1720693600_creating_designs_by_leveraging_openai_and_gradio_ui

December 28, 2024

1 Netflix Campaign Design Generator

1.1 Using OpenAI DALL-E and Gradio UI

This notebook implements an AI-powered design generation tool for Netflix marketing campaigns. We'll build this step by step with detailed explanations.

1.2 1. Initial Setup and Imports

This section sets up our development environment by importing necessary Python libraries:

Code Breakdown: - `os`: Provides functions for interacting with the operating system (used for environment variables) - `dataclasses`: Provides the @dataclass decorator for creating data classes - `typing`: Provides type hints (Dict, List, Optional, Tuple) for better code documentation - `dotenv`: Helps manage environment variables from a .env file - `openai`: The OpenAI API client for accessing DALL-E - `gradio`: Creates web-based interfaces for machine learning models - `requests`: Handles HTTP requests - `PIL`: Python Imaging Library for image processing - `BytesIO`: Handles binary data streams - `logging`: Provides logging capabilities - `time`: Provides time-related functions - `re`: Regular expressions for text processing

The logging configuration sets up error tracking and debugging information.

[1]:
```
# Install required packages if not already installed
# !pip install openai gradio Pillow python-dotenv requests

import os
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple
from dotenv import load_dotenv
from openai import OpenAI
import gradio as gr
import requests
from PIL import Image
from io import BytesIO
import logging
import time
```

```
import re

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
```

1.3 2. Configuration Settings

This section defines our application's configuration using a dataclass.

Code Breakdown: - `@dataclass`: A decorator that automatically adds generated special methods to the class - `Config` class contains: - `IMAGE_SIZES`: List of available image dimensions - `DEFAULT_SIZE`: The default image size if none specified - `STYLE_PRESETS`: Dictionary mapping style names to prompt templates - `CACHE_EXPIRY`: How long cached images remain valid (in seconds) - `MAX_CACHE_SIZE`: Maximum number of images to keep in cache - `load_env()`: Static method that loads the OpenAI API key from environment variables

The style presets use Python's string formatting with {prompt} as a placeholder that gets replaced with the user's input.

[2]:
```
from dataclasses import dataclass, field
from typing import Dict, List

@dataclass
class Config:
    """Configuration settings for the application"""
    IMAGE_SIZES: List[str] = field(default_factory=lambda: ["1024x1024",␣
 ↪"512x512"])
    DEFAULT_SIZE: str = "1024x1024"
    STYLE_PRESETS: Dict[str, str] = field(default_factory=lambda: {
        "Standard": "Create a high-quality image of {prompt}",
        "Movie Poster": "Create a dramatic movie poster style image with␣
 ↪{prompt}. Include cinematic lighting and theatrical elements",
        "Netflix Banner": "Create a wide Netflix-style banner featuring␣
 ↪{prompt}. Use dramatic lighting and Netflix's signature look",
        "Abstract Art": "Generate an abstract artistic interpretation of␣
 ↪{prompt} with bold colors and striking composition",
    })
    CACHE_EXPIRY: int = 3600  # Cache expiry in seconds (1 hour)
    MAX_CACHE_SIZE: int = 100  # Maximum number of items in cache

    @staticmethod
    def load_env():
        """Load and validate environment variables"""
```

```
        load_dotenv(verbose=True)
        api_key = os.getenv('OPENAI_API_KEY')
        if not api_key:
            raise ValueError("OPENAI_API_KEY not found in environment␣
 ↪variables")
        return api_key

# Create Config instance
config = Config()
```

1.4 3. Caching System

This section implements a simple caching system to store and retrieve generated images.

Code Breakdown: - `ImageCache` class contains: - `__init__`: Initializes an empty dictionary to store cached images - `get(key)`: - Checks if an image exists in cache - Verifies if it hasn't expired - Returns the image or None - `set(key, value)`: - Stores an image with timestamp - Removes oldest items if cache is full - Uses dictionary with nested structure: {key: {'data': image, 'timestamp': time}}

The cache helps reduce API calls and improves response time for repeated requests.

[3]:
```
class ImageCache:
    """Simple caching system for generated images"""
    def __init__(self):
        self._cache: Dict[str, Dict[str, Any]] = {}

    def get(self, key: str) -> Optional[Image.Image]:
        """Retrieve an image from cache if it exists and hasn't expired"""
        if key in self._cache:
            item = self._cache[key]
            if time.time() - item['timestamp'] < Config.CACHE_EXPIRY:
                return item['data']
            else:
                del self._cache[key]
        return None

    def set(self, key: str, value: Image.Image):
        """Store an image in the cache"""
        self._cache[key] = {
            'data': value,
            'timestamp': time.time()
        }

        # Remove oldest items if cache is too large
        if len(self._cache) > Config.MAX_CACHE_SIZE:
            oldest_key = min(self._cache.keys(),
```

```
                            key=lambda k: self._cache[k]['timestamp'])
            del self._cache[oldest_key]

# Initialize cache
image_cache = ImageCache()
```

1.5 4. Image Processing Utilities

This section handles image processing tasks through the ImageProcessor class.

Code Breakdown: - `ImageProcessor` class contains two static methods: - `process_image(image_url)`: - Downloads image from URL using requests - Converts to PIL Image object - Optimizes image format - Handles errors with logging - `create_error_image(message)`: - Creates a new blank image - Adds error message text - Used when image generation fails

Static methods are used because no instance state is needed for these operations.

[4]:
```
class ImageProcessor:
    """Handles image processing operations"""

    @staticmethod
    def process_image(image_url: str) -> Optional[Image.Image]:
        """Download and process an image from a URL"""
        try:
            response = requests.get(image_url)
            response.raise_for_status()

            # Create PIL Image from response content
            img = Image.open(BytesIO(response.content))

            # Optimize image
            if img.mode in ('RGBA', 'P'):
                img = img.convert('RGB')

            return img

        except Exception as e:
            logging.error(f"Error processing image: {str(e)}")
            return None

    @staticmethod
    def create_error_image(message: str) -> Image.Image:
        """Create a professional-looking error image"""
        img = Image.new('RGB', (512, 512), color=(240, 240, 240))
        from PIL import ImageDraw
        draw = ImageDraw.Draw(img)
        draw.text((20, 20), f"Error:\n{message}", fill=(33, 33, 33))
```

```
        return img
```

1.6 5. Utility Functions

This section contains helper functions for input processing and validation.

Code Breakdown: - `sanitize_prompt(prompt)`: - Removes extra whitespace using split() and join() - Removes special characters with regex - Validates non-empty result - `apply_style_template(style, prompt)`: - Looks up style template from Config - Formats prompt into template - Falls back to original prompt if style not found - `validate_size(size)`: - Checks if size is in allowed list - Returns default size if invalid

These functions ensure clean, safe input before processing.

[5]:
```python
def sanitize_prompt(prompt: str) -> str:
    """Clean and validate the input prompt"""
    # Remove extra whitespace
    cleaned = ' '.join(prompt.split())

    # Remove any potentially harmful characters
    cleaned = re.sub(r'[^\w\s,.!?-]', '', cleaned)

    if not cleaned:
        raise ValueError("Prompt cannot be empty after sanitization")

    return cleaned

def apply_style_template(style: str, prompt: str) -> str:
    """Apply a style template to the prompt"""
    template = config.STYLE_PRESETS.get(style)  # Use instance attribute
    if not template:
        return prompt
    return template.format(prompt=prompt)

def validate_size(size: str) -> str:
    """Validate the requested image size"""
    if size in config.IMAGE_SIZES:  # Use instance attribute
        return size
    return config.DEFAULT_SIZE  # Use instance attribute
```

1.7 6. Main Image Generation Function

This is the core function that coordinates the entire image generation process.

Code Breakdown: - Function accepts three parameters: - prompt: User's text description - size: Desired image size - style: Selected style preset - Process flow: 1. Validates and sanitizes inputs

2. Checks cache for existing image 3. Initializes OpenAI client 4. Calls DALL-E API to generate image 5. Processes and caches the result - Error handling: - Catches all exceptions - Creates error image if something fails - Returns tuple of (image, status message)

The function uses type hints and returns a tuple containing the image and a status message.

[6]:
```python
def generate_image(
    prompt: str,
    size: str = config.DEFAULT_SIZE,  # Use instance attribute here
    style: str = "Standard"
) -> Tuple[Optional[Image.Image], str]:
    """Generate an image based on the prompt with specified size and style"""
    try:
        # Validate and sanitize inputs
        cleaned_prompt = sanitize_prompt(prompt)
        validated_size = validate_size(size)
        styled_prompt = apply_style_template(style, cleaned_prompt)

        # Check cache
        cache_key = f"{styled_prompt}_{validated_size}"
        cached_image = image_cache.get(cache_key)
        if cached_image:
            return cached_image, "Retrieved from cache"

        # Initialize OpenAI client
        client = OpenAI(api_key=Config.load_env())

        # Generate image
        response = client.images.generate(
            model='dall-e-3',
            prompt=styled_prompt,
            size=validated_size,
            quality="standard",
            n=1,
        )

        # Process the image
        image_url = response.data[0].url
        processed_image = ImageProcessor.process_image(image_url)

        if not processed_image:
            raise Exception("Failed to process generated image")

        # Cache the result
        image_cache.set(cache_key, processed_image)

        return processed_image, "Successfully generated new image"
```

```
    except Exception as e:
        logger.error(f"Error generating image: {str(e)}")
        error_image = ImageProcessor.create_error_image(str(e))
        return error_image, f"Error: {str(e)}"
```

1.8 7. Gradio Interface Setup

This section creates the web-based user interface using Gradio.

Code Breakdown: - Creates Gradio Interface with: - Input components: - Textbox: Multi-line input for prompt - Dropdown: Image size selection - Dropdown: Style preset selection - Output components: - Image: Displays generated image - Markdown: Shows status message - Additional features: - Title and description - Example inputs for demonstration - The launch() method starts the web server

Gradio automatically creates a user-friendly interface from these specifications.

[7]:
```
interface = gr.Interface(
    fn=generate_image,
    inputs=[
        gr.Textbox(
            label="Prompt",
            placeholder="Describe your desired Netflix campaign image...",
            lines=3
        ),
        gr.Dropdown(
            choices=config.IMAGE_SIZES,  # Use instance attribute
            label="Image Size",
            value=config.DEFAULT_SIZE    # Use instance attribute
        ),
        gr.Dropdown(
            choices=list(config.STYLE_PRESETS.keys()),  # Use instance attribute
            label="Style Preset",
            value="Standard"
        )
    ],
    outputs=[
        gr.Image(label="Generated Image"),
        gr.Markdown(label="Generation Details")
    ],
    title="Netflix Campaign Design Generator",
    description=(
        "Create professional campaign designs for Netflix content using AI. "
        "Choose from different styles and customize your generation."
    ),
    examples=[
```

```
        ["A dramatic scene from a sci-fi series with dark atmosphere",␣
 ↪"1024x1024", "Netflix Banner"],
        ["A mysterious detective standing in rain at night", "1024x1024",␣
 ↪"Movie Poster"],
    ]
)

# Launch the interface
interface.launch()
```

```
2024-12-28 11:17:27,762 - httpx - INFO - HTTP Request: GET
http://127.0.0.1:7860/gradio_api/startup-events "HTTP/1.1 200 OK"
2024-12-28 11:17:27,771 - httpx - INFO - HTTP Request: HEAD
http://127.0.0.1:7860/ "HTTP/1.1 200 OK"
```

```
* Running on local URL:  http://127.0.0.1:7860

To create a public link, set `share=True` in `launch()`.
```

```
<IPython.core.display.HTML object>
```

[7]:

```
2024-12-28 11:17:27,886 - httpx - INFO - HTTP Request: GET
https://api.gradio.app/pkg-version "HTTP/1.1 200 OK"
2024-12-28 11:18:29,884 - httpx - INFO - HTTP Request: POST
https://api.openai.com/v1/images/generations "HTTP/1.1 400 Bad Request"
2024-12-28 11:18:29,889 - __main__ - ERROR - Error generating image: Error code:
400 - {'error': {'code': 'content_policy_violation', 'message': 'Your request
was rejected as a result of our safety system. Your prompt may contain text that
is not allowed by our safety system.', 'param': None, 'type':
'invalid_request_error'}}
2024-12-28 11:18:59,443 - httpx - INFO - HTTP Request: POST
https://api.openai.com/v1/images/generations "HTTP/1.1 200 OK"
2024-12-28 11:19:43,695 - httpx - INFO - HTTP Request: POST
https://api.openai.com/v1/images/generations "HTTP/1.1 200 OK"
```