UNIVERSITY CHATBOT SYSTEM

Using Custom TF-IDF Implementation Without AI Libraries

Submitted By	Zara Imran
SAP ID	70149236
Section	BSIET-5I
Submitted To	Sir Hamedoon
Course	Artificial Intelligence
Date	January 2026

PROJECT REPORT

---

1. PROJECT DESCRIPTION

1.1 Introduction

The *University Chatbot System* is an intelligent web-based application designed to automatically answer common university-related questions. It helps students, parents, and visitors obtain information about admissions, courses, fees, facilities, and other services without requiring manual staff support.

This project is implemented entirely from scratch without using any AI or machine learning libraries such as scikit-learn, TensorFlow, or spaCy. Instead, the system manually implements the **TF-IDF (Term Frequency–Inverse Document Frequency)** algorithm and **cosine similarity** to match user queries with the most relevant answers.

1.2 Problem Statement

Universities receive hundreds of repetitive inquiries daily such as:

- How to apply for admission?
- What courses are available?
- What are the fees?
- Where is the library?
- When does the semester start?

Handling these queries manually is time-consuming and resource-intensive. Students often experience delays and frustration. An automated chatbot can significantly reduce this workload while providing instant responses.

1.3 Project Objectives

The key objectives of this project are:

1. To develop a functional chatbot without using any AI libraries.
2. To manually implement TF-IDF for text vectorization.
3. To compute cosine similarity using mathematical formulas.
4. To provide accurate responses based on similarity matching.

5. To create a user-friendly web interface.
6. To maintain conversation logs and performance statistics.
7. To allow easy expansion of the knowledge base.

1.4 Key Features

Technical Features:

- Custom-built TF-IDF vectorizer
- Manual cosine similarity computation
- Text preprocessing (tokenization, stop-word removal)
- Flask-based RESTful API
- JSON-based knowledge base
- Real-time responses
- Conversation history tracking

User Features:

- Simple chat interface
- Instant automated replies
- Similarity score for confidence
- Topic browsing
- Conversation history viewing
- Web-based accessibility

1.5 Technologies Used

- **Programming Language:** Python 3.8+
- **Framework:** Flask, Flask-CORS
- **Frontend:** HTML, CSS, JavaScript
- **Data Storage:** JSON
- **Libraries:** Only Flask-related (no AI/ML libraries)

1.6 Importance of the Project

Educational Value:

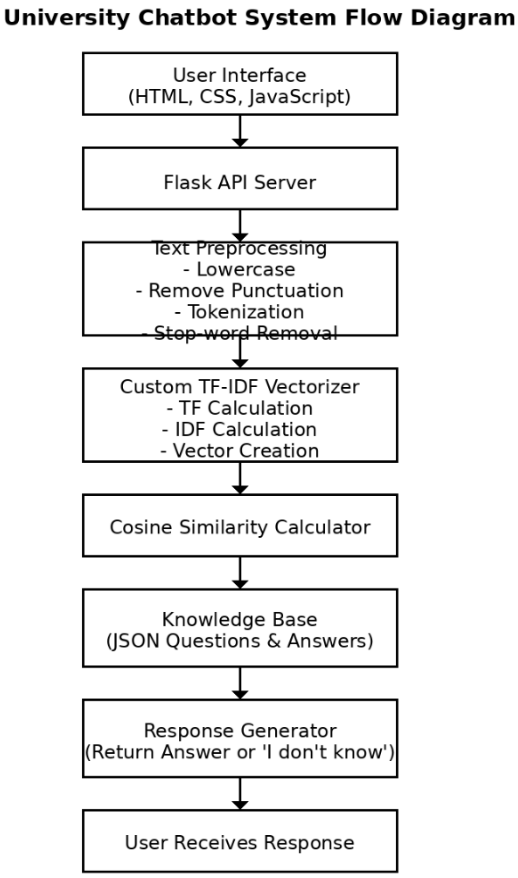
- Demonstrates fundamental NLP concepts
- Explains TF-IDF and vector mathematics
- Builds algorithmic problem-solving skills

Practical Applications:

- Reduces staff workload
- Provides 24/7 student support
- Can be adapted for hospitals, government offices, etc.

---

2. PROJECT FLOW DIAGRAM



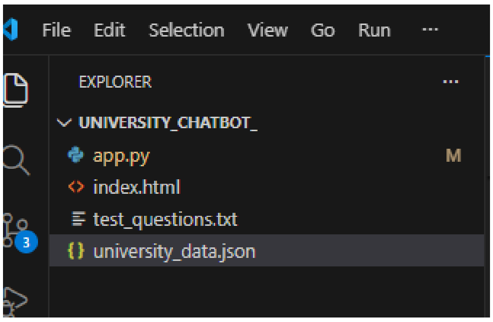
2.2 Step-by-Step Workflow

1. User enters a question in the chat interface.
2. JavaScript sends the request to Flask API.
3. Server receives input via /api/chat.
4. Text preprocessing is applied.
5. Query is converted into TF-IDF vector.
6. Similarity is calculated against all stored questions.
7. Best match is selected based on highest score.
8. If score > threshold, corresponding answer is returned.
9. Conversation is logged with timestamp and similarity score.
10. Response is sent back to frontend and displayed.

---

3. PROJECT CODE (SCREENSHOT REFERENCES)

3.1 File Structure



3.2 Major Components

1. **CustomVectorizer Class**
   - Handles tokenization, TF, IDF, and vector creation.
2. **UniversityChatbot Class**
   - Controls chatbot logic, training, and matching.
3. **Flask API**
   - Manages HTTP requests and responses.

3.3 Core Algorithm Summary

TF (Term Frequency):
TF = (Word count in document) / (Total words)

IDF (Inverse Document Frequency):
IDF = log(Total documents / Documents containing the word)

TF-IDF:
TF-IDF = TF × IDF

Cosine Similarity:
Similarity = (A · B) / (|A| × |B|)

3.4 Code Screenshots

➢ CustomVectorizer Class (TF-IDF Implementation)

```python
class CustomVectorizer:
    """Custom TF-IDF implementation without using sklearn"""

    def __init__(self):
        self.vocabulary = {}  # word -> index mapping
        self.idf_values = {}  # word -> IDF score
        self.documents = []   # store original documents

    def _tokenize(self, text):
        """Convert text to lowercase and split into words"""
        text = text.lower()
        text = re.sub(r'[^\w\s]', '', text)  # remove punctuation
        words = text.split()
        # Remove common stop words manually
        stop_words = {'the', 'a', 'an', 'and', 'or', 'but', 'in', 'on', 'at',
                      'to', 'for', 'of', 'with', 'is', 'are', 'was', 'were',
                      'be', 'been', 'being', 'have', 'has', 'had', 'do', 'does',
                      'did', 'will', 'would', 'could', 'should', 'may', 'might',
                      'can', 'this', 'that', 'these', 'those', 'i', 'you', 'he',
                      'she', 'it', 'we', 'they', 'what', 'which', 'who', 'when',
                      'where', 'why', 'how'}
        return [w for w in words if w not in stop_words and len(w) > 1]

    def _calculate_tf(self, words):
        """Calculate Term Frequency for a document"""
        word_count = len(words)
        word_freq = Counter(words)
        tf_scores = {}
        for word, count in word_freq.items():
            tf_scores[word] = count / word_count
        return tf_scores

    def _calculate_idf(self):
        """Calculate Inverse Document Frequency for all words"""
        num_docs = len(self.documents)
        word_doc_count = {}
```

```python
        # Count how many documents contain each word
        for doc in self.documents:
            unique_words = set(doc)
            for word in unique_words:
                word_doc_count[word] = word_doc_count.get(word, 0) + 1

        # Calculate IDF: log(total_docs / docs_containing_word)
        for word, doc_count in word_doc_count.items():
            self.idf_values[word] = math.log(num_docs / doc_count)

    def fit_transform(self, texts):
        """Train on documents and return TF-IDF vectors"""
        # Tokenize all documents
        self.documents = [self._tokenize(text) for text in texts]

        # Build vocabulary
        all_words = set()
        for doc in self.documents:
            all_words.update(doc)

        self.vocabulary = {word: idx for idx, word in enumerate(sorted(all_words))}

        # Calculate IDF values
        self._calculate_idf()

        # Create TF-IDF vectors for all documents
        vectors = []
        for doc in self.documents:
            vectors.append(self._create_vector(doc))

        return vectors

    def transform(self, texts):
        """Transform new texts to TF-IDF vectors"""
        vectors = []
        for text in texts:
            words = self._tokenize(text)
```

```python
            vectors.append(self._create_vector(words))
        return vectors

    def _create_vector(self, words):
        """Create TF-IDF vector for a document"""
        # Initialize vector with zeros
        vector = [0.0] * len(self.vocabulary)

        # Calculate TF for this document
        tf_scores = self._calculate_tf(words)

        # Calculate TF-IDF for each word
        for word, tf in tf_scores.items():
            if word in self.vocabulary:
                idx = self.vocabulary[word]
                idf = self.idf_values.get(word, 0)
                vector[idx] = tf * idf

        return vector
```

➢ Manual Cosine Similarity Function

```python
def cosine_similarity_manual(vec1, vec2):
    """Calculate cosine similarity between two vectors manually"""
    # Dot product
    dot_product = sum(a * b for a, b in zip(vec1, vec2))

    # Magnitude of vec1
    magnitude1 = math.sqrt(sum(a * a for a in vec1))

    # Magnitude of vec2
    magnitude2 = math.sqrt(sum(b * b for b in vec2))

    # Avoid division by zero
    if magnitude1 == 0 or magnitude2 == 0:
        return 0.0

    # Cosine similarity
    return dot_product / (magnitude1 * magnitude2)
```

➢ University Chatbot Matching Logic (Best Question Selection)

```python
    def find_best_match(self, user_question):
        """Find the best matching question manually"""
        if not self.is_trained:
            return {
                "answer": "Chatbot is not trained yet.",
                "similarity_score": 0.0,
                "matched_question": None
            }

        # Convert user question to vector using our custom vectorizer
        user_vector = self.vectorizer.transform([user_question])[0]

        # Manually calculate similarity with each stored question
        similarities = []
        for stored_vector in self.question_vectors:
            sim = cosine_similarity_manual(user_vector, stored_vector)
            similarities.append(sim)

        # Find the best match manually
        best_match_idx = 0
        best_score = similarities[0]
        for i, score in enumerate(similarities):
            if score > best_score:
                best_score = score
                best_match_idx = i
```

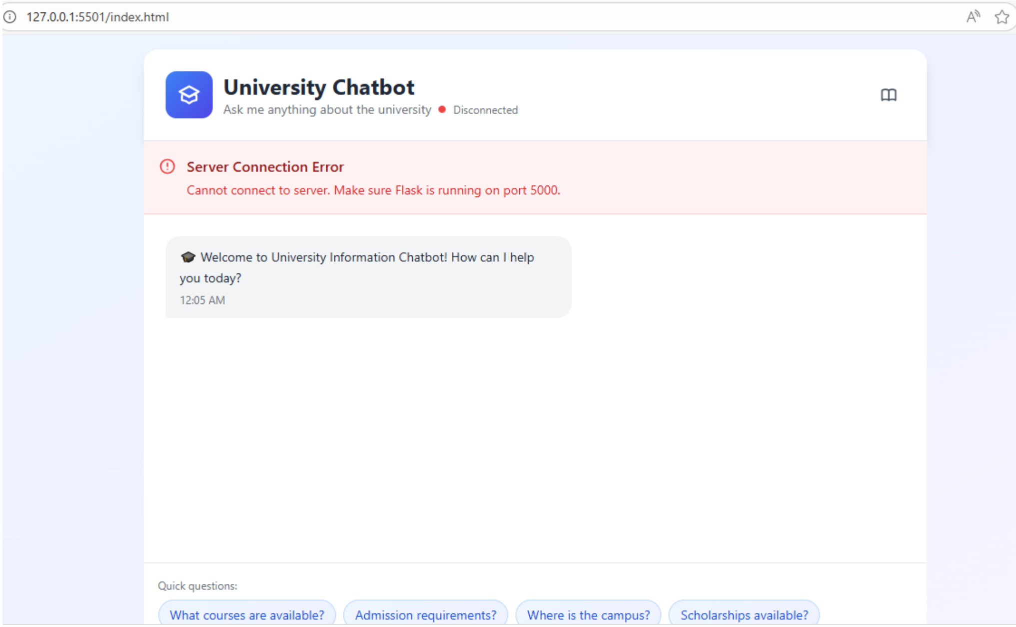
➢ Flask API Routes

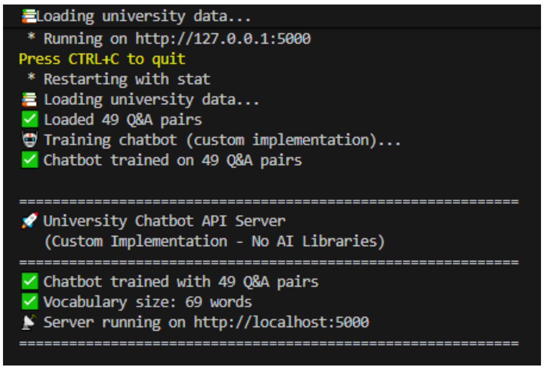
```python
# API Routes
@app.route('/api/chat', methods=['POST'])
def chat():
    """Handle chat messages"""
    try:
        data = request.get_json()
        user_question = data.get('question', '')

        if not user_question:
            return jsonify({"error": "No question provided"}), 400

        response = chatbot.find_best_match(user_question)
        return jsonify(response), 200

    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/api/stats', methods=['GET'])
def get_stats():
    """Get chatbot statistics"""
    try:
        stats = chatbot.get_stats()
        return jsonify(stats), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/api/topics', methods=['GET'])
def get_topics():
    """Get available topics"""
    try:
        topics = chatbot.get_topics()
        return jsonify(topics), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

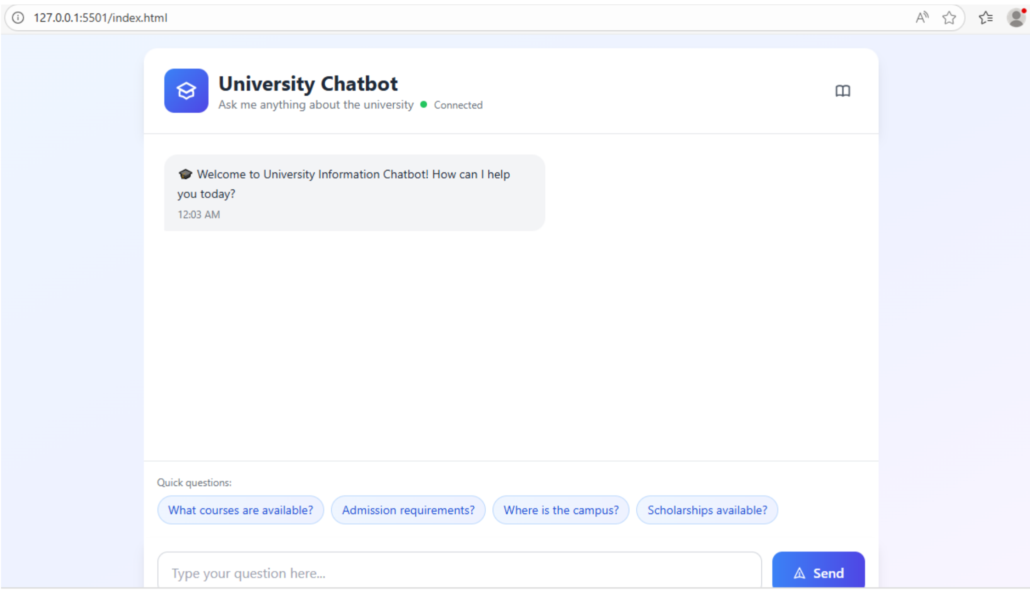
```python

@app.route('/api/history', methods=['GET'])
def get_history():
    """Get conversation history"""
    try:
        return jsonify(chatbot.conversation_history), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/api/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        "status": "healthy",
        "is_trained": chatbot.is_trained,
        "knowledge_base_size": len(chatbot.knowledge_base),
        "implementation": "Custom (No AI Libraries)"
    }), 200


@app.route('/api/add_data', methods=['POST'])
def add_data():
    """Add new data to knowledge base"""
    try:
        data = request.get_json()
        chatbot.add_data(data)
        chatbot.train()
        return jsonify({"message": "Data added successfully"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500

```

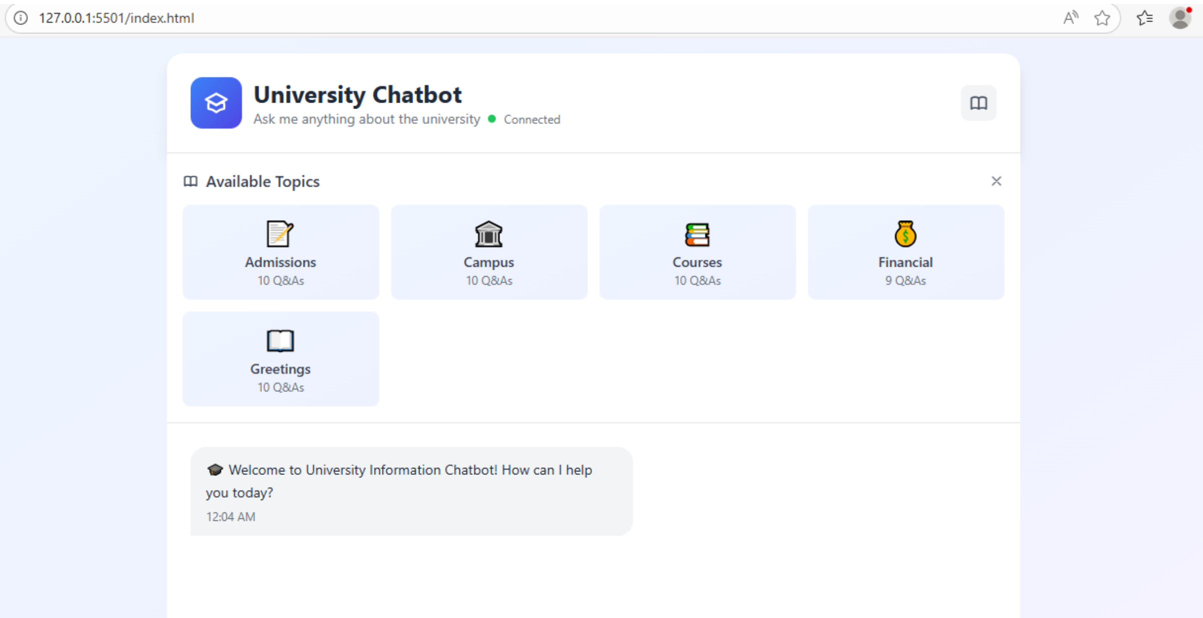
4. OUTPUT (SCREENSHOTS)

4.1 Chat Interface









---

5. SCOPE OF WORK

5.1 Backend Development

- Implemented custom TF-IDF without AI libraries
- Built preprocessing system
- Developed cosine similarity algorithm
- Created JSON-based knowledge base
- Designed Flask API with 6 endpoints
- Implemented conversation logging and statistics

5.2 Frontend Development

- Chat interface design
- Statistics dashboard
- Topics browser
- History viewer
- AJAX-based API communication

5.3 Algorithms Implemented

- Term Frequency (TF)
- Inverse Document Frequency (IDF)
- TF-IDF weighting
- Vector Space Model
- Cosine Similarity

5.4 Skills Demonstrated

- Python programming
- Object-Oriented Programming
- Flask framework
- RESTful API design
- Web development
- NLP fundamentals
- Algorithm implementation
- Data handling with JSON

5.5 Challenges and Solutions

- **Understanding TF-IDF:** Solved by step-by-step testing.
- **Slow Vector Comparisons:** Optimized using efficient loops.
- **Different Question Phrasing:** Improved preprocessing and stop-word list.
- **Low Similarity Scores:** Adjusted threshold and debugging vectors.

5.6 Testing and Validation

- Unit testing of algorithms
- Integration testing of APIs
- User testing and feedback

- Performance testing for scalability

5.7 Future Enhancements

- Spell checking
- Voice interaction
- Multi-language support
- Mobile app version
- Database integration
- Authentication system
- Admin dashboard
- Cloud deployment

---

6. GITHUB REPOSITORY LINK

Repository URL:
https://github.com/zaraimran03/AI_PROJECT.git

Installation Instructions:

1. Clone repository
   git clone https://github.com/[username]/university-chatbot.git
2. Install dependencies
   pip install flask flask-cors
3. Run application
   python app.py
4. Open browser
   http://localhost:5000

CONCLUSION

This project successfully demonstrates the implementation of a chatbot system using fundamental NLP techniques without relying on AI libraries. By manually implementing TF-IDF and cosine similarity, the system provides accurate, fast, and scalable responses to user queries.

Key Achievements:

- Built custom TF-IDF algorithm
- Implemented manual cosine similarity
- Developed REST API using Flask
- Designed user-friendly interface
- Created comprehensive documentation

This project reflects strong understanding of AI fundamentals, software engineering principles, and full-stack development.
