Edu Tutor AI: Personalized Learning

Project Documentation

1. Introduction :

Project title : Citizen Ai: Intelligent Citizen Enagagement Platform.

- Team member : DHANUSH KUMAR R
- Team member : BHARATH S
- Team member : ISAACRICHARD S
- Team member : ARUN K

2. Project Overview:

- **Purpose:** EduTutor AI uses the Granite model from Hugging Face to create simple, personalized learning tools like concept explainers, quizzes generator and add more functionalities that you like. This project is deployed in Google Colab using Granite for low setup effort and reliable performance.

- **Features:**
- **Adaptive Learning Paths**
  - Automatically adjust curriculum pacing and content difficulty based on learner's performance, background, pace of learning.
  - Branching lessons: when a student struggles on a topic, give remedial content; when they excel, offer challenging extensions.
- **Personalized Feedback & Hints**
  - Instant feedback on quizzes / exercises, highlighting mistakes and why.
  - Provide hints instead of full solutions to encourage reasoning. member
  - Use natural language explanations suitable to student's level.
- **Natural Language Dialogue / Conversational Tutor**
  - Chat interface where students can ask questions in everyday language.
  - Support for follow-ups, clarifications, multiple rounds of questioning.
  - Multilingual support or at least localization.
- **24/7 Availability & Scalability**
  - Always-on support: students can use at any time.
  - Handle routine / administrative queries automatically so human tutors can focus on higher-value tasks.
- **Learning Analytics & Insights**
  - Track metrics: time spent, mistakes, progress through topics, retention.
  - Predictive analytics to detect weak areas or risk of falling behind.
  - Dashboards for students and tutors / instructors.

- **Content Recommendations & Supplementary Resources**
  - Suggest extra exercises, videos, readings tailored to gaps.
  - Use generative content to create practice questions, flashcards, summaries.
- **Interactive & Multimodal Learning Content**
  - Use not just text: images, audio, video, simulations.
  - Possibly use voice or speech recognition for spoken practice / pronunciation.
- **Customization / Personalization**
  - Let students set preferences: learning style (visual / auditory), pace, areas of interest.
  - Ability for instructors / content creators to configure content domain, style, difficulty.
- **Assessment & Mastery Tracking**
  - Regular formative assessments embedded in the flow (small quizzes, check-ins).
  - Mastery thresholds for moving forward.
  - Summative assessments with feedback.
- **Virtual Assistant for Administrative & Logistical Support**
  - Answer student questions about deadlines, exams, schedule, account issues, etc.
  - Reduce load on support staff.
- **Security, Privacy, Data Control**
  - Secure handling of student data; compliance with relevant regulations (e.g., GDPR, local laws).
  - Transparent sources for content (no hallucinations or wrong info).
  - Ability to control/curate the knowledge base.
- **Human-in-the-Loop & Escalation**
  - When AI is unsure or when student needs deeper help, escalate to a human tutor.
  - Allow tutor review of AI responses.
- **Motivation, Gamification, Engagement**
  - Badges, rewards, leaderboards, progress bars.
  - Interactive quizzes, games, challenges.
- **Offline / Low-Connectivity Support**
  - Design for students who have intermittent internet. Cached content; ability to sync when online.
- **Scalability & Maintainability**
  - Modular architecture so content can be updated per subject, per level.
  - Efficient model updating retraining, content versioning.

3. Architecture:

- **Frontend (Streamlit):** The frontend is an interactive web UI with multiple pages for dashboards, file uploads, a chat interface, feedback forms, and report viewers. It uses the Streamlit-option-menu library for sidebar navigation, and each page is modularized for scalability.
- **Backend (FastAPI):** This serves as the REST framework for API endpoints that handle document processing, chat, eco-tip generation, and

more. It is optimized for asynchronous performance and easy Swagger integration.

- **LLM Integration (IBM Watsonx Granite):** The project uses Granite LLM models from IBM Watsonx for natural language understanding and generation. Prompts are specifically designed to produce summaries, reports, and sustainability tips.
- **Vector Search (Pinecone):** Uploaded policy documents are converted into embeddings using Sentence Transformers and stored in Pinecone. Semantic search is enabled via cosine similarity, letting users search documents using natural language queries.
- **ML Modules (Forecasting and Anomaly Detection):** Lightweight ML models from Scikit-learn are used for forecasting and anomaly detection. Time-series data is parsed, modeled, and visualized using pandas and matplotlib.

4. Setup Instructions:

- **Prerequisites:**
  - Python 3.9 or later
  - pip and virtual environment tools
  - API keys for IBM Watsonx and Pinecone
  - Internet access for cloud services
- **Installation Process:**
  - Clone the repository.
  - Install dependencies from

    requirements.txt.

  - Create and configure a

    .env file with credentials.

  - Run the backend server using FastAPI.
  - Launch the frontend via Streamlit.
  - Upload data and interact with the modules.

5. Folder Structure:

- app/ - Contains all FastAPI backend logic, including routers, models, and integration modules.
- app/api/ - Subdirectory for modular API routes like chat, feedback, and document vectorization.
- ui/ - Contains frontend components for Streamlit pages and form UIs.

- smart_dashboard.py - The entry script for the main Streamlit dashboard.
- granite_llm.py - Handles all communication with the IBM Watsonx Granite model.
- document_embedder.py - Converts documents to embeddings and stores them in Pinecone.
- kpi_file_forecaster.py - Forecasts future trends for energy/water using regression.
- anomaly_file_checker.py - Flags unusual values in uploaded KPI data.
- report_generator.py - Constructs AI-generated sustainability reports.

6. Running the Application:

- To start the project, launch the FastAPI server and then run the Streamlit dashboard.
- Navigate through the pages using the sidebar.
- Users can upload documents or CSVs, interact with the chat assistant, and view outputs like reports, summaries, and predictions.
- All interactions are real-time, with the frontend dynamically updating via backend APIs.

7. API Documentation:

- The backend APIs include:
  - POST /chat/ask - Accepts a user query and returns an AI-generated message.
  - POST /upload-doc - Uploads and embeds documents in Pinecone.
  - GET /search-docs - Returns semantically similar policies to a user query.
  - GET /get-eco-tips - Provides sustainability tips on selected topics.
  - POST /submit-feedback - Stores citizen feedback.
- Each endpoint is documented and tested in Swagger UI.

8. Authentication:

- For demonstration purposes, this version of the project runs in an open environment.
- Secure deployments can include:
  - Token-based authentication (JWT or API keys).
  - OAuth2 with IBM Cloud credentials.
  - Role-based access for different user types (admin, citizen, researcher).
- Future enhancements will include user sessions and history tracking.

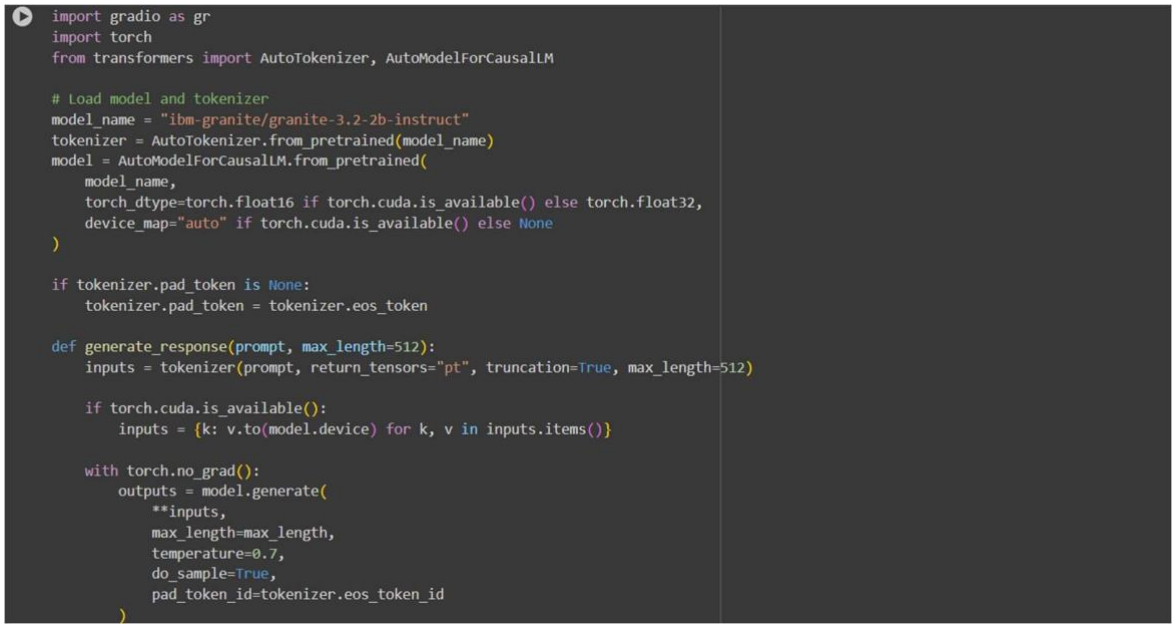
9. User Interface:

- The interface is minimalist and designed for accessibility for non-technical users.
- Key elements include:
  - A sidebar for navigation.
  - KPI visualizations with summary cards.
  - Tabbed layouts for chat, eco tips, and forecasting.
  - Real-time form handling.
  - PDF report download capability.

10. Testing:

- Testing was conducted in several phases:
  - **Unit Testing:** For prompt engineering functions and utility scripts.
  - **API Testing:** Done via Swagger UI, Postman, and test scripts.
  - **Manual Testing:** To validate file uploads, chat responses, and output consistency.
  - **Edge Case Handling:** To address malformed inputs, large files, and invalid API keys.
- Each function was validated to ensure reliability in both offline and API-connected modes.

11. Source Code Screenshots:

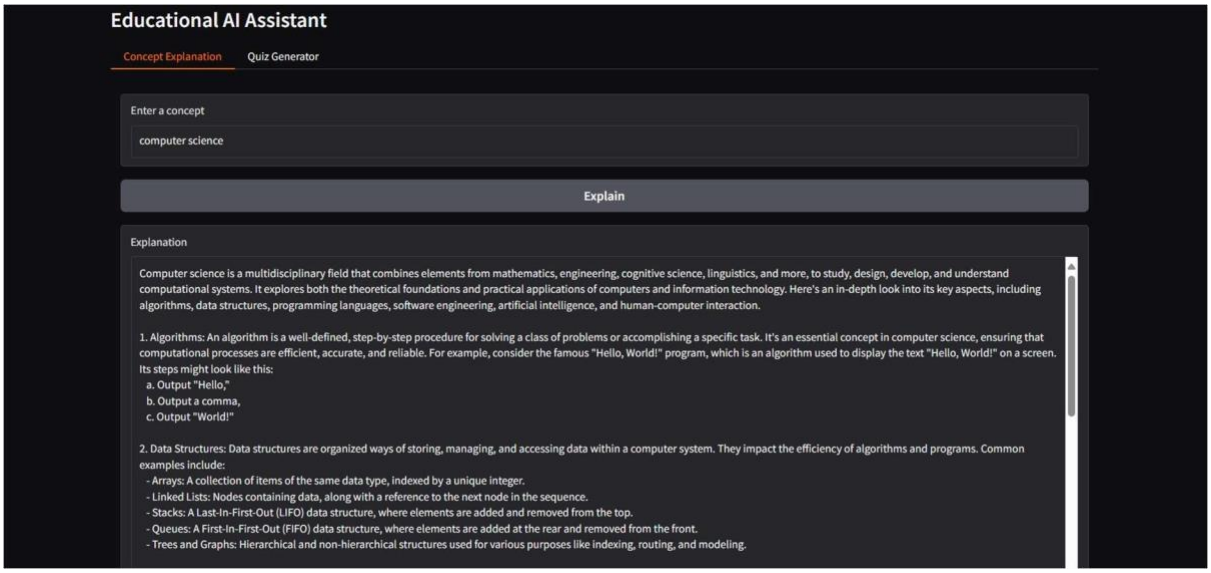


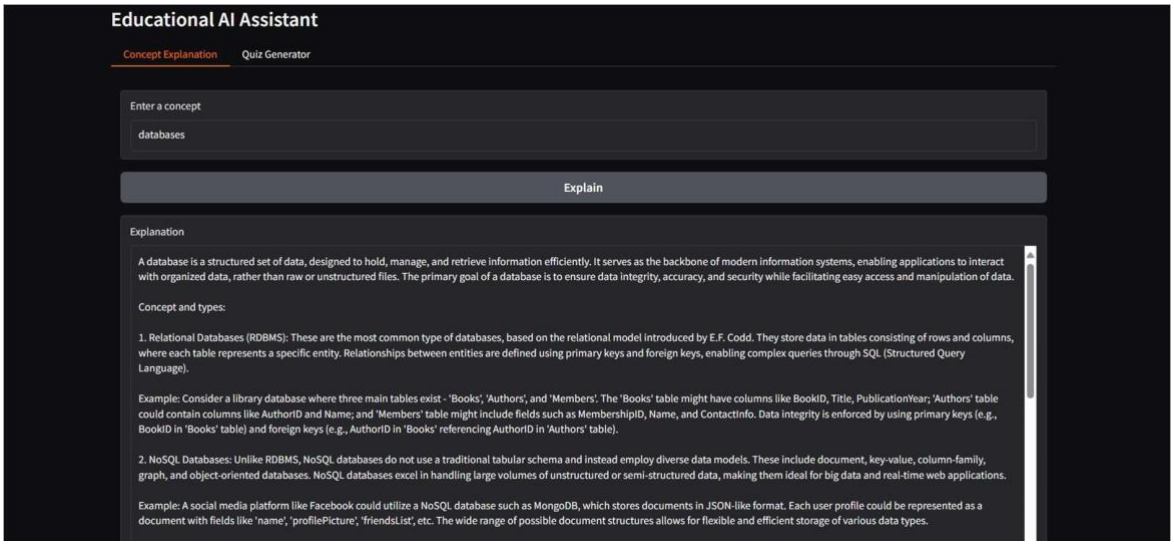
```python
import gradio as gr
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

# Load model and tokenizer
model_name = "ibm-granite/granite-3.2-2b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto" if torch.cuda.is_available() else None
)

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def generate_response(prompt, max_length=512):
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=512)

    if torch.cuda.is_available():
        inputs = {k: v.to(model.device) for k, v in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
```

```
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    response = response.replace(prompt, "").strip()
    return response

def concept_explanation(concept):
    prompt = f"Explain the concept of {concept} in detail with examples:"
    return generate_response(prompt, max_length=800)

def quiz_generator(concept):
    prompt = f"Generate 5 quiz questions about {concept} with different question types (multiple choice, true/false, short answer). At the end, provide all the answers in a sepa
    return generate_response(prompt, max_length=1000)

# Create Gradio interface
with gr.Blocks() as app:
    gr.Markdown("# Educational AI Assistant")

    with gr.Tabs():
        with gr.TabItem("Concept Explanation"):
            concept_input = gr.Textbox(label="Enter a concept", placeholder="e.g., machine learning")
            explain_btn = gr.Button("Explain")
            explanation_output = gr.Textbox(label="Explanation", lines=10)

            explain_btn.click(concept_explanation, inputs=concept_input, outputs=explanation_output)

        with gr.TabItem("Quiz Generator"):
            quiz_input = gr.Textbox(label="Enter a topic", placeholder="e.g., physics")
            quiz_btn = gr.Button("Generate Quiz")
            quiz_output = gr.Textbox(label="Quiz Questions", lines=15)

            quiz_btn.click(quiz_generator, inputs=quiz_input, outputs=quiz_output)
```


```
            explain_btn.click(concept_explanation, inputs=concept_input, outputs=explanation_output)

        with gr.TabItem("Quiz Generator"):
            quiz_input = gr.Textbox(label="Enter a topic", placeholder="e.g., physics")
            quiz_btn = gr.Button("Generate Quiz")
            quiz_output = gr.Textbox(label="Quiz Questions", lines=15)

            quiz_btn.click(quiz_generator, inputs=quiz_input, outputs=quiz_output)

app.launch(share=True)
```

12. Source Output:





13. Future Enhancements:

1. Real-Time Emotional Intelligence (Affective AI)

- Detect student emotion via webcam or voice tone (e.g., confusion, frustration, boredom).
- Adapt responses and learning materials accordingly.
- Integrate IBM Watson Tone Analyzer or third-party emotion AI.

2. Voice-Enabled Tutoring (Speech-to-Text + Text-to-Speech)

- Students interact via speech instead of typing.
- Useful for younger students, visually impaired users, or language learners.
- IBM Watson Speech services can be leveraged here.

3. Multimodal Input & Output

- Support diagrams, charts, handwritten math (via OCR), interactive notebooks.
- Output can include explainer videos, audio, AR/VR objects.
- Combine IBM's Visual Recognition with third-party tools like Unity/Unreal.