Comprehensive Documentation for smartsdlc.py

The `smartsdlc.py` script is a Python-based application built using Gradio that provides two primary functionalities: 1. Requirement analysis: Extracts and analyzes software requirements from PDF documents or text input. 2. Code generation: Generates code snippets in various programming languages based on provided requirements. The script integrates IBM Granite language models (via Hugging Face's `transformers` library) for text analysis and generation.

Libraries Used:
- **Gradio:** To create an interactive web UI for the app.
- **PyTorch:** For running the deep learning models efficiently.
- **Transformers:** To load and use IBM Granite language models.
- **PyPDF2:** To parse and extract text content from uploaded PDF documents.

Model Details:
The script uses the `ibm-granite/granite-3.2-2b-instruct` model for text generation tasks. It dynamically switches between GPU (fp16 precision) and CPU (fp32 precision) based on system availability.

Function: generate_response

This function takes a text prompt and generates a response using the Granite model. It tokenizes input, manages device placement, and applies text generation parameters like temperature and sampling.

Parameters:
- prompt (str): The text input for the model.
- max_length (int): Maximum number of tokens to generate.
Returns: str: Generated response text.

Function: extract_text_from_pdf

Extracts all text from a given PDF file using PyPDF2.

Parameters:
- pdf_file (file): Uploaded PDF file.
Returns: str: Extracted text content.

Function: requirement_analysis

Analyzes text or PDF content to identify software requirements.
Parameters:
- pdf_file (file): Optional PDF document.
- prompt_text (str): Raw requirement text.
Returns: str: Analysis of functional, non-functional, and technical requirements.

Function: code_generation

Generates code snippets in a specified programming language.
Parameters:
- prompt (str): Description of what code to generate.

- language (str): Desired programming language.
Returns: str: Generated code snippet.

Gradio UI:
The app launches with two tabs: 1. **Code Analysis**: Upload PDFs or input requirements text, then analyze. 2. **Code Generation**: Provide prompts and choose programming language to generate code. Each tab has respective buttons that trigger backend functions.

Full Source Code

```
import gradio as gr import torch from transformers import AutoTokenizer,
AutoModelForCausalLM import PyPDF2 import io # 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=1024): 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 extract_text_from_pdf(pdf_file): if pdf_file is None:
return "" try: pdf_reader = PyPDF2.PdfReader(pdf_file) text = "" for page in
pdf_reader.pages: text += page.extract_text() + "\n" return text except Exception as e:
return f"Error reading PDF: {str(e)}" def requirement_analysis(pdf_file, prompt_text): #
Get text from PDF or prompt if pdf_file is not None: content =
extract_text_from_pdf(pdf_file) analysis_prompt = f"Analyze the following document and
extract key software requirements. Organize them into functional requirements,
non-functional requirements, and technical specifications:\n\n{content}" else:
analysis_prompt = f"Analyze the following requirements and organize them into functional
requirements, non-functional requirements, and technical specifications:\n\n{prompt_text}"
return generate_response(analysis_prompt, max_length=1200) def code_generation(prompt,
language): code_prompt = f"Generate {language} code for the following
requirement:\n\n{prompt}\n\nCode:" return generate_response(code_prompt, max_length=1200) #
Create Gradio interface with gr.Blocks() as app: gr.Markdown("# AI Code Analysis &
Generator") with gr.Tabs(): with gr.TabItem("Code Analysis"): with gr.Row(): with
gr.Column(): pdf_upload = gr.File(label="Upload PDF", file_types=[".pdf"]) prompt_input =
gr.Textbox( label="Or write requirements here", placeholder="Describe your software
requirements...", lines=5 ) analyze_btn = gr.Button("Analyze") with gr.Column():
analysis_output = gr.Textbox(label="Requirements Analysis", lines=20)
analyze_btn.click(requirement_analysis, inputs=[pdf_upload, prompt_input],
outputs=analysis_output) with gr.TabItem("Code Generation"): with gr.Row(): with
gr.Column(): code_prompt = gr.Textbox( label="Code Requirements", placeholder="Describe
what code you want to generate...", lines=5 ) language_dropdown = gr.Dropdown(
choices=["Python", "JavaScript", "Java", "C++", "C#", "PHP", "Go", "Rust"],
label="Programming Language", value="Python" ) generate_btn = gr.Button("Generate Code")
with gr.Column(): code_output = gr.Textbox(label="Generated Code", lines=20)
generate_btn.click(code_generation, inputs=[code_prompt, language_dropdown],
outputs=code_output) app.launch(share=True)
```