CAPStone Project Ideas

1. **Intelligent Document Assistant**
    - Build a RAG-based system that can analyse and answer questions about technical documents
    - Incorporates: RAG, LLMs, Python, Neural Networks
    - Key features:
        - Document preprocessing and embedding
        - Context-aware question answering
        - Basic UI using Streamlit
        - Performance evaluation metrics

2. **Customer Service Automation Platform**
    - Create a multi-agent system for handling customer queries
    - Incorporates: Agentic workflows, Langchain, LLMs, Cloud deployment
    - Key features:
        - Query classification using supervised learning
        - Conversational AI using LLMs
        - Multiple specialised agents (billing, technical support, etc.)
        - Integration with a knowledge base

3. **Predictive Maintenance System**
    - Develop a system that predicts equipment failures using sensor data
    - Incorporates: Deep Learning, Supervised Learning, Model Evaluation
    - Key features:
        - Time series analysis using RNNs
        - Anomaly detection using unsupervised learning
        - Real-time monitoring dashboard
        - Alert system with explainable predictions

4. **Content Generation and Moderation Platform**
    - Build a system that can generate and moderate content
    - Incorporates: Generative AI, Neural Networks, Ethics

- Key features:
    - Text generation using LLMs
    - Content classification for moderation
    - Bias detection and mitigation
    - User feedback integration

5. **Healthcare Diagnosis Assistant**
    - Create a system to assist in medical diagnosis using patient data
    - Incorporates: CNN, Supervised Learning, Ethics, Model Evaluation
    - Key features:
        - Image classification for medical scans
        - Patient data analysis using classical ML
        - Explainable AI components
        - Privacy-preserving data handling

For each project, cohorts should:

- Work in groups of 3-4
- Create proper documentation
- Implement testing and evaluation metrics
- Consider ethical implications
- Deploy a working demo
- Present their findings and challenges

1. Intelligent Document Assistant Implementation Guide

Description: Build a RAG-based system that can analyze and answer questions about technical documents
Incorporates: RAG, LLMs, Python, Neural Networks
Key features:

- Document preprocessing and embedding
- Context-aware question answering
- Basic UI using Streamlit
- Performance evaluation metrics

Phase 1: Project Setup and Document Processing (Week 1)

1.1 Environment Setup

1. Create a new Python virtual environment
2. Install required packages:
   - langchain
   - streamlit
   - sentence-transformers
   - python-dotenv
   - chromadb
   - pypdf (for PDF processing)
   - beautifulsoup4 (for HTML processing)
   - pytorch

1.2 Document Preprocessing Pipeline

1. Create document loader classes:
   - PDF document loader
   - Text document loader
   - HTML document loader
2. Implement text cleaning functions:
   - Remove special characters
   - Handle formatting
   - Split into chunks (consider overlap)
3. Build document metadata extraction:
   - Document type
   - Creation date
   - Author information
   - Section headers

Phase 2: Vector Store and Embeddings (Week 2)

2.1 Document Embedding System

1. Implement document vectorization:
   - Choose embedding model (e.g., BERT, Sentence Transformers)
   - Create embedding pipeline
   - Add batch processing capability
2. Set up vector store:
   - Initialize ChromaDB
   - Create collection structure
   - Implement CRUD operations

2.2 Retrieval System

1. Build similarity search:
   - Implement k-nearest neighbors
   - Add filtering capabilities
   - Create ranking system
2. Create caching mechanism:
   - Store frequent queries
   - Implement cache invalidation
   - Add cache size management

Phase 3: Question-Answering System (Week 3)

3.1 Context Processing

1. Implement context window management:
   - Dynamic window sizing
   - Context merging
   - Relevance scoring
2. Create prompt engineering system:
   - Template management
   - Dynamic prompt generation
   - Context injection

3.2 Answer Generation

1. Set up LLM integration:
   - Configure API connections
   - Implement retry mechanism
   - Add error handling
2. Build answer processing:
   - Source attribution
   - Confidence scoring
   - Answer formatting

Phase 4: UI Development and Integration (Week 4)

4.1 Streamlit Interface

1. Create main application layout:

```python
def main():
    st.title("Intelligent Document Assistant")
    st.sidebar.title("Controls")

    # File upload section
    uploaded_file = st.file_uploader("Upload Document",
        type=['pdf', 'txt', 'html'])

    # Question input
    user_question = st.text_input("Ask a question about your document")

    # Display answers
    if user_question:
        answer = get_answer(user_question, context)
        st.write(answer)
```

2. Add interactive components:
   - Document upload
   - Question input
   - Answer display
   - System status
3. Implement progress tracking:
   - Processing indicators
   - Error messages
   - Success notifications

4.2 Performance Monitoring

1. Implement evaluation metrics:
   - Response time tracking
   - Answer quality assessment
   - User feedback collection
2. Create monitoring dashboard:
   - Performance graphs
   - Usage statistics
   - Error tracking

Phase 5: Testing and Optimization

5.1 Testing Framework

1. Unit tests:
   - Document processing
   - Embedding generation
   - Answer retrieval
2. Integration tests:
   - End-to-end workflows
   - API integration
   - UI functionality

5.2 Optimization

1. Performance tuning:
   - Batch processing
   - Caching strategies
   - Query optimization
2. Resource management:
   - Memory usage
   - API call efficiency
   - Storage optimization

Bonus Features

Advanced Features (Optional)

1. Multi-language support:
   - Language detection
   - Translation integration
   - Language-specific models
2. Document comparison:
   - Similarity analysis
   - Version tracking
   - Change detection
3. Export functionality:
   - PDF report generation
   - Answer history export
   - Usage analytics

Evaluation Criteria

Project Assessment

1. Technical Implementation (40%):
   - Code quality
   - System architecture
   - Performance optimization
2. Documentation (20%):
   - Setup instructions
   - API documentation
   - Usage guidelines
3. User Experience (20%):
   - Interface design
   - Responsiveness
   - Error handling
4. Innovation (20%):
   - Creative solutions
   - Additional features
   - Performance improvements

Initial Code Setup:

```
from typing import List, Dict
import PyPDF2
from bs4 import BeautifulSoup
from sentence_transformers import SentenceTransformer
import chromadb
from langchain.text_splitter import RecursiveCharacterTextSplitter

class DocumentProcessor:
    def __init__(self):
        self.text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000,
            chunk_overlap=200,
            length_function=len,
        )
        self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
        self.chroma_client = chromadb.Client()
        self.collection = self.chroma_client.create_collection(name="documents")

    def process_pdf(self, file_path: str) -> List[str]:
        """Process PDF file and return chunks of text."""
```

```
        with open(file_path, 'rb') as file:
            reader = PyPDF2.PdfReader(file)
            text = ""
            for page in reader.pages:
                text += page.extract_text()
        return self.text_splitter.split_text(text)

    def process_text(self, text: str) -> List[str]:
        """Process raw text and return chunks."""
        return self.text_splitter.split_text(text)

    def create_embeddings(self, chunks: List[str]) -> List[List[float]]:
        """Create embeddings for text chunks."""
        return self.embedding_model.encode(chunks).tolist()

    def store_embeddings(self, embeddings: List[List[float]], chunks: List[str],
metadata: Dict):
        """Store embeddings in ChromaDB."""
        self.collection.add(
            embeddings=embeddings,
            documents=chunks,
            metadatas=[metadata] * len(chunks),
            ids=[f"doc_{i}" for i in range(len(chunks))]
        )

    def process_document(self, file_path: str, metadata: Dict):
        """Complete document processing pipeline."""
        # Process document based on file type
        if file_path.endswith('.pdf'):
            chunks = self.process_pdf(file_path)
        else:
            with open(file_path, 'r') as file:
                chunks = self.process_text(file.read())

        # Create and store embeddings
        embeddings = self.create_embeddings(chunks)
        self.store_embeddings(embeddings, chunks, metadata)
        return len(chunks)

# Example usage
if __name__ == "__main__":
    processor = DocumentProcessor()
```

```
# Example metadata
metadata = {
    "source": "example.pdf",
    "author": "John Doe",
    "date": "2024-10-23"
}

# Process a document
num_chunks = processor.process_document("example.pdf", metadata)
print(f"Processed document into {num_chunks} chunks")
```

2. Customer Service Automation Platform Implementation Guide

Description: Create a multi-agent system for handling customer queries
Incorporates: Agentic workflows, Langchain, LLMs, Cloud deployment
Key features:

- Query classification using supervised learning
- Conversational AI using LLMs
- Multiple specialised agents (billing, technical support, etc.)
- Integration with a knowledge base

Phase 1: System Architecture Setup (Week 1)

1.1 Project Structure

```
customer_service_platform/
├── agents/
│   ├── billing_agent.py
│   ├── technical_agent.py
│   ├── routing_agent.py
│   └── base_agent.py
├── knowledge_base/
│   ├── kb_manager.py
│   └── data/
├── models/
│   ├── classifier.py
│   └── embeddings.py
├── api/
│   └── endpoints.py
├── utils/
│   ├── logger.py
│   └── validators.py
└── config.py
```

1.2 Environment Setup

1. Create virtual environment
2. Install required packages:
   - langchain
   - fastapi

- pydantic
- sentence-transformers
- pytorch
- chromadb
- uvicorn

Phase 2: Knowledge Base Development (Week 1-2)

2.1 Knowledge Base Structure

1. Create knowledge categories:
   - Billing FAQs
   - Technical Documentation
   - Product Information
   - Policy Documents

2.2 Implementation Steps

1. Build KB Manager:

```python
class KnowledgeBaseManager:
    def __init__(self):
        self.embeddings_model = SentenceTransformer('all-MiniLM-L6-v2')
        self.db = chromadb.Client()
        self.collection = self.db.create_collection("customer_service_kb")

    def add_document(self, content, metadata):
        embeddings = self.embeddings_model.encode([content]).tolist()
        self.collection.add(
            embeddings=embeddings,
            documents=[content],
            metadatas=[metadata],
            ids=[str(uuid.uuid4())]
        )

    def search(self, query, filter_metadata=None):
        query_embedding = self.embeddings_model.encode([query]).tolist()
        results = self.collection.query(
            query_embeddings=query_embedding,
            n_results=3,
            where=filter_metadata
```

```
        )
        return results
```

2. Implement document processing
3. Add search functionality
4. Create update mechanisms

Phase 3: Agent System Development (Week 2)

3.1 Base Agent Implementation

1. Create base agent class:

```python
class BaseAgent:
    def __init__(self, kb_manager):
        self.kb_manager = kb_manager
        self.llm = ChatOpenAI(temperature=0.7)
        self.conversation_history = []

    async def process_query(self, query):
        context = self.get_relevant_context(query)
        response = await self.generate_response(query, context)
        self.conversation_history.append({
            "query": query,
            "response": response
        })
        return response

    def get_relevant_context(self, query):
        return self.kb_manager.search(query)
```

3.2 Specialized Agents

1. Billing Agent:
   - Payment processing
   - Invoice queries
   - Refund handling

2. Technical Support Agent:
   - Troubleshooting
   - Setup assistance
   - Bug reporting

3. Routing Agent:
   - Query classification
   - Agent selection
   - Handoff management

Phase 4: Query Classification System (Week 3)

4.1 Classifier Development

1. Implement query classifier:

```python
class QueryClassifier:
    def __init__(self):
        self.model = pipeline(
            "text-classification",
            model="distilbert-base-uncased-finetuned-sst-2-english"
        )

    def classify_query(self, query):
        categories = {
            'billing': ['payment', 'invoice', 'refund', 'charge'],
            'technical': ['error', 'bug', 'install', 'setup'],
            'general': ['info', 'policy', 'account']
        }

        # Implement classification logic
        classification = self.model(query)[0]
        return self.map_to_category(classification)
```

2. Train on customer service data
3. Implement confidence scoring
4. Add feedback loop

Phase 5: Integration and API Development (Week 3-4)

5.1 FastAPI Implementation

1. Create API endpoints:

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.post("/query")
```

```
async def process_customer_query(query: CustomerQuery):
    try:
        # Classify query
        category = query_classifier.classify_query(query.text)

        # Route to appropriate agent
        agent = agent_router.get_agent(category)

        # Process query
        response = await agent.process_query(query.text)

        return {
            "response": response,
            "category": category,
            "confidence": response.confidence
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

2. Add authentication
3. Implement rate limiting
4. Add monitoring endpoints

5.2 WebSocket Support

1. Implement real-time chat
2. Add presence detection
3. Handle disconnections

Phase 6: Monitoring and Analytics (Week 4)

6.1 Performance Metrics

1. Response time tracking
2. Query classification accuracy
3. Customer satisfaction metrics
4. Agent performance metrics

6.2 Analytics Dashboard

1. Query volume trends
2. Common issue identification
3. Agent utilization stats
4. Knowledge base coverage

Evaluation Criteria

1. System Performance (30%)
- Response accuracy
- Processing time
- Scalability
- Error handling

2. Implementation Quality (30%)
- Code organization
- Documentation
- Testing coverage
- Best practices

3. Features (20%)
- Agent capabilities
- Integration completeness
- UI/UX design
- Analytics implementation

4. Innovation (20%)
- Novel solutions
- Additional features
- Performance optimizations
- User experience enhancements

Bonus Features

Advanced Capabilities
1. Multi-language support
2. Voice interface
3. Sentiment analysis
4. Automated escalation
5. Custom analytics dashboards

Initital Code Setup:

```
from typing import Dict, List, Optional
from enum import Enum
from pydantic import BaseModel
import asyncio
from datetime import datetime

class QueryCategory(Enum):
    BILLING = "billing"
    TECHNICAL = "technical"
    GENERAL = "general"

class CustomerQuery(BaseModel):
    text: str
    category: Optional[QueryCategory]
    user_id: str
    timestamp: datetime = datetime.now()

class AgentRouter:
    def __init__(self, kb_manager):
        self.kb_manager = kb_manager
        self.agents: Dict[QueryCategory, BaseAgent] = {}
        self.initialize_agents()

    def initialize_agents(self):
        """Initialize specialized agents for each category."""
        self.agents = {
            QueryCategory.BILLING: BillingAgent(self.kb_manager),
            QueryCategory.TECHNICAL: TechnicalAgent(self.kb_manager),
            QueryCategory.GENERAL: GeneralAgent(self.kb_manager)
        }

    async def route_query(self, query: CustomerQuery) -> dict:
        """Route query to appropriate agent and get response."""
        try:
            # If category not provided, classify query
            if not query.category:
                query.category = self.classify_query(query.text)

            # Get appropriate agent
            agent = self.agents.get(query.category)
            if not agent:
```

```
                raise ValueError(f"No agent available for category:
{query.category}")

            # Process query
            response = await agent.process_query(query.text)

            # Log interaction
            self.log_interaction(query, response)

            return {
                "response": response,
                "category": query.category.value,
                "timestamp": datetime.now(),
                "query_id": str(uuid.uuid4())
            }

        except Exception as e:
            # Log error and return to general agent
            self.log_error(query, str(e))
            return await self.handle_fallback(query)

    def classify_query(self, text: str) -> QueryCategory:
        """Classify incoming query to determine appropriate agent."""
        # Implement classification logic
        # For now, using simple keyword matching
        keywords = {
            QueryCategory.BILLING: ["payment", "bill", "charge", "refund",
"invoice"],
            QueryCategory.TECHNICAL: ["error", "bug", "broken", "help", "how to"],
        }

        text_lower = text.lower()
        for category, words in keywords.items():
            if any(word in text_lower for word in words):
                return category

        return QueryCategory.GENERAL

    async def handle_fallback(self, query: CustomerQuery) -> dict:
        """Handle queries when primary routing fails."""
        general_agent = self.agents[QueryCategory.GENERAL]
        response = await general_agent.process_query(query.text)
```

```
            return {
                "response": response,
                "category": QueryCategory.GENERAL.value,
                "timestamp": datetime.now(),
                "query_id": str(uuid.uuid4()),
                "fallback": True
            }

        def log_interaction(self, query: CustomerQuery, response: dict):
            """Log interaction for analysis and improvement."""
            # Implement logging logic
            pass

        def log_error(self, query: CustomerQuery, error: str):
            """Log errors for monitoring and debugging."""
            # Implement error logging
            pass

    # Example usage
    if __name__ == "__main__":
        # Initialize knowledge base
        kb_manager = KnowledgeBaseManager()

        # Initialize router
        router = AgentRouter(kb_manager)

        # Example query
        query = CustomerQuery(
            text="I need help with my latest bill",
            user_id="user123"
        )

        # Process query
        async def main():
            response = await router.route_query(query)
            print(f"Response: {response}")

        asyncio.run(main())
```

3. Predictive Maintenance System Implementation Guide

Description: Develop a system that predicts equipment failures using sensor data
Incorporates: Deep Learning, Supervised Learning, Model Evaluation
Key features:

- Time series analysis using RNNs
- Anomaly detection using unsupervised learning
- Real-time monitoring dashboard
- Alert system with explainable predictions

Phase 1: Data Pipeline Setup (Week 1)

1.1 Project Structure

```
predictive_maintenance/
├── data/
│   ├── raw/
│   ├── processed/
│   └── models/
├── src/
│   ├── data_ingestion/
│   ├── preprocessing/
│   ├── models/
│   ├── visualization/
│   └── api/
├── notebooks/
├── tests/
└── config/
```

1.2 Environment Setup
1. Required packages:
  - pandas
  - numpy
  - scikit-learn
  - pytorch
  - plotly
  - streamlit
  - pytest
  - fastapi

- influxdb-client (for time series)

Phase 2: Data Processing Pipeline (Week 1-2)

2.1 Data Ingestion

1. Implement sensor data collectors
2. Set up real-time data streaming
3. Create data validation systems
4. Implement data storage solutions

2.2 Data Preprocessing

1. Time series alignment
2. Missing value handling
3. Anomaly detection
4. Feature engineering
5. Data normalization

Phase 3: Model Development (Week 2)

3.1 Time Series Analysis

1. Implement trend analysis
2. Seasonal decomposition
3. Pattern recognition
4. Feature importance analysis

3.2 Model Architecture

1. Build base model classes
2. Implement different model types:
   - RNN for sequence prediction
   - Anomaly detection models
   - Classification models for failure types
   - Regression models for time-to-failure

3.3 Model Training Pipeline

1. Data splitting strategies
2. Cross-validation setup
3. Model evaluation metrics
4. Hyperparameter optimization

Phase 4: Alert System (Week 3)

4.1 Alert Generation

1. Threshold-based alerts

2. Anomaly-based alerts
3. Prediction-based alerts
4. Alert prioritization

4.2 Notification System

1. Email notifications
2. SMS alerts
3. Dashboard notifications
4. Alert logging

Phase 5: Dashboard Development (Week 3-4)

5.1 Main Dashboard

1. Real-time monitoring
2. Historical analysis
3. Prediction visualization
4. Alert management

5.2 Report Generation

1. Automated reporting
2. Custom report builder
3. Export functionality
4. Scheduling system

Phase 6: System Integration (Week 4)

6.1 API Development

1. Data ingestion endpoints
2. Prediction endpoints
3. Alert management endpoints
4. Report generation endpoints

6.2 Testing and Validation

1. Unit tests
2. Integration tests
3. Load testing
4. System validation

Evaluation Criteria

1. Technical Implementation (40%)
   - Model accuracy
   - System reliability

- Code quality
- Performance optimization

2. Features (30%)
   - Real-time monitoring
   - Alert system
   - Reporting capabilities
   - User interface

3. Documentation (15%)
   - Code documentation
   - API documentation
   - System architecture
   - User guide

4. Innovation (15%)
   - Novel approaches
   - Additional features
   - Performance improvements
   - UX enhancements

Bonus Features

1. Mobile app integration
2. Advanced visualization
3. Automated maintenance scheduling
4. Cost impact analysis

Initial Code Setup:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from torch import nn
import torch
from typing import List, Dict, Tuple
import logging
from datetime import datetime

class TimeSeriesPreprocessor:
    def __init__(self, window_size: int = 24):
        self.window_size = window_size
        self.scaler = StandardScaler()

    def create_sequences(self, data: pd.DataFrame) -> Tuple[np.ndarray, np.ndarray]:
        """Create sequences for time series prediction."""
        sequences = []
        targets = []

        for i in range(len(data) - self.window_size):
            sequence = data.iloc[i:(i + self.window_size)].values
            target = data.iloc[i + self.window_size].values
            sequences.append(sequence)
            targets.append(target)

        return np.array(sequences), np.array(targets)

    def normalize_data(self, data: pd.DataFrame) -> pd.DataFrame:
        """Normalize the sensor data."""
        normalized_data = pd.DataFrame(
            self.scaler.fit_transform(data),
            columns=data.columns,
            index=data.index
        )
        return normalized_data

class MaintenancePredictor(nn.Module):
    def __init__(self, input_size: int, hidden_size: int, num_layers: int = 2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
        # LSTM layer
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=0.2
        )

        # Prediction layers
        self.fc = nn.Sequential(
            nn.Linear(hidden_size, 50),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(50, 1)
        )

    def forward(self, x):
        # LSTM forward pass
        lstm_out, _ = self.lstm(x)

        # Use last lstm output for prediction
        last_output = lstm_out[:, -1, :]

        # Make prediction
        prediction = self.fc(last_output)
        return prediction

class AlertSystem:
    def __init__(self, threshold: float = 0.8):
        self.threshold = threshold
        self.alerts = []

    def check_threshold(self, prediction: float, actual: float) -> bool:
        """Check if prediction exceeds threshold."""
        return abs(prediction - actual) > self.threshold

    def generate_alert(self, sensor_id: str, prediction: float, actual: float) -> Dict:
        """Generate alert if threshold is exceeded."""
        if self.check_threshold(prediction, actual):
            alert = {
                'sensor_id': sensor_id,
```

```
                'timestamp': datetime.now(),
                'prediction': prediction,
                'actual': actual,
                'severity': self.calculate_severity(prediction, actual)
            }
            self.alerts.append(alert)
            return alert
        return None

    def calculate_severity(self, prediction: float, actual: float) -> str:
        """Calculate alert severity based on deviation."""
        deviation = abs(prediction - actual)
        if deviation > self.threshold * 2:
            return 'HIGH'
        elif deviation > self.threshold * 1.5:
            return 'MEDIUM'
        return 'LOW'

class MaintenanceSystem:
    def __init__(self, model: MaintenancePredictor, preprocessor:
TimeSeriesPreprocessor):
        self.model = model
        self.preprocessor = preprocessor
        self.alert_system = AlertSystem()
        self.logger = logging.getLogger(__name__)

    def process_sensor_data(self, data: pd.DataFrame) -> Dict:
        """Process incoming sensor data and generate predictions."""
        try:
            # Preprocess data
            normalized_data = self.preprocessor.normalize_data(data)
            sequences, targets = self.preprocessor.create_sequences(normalized_data)

            # Convert to torch tensors
            sequences_tensor = torch.FloatTensor(sequences)

            # Generate predictions
            with torch.no_grad():
                predictions = self.model(sequences_tensor)

            # Check for alerts
            alerts = []
            for i, (pred, actual) in enumerate(zip(predictions, targets)):
```

```
                alert = self.alert_system.generate_alert(
                    f"sensor_{i}",
                    pred.item(),
                    actual.item()
                )
                if alert:
                    alerts.append(alert)

            return {
                'predictions': predictions.numpy().tolist(),
                'alerts': alerts
            }

        except Exception as e:
            self.logger.error(f"Error processing sensor data: {str(e)}")
            raise

# Example usage
if __name__ == "__main__":
    # Initialize components
    preprocessor = TimeSeriesPreprocessor(window_size=24)
    model = MaintenancePredictor(input_size=10, hidden_size=64)
    system = MaintenanceSystem(model, preprocessor)

    # Example data processing
    sample_data = pd.DataFrame(
        np.random.randn(100, 10),
        columns=[f'sensor_{i}' for i in range(10)]
    )

    # Process data and get predictions
    results = system.process_sensor_data(sample_data)
    print(f"Generated {len(results['alerts'])} alerts")
```

Streamlit Dashboard Setup:

```
import streamlit as st
import plotly.graph_objects as go
import plotly.express as px
from datetime import datetime, timedelta
import pandas as pd
import numpy as np
```

```
class MaintenanceDashboard:
    def __init__(self, maintenance_system):
        self.maintenance_system = maintenance_system
        self.setup_page()

    def setup_page(self):
        """Configure the Streamlit page."""
        st.set_page_config(
            page_title="Predictive Maintenance Dashboard",
            page_icon="🔧",
            layout="wide"
        )
        st.title("Predictive Maintenance Dashboard")

    def run(self):
        """Run the dashboard."""
        # Sidebar controls
        st.sidebar.header("Controls")
        selected_view = st.sidebar.selectbox(
            "Select View",
            ["Overview", "Sensor Details", "Alerts", "Reports"]
        )

        # Display selected view
        if selected_view == "Overview":
            self.show_overview()
        elif selected_view == "Sensor Details":
            self.show_sensor_details()
        elif selected_view == "Alerts":
            self.show_alerts()
        else:
            self.show_reports()

    def show_overview(self):
        """Display system overview."""
        col1, col2 = st.columns(2)

        with col1:
            st.subheader("System Health")
            self.plot_system_health()

        with col2:
            st.subheader("Recent Alerts")
```

```
            self.show_recent_alerts()

        # Predictions timeline
        st.subheader("Maintenance Predictions")
        self.plot_predictions_timeline()

    def plot_system_health(self):
        """Plot overall system health metrics."""
        # Sample data - replace with real data
        health_metrics = {
            "Sensor 1": 0.92,
            "Sensor 2": 0.85,
            "Sensor 3": 0.78,
            "Sensor 4": 0.95
        }

        fig = go.Figure()
        for sensor, health in health_metrics.items():
            fig.add_trace(go.Indicator(
                mode="gauge+number",
                value=health * 100,
                title={'text': sensor},
                domain={'row': 0, 'column': 0},
                gauge={
                    'axis': {'range': [None, 100]},
                    'steps': [
                        {'range': [0, 60], 'color': "red"},
                        {'range': [60, 80], 'color': "yellow"},
                        {'range': [80, 100], 'color': "green"}
                    ]
                }
            ))

        fig.update_layout(
            grid={'rows': 2, 'columns': 2, 'pattern': "independent"},
            height=400
        )
        st.plotly_chart(fig)

    def show_recent_alerts(self):
        """Display recent system alerts."""
        alerts = self.maintenance_system.alert_system.alerts[-5:]
        if alerts:
```

```
            for alert in alerts:
                with st.container():
                    severity_color = {
                        'HIGH': 'red',
                        'MEDIUM': 'orange',
                        'LOW': 'blue'
                    }[alert['severity']]

                    st.markdown(
                        f"<div style='padding:10px;border-left:5px solid
{severity_color};margin:5px'>"
                        f"<strong>{alert['sensor_id']}</strong><br>"
                        f"Severity: {alert['severity']}<br>"
                        f"Time: {alert['timestamp']}"
                        "</div>",
                        unsafe_allow_html=True
                    )
        else:
            st.info("No recent alerts")

    def plot_predictions_timeline(self):
        """Plot predictions timeline."""
        # Sample data - replace with real data
        dates = pd.date_range(start='2024-01-01', periods=30, freq='D')
        predictions = np.random.normal(loc=0.8, scale=0.1, size=30)
        actuals = predictions + np.random.normal(loc=0, scale=0.05, size=30)

        df = pd.DataFrame({
            'date': dates,
            'prediction': predictions,
            'actual': actuals
        })

        fig = px.line(df, x='date', y=['prediction', 'actual'],
                      title='Maintenance Predictions vs Actuals')
        st.plotly_chart(fig)

    def show_sensor_details(self):
        """Display detailed sensor information."""
        selected_sensor = st.selectbox(
            "Select Sensor",
            [f"Sensor {i+1}" for i in range(10)]
        )
```

```
        col1, col2 = st.columns(2)

        with col1:
            st.subheader("Sensor Metrics")
            self.plot_sensor_metrics(selected_sensor)

        with col2:
            st.subheader("Failure Probability")
            self.plot_failure_probability(selected_sensor)

    def plot_sensor_metrics(self, sensor):
        """Plot detailed sensor metrics."""
        # Implement sensor-specific visualizations
        pass

    def plot_failure_probability(self, sensor):
        """Plot failure probability over time."""
        # Implement failure probability visualization
        pass

# Example usage
if __name__ == "__main__":
    # Initialize maintenance system
    preprocessor = TimeSeriesPreprocessor(window_size=24)
    model = MaintenancePredictor(input_size=10, hidden_size=64)
    maintenance_system = MaintenanceSystem(model, preprocessor)

    # Create and run dashboard
    dashboard = MaintenanceDashboard(maintenance_system)
    dashboard.run()
```

4. Content Generation and Moderation Platform Implementation Guide

Description: Build a system that can generate and moderate content
Incorporates: Generative AI, Neural Networks, Ethics
Key features:

- Text generation using LLMs
- Content classification for moderation
- Bias detection and mitigation
- User feedback integration

Phase 1: System Architecture Setup (Week 1)

1.1 Project Structure

```
content_platform/
├── src/
│   ├── generation/
│   │   ├── text_generator.py
│   │   ├── prompt_manager.py
│   │   └── content_validator.py
│   ├── moderation/
│   │   ├── content_classifier.py
│   │   ├── bias_detector.py
│   │   └── toxicity_analyzer.py
│   ├── api/
│   │   ├── routes.py
│   │   └── middleware.py
│   └── utils/
│       ├── logger.py
│       └── metrics.py
├── models/
│   ├── classifiers/
│   └── embeddings/
├── tests/
├── config/
└── web/
    ├── frontend/
    └── templates/
```

1.2 Environment Setup

1. Required packages:

```python
requirements.txt:
langchain>=0.1.0
transformers>=4.30.0
pytorch>=2.0.0
fastapi>=0.100.0
streamlit>=1.25.0
tiktoken>=0.4.0
python-dotenv>=1.0.0
pandas>=2.0.0
scikit-learn>=1.3.0
redis>=4.5.0
sqlalchemy>=2.0.0
pytest>=7.4.0
```

Phase 2: Content Generation System (Week 1-2)

2.1 Text Generator Implementation

1. LLM Integration
2. Prompt Engineering
3. Content Formatting
4. Output Validation

2.2 Generation Features

1. Template-based generation
2. Style customization
3. Length control
4. Topic management

Phase 3: Content Moderation System (Week 2)

3.1 Content Classification

1. Category detection
2. NSFW content detection
3. Sentiment analysis
4. Language detection

3.2 Bias Detection

1. Gender bias analysis
2. Cultural bias detection

3. Age-related bias
4. Geographical bias

3.3 Toxicity Analysis

1. Hate speech detection
2. Harassment detection
3. Profanity filtering
4. Threat assessment

Phase 4: Feedback Integration (Week 3)

4.1 User Feedback System

1. Rating mechanism
2. Report system
3. Content flagging
4. Appeal process

4.2 Model Improvement

1. Feedback incorporation
2. Model fine-tuning
3. Performance tracking
4. Quality metrics

Phase 5: API and Frontend Development (Week 3-4)

5.1 API Development

1. Content generation endpoints
2. Moderation endpoints
3. Feedback endpoints
4. Analytics endpoints

5.2 Frontend Implementation

1. Content creation interface
2. Moderation dashboard
3. Analytics visualization
4. User management

Phase 6: Analytics and Reporting (Week 4)

6.1 Performance Metrics

1. Generation quality metrics
2. Moderation accuracy
3. Response times

4. User satisfaction

6.2 Reporting System

1. Content analysis reports
2. Moderation statistics
3. User engagement metrics
4. System performance reports

Evaluation Criteria

1. Technical Implementation (35%)

- Code quality
- System architecture
- Performance optimization
- Error handling

2. Content Quality (25%)

- Generation accuracy
- Moderation effectiveness
- Bias detection accuracy
- Content relevance

3. User Experience (20%)

- Interface design
- System responsiveness
- Feature accessibility
- Error feedback

4. Innovation (20%)

- Novel approaches
- Additional features
- Performance improvements
- Creative solutions

Bonus Features

1. Multi-language support
2. Content summarization
3. Automated tagging
4. Version control for content

Core implementation code:

```
import torch
from transformers import pipeline, AutoModelForSequenceClassification,
AutoTokenizer
from typing import List, Dict, Optional
import numpy as np
from datetime import datetime
import logging
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from pydantic import BaseModel

class ContentGenerator:
    def __init__(self, model_name: str = "gpt-3.5-turbo"):
        self.llm = ChatOpenAI(model_name=model_name)
        self.logger = logging.getLogger(__name__)

    def generate_content(self, prompt: str, style: Dict = None, max_length: int =
1000) -> str:
        """Generate content based on prompt and style guidelines."""
        try:
            # Create system message with style guidelines
            system_message = self._create_system_message(style)

            # Create prompt template
            prompt_template = ChatPromptTemplate.from_messages([
                ("system", system_message),
                ("user", "{input_prompt}")
            ])

            # Generate content
            chain = prompt_template | self.llm
            response = chain.invoke({"input_prompt": prompt})

            # Validate and format response
            formatted_content = self._format_content(response.content, max_length)

            return formatted_content

        except Exception as e:
            self.logger.error(f"Error generating content: {str(e)}")
            raise
```

```
    def _create_system_message(self, style: Optional[Dict] = None) -> str:
        """Create system message with style guidelines."""
        base_message = "You are a professional content creator."
        if style:
            style_guidelines = [f"- {key}: {value}" for key, value in style.items()]
            return f"{base_message}\nStyle guidelines:\n" + "\n".join(style_guidelines)
        return base_message

    def _format_content(self, content: str, max_length: int) -> str:
        """Format and validate generated content."""
        # Trim to max length while preserving complete sentences
        if len(content) > max_length:
            content = content[:max_length]
            last_period = content.rfind('.')
            if last_period > 0:
                content = content[:last_period + 1]

        return content.strip()

class ContentModerator:
    def __init__(self):
        # Initialize various classification models
        self.toxicity_model = pipeline("text-classification",
                                       model="unitary/toxic-bert")
        self.bias_model = pipeline("text-classification",
                                   model="microsoft/deberta-v3-base")
        self.nsfw_model = pipeline("text-classification",
                                   model="facebook/roberta-hate-speech-detector")

    def moderate_content(self, content: str) -> Dict:
        """Perform comprehensive content moderation."""
        results = {
            'toxicity': self._check_toxicity(content),
            'bias': self._check_bias(content),
            'nsfw': self._check_nsfw(content),
            'timestamp': datetime.now(),
            'content_hash': hash(content)
        }

        # Calculate overall safety score
        results['safety_score'] = self._calculate_safety_score(results)
```

```
        return results

    def _check_toxicity(self, content: str) -> Dict:
        """Check content for toxic elements."""
        result = self.toxicity_model(content)
        return {
            'score': result[0]['score'],
            'label': result[0]['label'],
            'is_toxic': result[0]['score'] > 0.7
        }

    def _check_bias(self, content: str) -> Dict:
        """Check content for various types of bias."""
        # Implement bias detection logic
        return {
            'gender_bias': self._detect_gender_bias(content),
            'cultural_bias': self._detect_cultural_bias(content),
            'age_bias': self._detect_age_bias(content)
        }

    def _check_nsfw(self, content: str) -> Dict:
        """Check content for NSFW elements."""
        result = self.nsfw_model(content)
        return {
            'score': result[0]['score'],
            'label': result[0]['label'],
            'is_nsfw': result[0]['score'] > 0.7
        }

    def _calculate_safety_score(self, results: Dict) -> float:
        """Calculate overall safety score."""
        weights = {
            'toxicity': 0.4,
            'bias': 0.3,
            'nsfw': 0.3
        }

        toxicity_score = 1 - results['toxicity']['score']
        bias_score = 1 - max([v for v in results['bias'].values() if isinstance(v, float)],
default=0)
        nsfw_score = 1 - results['nsfw']['score']

        return (
```

```
            weights['toxicity'] * toxicity_score +
            weights['bias'] * bias_score +
            weights['nsfw'] * nsfw_score
        )

class ContentPlatform:
    def __init__(self):
        self.generator = ContentGenerator()
        self.moderator = ContentModerator()
        self.logger = logging.getLogger(__name__)

    async def create_content(self,
                             prompt: str,
                             style: Dict = None,
                             max_length: int = 1000) -> Dict:
        """Generate and moderate content."""
        try:
            # Generate content
            content = self.generator.generate_content(
                prompt=prompt,
                style=style,
                max_length=max_length
            )

            # Moderate content
            moderation_results = self.moderator.moderate_content(content)

            # Prepare response
            response = {
                'content': content,
                'moderation': moderation_results,
                'timestamp': datetime.now(),
                'prompt': prompt,
                'style': style
            }

            # Log generation
            self._log_generation(response)

            return response

        except Exception as e:
            self.logger.error(f"Error in content creation: {str(e)}")
```

```
            raise

    def _log_generation(self, response: Dict):
        """Log content generation details."""
        # Implement logging logic
        pass

# Example usage
if __name__ == "__main__":
    # Initialize platform
    platform = ContentPlatform()

    # Example content generation
    async def generate_example():
        prompt = "Write a brief article about sustainable energy."
        style = {
            "tone": "professional",
            "length": "medium",
            "audience": "general public"
        }

        response = await platform.create_content(prompt, style)
        print(f"Content generated with safety score:
{response['moderation']['safety_score']}")

    # Run example
    import asyncio
    asyncio.run(generate_example())
```

Streamlit Interface implementation:

```
import streamlit as st
import plotly.graph_objects as go
import plotly.express as px
from datetime import datetime, timedelta
import asyncio
from typing import Dict, List

class ContentPlatformUI:
    def __init__(self, platform: ContentPlatform):
        self.platform = platform
        self.setup_page()
```

```
def setup_page(self):
    """Configure the Streamlit page."""
    st.set_page_config(
        page_title="Content Generation & Moderation Platform",
        page_icon="📝",
        layout="wide"
    )
    st.title("Content Generation & Moderation Platform")

def run(self):
    """Run the main UI loop."""
    # Sidebar navigation
    page = st.sidebar.selectbox(
        "Select Page",
        ["Content Generation", "Moderation Dashboard", "Analytics"]
    )

    if page == "Content Generation":
        self.show_generation_page()
    elif page == "Moderation Dashboard":
        self.show_moderation_dashboard()
    else:
        self.show_analytics()

def show_generation_page(self):
    """Display content generation interface."""
    st.header("Content Generation")

    # Input form
    with st.form("generation_form"):
        prompt = st.text_area("Enter your prompt", height=100)

        col1, col2, col3 = st.columns(3)
        with col1:
            tone = st.selectbox("Tone",
                ["Professional", "Casual", "Academic", "Creative"])
        with col2:
            length = st.selectbox("Length",
                ["Short", "Medium", "Long"])
        with col3:
            audience = st.selectbox("Target Audience",
                ["General", "Technical", "Business", "Academic"])
```

```
        submit = st.form_submit_button("Generate Content")

    if submit and prompt:
        style = {
            "tone": tone.lower(),
            "length": length.lower(),
            "audience": audience.lower()
        }

        with st.spinner("Generating content..."):
            response = asyncio.run(
                self.platform.create_content(prompt, style)
            )

        # Display generated content
        st.subheader("Generated Content")
        st.write(response['content'])

        # Display moderation results
        st.subheader("Moderation Results")
        self.display_moderation_results(response['moderation'])

def display_moderation_results(self, results: Dict):
    """Display moderation results in a user-friendly format."""
    col1, col2, col3 = st.columns(3)

    with col1:
        self.create_gauge(
            "Safety Score",
            results['safety_score'],
            0, 1
        )

    with col2:
        self.create_gauge(
            "Toxicity",
            1 - results['toxicity']['score'],
            0, 1
        )

    with col3:
        self.create_gauge(
```

```
            "NSFW Score",
            1 - results['nsfw']['score'],
            0, 1
        )

    # Display bias analysis
    st.subheader("Bias Analysis")
    bias_data = results['bias']
    fig = px.bar(
        x=list(bias_data.keys()),
        y=list(bias_data.values()),
        title="Bias Detection Results"
    )
    st.plotly_chart(fig)

def create_gauge(self, title: str, value: float, min_val: float, max_val: float):
    """Create a gauge chart for metrics."""
    fig = go.Figure(go.Indicator(
        mode="gauge+number",
        value=value,
        title={'text': title},
        gauge={
            'axis': {'range': [min_val, max_val]},
            'steps': [
                {'range': [min_val, max_val/2], 'color': "red"},
                {'range': [max_val/2, max_val*0.8], 'color': "yellow"},
                {'range': [max_val*0.8, max_val], 'color': "green"}
            ],
            'bar': {'color': "darkblue"}
        }
    ))
    st.plotly_chart(fig)

def show_moderation_dashboard(self):
    """Display moderation dashboard."""
    st.header("Content Moderation Dashboard")

    # Add moderation dashboard components
    # Implementation depends on your specific needs

def show_analytics(self):
    """Display analytics dashboard."""
```

5. Healthcare Diagnosis Assistant Implementation Guide

Description: Create a system to assist in medical diagnosis using patient data
Incorporates: CNN, Supervised Learning, Ethics, Model Evaluation
Key features:

- Image classification for medical scans
- Patient data analysis using classical ML
- Explainable AI components
- Privacy-preserving data handling

Important Disclaimer
This system is designed for educational purposes and should be used as a study/learning tool only. It is NOT intended for actual medical diagnosis or to replace professional medical advice. All implementations should include clear disclaimers and require appropriate medical professional oversight.

Phase 1: Project Setup (Week 1)

1.1 Project Structure

```
healthcare_assistant/
├── src/
│   ├── data/
│   │   ├── processors/
│   │   └── validators/
│   ├── models/
│   │   ├── image/
│   │   ├── tabular/
│   │   └── ensemble/
│   ├── explainability/
│   ├── privacy/
│   ├── api/
│   └── utils/
├── tests/
├── config/
├── notebooks/
└── web_interface/
```

1.2 Environment Setup

```
```

requirements.txt:
torch>=2.0.0
torchvision>=0.15.0
scikit-learn>=1.3.0
pandas>=2.0.0
numpy>=1.24.0
lime>=0.2.0
shap>=0.41.0
streamlit>=1.25.0
pillow>=9.5.0
pydicom>=2.3.0
opencv-python>=4.7.0
fastapi>=0.100.0
pytest>=7.4.0
great-expectations>=0.15.0
```

**Phase 2: Data Processing Pipeline (Week 1-2)**

2.1 Data Handling

1. Medical image processing
2. Patient data standardization
3. Privacy preservation
4. Data validation

2.2 Privacy Implementation

1. Data anonymization
2. Encryption
3. Access control
4. Audit logging

**Phase 3: Model Development (Week 2)**

3.1 Image Analysis Model

1. CNN architecture
2. Transfer learning
3. Model validation
4. Performance metrics

3.2 Patient Data Analysis

1. Feature engineering
2. Model selection
3. Validation pipeline
```

4. Uncertainty quantification

3.3 Explainability System

1. LIME integration
2. SHAP analysis
3. Feature importance
4. Decision path visualization

Phase 4: Integration Systems (Week 3)

4.1 Model Ensemble

1. Voting system
2. Confidence scoring
3. Cross-validation
4. Error analysis

4.2 Decision Support

1. Risk assessment
2. Recommendation system
3. Confidence metrics
4. Warning systems

Phase 5: User Interface (Week 3-4)

5.1 Clinical Interface

1. Patient data entry
2. Image upload
3. Analysis results
4. Explanation display

5.2 Monitoring Dashboard

1. System metrics
2. Performance tracking
3. Usage statistics
4. Error monitoring

Evaluation Criteria

1. Technical Implementation (35%)
   - Model accuracy
   - Privacy protection
   - System reliability

- Code quality

2. Clinical Relevance (25%)
    - Medical accuracy
    - Practical utility
    - Safety measures
    - Ethical compliance

3. Explainability (20%)
    - Decision transparency
    - Result interpretation
    - Visualization quality
    - Documentation

4. User Experience (20%)
    - Interface design
- Workflow efficiency
- Error handling
- Response time

Bonus Features

1. Multiple imaging modalities
2. Temporal analysis
3. Report generation
4. Collaboration tools

Core implementation Code:

```
import torch
import torch.nn as nn
import torchvision.models as models
from typing import Dict, List, Tuple, Optional
import numpy as np
import pandas as pd
from datetime import datetime
import logging
import lime
import shap
from dataclasses import dataclass
from PIL import Image
import cv2
```

```
@dataclass
class PatientData:
    """Structure for patient data."""
    id: str
    age: int
    gender: str
    symptoms: List[str]
    medical_history: List[str]
    vitals: Dict[str, float]
    images: Optional[List[str]] = None

class ImageAnalyzer(nn.Module):
    def __init__(self, num_classes: int):
        super().__init__()

        # Load pre-trained model
        self.model = models.resnet50(pretrained=True)

        # Modify final layer
        num_features = self.model.fc.in_features
        self.model.fc = nn.Sequential(
            nn.Linear(num_features, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, num_classes)
        )

        # Add attention mechanism
        self.attention = nn.MultiheadAttention(512, 8)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Extract features
        features = self.model.conv1(x)
        features = self.model.bn1(features)
        features = self.model.relu(features)
        features = self.model.maxpool(features)

        features = self.model.layer1(features)
        features = self.model.layer2(features)
        features = self.model.layer3(features)
        features = self.model.layer4(features)
```

```
        # Apply attention
        features = features.flatten(2).permute(2, 0, 1)
        attended_features, _ = self.attention(features, features, features)

        # Global average pooling
        out = torch.mean(attended_features, dim=0)

        # Classification
        out = self.model.fc(out)
        return out

class PatientDataAnalyzer:
    def __init__(self):
        self.feature_processor = self._create_feature_processor()
        self.classifier = self._create_classifier()
        self.explainer = shap.TreeExplainer(self.classifier)

    def _create_feature_processor(self):
        """Create feature processing pipeline."""
        from sklearn.pipeline import Pipeline
        from sklearn.preprocessing import StandardScaler, OneHotEncoder
        from sklearn.compose import ColumnTransformer

        numeric_features = ['age'] + [f'vital_{i}' for i in range(5)]
        categorical_features = ['gender'] + [f'symptom_{i}' for i in range(10)]

        numeric_transformer = Pipeline(steps=[
            ('scaler', StandardScaler())
        ])

        categorical_transformer = Pipeline(steps=[
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ])

        preprocessor = ColumnTransformer(
            transformers=[
                ('num', numeric_transformer, numeric_features),
                ('cat', categorical_transformer, categorical_features)
            ])

        return preprocessor

    def _create_classifier(self):
```

```
        """Create main classifier."""
        from sklearn.ensemble import GradientBoostingClassifier
        return GradientBoostingClassifier(
            n_estimators=100,
            learning_rate=0.1,
            max_depth=5
        )

    def analyze_patient_data(self, data: PatientData) -> Dict:
        """Analyze patient data and return predictions with explanations."""
        # Process features
        features = self._prepare_features(data)

        # Make prediction
        prediction = self.classifier.predict_proba(features)

        # Generate explanation
        explanation = self._generate_explanation(features)

        return {
            'prediction': prediction,
            'explanation': explanation,
            'confidence': self._calculate_confidence(prediction)
        }

    def _generate_explanation(self, features) -> Dict:
        """Generate SHAP-based explanation."""
        shap_values = self.explainer.shap_values(features)

        return {
            'shap_values': shap_values,
            'feature_importance': self._get_feature_importance(shap_values)
        }

    def _calculate_confidence(self, prediction: np.ndarray) -> float:
        """Calculate confidence score for prediction."""
        return np.max(prediction)

class DiagnosisAssistant:
    def __init__(self):
        self.image_analyzer = ImageAnalyzer(num_classes=10)
        self.patient_analyzer = PatientDataAnalyzer()
        self.logger = logging.getLogger(__name__)
```

```
def analyze_case(self, patient_data: PatientData) -> Dict:
    """Analyze patient case and provide diagnosis assistance."""
    try:
        results = {
            'timestamp': datetime.now(),
            'patient_id': patient_data.id
        }

        # Analyze patient data
        patient_analysis = self.patient_analyzer.analyze_patient_data(
            patient_data
        )
        results['patient_analysis'] = patient_analysis

        # Analyze images if available
        if patient_data.images:
            image_analysis = self._analyze_images(patient_data.images)
            results['image_analysis'] = image_analysis

        # Combine analyses
        combined_results = self._combine_analyses(results)
        results['combined_analysis'] = combined_results

        # Add confidence metrics
        results['confidence_metrics'] = self._calculate_confidence_metrics(
            results
        )

        # Add recommendations
        results['recommendations'] = self._generate_recommendations(
            combined_results
        )

        return results

    except Exception as e:
        self.logger.error(f"Error in case analysis: {str(e)}")
        raise

def _analyze_images(self, image_paths: List[str]) -> Dict:
    """Analyze medical images."""
    results = []
```

```
        for path in image_paths:
            # Load and preprocess image
            image = self._preprocess_image(path)

            # Get prediction
            with torch.no_grad():
                prediction = self.image_analyzer(image)

            # Generate explanation
            explanation = self._generate_image_explanation(image)

            results.append({
                'path': path,
                'prediction': prediction,
                'explanation': explanation
            })

        return results

    def _combine_analyses(self, results: Dict) -> Dict:
        """Combine different analyses into final results."""
        # Implement ensemble logic
        pass

    def _calculate_confidence_metrics(self, results: Dict) -> Dict:
        """Calculate confidence metrics for the analysis."""
        # Implement confidence calculation
        pass

    def _generate_recommendations(self, analysis: Dict) -> List[str]:
        """Generate recommendations based on analysis."""
        # Implement recommendation logic
        pass

# Example usage
if __name__ == "__main__":
    # Initialize assistant
    assistant = DiagnosisAssistant()

    # Example patient data
    patient = PatientData(
        id="P12345",
```

```
        age=45,
        gender="F",
        symptoms=["headache", "fever"],
        medical_history=["hypertension"],
        vitals={"blood_pressure": 130/85, "heart_rate": 75}
    )

    # Analyze case
    results = assistant.analyze_case(patient)
    print(f"Analysis completed for patient {results['patient_id']}")
```

Streamlit Interface implementation:

```
import streamlit as st
import plotly.graph_objects as go
import plotly.express as px
from datetime import datetime
import pandas as pd
import numpy as np
from typing import Dict, List

class HealthcareAssistantUI:
    def __init__(self, assistant: DiagnosisAssistant):
        self.assistant = assistant
        self.setup_page()

    def setup_page(self):
        """Configure the Streamlit page."""
        st.set_page_config(
            page_title="Healthcare Diagnosis Assistant",
            page_icon="🏥",
            layout="wide"
        )
        st.title("Healthcare Diagnosis Assistant")

        # Add disclaimer
        st.warning(
            "⚠️ EDUCATIONAL TOOL ONLY: This system is for educational and research
"
            "purposes only. It is not intended for actual medical diagnosis. Always "
            "consult with qualified healthcare professionals for medical advice."
        )
```

```
def run(self):
    """Run the main UI loop."""
    # Sidebar navigation
    page = st.sidebar.selectbox(
        "Select Page",
        ["Patient Analysis", "Image Analysis", "Analytics Dashboard"]
    )

    if page == "Patient Analysis":
        self.show_patient_analysis()
    elif page == "Image Analysis":
        self.show_image_analysis()
    else:
        self.show_analytics()

def show_patient_analysis(self):
    """Display patient analysis interface."""
    st.header("Patient Analysis")

    # Patient data form
    with st.form("patient_form"):
        col1, col2 = st.columns(2)

        with col1:
            patient_id = st.text_input("Patient ID")
            age = st.number_input("Age", min_value=0, max_value=120)
            gender = st.selectbox("Gender", ["M", "F", "Other"])

        with col2:
            # Vital signs
            st.subheader("Vital Signs")
            bp_sys = st.number_input("Systolic BP", min_value=0, max_value=300)
            bp_dia = st.number_input("Diastolic BP", min_value=0, max_value=200)
            heart_rate = st.number_input("Heart Rate", min_value=0, max_value=250)

        # Symptoms and medical history
        symptoms = st.multiselect(
            "Symptoms",
            ["Fever", "Cough", "Fatigue", "Pain", "Nausea", "Headache"]
        )

        medical_history = st.multiselect(
```

```
            "Medical History",
            ["Hypertension", "Diabetes", "Asthma", "Heart Disease"]
        )

        # Image upload
        uploaded_files = st.file_uploader(
            "Upload Medical Images",
            accept_multiple_files=True,
            type=['png', 'jpg', 'jpeg']
        )

        submit = st.form_submit_button("Analyze")

    if submit:
        # Create patient data object
        patient = PatientData(
            id=patient_id,
            age=age,
            gender=gender,
            symptoms=symptoms,
            medical_history=medical_history,
            vitals={
                "blood_pressure": f"{bp_sys}/{bp_dia}",
                "heart_rate": heart_rate
            },
            images=[f.name for f in uploaded_files] if uploaded_files else None
        )

        # Perform analysis
        with st.spinner("Analyzing patient data..."):
            results = self.assistant.analyze_case(patient)

        # Display results
        self.display_analysis_results(results)

def display_analysis_results(self, results: Dict):
    """Display analysis results."""
    st.subheader("Analysis Results")

    # Create tabs for different aspects of results
    tabs = st.tabs([
        "Summary",
        "Patient Analysis",
```

```
    "Image Analysis",
    "Recommendations"
])

with tabs[0]:
    self.display_summary(results)

with tabs[1]:
    self.display_patient_analysis(results['patient_analysis'])

with tabs[2]:
    if 'image_analysis' in results:
        self.display_image_analysis(results['image_analysis'])
    else:
        st.info("No images were provided for analysis")

with
```