# PROJECT REPORT

## ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEM

## CT-361


**GROUP MEMBERS:**

**CT-22056 YUSRA HAQUE**

**CT-22062 FARIA SHAHID**

**CT-22066 HAMDIA NOUMAN**

**CT-22069 NADEEYAH JAMIL**

# AI URDU TUTOR

**TABLE OF CONTENTS**

# 1.Introduction:

### 1.1 PROBLEM STATEMENT:

In Pakistan and other Urdu-speaking regions, many children face challenges in accessing engaging and interactive learning resources in their native language. Traditional educational methods often lack personalization, interactivity, and accessibility—especially for young learners. Additionally, there is a scarcity of AI-powered educational tools specifically designed for kids in Urdu, which can make language learning and comprehension more difficult for early learners. This gap hinders not only linguistic development but also the adoption of modern learning methods in local contexts.

### 1.2 PROJECT OVERVIEW:

The "AI Urdu Tutor for Kids" is an intelligent, interactive educational tool designed to help children learn and understand the Urdu language through an engaging conversational interface. The system leverages artificial intelligence to provide real-time answers to children's questions using a curated knowledge base that includes an Urdu dictionary and a collection of age-appropriate stories. Children can ask the AI questions in Urdu, and it will respond with accurate and easy-to-understand answers, making language learning more interactive and enjoyable. The project aims to promote Urdu literacy in a fun and accessible way, especially for early-age learners, by combining modern AI technology with culturally relevant content.

## 2.Technologies & Libraries Used:

| Tool / Library | Purpose |
|---|---|
| **streamlit** | To build the user interface |
| **speech_recognition** | Convert Urdu audio to text |
| **gtts** | Convert Urdu text to speech |
| **fitz (PyMuPDF)** | Read and extract text from PDF |
| **SentenceTransformer** | Convert sentences to embeddings (numbers) |
| **faiss** | Fast vector-based search |
| **langchain** | To handle prompts, memory, and chains |
| **ChatGroq + LLaMA 3** | The actual LLM generating the response |
| **audio_recorder_streamlit** | For recording audio in Streamlit |
| **tempfile, os, BytesIO** | File and memory handling |

## 3. Features :

| | |
|---|---|
| **RAG (Retrieval-Augmented Generation)** | Combines knowledge from a PDF with AI's intelligence to answer accurately. |
| **Urdu PDF Support** | Upload and read Urdu PDF files. |
| **Voice Input** | User can speak in Urdu to ask questions. |
| **Voice Output (TTS)** | Assistant speaks back the answers. |
| **Memory** | Remembers previous questions and answers (context-aware). |
| **Caching** | If the same question is asked again, it gives a faster answer using stored results. |
| **Control Buttons** | Options to clear memory, toggle memory visibility, etc. |

## 4..How It Works:

1. User uploads an Urdu PDF.
2. Bot processes the PDF and creates a searchable vector store.
3. User asks a question (voice or text).
4. Bot converts voice to Urdu text incase of voice input.
5. Bot retrieves related chunks from the PDF using FAISS.
6. Bot generates an answer using LLaMA 3 (Groq) with context + memory.
7. Bot converts the Urdu answer to audio and shows + speaks it.

## 5.Code:

```
import streamlit as st
import speech_recognition as sr
from gtts import gTTS
from io import BytesIO
import fitz  # PyMuPDF
import os
import numpy as np
import faiss
from sentence_transformers import SentenceTransformer
from langchain_groq import ChatGroq
from langchain.text_splitter import RecursiveCharacterTextSplitter
import torch
from audio_recorder_streamlit import audio_recorder
import tempfile
import time
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
```

### 1. User Interface (UI)

- **streamlit**:
  Builds the interactive web app interface with chat input, buttons, and audio playback.

- **audio_recorder_streamlit**:
  Adds a user-friendly audio recording button directly to the Streamlit app.

### 2. Voice Interaction

- **speech_recognition**:
  Converts spoken Urdu input into text using Google's Speech API.

- **gTTS (Google Text-to-Speech)**:
  Converts Urdu text responses to audio for playback.

- **BytesIO**:
  Handles audio input/output entirely in memory (no file writing needed).

- **tempfile**:
  Manages temporary files, especially for handling audio during text-to-speech processing.

**3. Document Processing**

- **fitz (PyMuPDF)**:
  Extracts text from uploaded PDF files, including support for right-to-left Urdu.

- **RecursiveCharacterTextSplitter**:
  Splits large documents into smaller, manageable chunks for retrieval.

**4. Natural Language Processing & Retrieval**

- **SentenceTransformer**:
  Converts each text chunk into vector embeddings for similarity search.

- **numpy**:
  Formats embeddings into compatible structures for FAISS.

- **faiss**:
  Performs efficient vector-based retrieval to find the most relevant document chunks for a user's question.

**5. Language Model Integration (LLM)**

- **ChatGroq via LangChain**:
  Connects to Groq's LLaMA3 model to generate answers based on context and conversation history.

- **PromptTemplate**:
  Defines the structure of inputs sent to the language model (history + context + question).

- **LLMChain**:
  Links the prompt, memory, and model together to execute the query pipeline.

- **ConversationChain**:
  Manages the overall conversational flow with memory and model response handling.

### 6. Memory & Conversation Handling

- **ConversationBufferMemory**:
  Stores the full chat history, enabling context-aware and follow-up questions.

### 7. Utility Libraries

- **torch**:
  Supports PyTorch-based models, useful for GPU checks or model compatibility.

- **time**:
  Handles delays, tracks performance, or controls animation effects (like loading spinners).

```python
@st.cache_resource
def load_models():
    device = "cuda" if torch.cuda.is_available() else "cpu"
    EMBED_MODEL = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-v2", device=device)
    GROQ_API_KEY = "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"
    os.environ['GROQ_API_KEY'] = GROQ_API_KEY
    groq_model = ChatGroq(model_name="llama3-70b-8192", temperature=0.3, max_tokens=512)
    urdu_splitter = RecursiveCharacterTextSplitter(
        separators=["۔", "\n\n", "\n", "،", " "],
        chunk_size=400,
        chunk_overlap=100,
        length_function=len
    )
    return EMBED_MODEL, groq_model, urdu_splitter
```

st.cache _resource  helps to remember (or cache) expensive operations  like loading big machine learning models  so they don't run again and again every time the user interacts with the app.Without caching, the app would reload everything every time a button is clicked.With @st.cache_resource, the app only loads it once, and reuses it

The first time load_model() runs, it loads the model and stores it in memory.After that, any time you call load_model(), it uses the already-loaded model.

This function load_models() is designed to prepare and load everything your chatbot needs to work efficiently. First, it checks if a GPU is available (using CUDA) and chooses the best device (GPU or CPU) for faster performance. Then it loads a multilingual sentence embedding model called paraphrase-multilingual-MiniLM-L12-v2, which helps the app convert user input or PDF content into vector format for smart searching. After that, it sets up an API key for using Groq's powerful language model (LLaMA 3) to generate intelligent replies. It creates a ChatGroq object with specific settings like how creative the response should be (temperature) and how long the replies can be (max_tokens). Finally, it defines how to split Urdu text into smaller parts using the RecursiveCharacterTextSplitter, so long texts (like PDFs) can be broken down and processed effectively. In the end, it returns all three tools — the embedding model, the chatbot model, and the text splitter — ready to use in your app.

```python
def create_faiss_vectorstore(text, EMBED_MODEL):
    chunks = urdu_splitter.split_text(text[:100000])
    embeddings = EMBED_MODEL.encode(chunks, show_progress_bar=True)
    embeddings = np.array(embeddings).astype('float32')
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)
    index.add(embeddings)
    return {"chunks": chunks, "index": index}
```

The `create_faiss_vectorstore` function is designed to process a large Urdu text and prepare it for efficient semantic search using FAISS. It first splits the input text into smaller chunks using an Urdu-specific text splitter. These chunks are then converted into vector embeddings using a provided embedding model (`EMBED_MODEL`). Since FAISS requires embeddings to be in `float32` format, the function ensures this conversion. It then initializes a FAISS index using L2 (Euclidean) distance and adds the generated embeddings to this index. Finally, the function returns a dictionary containing both the list of text chunks and the FAISS index, which can later be used for searching relevant information from the original text.

```python
def init_memory():
    return ConversationBufferMemory(
        memory_key="history",
        input_key="input",
        return_messages=True
    )
```

This function sets up conversation memory using ConversationBufferMemory, which helps the chatbot remember previous user inputs and responses. This makes the conversation more natural and coherent  like how a human would remember what was said earlier.

```python
# Session State Setup
if "messages" not in st.session_state:
    st.session_state.messages = []
if "processing" not in st.session_state:
    st.session_state.processing = False
if "vectorstore" not in st.session_state:
    st.session_state.vectorstore = None
if "last_processed" not in st.session_state:
    st.session_state.last_processed = None
if "audio_cooldown" not in st.session_state:
    st.session_state.audio_cooldown = 0
if "qa_cache" not in st.session_state:
    st.session_state.qa_cache = {}
if "conversation_memory" not in st.session_state:
    st.session_state.conversation_memory = init_memory()
if "show_memory" not in st.session_state:
    st.session_state.show_memory = False
```

Streamlit resets everything on each user interaction, which means that without storing information in st.session_state, all previous data would be lost after each action. To prevent this and maintain the chatbot's functionality, the code initializes several session variables. The messages variable keeps track of the conversation history, while processing acts as a flag to indicate if the chatbot is currently working on a response, which can be helpful for showing loading animations. The vectorstore is used to store vector embeddings, such as those generated from PDF text, enabling quick and efficient semantic searches. The last_processed variable stores the name or timestamp of the most recently processed file to avoid redundant operations. audio_cooldown ensures that the audio recording feature is not triggered too frequently, improving usability. The qa_cache holds previously asked questions and their answers to avoid repeating computations and to improve performance. The conversation_memory stores past inputs and responses using the init_memory() function so that the chatbot can maintain context across the conversation. Lastly, show_memory is a toggle that allows developers to show or hide memory-related information in the user interface. Additionally, the line EMBED_MODEL, groq_model, urdu_splitter = load_models() loads the core components of the app: EMBED_MODEL converts text into numerical vectors, groq_model generates responses using Groq's powerful language model, and urdu_splitter breaks long Urdu text into manageable chunks for better processing and retrieval.

```python
with st.sidebar:
    st.title("Settings")
    uploaded_file = st.file_uploader("Upload Urdu PDF", type=["pdf"])
    if uploaded_file is not None and st.session_state.vectorstore is None:
        with st.spinner("Processing PDF..."):
            with BytesIO(uploaded_file.getvalue()) as pdf_data:
                doc = fitz.open(stream=pdf_data, filetype="pdf")
                raw_text = "".join([page.get_text() for page in doc])
                doc.close()
                st.session_state.vectorstore = create_faiss_vectorstore(raw_text, EMBED_MODEL)
                st.success("PDF loaded successfully!")

    if st.button("Clear Conversation Memory"):
        st.session_state.conversation_memory.clear()
        st.success("Conversation memory cleared!")
    st.sidebar.checkbox("Show Memory", key="show_memory")
```

This sidebar section titled "Settings" allows users to upload Urdu PDF files, which are processed only if they haven't been handled before. When a PDF is uploaded, the app reads and extracts its text, converts it into vector embeddings using the embedding model, and stores these vectors for fast searching. A loading spinner informs the user during processing, and a success message confirms when the PDF is ready. Additionally, there is a button to clear the chatbot’s conversation memory, allowing the chat history to be reset, with a confirmation message shown after clearing. Finally, a checkbox lets users toggle whether the conversation memory details are displayed within the app interface.

```python
def urdu_rag_query(question: str) -> str:  2 usages
    # Return cached answer if available
    if question in st.session_state.qa_cache:
        return st.session_state.qa_cache[question]
    # Ensure vectorstore is loaded
    vectorstore = st.session_state.vectorstore
    # Retrieve context
    q_emb = EMBED_MODEL.encode([question])
    q_emb = np.array(q_emb).astype('float32')
    distances, indices = vectorstore["index"].search(q_emb, 15)
    context = "\n\n".join([vectorstore["chunks"][i] for i in indices[0]])

    # Initialize ConversationChain if needed
    if "conversation_chain" not in st.session_state:
        prompt = PromptTemplate(
            input_variables=["history", "input", "context"],
            template="""
            آپ کا کردار: ایک ماہر اردو استاد

            ہدایات:
```

```
            1. پچھلی گفتگو اور دیا گیا سیاق و سباق استعمال کریں۔
            2. اگر سوال مختصر اور معلوماتی ہو، تو مختصر جواب دیں۔
            3. اگر سوال میں وضاحت یا خلاصہ مانگا گیا ہو، تو تفصیلی جواب دیں۔
            4. جہاں ضروری ہو وہاں مشکل اردو الفاظ کی وضاحت کریں۔
            5. جواب کے شروع میں 'جواب:' نہ لکھیں۔

            پچھلی گفتگو:
            {history}

            سیاق:
            {context}

            سوال:
            {input}

            جواب:
            """
        )
        st.session_state.conversation_chain = LLMChain(
```

```
        st.session_state.conversation_chain = LLMChain(
            llm=groq_model,
            prompt=prompt,
            memory=st.session_state.conversation_memory,
            verbose=False
        )
    # Build history string from memory
    history = "\n".join([
        f"انسان: {m.content}" if m.type == "human" else f"AI: {m.content}"
        for m in st.session_state.conversation_memory.chat_memory.messages
    ])
    resp = st.session_state.conversation_chain.predict(
        input=question,
        context=context
    )
    st.session_state.qa_cache[question] = resp
    return resp
```

This function urdu_rag_query is designed to answer user questions in Urdu by combining semantic search with a powerful language model, creating a smooth and context-aware chatbot experience.

1. **Caching for Efficiency:** It first checks if the question was already asked and answered before, using a cache stored in st.session_state.qa_cache. If yes, it returns the cached answer instantly, saving time and resources.
2. **Vector Search for Context Retrieval:** If the question is new, it converts the question into a vector embedding using the embedding model. It then searches a pre-built vector index (stored in vectorstore) to find the most relevant text chunks from the uploaded Urdu PDF or other sources. These chunks provide the context needed to answer the question accurately.
3. **Dynamic Prompt Setup:** If the conversation chain (which ties the language model, memory, and prompts together) hasn't been initialized yet, it sets up a custom prompt template in Urdu. This prompt instructs the language model to behave like an expert

Urdu teacher, use the conversation history and retrieved context, and give clear, detailed, or concise answers depending on the question.

4. **Memory for Conversation Flow:** The function builds a history of the conversation from previous messages, allowing the chatbot to maintain context and respond naturally, not just in isolation.
5. **Generating Response:** The language model (Groq-powered) is called with the question, the retrieved context, and the conversation history to generate a coherent and relevant reply in Urdu.
6. **Caching the Answer:** The new response is saved in the cache to speed up future identical queries.

```
# Text-to-speech helper
def text_to_speech(text: str, lang: str = "ur") -> BytesIO:  2 usages
    tts = gTTS(text=text, lang=lang, slow=False)
    b = BytesIO(); tts.write_to_fp(b); b.seek(0)
    return b
```

The text_to_speech function converts written text into spoken audio using the Google Text-to-Speech (gTTS) library, with Urdu as the default language. It takes the input text and generates speech at a normal speed, then saves the audio not as a file on disk but directly into an in-memory buffer called BytesIO. This approach allows the audio to be stored temporarily in the computer's memory, making it quick and easy to access without needing to read or write files on disk. After writing the audio data to this buffer, the function resets the reading position to the start so the audio can be played or processed immediately. This function is useful in applications like chatbots or web apps where you want to provide instant spoken responses without any delay caused by file handling.

```python
st.title("📚 Urdu Tutor")
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(
            f'<div style="text-align: right; direction: rtl;">{msg["content"]}</div>',
            unsafe_allow_html=True
        )
        if msg.get("audio"): st.audio(msg["audio"], format="audio/mp3")
if st.session_state.show_memory:
    st.subheader("Conversation Memory")
    st.code(st.session_state.conversation_memory.buffer_as_str, language="text")
t1, t2 = st.columns([5,1])
with t1:
    user_input = st.chat_input("Type your question...", key="input")
with t2:
    audio_bytes = None
    if time.time() - st.session_state.audio_cooldown > 3:
        audio_bytes = audio_recorder(text="", pause_threshold=2.5, key="recorder")
    else:
        st.warning("Please wait...")
```

This Streamlit code builds an Urdu chat tutor interface showing past messages aligned for Urdu reading. It supports text input and audio recording, but limits audio recording frequency to avoid overload. Users can also toggle viewing the full conversation memory.

```python
if audio_bytes and len(audio_bytes)>0 and not st.session_state.processing and current - st.session_state.audio_cooldown>3
    st.session_state.processing=True; st.session_state.audio_cooldown=current
    with st.spinner("Transcribing..."):
        try:
            with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as tmp:
                tmp.write(audio_bytes); path=tmp.name
            r=sr.Recognizer();
            with sr.AudioFile(path) as source:
                audio=r.record(source); text=r.recognize_google(audio, language="ur-PK")
            os.remove(path)
            if text and text!=st.session_state.last_processed:
                st.session_state.messages.append({"role":"user","content":text})
                st.session_state.last_processed=text
                if st.session_state.vectorstore:
                    ans=urdu_rag_query(text)
                    audio=text_to_speech(ans)
                    st.session_state.messages.append({"role":"assistant","content":ans,"audio":audio})
                    st.rerun()
                else: st.warning("Upload a PDF first.")
```

```python
        except Exception as e:
            st.error(f"Error: {e}")
        finally: st.session_state.processing=False
```

This code handles voice input by converting recorded Urdu audio into text, then uses that text to get an answer from the chatbot if a PDF is loaded. It adds both the text and audio reply to the chat and prevents multiple rapid inputs using timers and flags, showing errors if anything goes wrong.

```python
# Process text input
if user_input and not st.session_state.processing and user_input!=st.session_state.last_processed:
    st.session_state.processing=True
    st.session_state.messages.append({"role":"user","content":user_input})
    st.session_state.last_processed=user_input
    if st.session_state.vectorstore:
        with st.spinner("Generating answer..."):
            ans=urdu_rag_query(user_input)
            audio=text_to_speech(ans)
            st.session_state.messages.append({"role":"assistant","content":ans,"audio":audio})
    else:
        st.warning("Upload a PDF first.")
    st.session_state.processing=False
    st.rerun()
```

This code processes the user's typed question. If the input is new and the system isn't already processing, it stores the question, checks if a PDF is uploaded, and then generates an answer using the chatbot. The answer is converted to audio, both are saved in the chat, and the app refreshes to display the response.

## 6. Conclusion:

This Urdu RAG Chatbot integrates multiple advanced components to provide an interactive, context-aware tutoring experience in Urdu. It efficiently processes uploaded Urdu PDFs by splitting the text and creating a FAISS vector store for fast semantic retrieval. The chatbot uses an embedding model combined with a powerful LLM (Groq) to generate contextually relevant answers while maintaining conversation memory for a natural, coherent dialogue flow.The system supports both text and voice input, utilizing speech recognition for Urdu speech-to-text and text-to-speech for audio responses, enhancing accessibility and user engagement. Caching mechanisms optimize repeated queries for faster response times. The Streamlit UI offers seamless interaction, including conversation memory control and PDF uploads, making the chatbot a robust tool for Urdu language learning and information retrieval.