llm_model="gpt-3.5-turbo"):
        self.retrieval_system =
retrieval_system
        self.llm = LLMInterface(model=
llm_model)
        self.conversation_history = []

    def process_query(self, user_query):
        # Add query to conversation history
        self.conversation_history.append({"
role": "user", "content": user_query})

```

```
        # Retrieve relevant documents
        retrieved_docs = self.retrieval_system.search(user_query, top_k=5)

        # Construct prompt with retrieved information
        prompt = self._construct_prompt(user_query, retrieved_docs)

        # Generate response using LLM
        response = self.llm.generate(
            prompt,
            conversation_history=self.conversation_history
        )

        # Add response to history
        self.conversation_history.append({"role": "assistant", "content": response})

        return response

    def _construct_prompt(self, query, documents):
        context = "\n\n".join([doc.content for doc in documents])
        prompt = f"""
        You are a fitness and nutrition expert assistant for NutriQuest.
        Answer the user's question based on the following information:

        {context}

        If the information doesn't contain the answer, say so honestly and
        suggest related topics the user might explore instead.

        User question: {query}
        """
        return prompt
```

Listing 7. AI Chatbot Integration

B. Personalized AI Recommendations

NutriQuest implements a personalized recommendation system that leverages user preferences, search history, and fitness goals to provide tailored content recommendations.

Fig. 6. AI-Powered Recommendation System Architecture

The recommendation system employs a hybrid approach combining:

- **Content-Based Filtering**: Analyzing document features and user preferences
- **Collaborative Filtering**: Identifying patterns among similar users
- **Deep Learning**: Embedding-based similarity using neural networks
- **Contextual Awareness**: Adapting recommendations based on user's current goals

The recommendation algorithm updates user profiles based on implicit and explicit feedback:

Algorithm 2 Personalized Recommendation Algorithm

procedure GENERATERECOMMENDATIONS($user\_id$)
  $profile \leftarrow$ GetUserProfile($user\_id$)
  $history \leftarrow$ GetUserHistory($user\_id$)
  $embeddings \leftarrow$ GenerateEmbeddings($profile, history$)
  $candidates \leftarrow$ RetrieveCandidateDocuments()
  $scores \leftarrow []$
  for doc in $candidates$ **do**
    $content\_score \leftarrow$ ComputeContentSimilarity($doc, profile$)
    $collab\_score \leftarrow$ ComputeCollaborativeScore($doc, user\_id$)
    $context\_score \leftarrow$ ComputeContextualRelevance($doc, profile.goals$)
    $final\_score \leftarrow 0.4 * content\_score + 0.3 * collab\_score + 0.3 * context\_score$
    $scores.append((doc, final\_score))$
  end for
  $scores.sort(reverse = True)$
  return $scores[0:10]$
end procedure

C. AI for Content Quality Verification

To address the challenge of misinformation in fitness and nutrition, an AI-powered content verification system was implemented to assess the reliability and accuracy of information.

Fig. 7. AI Content Verification Pipeline

The content verification system:

- **Source Credibility Assessment**: Evaluates the reputation and credentials of content sources
- **Claim Detection**: Identifies specific fitness and nutrition claims within content
- **Evidence Matching**: Searches for scientific evidence supporting or contradicting claims
- **Uncertainty Quantification**: Highlights areas where scientific consensus is limited
- **Content Labeling**: Provides reliability indicators alongside search results

```
class ContentVerifier:
    def __init__(self, scientific_database, credibility_model):
        self.scientific_db = scientific_database
        self.credibility_model = credibility_model
        self.claim_detector = ClaimDetectionModel()

    def verify_document(self, document):
        # Extract claims from document
```

```python
        claims = self.claim_detector.extract_claims(document.text)

        verification_results = []
        for claim in claims:
            # Search for scientific evidence
            evidence = self.scientific_db.search_evidence(claim.text)

            # Assess source credibility
            source_score = self.credibility_model.evaluate_source(document.source)

            # Compute overall reliability score
            evidence_strength = self._calculate_evidence_strength(evidence)
            reliability_score = 0.6 * evidence_strength + 0.4 * source_score

            verification_results.append({
                'claim': claim.text,
                'reliability_score': reliability_score,
                'evidence': evidence[:3],  # Top 3 pieces of evidence
                'confidence': self._calculate_confidence(evidence)
            })

        return verification_results

    def _calculate_evidence_strength(self, evidence_list):
        # Implementation of evidence strength calculation
        # based on study types, sample sizes, and consistency
        pass

    def _calculate_confidence(self, evidence_list):
        # Implementation of confidence calculation
        # based on amount and quality of evidence
        pass
```

Listing 8. Content Verification Implementation

D. AI-Enhanced Query Understanding

The query understanding module uses advanced natural language processing to better interpret user intent and provide more relevant results.

Fig. 8. AI-Enhanced Query Understanding Process

Key components include:

- **Intent Recognition**: Classifies queries into categories (e.g., information-seeking, advice-seeking)
- **Entity Extraction**: Identifies fitness concepts, exercises, nutrients, etc.
- **Query Reformulation**: Expands queries with domain-specific knowledge
- **Ambiguity Resolution**: Clarifies ambiguous terms in the fitness context

The system uses a transformer-based architecture to encode and understand user queries:

```python
class QueryUnderstanding:
    def __init__(self, fitness_ontology, transformer_model="bert-base-uncased"):
        self.ontology = fitness_ontology
        self.transformer = AutoModel.from_pretrained(transformer_model)
        self.tokenizer = AutoTokenizer.from_pretrained(transformer_model)
        self.intent_classifier = IntentClassificationHead(self.transformer.config.hidden_size)
        self.entity_extractor = EntityExtractionModel()

    def process_query(self, query_text):
        # Tokenize and encode query
        tokens = self.tokenizer(query_text, return_tensors="pt", padding=True, truncation=True)
        with torch.no_grad():
            outputs = self.transformer(**tokens)

        # Extract features
        features = outputs.last_hidden_state

        # Classify intent
        intent = self.intent_classifier(features[:, 0, :])  # Use CLS token

        # Extract entities
        entities = self.entity_extractor.extract(query_text)

        # Expand query with ontology knowledge
        expanded_terms = []
        for entity in entities:
            related_concepts = self.ontology.get_related_concepts(entity.text, entity.type)
            expanded_terms.extend(related_concepts)

        # Handle ambiguity
        ambiguous_terms = self.identify_ambiguous_terms(query_text, entities)

        return {
            'original_query': query_text,
            'intent': intent,
            'entities': entities,
            'expanded_terms': expanded_terms,
            'ambiguous_terms': ambiguous_terms
        }

    def identify_ambiguous_terms(self, query, entities):
        # Implementation of ambiguity
```

```
    detection
        pass
```

Listing 9. Query Understanding Implementation

Figure 9 shows the planned roadmap for AI feature integration in NutriQuest:

Fig. 9. NutriQuest AI Enhancement Roadmap

Preliminary experiments with multimodal understanding have shown promising results:

Task	Accuracy	Processing Time (ms)
Exercise Recognition	92.3%	156
Form Analysis	85.7%	289
Food Recognition	89.1%	175
Nutrition Estimation	78.4%	210

TABLE VI
PRELIMINARY RESULTS OF MULTIMODAL AI FEATURES