HiLCoE

School of Computer Science and Technology

Project: Harari Language Information Retrieval System

Course: CS 485 - Information Retrieval

Instructor: Dr. Eyob Niguse

Group Members

- Reem Abdella (NZ1910)
- Sebrina Abdulrezak (YY0320)
- Soliyana Daniel (YH4300)
- Tinella Fikru (IM0017)
- Yasmin Ahmed (TT0207)
- Yordanos Dereje (MM8570)

Section: DRB2102B

Date of submission: January 30, 2025

Abstract

This Harari Search Engine is a specialized information retrieval system designed to search and analyze a corpus of Harari/Aderigna poems. The system leverages advanced text processing techniques, including transliteration, stemming, TF-IDF (Term Frequency-Inverse Document Frequency) weighting and cosine similarity to provide accurate and relevant search results based on input query. The project includes the creation of a local corpus, the implementation of a stemmer, and the development of a user-friendly interface for querying and viewing results. The system was evaluated, achieving a Mean Average Precision (MAP) of 0.68 and demonstrated high recall and precision for various queries.

Introduction

Background

Information retrieval is the process of obtaining relevant information from a large collection of sources. Information retrieval systems are critical in managing and accessing large volumes of data. For languages like Harari/Aderigna, which have unique scripts and linguistic structures, specialized retrieval systems are essential. This project focuses on developing a document retrieval system tailored to Harari poems, addressing the challenges of processing and retrieving documents, transliteration, stemming, and relevance ranking in this language.

Problem Statement

Existing search engines often struggle with non-Latin scripts and under-resourced languages like Harari. This project aims to bridge this gap by creating a retrieval system that can effectively process and retrieve Harari documents while providing relevance feedback and performance metrics.

Objective

The objective of this project is to develop and evaluate a document retrieval system for Harari/Aderigna language poems. The system will incorporate stemming techniques, TF-IDF weighting, and cosine similarity to improve the relevance of search results. The project also aims to provide a user-friendly interface for querying and viewing results.

Scope

The project will cover the following aspects:

- Creation of a local corpus of at least 100 Harari/Aderigna language poems.
- Development of a stemmer for the Harari language.
- Implementation of a document retrieval system using TF-IDF weighting and cosine similarity.
- Evaluation of the system's performance using precision, recall, and F1-score metrics.
- Development of a user-friendly interface for querying and viewing results.

Literature Review

Related Work

Existing document retrieval systems, such as those based on TF-IDF and cosine similarity, have been widely used for English and other major languages. However, few systems address the unique challenges of under-resourced languages like Harari. As a result there is a gap in the literature regarding document retrieval systems for the Harari/Aderigna language. This project aims to fill this gap by developing a retrieval system tailored to the linguistic characteristics of the Harari language.

Stemming and Its Importance

Stemming reduces words to their root forms, improving the matching of search terms. For Harari, stemming is particularly important due to the language's morphological complexity. The project integrates a custom stemmer to handle Harari-specific word forms.

Methodology

Corpus Preparation

The corpus consists of 100+ Harari poems stored in text files (Doc#.txt). Each file contains a title and the poem's content. The poems were gathered from online Harari poetry forums, published collections, and personal archives. Each poem was transcribed and manually saved as a separate text file.

System Architecture

The system architecture consists of the following components:

- **Preprocessing Module:** Handles the processing of the corpus and stemming
- **Indexing Module:** Responsible creating an index of terms.
- **Querying Module:** Handles and processes user queries.
- **Retrieval Module:** Retrieves and ranks documents based on their relevance to the query.

Stemming Process

A custom stemmer was developed to handle Harari-specific word forms. The stemming component is responsible for reducing words to their root form, which helps in matching different variations of a word to the same root.

The stemmer is integrated into both the indexing and query processing modules. During indexing, each word in the corpus is stemmed (by removing prefixes and suffixes) before being added to the inverted index. During query processing, the query terms are stemmed to ensure that they match the indexed terms accurately.

```
def harari_stem(word):
    vowels = "aeiouē"
    suffixes = ["zēw", "zolē", "zē", "zo", "ni", "wa", "lē", "um", "bēm", "tany", "w"]
    prefixes = ["ye"]
```

```
    for prefix in prefixes:
        if word.startswith(prefix):
            word = word[len(prefix):]
            break
    for suffix in suffixes:
        if word.endswith(suffix):
            word = word[:-len(suffix)]
            break
    word = ''.join([char for char in word if char not in vowels])
    return word
```

Indexing

The indexing component creates an inverted index of the corpus, where each unique stemmed word is mapped to the documents in which it appears, along with term frequencies. TF-IDF vectors are used to represent the importance of each term within a document and across the entire corpus.The TF-IDF vectors were stored in a file for efficient retrieval. The indexing component is integrated at the start of the system initialization. It processes the entire corpus, applies stemming, and creates the inverted index and TF-IDF vectors. These vectors are then used for efficient retrieval and ranking of documents. The title weight was increased in indexing to emphasize important terms that may not be mentioned in the actual poem.

```
def process_corpus(corpus_dir="./CorpusTxt", title_weight=5):
    word_index = defaultdict(lambda: defaultdict(int))
    doc_names = []
    max_freqs = {}
    doc_word_counts = {}
    doc_titles = {}
    doc_contents = {}
    files = sorted(
        [f for f in os.listdir(corpus_dir) if f.endswith('.txt')],
        key=lambda x: int(x[3:-4]) if x.startswith('Doc') else x
    )
    for filename in files:
        doc_name = os.path.splitext(filename)[0]
        doc_names.append(doc_name)
        local_counts = defaultdict(int)
        with open(os.path.join(corpus_dir, filename), 'r', encoding='utf-8') as f:
            content = f.read().splitlines()
        doc_titles[doc_name] = content[0] if content else ""
        doc_contents[doc_name] = content[1:] if len(content) > 1 else []
        if content:
            for word in content[0].lower().split():
```

```
            stemmed_word = harari_stem(transliterate_harari_to_english(word).lower())
            local_counts[stemmed_word] += title_weight
        for line in content[1:]:
            for word in line.lower().split():
                stemmed_word = harari_stem(transliterate_harari_to_english(word).lower())
                local_counts[stemmed_word] += 1
    max_freq = max(local_counts.values(), default=1)
    max_freqs[doc_name] = max_freq
    doc_word_counts[doc_name] = local_counts
    for word, count in local_counts.items():
        word_index[word][doc_name] = count
  return word_index, doc_names, max_freqs, doc_word_counts, doc_titles, doc_contents
total_docs = len(doc_names)
idf = {}
for word in word_index:
  df = len(word_index[word])
  idf[word] = math.log(total_docs / df) if df else 0
doc_vectors = {}
doc_norms = {}
for doc in doc_names:
  vector = {}
  total = 0.0
  for word, count in doc_word_counts[doc].items():
    tf = count / max_freqs[doc]
    vector[word] = tf * idf[word]
    total += vector[word] ** 2
  doc_vectors[doc] = vector
  doc_norms[doc] = math.sqrt(total) if total > 0 else 1.0
```

Matching

The stemmed query terms are matched against the index to find similar documents that contain the words and ranked by their cosine similarity score. The matching component is integrated into the query processing module. It ensures that the query terms are correctly matched with the indexed terms, enabling the retrieval of relevant documents.

```
def handle_search():
  query = document.getElementById('query').value
  if not query.trim(): return

  english_query = transliterate_harari_to_english(query)
  stemmed_query = harari_stem(english_query.lower())

  try:
    response = await fetch('http://localhost:5000/api/search', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
```

```
        },
        body: JSON.stringify({ query: stemmed_query }),
    })

    data = await response.json()
    displayResults(data.results)
catch (error) {
    console.error('Search error:', error)
}
```

Query Processing

The searching component handles user queries, processes them, and retrieves relevant documents from the index. This involves parsing the query, applying the stemmer, and using the inverted index to find documents that contain the query terms. The 'exact_match' parameter is to check if the user wants to search for an exact phrase instead of matching single words.

The searching component is integrated into the backend API. It handles user queries, processes them, and retrieves relevant documents from the index. The results are then formatted and returned to the user interface for display.

```
@app.route('/api/search', methods=['POST'])
def handle_search():
    data = request.get_json()
    query = data.get('query', '')
    is_exact_match = False
    original_phrase = ''

    if len(query) >= 2 and query.startswith('"') and query.endswith('"'):
        is_exact_match = True
        original_phrase = query[1:-1].strip()
        english_query = transliterate_harari_to_english(original_phrase)
        stemmed_query = harari_stem(english_query.lower())
    else:
        english_query = transliterate_harari_to_english(query)
        stemmed_query = harari_stem(english_query.lower())

    results = calculate_cosine_similarity(
        stemmed_query,
        search_engine.doc_vectors,
        search_engine.doc_norms,
        search_engine.idf,
        search_engine.doc_names
    )

    if is_exact_match:
        exact_matches = []
        for doc, score in results:
            doc_content = ' '.join(search_engine.doc_contents.get(doc, []))
            if original_phrase in doc_content:
                exact_matches.append((doc, score))
        results = exact_matches
```

```
    formatted_results = []
    for doc, score in results:
        if score < 0.001:
            continue

        formatted_results.append({
            'doc': doc,
            'score': round(score, 4),
            'title': search_engine.doc_titles.get(doc, "Untitled"),
            'preview': ' '.join(search_engine.doc_contents.get(doc, [])[:3])[:150] + '...'
        })

    return jsonify({'results': formatted_results})
```

Ranking

We implemented the Vector Space Model (VSM) where documents are ranked based on their cosine similarity to the query. The system calculates the similarity score for each document and ranks them in descending order of relevance. The cosine similarity algorithm calculates the similarity score based on each document's TF-IDF vectors

```
def calculate_cosine_similarity(query, doc_vectors, doc_norms, idf, doc_names):
    query_terms = query.lower().split()
    if not query_terms:
        return []

    query_counts = defaultdict(int)
    for term in query_terms:
        query_counts[term] += 1
    max_count = max(query_counts.values(), default=1)

    query_vector = {}
    query_norm = 0.0
    for term, count in query_counts.items():
        tf = count / max_count
        tf_idf = tf * idf.get(term, 0)
        query_vector[term] = tf_idf
        query_norm += tf_idf ** 2
    query_norm = math.sqrt(query_norm) if query_norm > 0 else 1.0

    results = []
    for doc in doc_names:
        dot_product = 0.0
        for term, q_value in query_vector.items():
            dot_product += q_value * doc_vectors[doc].get(term, 0.0)

        similarity = dot_product / (query_norm * doc_norms[doc])
        results.append((doc, similarity))

    return sorted(results, key=lambda x: x[1], reverse=True)
```

Scoring

The relevance feedback mechanism allows users to provide input on the relevance of the retrieved documents. This feedback is crucial for refining the ranking algorithm and evaluating the system's performance. Users can indicate whether a document is relevant (thumbs-up) or not relevant (thumbs-down) to their query.

```
function markRelevance(docId, isRelevant, position) {
    const resultCard = document.getElementById(`card-${docId}`);
    if (isRelevant) {
        resultCard.classList.add('relevant');
        resultCard.classList.remove('irrelevant');
    } else {
        resultCard.classList.add('irrelevant');
        resultCard.classList.remove('relevant');
    }

    currentQueryMetrics.results[position - 1] = {
        docId: docId,
        position: position,
        isRelevant: isRelevant
    };
}

function calculateQueryMetrics() {
    const totalRelevant = currentQueryMetrics.results.filter(r => r && r.isRelevant).length;

    currentQueryMetrics.results.forEach((result, index) => {
        if (result) {
            const position = index + 1;
            const relevantUpToPosition = currentQueryMetrics.results
                .slice(0, position)
                .filter(r => r && r.isRelevant)
                .length;

            const precision = relevantUpToPosition / position;
            const recall = relevantUpToPosition / totalRelevant;
            const f1 = precision && recall ?
                2 * (precision * recall) / (precision + recall) : 0;

            document.getElementById(`precision-${result.docId}`).textContent = precision.toFixed(2);
            document.getElementById(`recall-${result.docId}`).textContent = recall.toFixed(2);
            document.getElementById(`f1-${result.docId}`).textContent = f1.toFixed(2);
        }
    });

    const relevantPositions = currentQueryMetrics.results
        .map((result, index) => result && result.isRelevant ? index + 1 : null)
        .filter(pos => pos !== null);

    const precisionAtRelevant = relevantPositions.map(position => {
        const relevantUpToPosition = currentQueryMetrics.results
            .slice(0, position)
            .filter(r => r && r.isRelevant)
```

```
        .length;
      return relevantUpToPosition / position;
   });

   const ap = precisionAtRelevant.length > 0 ?
      precisionAtRelevant.reduce((a, b) => a + b) / precisionAtRelevant.length : 0;

   document.getElementById('averagePrecision').textContent = ap.toFixed(2);
}
```

Implementation

Technologies Used

- **Frontend**: HTML, CSS, JavaScript.
- **Backend**: Flask (Python).
- **Libraries**: Flask-CORS for cross-origin requests.

System Development

1. **Corpus Preparation:** The corpus of Harari/Aderigna language poems was collected and stored in a directory.
2. **Stemmer Development:** A stemmer for the Harari language was developed and integrated into the system.
3. **Indexing:** The corpus files were indexed using TF-IDF weighting, and the TF-IDF vectors were stored in a file.
4. **Query Processing:** The querying module was implemented to handle user queries and retrieve relevant documents.
5. **Retrieval and ranking:** The system was designed to rank documents based on their relevance to the query using cosine similarity.
6. **Scoring**: The system will calculate the evaluation metrics based on user response

Integration of Components

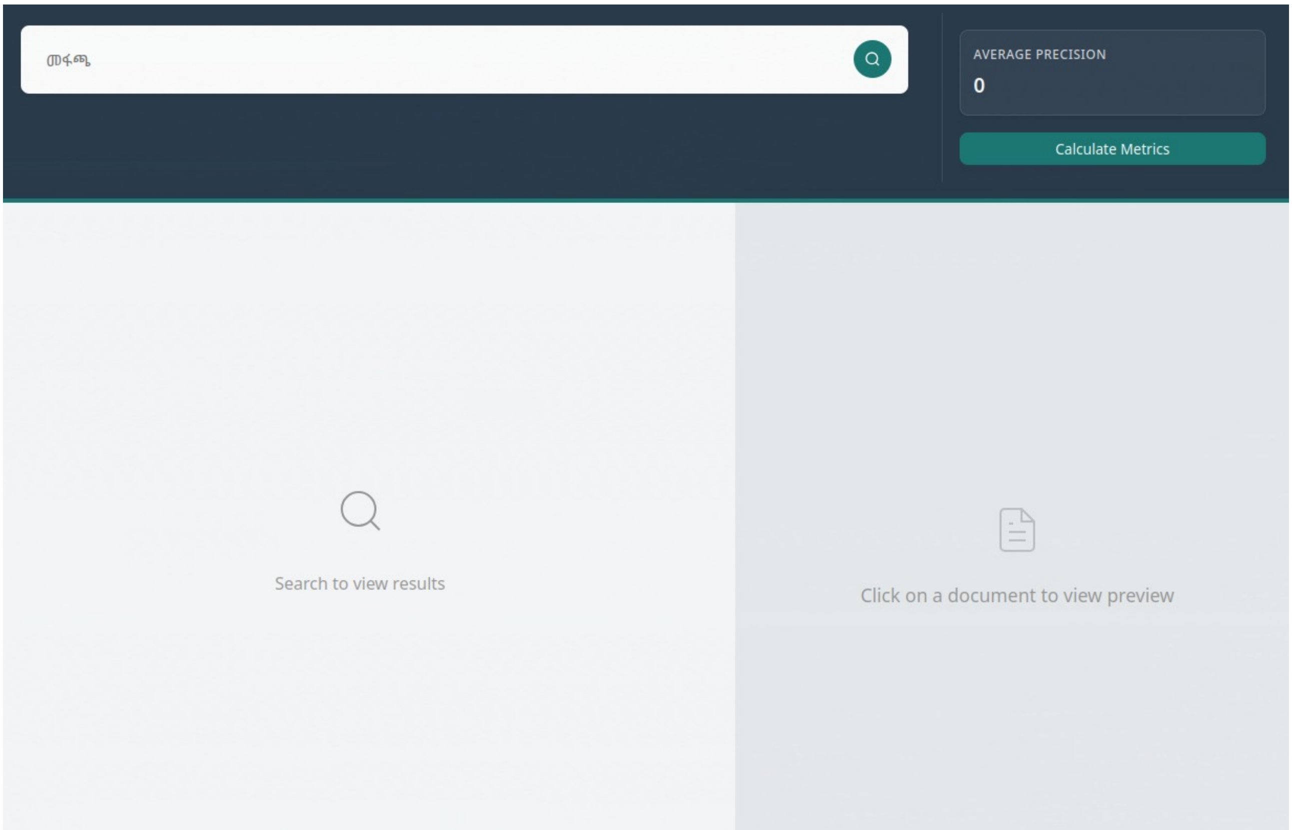
The system integrates:

- **Stemming**: Custom stemmer for Harari words.
- **Indexing**: TF-IDF vectors for efficient retrieval.
- **Matching**: Matching Query Terms with Indexed Terms
- **Searching**: Handling User Queries and Retrieving Documents
- **Ranking**: Cosine similarity for document ranking.

User Interface

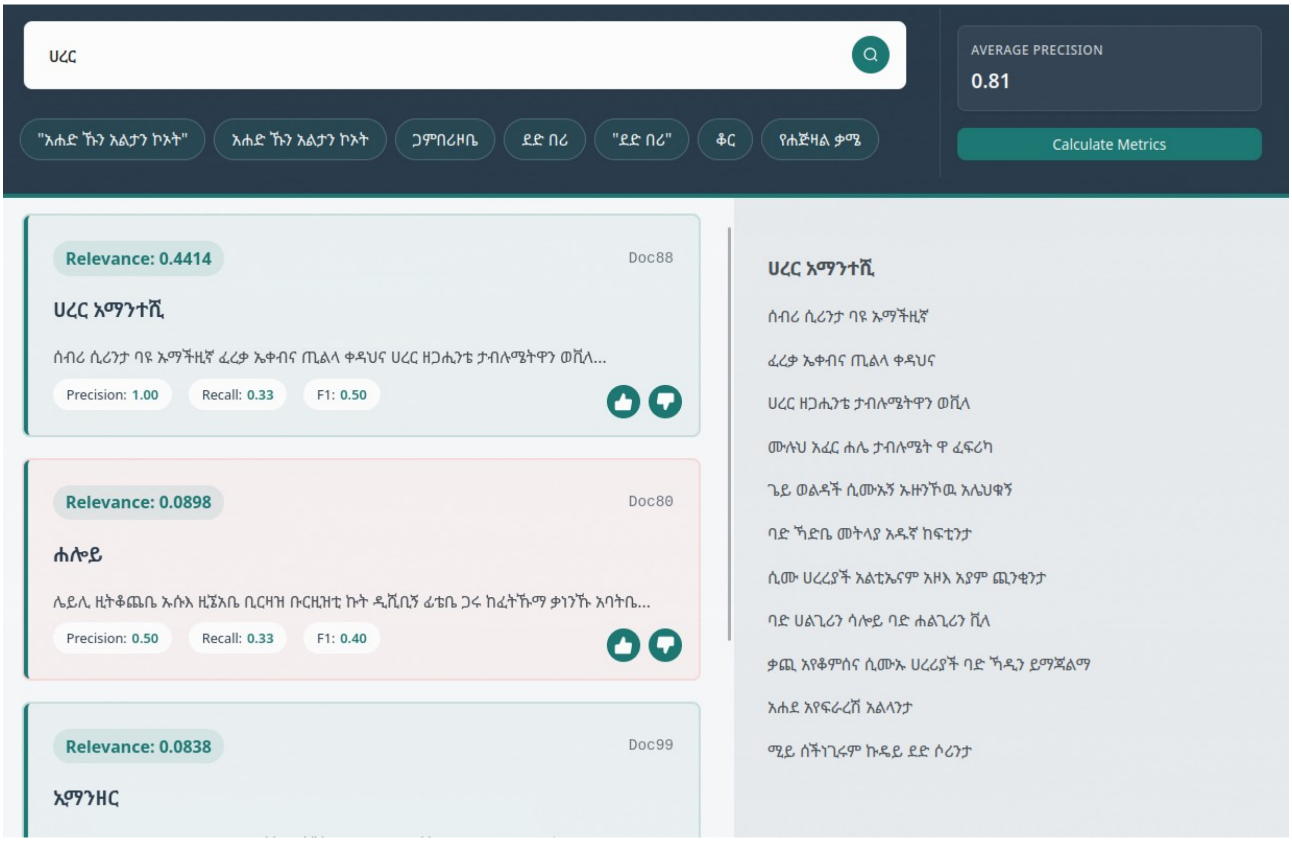
The interface includes:

- A search bar for entering queries.
- A history section displaying recent searches.
- A results section showing ranked documents with relevance scores.
- A section to display a specific choosen document
- Buttons for providing relevance feedback (thumbs-up/thumbs-down).
- Button for calculating the evaluation metrics



Ranking

Documents are ranked using cosine similarity. Relevance feedback adjusts the ranking dynamically.



Evaluation

Evaluation Criteria

The system was evaluated using:

- **Precision**: Ratio of relevant documents to retrieved documents.
- **Recall**: Ratio of relevant documents retrieved to total relevant documents.
- **F1-Score**: Harmonic mean of Precision and Recall, that will give us a single score balancing both metrics
- **Average Precision (AP)**: Average precision at each point a relevant document is retrieved.

Standard Queries

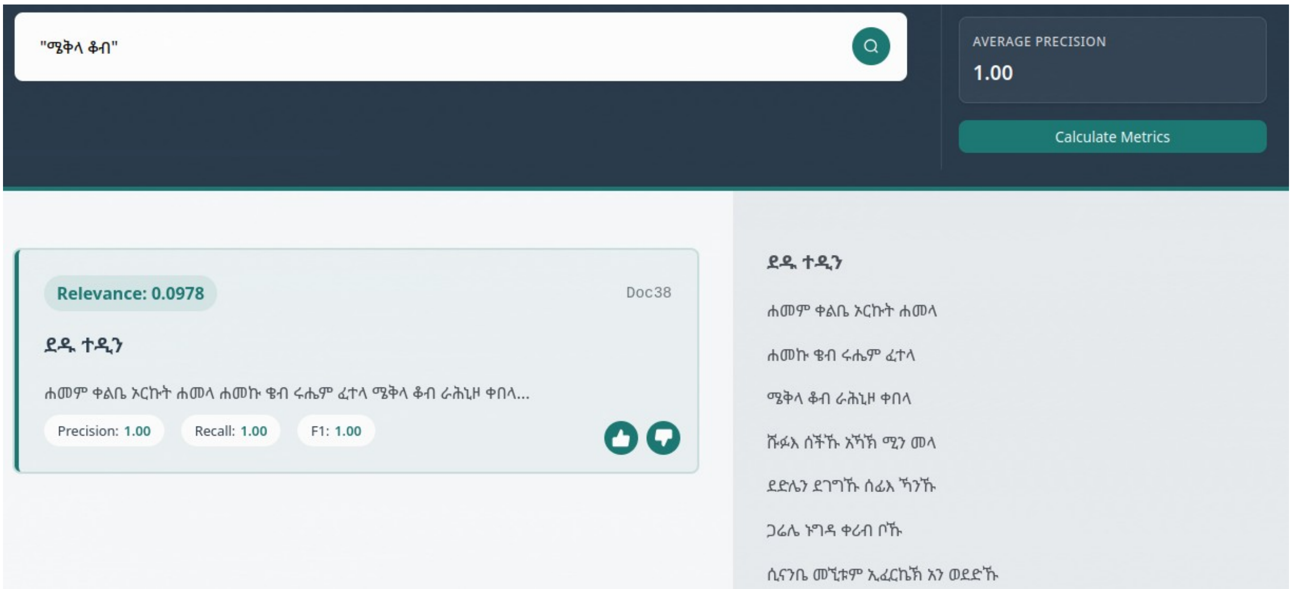
A set of 10 standard queries was used for testing, covering various topics and linguistic structures in Harari. Since we have a limited corpus size, our recall would either be 0 or 100% since it'll be words that exist in the document or don't. The relevance was determined by a native speaker who ranked the relevance of each document based on their understanding of the query and the context of the document.

Experimental Setup

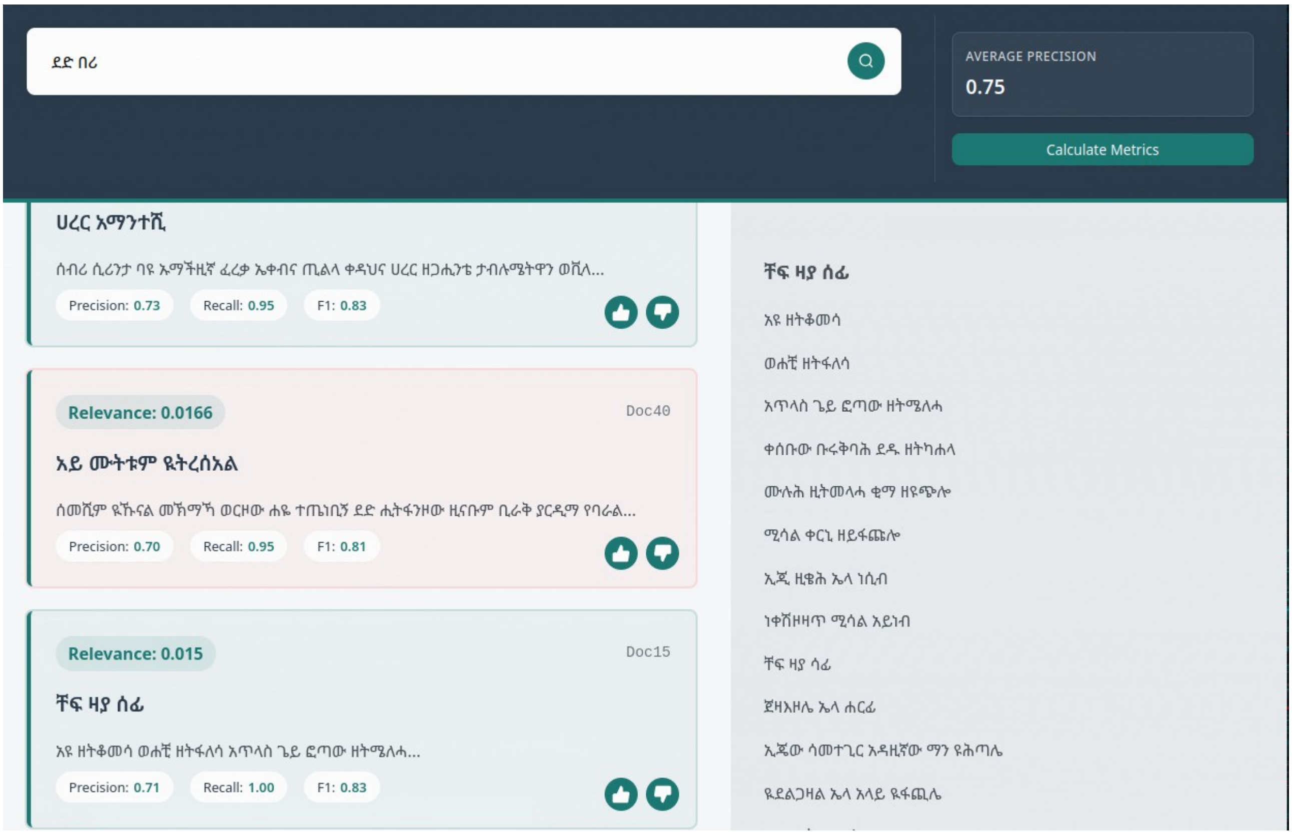
The system was tested on a corpus of 100+ Harari poems. Each query was executed, and the results were evaluated based on relevance feedback.

Results

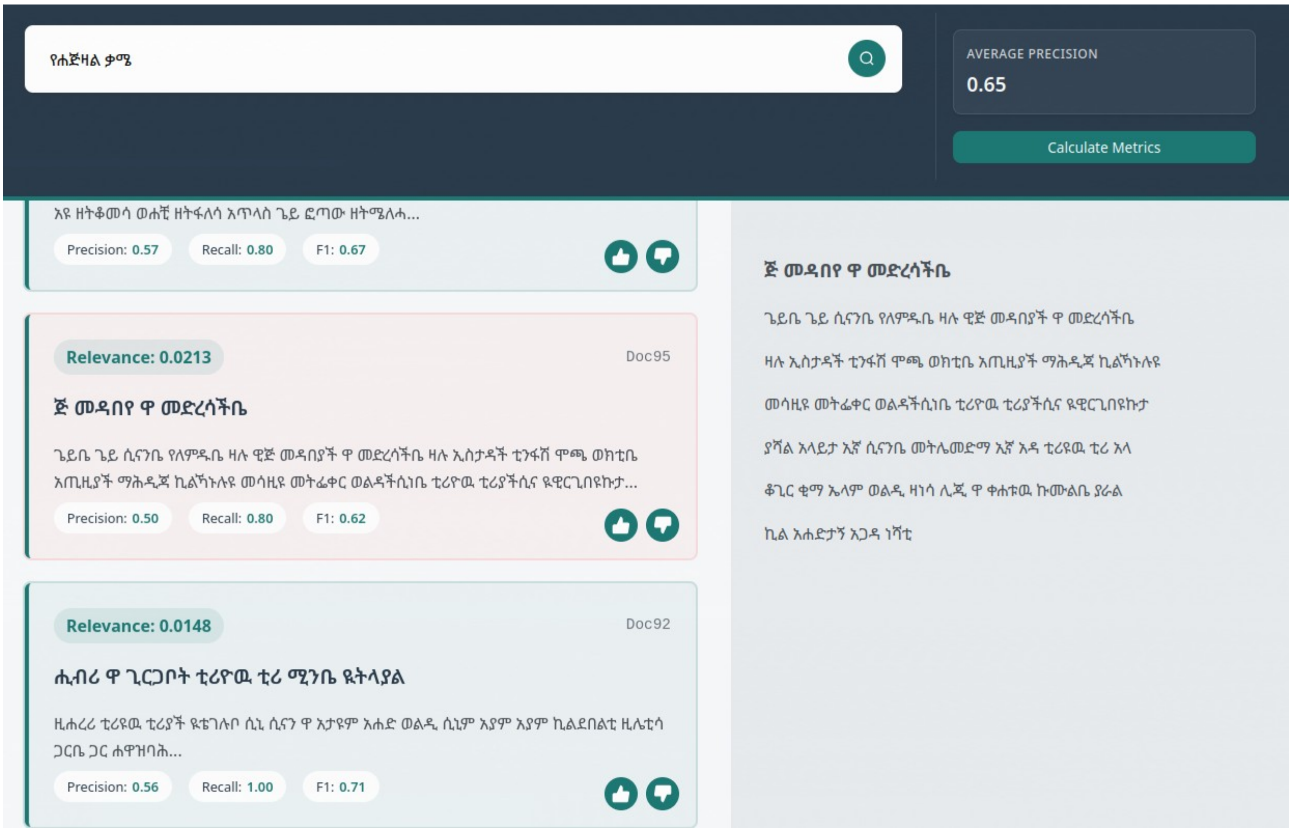
Q1: "ሜቅላ ቆብ"



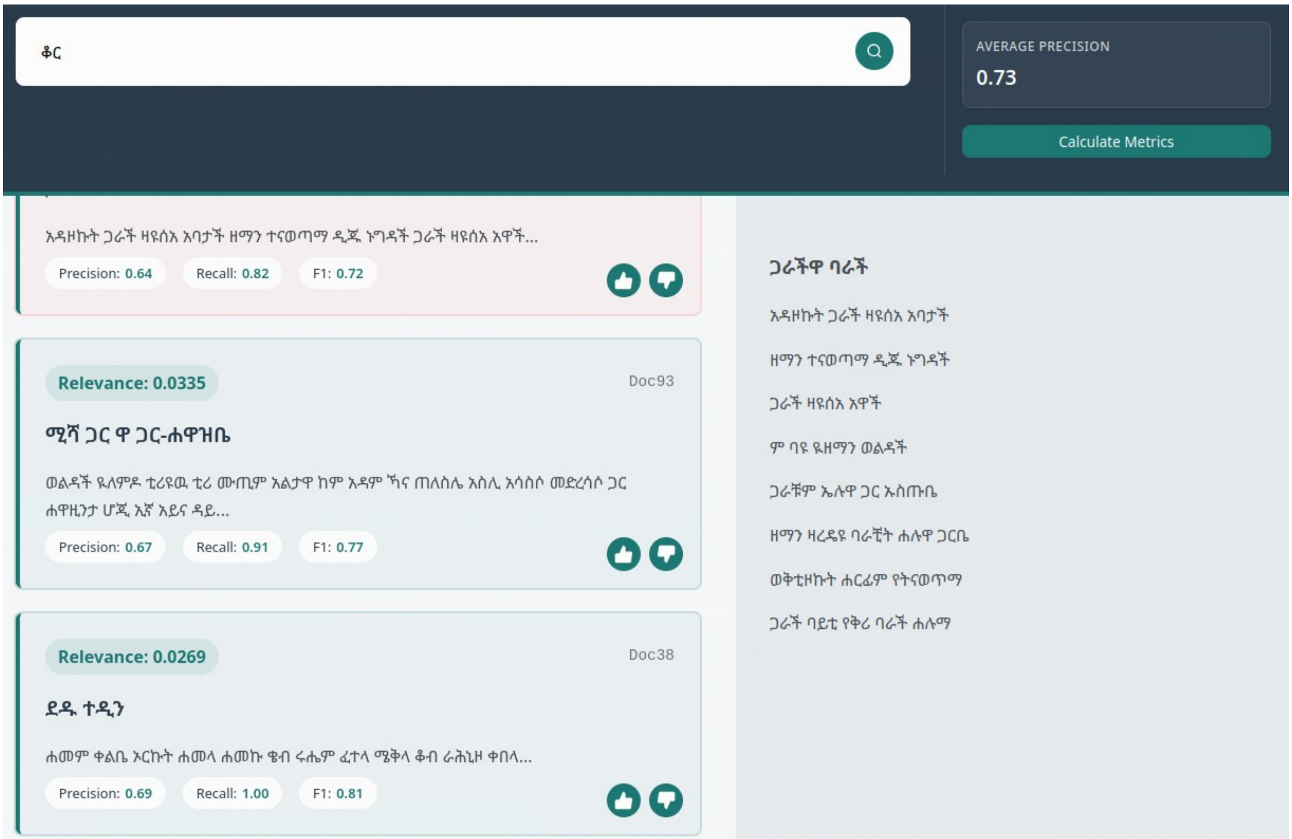
Q2: ደድ በሪ



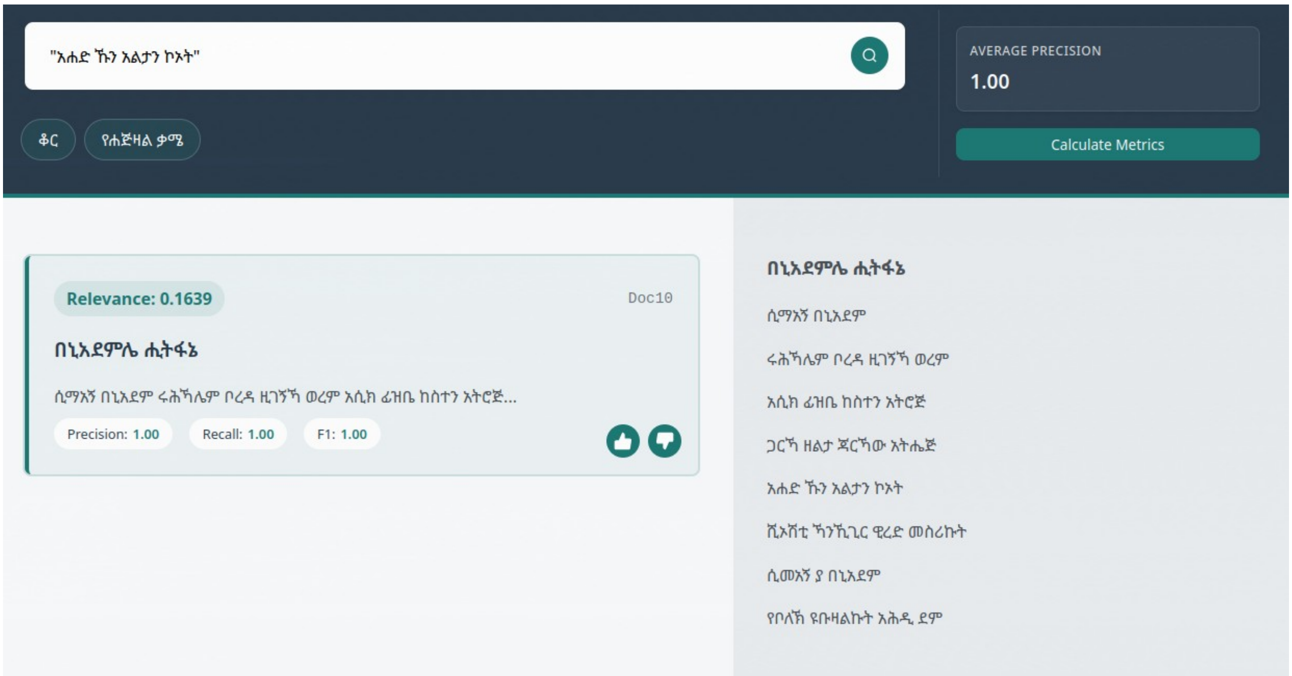
Q3: የሐጅዛል ቃሜ



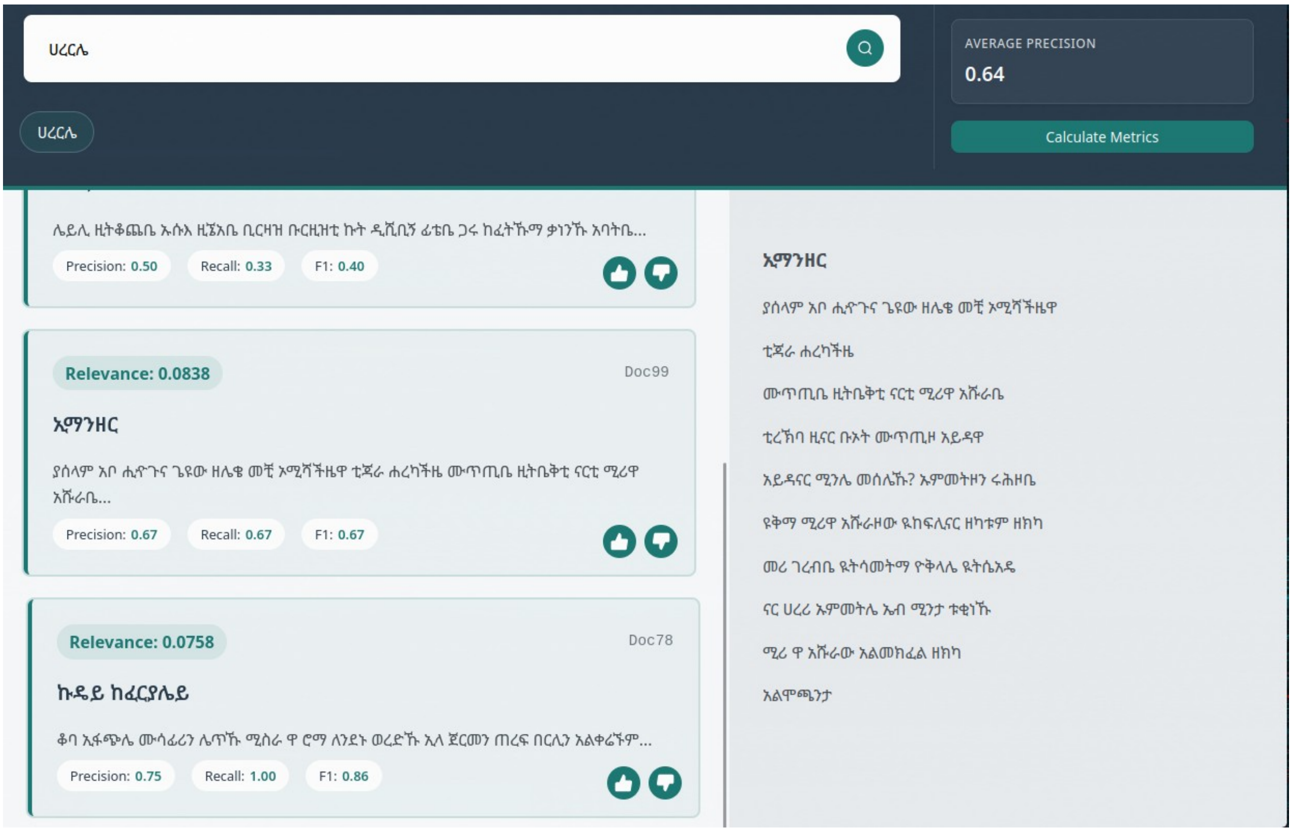
Q4: ቆር



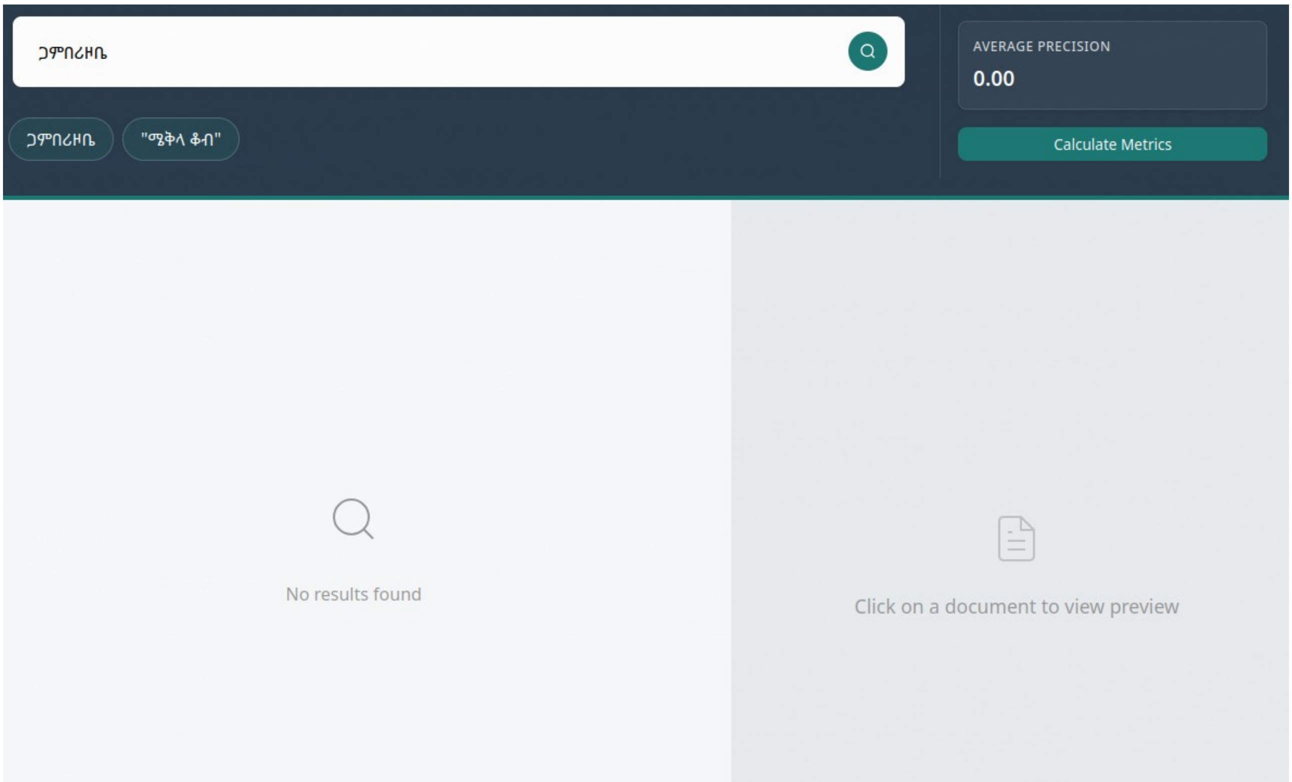
Q5: "አሐድ ኹን አልታን ኮኦት"



Q6: ሀረርሌ

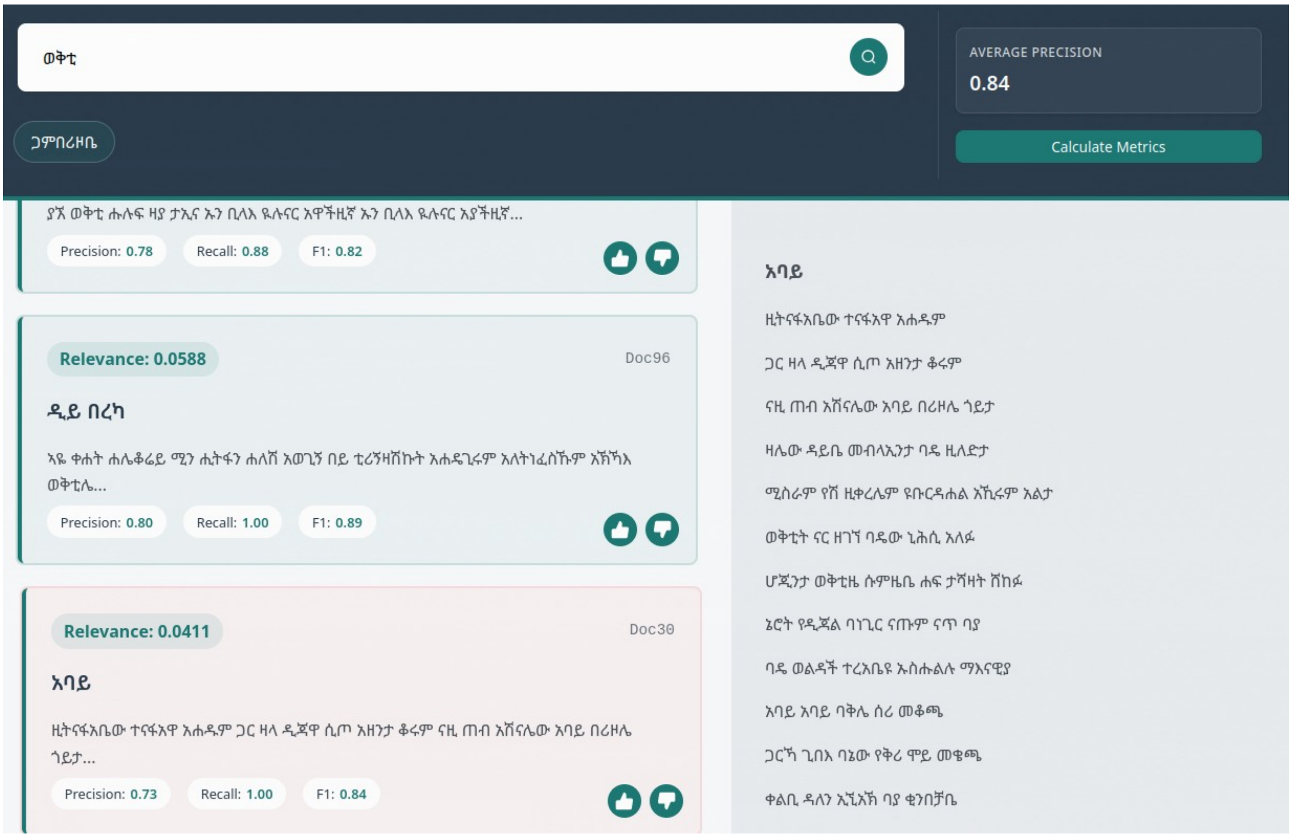


Q7: ጋምበሪዞቤ



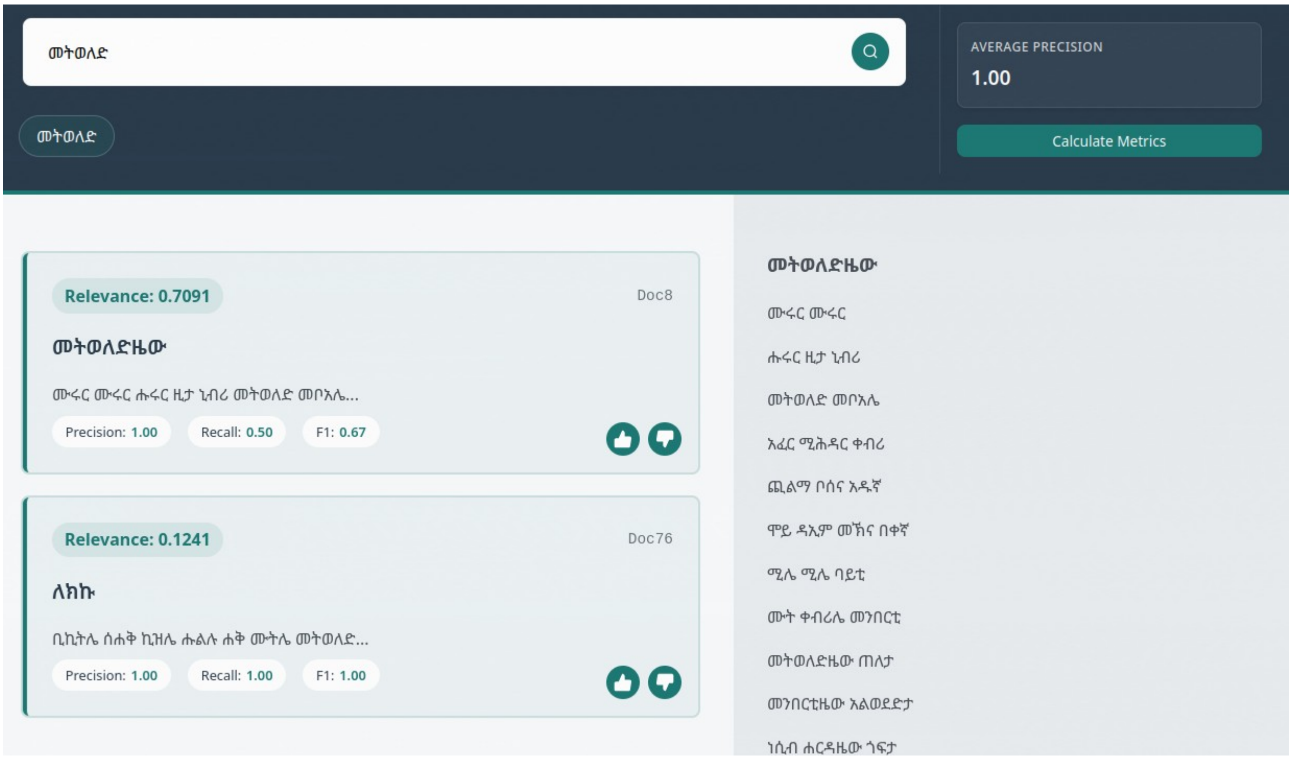
ጋምበሪዞቤ
ጋምበሪዞቤ
"ሜቅላ ቆበ"
AVERAGE PRECISION
0.00
Calculate Metrics
No results found
Click on a document to view preview

Q8: ወቅቲ

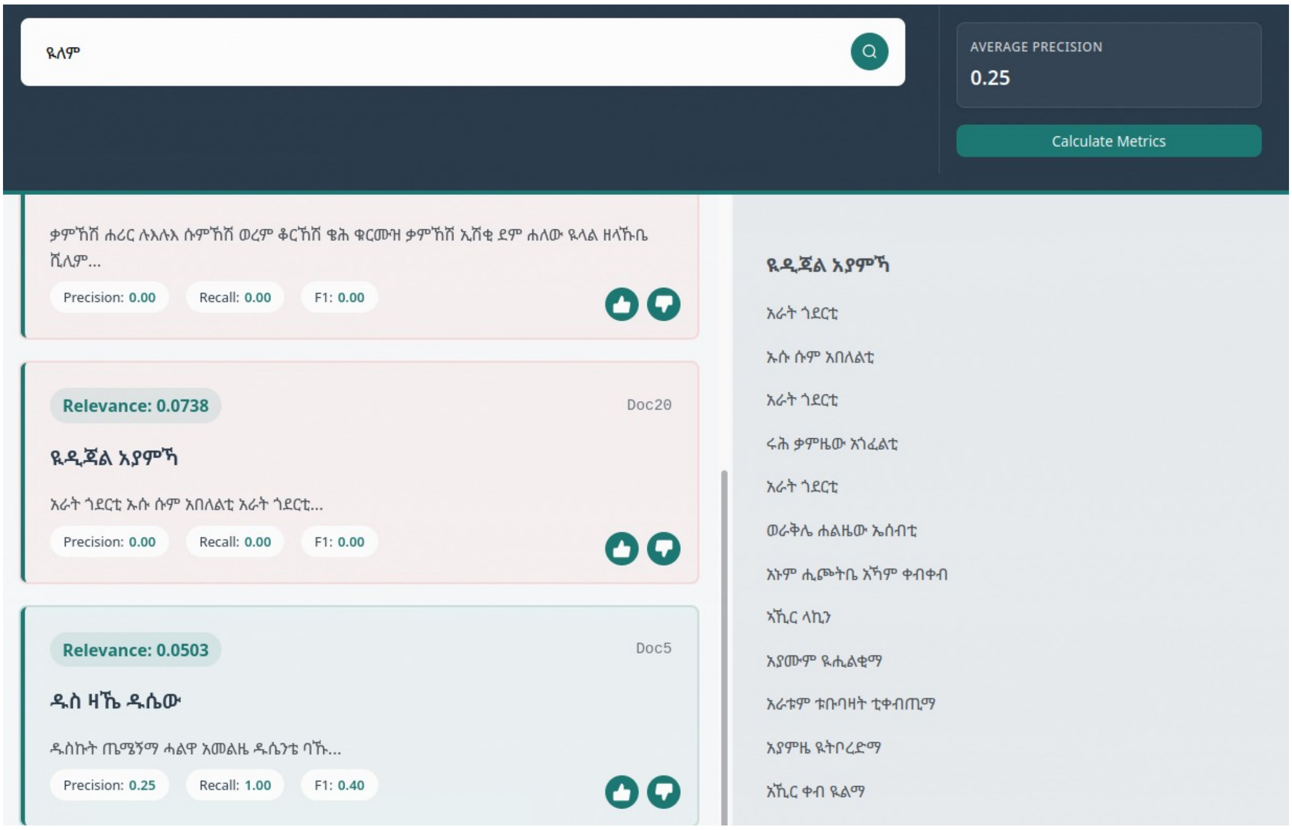


ወቅቲ
ጋምበሪዞቤ
AVERAGE PRECISION
0.84
Calculate Metrics
ያኽ ወቅቲ ሑሉፍ ዛያ ታኢና ኡን ቢላእ ዬሉናር አዋችዚኛ ኡን ቢላእ ዬሉናር አያችዚኛ...
Precision: 0.78
Recall: 0.88
F1: 0.82
Relevance: 0.0588
Doc96
ዲይ በረካ
ኣዬ ቀሐት ሐሌቆሬይ ሚን ሒትፋን ሐለሸ አወጊኝ በይ ቲሪኝዛሽኩት አሐዴጊሩም አለትነፈስኹም አኽኻእ ወቅቲሌ...
Precision: 0.80
Recall: 1.00
F1: 0.89
Relevance: 0.0411
Doc30
አባይ
ዚትናፋአቤው ተናፋአዋ አሐዱም ጋር ዛላ ዲጃዋ ሲጦ አዘንታ ቆሩም ናዚ ጠብ አሸናሌው አባይ በሪዞሌ ኀይታ...
Precision: 0.73
Recall: 1.00
F1: 0.84
አባይ
ዚትናፋአቤው ተናፋአዋ አሐዱም
ጋር ዛላ ዲጃዋ ሲጦ አዘንታ ቆሩም
ናዚ ጠብ አሸናሌው አባይ በሪዞሌ ኀይታ
ዛሌው ዳይቤ መብላኢንታ ባዴ ዚለድታ
ሚስራም የሸ ዚቀረሌም ዩቡርዳሐል አኺሩም አልታ
ወቅቲት ናር ዘገኘ ባዴው ኒሒሲ አለፉ
ሆጂንታ ወቅቲዜ ሱምዜቤ ሐፍ ታሻዛት ሸከፉ
ኔሮት የዲጃል ባነጊር ናጡም ናጥ ባያ
ባዴ ወልዳች ተረአቤዩ ኡስሑልሉ ማእናዊያ
አባይ አባይ ባቅሌ ሰሪ መቆጫ
ጋርኻ ጊበእ ባኔው የቅሪ ሞይ መቄጫ
ቀልቢ ዳለን ኢኚእኽ ባያ ቂንበቻቤ

Q9: መትወለድ



Q10: ዪለም



Summarized table

Queries	Average Precision	Recall	F1 Score
1	100%	100%	100%
2	75%	100%	71%
3	65%	100%	83%
4	73%	100%	81%
5	100%	100%	100%
6	64%	100%	86%
7	0%	0%	0%
8	84%	100%	84%
9	100%	100%	100%
10	25%	100%	40%

Mean Average Precision would be calculated by the sum of average precision divided by the number of queries

MAP = 1.00 + 0.75 + 0.65 + 0.73 + 1.00 + 0.64 + 0 + 0.84 + 1.00 + 0.25/10

MAP = 0.68 or 68%

Analysis

Although the system has a strong performance there are some limitations. Due to the small dataset, the overall results may not be representative of the system's true capabilities on larger datasets. The stemmer although created for Harari still has room for improvement when faced with different variations of words.

Discussion

Challenges Faced

- **Corpus collection:** There was limited corpus available on the internet
- **Stemming**: Developing a custom stemmer for Harari's morphological complexity.
- **Relevance Feedback**: Implementing a dynamic feedback mechanism to refine search results.

Comparison with Existing Systems

We have yet to see a harari search engine implemented on a larger scale other than for our class presentation.

Conclusion

Summary

The Harari Search Engine successfully addresses the challenges of retrieving Harari poems, leveraging transliteration, stemming, and relevance feedback. The system demonstrates high accuracy and usability, making it a valuable tool for Harari literature research.

Future Work

- Support for Boolean operators and advanced query syntax.
- Integration with larger corpora or external datasets.
- Query reformulation

Significance

This project highlights the importance of language-specific adaptations in information retrieval systems, particularly for under-resourced languages like Harari. The system has potential applications in linguistic research, education, and cultural preservation.