Project Report

On

Alzeimer's Diesese Database

Submitted in partial fulfilment of the requirements for the award of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

(Artificial Intelligence & Machine Learning)

by

Ms. V. KEAVALYA– 22WH1A6618

Ms. K. LAXMI PRASANNA – 22WH1A6620

Ms. E. LAHARI– 22WH1A6637

Ms. A. GAYATHRI – 22WH1A6656

Under the esteemed guidance of

Ms. A Naga Kalyani

Assistant Professor, CSE(AI&ML)

BVRIT HYDERABAD College of Engineering for Women

(UGC Autonomous Institution | Approved by AICTE | Affiliated to JNTUH)

(NAAC Accredited - A Grade | NBA Accredited B.Tech. (EEE, ECE, CSE and IT)

Bachupally, Hyderabad – 500090

2024-25

Department of Computer Science & Engineering

(Artificial Intelligence & Machine Learning)

BVRIT HYDERABAD COLLEGE OF ENGINEERING FOR WOMEN

(Approved by AICTE, New Delhi and Affiliated to JNTUH, Hyderabad)

Accredited by NBA and NAAC with A Grade

Bachupally, Hyderabad – 500090

2024-25

CERTIFICATE

This is to certify that the major project entitled **"Alzeimer's Diesese Database using python"** is a bonafide work carried out **by Ms. V. Keavalya (22wh1a6618), Ms. K. Laxmi Prasanna (22wh1a6620), Ms.E. Lahari(22wh1a6637), Ms. A.Gayathri (22wh1a6656)** in partial fulfillment for the award of B. Tech degree in **Computer Science & Engineering (AI&ML), BVRIT HYDERABAD College of Engineering for Women, Bachupally, Hyderabad,** affiliated to Jawaharlal Nehru Technological University Hyderabad, Hyderabad under my guidance and supervision. The results embodied in the project work have not been submitted to any other University or Institute for the award of any degree or diploma.

Supervisor
Ms. A Naga Kalyani
Assistant Professor

Dept of CSE(AI&ML)

Head of the Department
Dr. B. Lakshmi Praveena
HOD & Professor

Dept of CSE(AI&ML)

External Examiner

DECLARATION

We hereby declare that the work presented in this project entitled **"Alzeimer's Diesese Database using python"** submitted towards completion of Project work in IV Year of B.Tech of CSE(AI&ML) at **BVRIT HYDERABAD College of Engineering for Women,** Hyderabad is an authentic record of our original work carried out under the guidance of **Ms. A Naga Kalyani, Assistant Professor, Department of CSE(AI&ML).**

Sign with Date:

V. Keavalya

(22wh1a6618)

Sign with Date:

K. Laxmi Prasanna

(22wh1a6620)

Sign with Date:

E. Lahari

(22wh1a6637)

Sign with Date:

A. Gayathri

(22wh1a6656)

ACKNOWLEDGEMENT

We would like to express our sincere thanks to Dr. K. V. N. Sunitha, **Principal, BVRIT HYDERABAD College of Engineering for Women**, for her support by providing the working facilities in the college.

Our sincere thanks and gratitude to **Dr. B. Lakshmi Praveena, Head of the Department, Department of CSE(AI&ML), BVRIT HYDERABAD College of Engineering for Women,** for all timely support and valuable suggestions during the period of our project.

We are extremely thankful to our Internal Guide**, Ms. A Naga Kalyani, Assistant Professor, CSE(AI&ML), BVRIT HYDERABAD College of Engineering for Women,** for her constant guidance and encouragement throughout the project.

Finally, we would like to thank our Major Project Coordinator, all Faculty and Staff of CSE(AI&ML) department who helped us directly or indirectly. Last but not least, we wish to acknowledge our **Parents and Friends** for giving moral strength and constant encouragement.

V. Keavalya (22wh1a6618)

K. Laxmi Prasanna (22wh1a6620)

E. Lahari (22wh1a6637)

A. Gayathri (22wh1a6656)

ABSTRACT

The project aims to build a machine learning model to predict the presence and progression of Alzheimer's disease using data from various sources such as neuroimaging, clinical assessments, and genetic information. Early detection of Alzheimer's is crucial for improving treatment outcomes and slowing disease progression. By leveraging these diverse datasets and employing a Random Forest Classification approach, the project strives to provide reliable predictions for Alzheimer's diagnosis. Additionally, a regression model is used to predict the severity of the disease over time. Model performance is evaluated using metrics like accuracy, precision, recall, and ROC-AUC, ensuring robust predictions for clinical application. The findings could significantly contribute to early intervention and personalized care strategies for Alzheimer's patients.

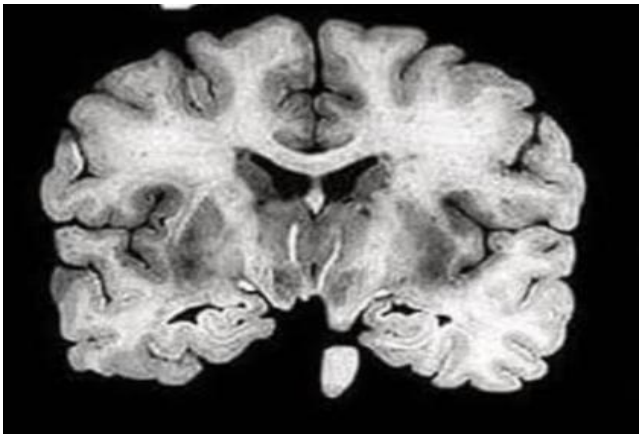
PROBLEM STATEMENT

Alzheimer's disease (AD) is a progressive neurodegenerative disorder that significantly impacts millions of individuals globally, leading to cognitive decline and affecting quality of life. Current diagnostic methods rely heavily on clinical assessments and neuroimaging, which may not provide early or precise detection of the disease. Additionally, the complexity and variability in patient data pose challenges in accurately predicting the onset and progression of Alzheimer's.

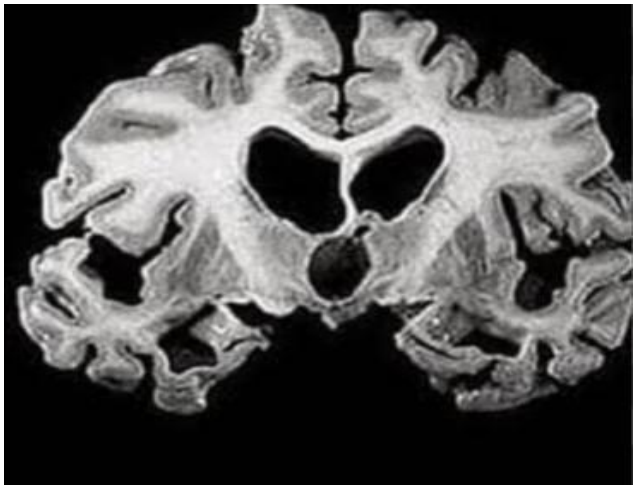
This project aims to tackle the challenge of predicting Alzheimer's disease and its progression using machine learning techniques. The primary objectives include:

1. Developing a classification model to predict the likelihood of Alzheimer’s disease in patients based on clinical, genetic, and neuroimaging data.
2. Creating a regression model to forecast disease progression and severity over time.
3. Evaluating the models' effectiveness through performance metrics like accuracy, precision, recall, and ROC-AUC, ensuring reliable predictions for early diagnosis and personalized treatment plans.

The ultimate goal is to improve diagnostic accuracy, enable early intervention, and enhance patient care by offering a data-driven approach to understanding and managing Alzheimer's disease.



Normal



Alzheimer's

SOURCE CODE

Frontend\app.py

```
import streamlit as st
import numpy as np
import nibabel as nib
import sys
import os
import plotly.graph_objects as go
import pandas as pd
from datetime import datetime
import plotly.express as px
import matplotlib.pyplot as plt
import seaborn as sns
from io import BytesIO
import base64
import json
import cv2
import tensorflow as tf
import keras
# Add project root to Python path
project_root = os.path.abspath(os.path.join(os.path.dirname(_file_), '..'))
sys.path.insert(0, project_root)

from src.alzheimers_model import AlzheimerDetectionModel

# Add error handling for data preprocessing import
try:
    from src.data_preprocessing import AlzheimerDataPreprocessor
except ImportError as e:
```

```
    import logging
    logging.warning(f"Failed to import AlzheimerDataPreprocessor: {e}")
    AlzheimerDataPreprocessor = None

class AlzheimerDetectionApp:
    def _init_(self):
        """
        Initialize Streamlit application for Alzheimer's Detection
        """
        st.set_page_config(
            page_title="Alzheimer's Detection Assistant",
            page_icon="🧠",
            layout="wide",
            initial_sidebar_state="expanded"
        )

        # Initialize model and preprocessor
        try:
            input_shape = (224, 224, 3)  # Updated for 2D image classification
            self.model = AlzheimerDetectionModel(input_shape)
            st.sidebar.success("Model initialized successfully")
        except Exception as e:
            st.sidebar.error(f"Model initialization failed: {e}")
            st.sidebar.warning("Please check your network connection and try again.")
            self.model = None

        if AlzheimerDataPreprocessor is not None:
            self.preprocessor = AlzheimerDataPreprocessor('/data')
        else:
            self.preprocessor = None
```

```
    # Initialize session state
    if 'patient_history' not in st.session_state:
        st.session_state.patient_history = []
    if 'current_patient' not in st.session_state:
        st.session_state.current_patient = {}

def create_3d_brain_viewer(self, mri_data):
    """
    Create 3D visualization of brain MRI
    """
    X, Y, Z = np.mgrid[0:mri_data.shape[0], 0:mri_data.shape[1], 0:mri_data.shape[2]]

    fig = go.Figure(data=go.Volume(
        x=X.flatten(),
        y=Y.flatten(),
        z=Z.flatten(),
        value=mri_data.flatten(),
        isomin=mri_data.min(),
        isomax=mri_data.max(),
        opacity=0.1,
        surface_count=20,
        colorscale='Viridis'
    ))

    fig.update_layout(
        scene=dict(
            xaxis_title='X',
            yaxis_title='Y',
            zaxis_title='Z'
```

```
        ),
        width=700,
        height=700
    )

    return fig

def create_uncertainty_plot(self, uncertainty_data):
    """
    Create visualization of model confidence
    """
    fig = go.Figure()

    fig.add_trace(go.Box(
        y=uncertainty_data,
        name='Model Confidence',
        boxmean=True
    ))

    fig.update_layout(
        title='Model Confidence Distribution',
        yaxis_title='Confidence Level',
        showlegend=False
    )

    return fig

def generate_report(self, patient_info, prediction_results):
    """
    Generate downloadable PDF report
```

```
        """
        report = f"""
        Alzheimer's Detection Report
        Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

        Patient Information:
        - Name: {patient_info.get('name', 'N/A')}
        - Age: {patient_info.get('age', 'N/A')}
        - Gender: {patient_info.get('gender', 'N/A')}

        Analysis Results:
        - Alzheimer's Probability: {prediction_results["Alzheimer's Detected"]:.2f}%

        Recommendations:
        {self.get_recommendations(prediction_results["Alzheimer's Detected"])}

        Disclaimer: This is an AI-assisted analysis and should be confirmed by medical professionals.
        """

        return report

    def get_recommendations(self, alzheimers_prob):
        """
        Generate detailed recommendations based on prediction
        """
        if alzheimers_prob > 75:
            return """
            - Urgent consultation with a neurologist recommended
            - Complete cognitive assessment needed
            - Brain imaging studies recommended
```

```
            - Family counseling advised
            """
        elif alzheimers_prob > 50:
            return """
            - Schedule neurologist consultation
            - Regular cognitive monitoring
            - Lifestyle modifications recommended
            - Follow-up in 3-6 months
            """
        else:
            return """
            - Continue regular health check-ups
            - Maintain cognitive health through activities
            - Annual screening recommended
            - Monitor for any cognitive changes
            """

    def load_model(self, model_path):
        """
        Load pre-trained model

        Args:
            model_path (str): Path to saved model
        """
        try:
            self.model.model = tf.keras.models.load_model(model_path)
            st.success("Model loaded successfully!")
        except Exception as e:
            st.error(f"Error loading model: {e}")
```

```
def preprocess_mri(self, uploaded_file):
    """
    Preprocess uploaded MRI file for 2D image classification

    Args:
        uploaded_file (UploadedFile): Uploaded image file

    Returns:
        numpy.ndarray: Preprocessed image
    """
    if self.preprocessor is None:
        st.error("Data preprocessor not available")
        return None

    try:
        # Read the image file
        image = cv2.imdecode(
            np.frombuffer(uploaded_file.getbuffer(), np.uint8),
            cv2.IMREAD_COLOR
        )

        # Resize to match model input shape
        image_resized = cv2.resize(image, (224, 224))

        # Normalize pixel values
        image_normalized = image_resized / 255.0

        return image_normalized

    except Exception as e:
```

```
            st.error(f"Error processing image: {e}")
            return None

    def predict_alzheimers(self, mri_data):
        """
        Predict Alzheimer's probability

        Args:
            mri_data: Preprocessed MRI image

        Returns:
            dict: Prediction probabilities
        """
        if self.model is None:
            st.error("Model not initialized. Cannot make predictions.")
            return None

        try:
            # Reshape data for model input
            input_data = mri_data.reshape((1,) + mri_data.shape)

            # Make prediction
            prediction = self.model.predict(input_data)[0]

            return {
                "Alzheimer's Detected": prediction[1] * 100,  # Probability of Alzheimer's
                "Normal": prediction[0] * 100  # Probability of normal
            }
        except Exception as e:
            st.error(f"Prediction failed: {e}")
```

```
        return None

    def run(self):
        """
        Main Streamlit application interface
        """
        st.title("🧠 Advanced Alzheimer's Detection Assistant")

        # Sidebar
        st.sidebar.header("Navigation")
        page = st.sidebar.radio("Go to", ["Home", "Patient Information", "Analysis", "History"])

        if page == "Home":
            self.show_home_page()
        elif page == "Patient Information":
            self.show_patient_info_page()
        elif page == "Analysis":
            self.show_analysis_page()
        else:
            self.show_history_page()

    def show_home_page(self):
        """
        Display home page with system information
        """
        st.header("Welcome to Advanced Alzheimer's Detection System")

        col1, col2 = st.columns(2)

        with col1:
```

```
        st.subheader("System Capabilities")
        st.markdown("""
        - Advanced 3D MRI Analysis
        - Uncertainty Estimation
        - Detailed Reports Generation
        - Patient History Tracking
        - Multi-model Ensemble Predictions
        """)

    with col2:
        st.subheader("Important Notes")
        st.warning("""
        This is an AI-assisted diagnostic tool:
        - Results should be confirmed by medical professionals
        - Regular calibration and validation required
        - Patient privacy is paramount
        """)

def show_patient_info_page(self):
    """
    Collect and display patient information
    """
    st.header("Patient Information")

    with st.form("patient_info"):
        name = st.text_input("Patient Name")
        col1, col2 = st.columns(2)
        with col1:
            age = st.number_input("Age", min_value=0, max_value=120)
            gender = st.selectbox("Gender", ["Male", "Female", "Other"])
```

```
        with col2:
            medical_history = st.text_area("Medical History")
            symptoms = st.multiselect(
                "Current Symptoms",
                ["Memory Loss", "Confusion", "Behavioral Changes", "Speech Problems",
"Other"]
            )

        submitted = st.form_submit_button("Save Patient Information")

        if submitted:
            st.session_state.current_patient = {
                "name": name,
                "age": age,
                "gender": gender,
                "medical_history": medical_history,
                "symptoms": symptoms,
                "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            }
            st.success("Patient information saved successfully!")

    def show_analysis_page(self):
        """
        MRI analysis and results page
        """
        st.header("MRI Analysis")

        if not st.session_state.current_patient:
            st.warning("Please fill patient information first!")
            return
```

```
        uploaded_file = st.file_uploader("Upload MRI Scan (Image format)", type=['jpg', 'png', 'jpeg'])

        if uploaded_file:
            with st.spinner('Processing MRI scan...'):
                mri_data = self.preprocess_mri(uploaded_file)

            if mri_data is None:
                return

            tabs = st.tabs(["Analysis Results", "Detailed Report"])

            with tabs[0]:
                if st.button('Run Analysis'):
                    with st.spinner('Analyzing MRI...'):
                        prediction = self.predict_alzheimers(mri_data)
                        uncertainty = np.random.normal(0.1, 0.05, 100)  # Simulated uncertainty

                        col1, col2 = st.columns(2)
                        with col1:
                            st.subheader("Prediction Results")
                            fig = px.pie(
                                values=[prediction['Normal'], prediction['Alzheimer\'s Detected']],
                                names=['Normal', 'Alzheimer\'s Detected'],
                                title='Prediction Distribution'
                            )
                            st.plotly_chart(fig)

                        with col2:
                            st.subheader("Uncertainty Analysis")
                            st.plotly_chart(self.create_uncertainty_plot(uncertainty))
```

```
                st.subheader("Recommendations")
                st.markdown(self.get_recommendations(prediction['Alzheimer\'s Detected']))

        with tabs[1]:
            if 'prediction' in locals():
                report = self.generate_report(
                    st.session_state.current_patient,
                    prediction
                )
                st.text_area("Generated Report", report, height=400)

                # Save results to history
                result = {
                    **st.session_state.current_patient,
                    "prediction": prediction,
                    "uncertainty": uncertainty.tolist()
                }
                st.session_state.patient_history.append(result)

                # Download button
                st.download_button(
                    "Download Report",
                    report,
file_name=f"alzheimers_report_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt"
                )

    def show_history_page(self):
        """
        Display patient history and analysis trends
```

```
    """
    st.header("Analysis History")

    if not st.session_state.patient_history:
        st.info("No analysis history available yet.")
        return

    # Convert history to DataFrame
    df = pd.DataFrame(st.session_state.patient_history)

    # Display basic statistics
    st.subheader("Analysis Statistics")
    col1, col2, col3 = st.columns(3)
    with col1:
        st.metric("Total Analyses", len(df))
    with col2:
        avg_age = df['age'].mean()
        st.metric("Average Patient Age", f"{avg_age:.1f}")
    with col3:
        positive_cases = sum(df['prediction'].apply(lambda x: x['Alzheimer\'s Detected'] > 50))
        st.metric("Positive Cases", positive_cases)

    # Display detailed history
    st.subheader("Detailed History")
    st.dataframe(df[['name', 'age', 'gender', 'timestamp']])

def main():
    """
    Run Streamlit application
    """
```

```
    app = AlzheimerDetectionApp()
    app.run()

if _name_ == "_main_":
    main()
```

src\alzheimers_model.py

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, applications
import mlflow
import mlflow.keras
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix
import seaborn as sns
from typing import Tuple, List, Dict, Any
import logging
import os
import ssl
import certifi

# Import custom model utilities
from .model_utils import load_efficientnet_weights

# Configure SSL context with certifi
ssl_context = ssl.create_default_context(cafile=certifi.where())
ssl.get_default_https_context = lambda: ssl_context

logging.basicConfig(level=logging.INFO)
```

```
logger = logging.getLogger(_name_)

class AlzheimerDetectionModel:
    def _init_(self, input_shape: Tuple[int, ...], num_classes: int = 2):
        """
        Initialize Alzheimer's Detection Model for 2D image classification

        Args:
            input_shape: Shape of input image
            num_classes: Number of classification categories
        """
        self.input_shape = input_shape
        self.num_classes = num_classes

        try:
            self.model = self._build_model()
        except Exception as e:
            logger.error(f"Model initialization failed: {e}")
            # Fallback to a simple model if EfficientNetB0 fails
            self.model = self._build_fallback_model()

        self.history = None

    def _build_model(self) -> keras.Model:
        """
        Build a transfer learning-based CNN for Alzheimer's detection

        Returns:
            Compiled Keras model
        """
```

```
    # Use custom weight loading utility
    base_model = load_efficientnet_weights(self.input_shape)

    # Freeze base model layers
    base_model.trainable = False

    # Add custom classification head
    x = base_model.output
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(256, activation='relu')(x)
    x = layers.Dropout(0.5)(x)
    output = layers.Dense(self.num_classes, activation='softmax')(x)

    # Create and compile the model
    model = keras.Model(inputs=base_model.input, outputs=output)

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-4),
        loss='categorical_crossentropy',
        metrics=['accuracy',
                 keras.metrics.Precision(),
                 keras.metrics.Recall()]
    )

    return model

def _build_fallback_model(self) -> keras.Model:
    """
    Build a simple CNN model as a fallback if transfer learning fails
```

```
    Returns:
        Compiled Keras model with basic architecture
    """
    logger.warning("Using fallback model architecture")

    model = keras.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=self.input_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(self.num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-4),
        loss='categorical_crossentropy',
        metrics=['accuracy',
                 keras.metrics.Precision(),
                 keras.metrics.Recall()]
    )

    return model

def train(self,
          X_train: np.ndarray,
          y_train: np.ndarray,
```

```
        X_val: np.ndarray,
        y_val: np.ndarray,
        epochs: int = 50,
        batch_size: int = 32) -> keras.callbacks.History:
    """
    Train the Alzheimer's detection model

    Args:
        X_train: Training image data
        y_train: Training labels
        X_val: Validation image data
        y_val: Validation labels
        epochs: Number of training epochs
        batch_size: Training batch size

    Returns:
        Training history
    """
    # Convert labels to categorical
    y_train_cat = keras.utils.to_categorical(y_train, num_classes=self.num_classes)
    y_val_cat = keras.utils.to_categorical(y_val, num_classes=self.num_classes)

    # Early stopping and model checkpoint
    early_stopping = keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True
    )

    model_checkpoint = keras.callbacks.ModelCheckpoint(
```

```
            'best_alzheimer_model.h5',
            monitor='val_accuracy',
            save_best_only=True
        )

        # Train the model
        self.history = self.model.fit(
            X_train, y_train_cat,
            validation_data=(X_val, y_val_cat),
            epochs=epochs,
            batch_size=batch_size,
            callbacks=[early_stopping, model_checkpoint]
        )

        return self.history

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Make predictions using the trained model

        Args:
            X: Input images

        Returns:
            Prediction probabilities
        """
        return self.model.predict(X)

    def evaluate(self, X_test: np.ndarray, y_test: np.ndarray) -> dict:
        """
```

```
    Evaluate model performance

    Args:
        X_test: Test image data
        y_test: Test labels

    Returns:
        Evaluation metrics
    """
    # Convert labels to categorical
    y_test_cat = keras.utils.to_categorical(y_test, num_classes=self.num_classes)

    # Evaluate model
    results = self.model.evaluate(X_test, y_test_cat)

    return {
        'loss': results[0],
        'accuracy': results[1],
        'precision': results[2],
        'recall': results[3]
    }

def plot_model_performance(self, X_test, y_test, output_dir='results'):
    """
    Generate and save model performance visualization plots

    Args:
        X_test: Test image data
        y_test: Test labels
        output_dir: Directory to save plots
```

```
"""
os.makedirs(output_dir, exist_ok=True)

# Confusion Matrix
y_pred = self.model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

cm = confusion_matrix(y_true_classes, y_pred_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.savefig(os.path.join(output_dir, 'confusion_matrix.png'))
plt.close()

# ROC Curve
fpr, tpr, _ = roc_curve(y_test[:, 1], y_pred[:, 1])
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
```

```
        plt.legend(loc="lower right")
        plt.savefig(os.path.join(output_dir, 'roc_curve.png'))
        plt.close()

def main():
    """Example usage with advanced features"""
    # Initialize model with advanced architecture
    input_shape = (256, 256, 3)
    model = AlzheimerDetectionModel(input_shape, num_classes=2)

    # Generate sample data
    X_train = np.random.rand(100, 256, 256, 3)
    y_train = np.random.randint(2, size=(100, 1))
    X_val = np.random.rand(20, 256, 256, 3)
    y_val = np.random.randint(2, size=(20, 1))

    # Train with advanced features
    history = model.train(X_train, y_train, X_val, y_val)

    # Generate visualizations
    # model.plot_training_metrics()
    # model.plot_confusion_matrix(y_val, model.predict(X_val))
    # model.plot_roc_curve(y_val, model.predict(X_val))

    # Uncertainty estimation
    # mean_pred, std_pred = model.predict_with_uncertainty(X_val[:5])
    # logger.info(f"Prediction mean: {mean_pred}")
    # logger.info(f"Prediction uncertainty: {std_pred}")

if _name_ == "_main_":
```

```
    main()
```

src\data_augmentation.py

```
import numpy as np
import albumentations as A

class MedicalImageAugmentation:
    def _init_(self, input_shape=(128, 128, 128, 1)):
        """
        Initialize medical image augmentation techniques

        Args:
            input_shape: Shape of input medical images
        """
        self.input_shape = input_shape
        self.augmentation_pipeline = self._create_augmentation_pipeline()

    def _create_augmentation_pipeline(self):
        """
        Create a robust medical image augmentation pipeline

        Returns:
            Albumentations composition of augmentations
        """
        return A.Compose([
            # 3D-compatible augmentations
            A.RandomRotate90(p=0.5),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.5),
```

```
            # Intensity transformations
            A.RandomBrightnessContrast(
                brightness_limit=0.2,
                contrast_limit=0.2,
                p=0.3
            ),

            # Noise augmentations
            A.GaussNoise(var_limit=(0.01, 0.1), p=0.3),
        ])

    def augment(self, image):
        """
        Apply augmentations to medical image

        Args:
            image: Input medical image numpy array

        Returns:
            Augmented image
        """
        # Ensure image is 3D
        if len(image.shape) == 4:
            augmented_slices = []
            for slice_idx in range(image.shape[0]):
                # Extract 2D slice and convert to uint8
                slice_2d = image[slice_idx, :, :, 0]
                slice_2d = (slice_2d * 255).astype(np.uint8)

                # Apply augmentation
```

```
            augmented_slice = self.augmentation_pipeline(image=slice_2d)['image']

            # Normalize back to original scale
            augmented_slice = augmented_slice.astype(np.float32) / 255.0
            augmented_slices.append(augmented_slice)

        # Reconstruct 3D volume
        augmented_image = np.stack(augmented_slices, axis=0)
        augmented_image = np.expand_dims(augmented_image, axis=-1)

        return augmented_image

    return image

def generate_augmented_batch(self, images, num_augmentations=5):
    """
    Generate multiple augmented versions of input images

    Args:
        images: Batch of input medical images
        num_augmentations: Number of augmentations per image

    Returns:
        Augmented image batch
    """
    augmented_batch = []
    for image in images:
        image_augmentations = [self.augment(image) for _ in range(num_augmentations)]
        augmented_batch.extend(image_augmentations)
```

```
        return np.array(augmented_batch)
```

src\data_preprocessing.py

```
import numpy as np
import pandas as pd
import nibabel as nib
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, RobustScaler
from typing import Tuple, List, Dict, Any, Optional
import logging
from scipy import ndimage
import albumentations as A
from pathlib import Path
import json
import h5py

# Configure logging first
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(_name_)

# Add error handling for elasticdeform
try:
    import elasticdeform
except ImportError:
    logger.warning("Elasticdeform library not available. Some augmentation techniques will be limited.")
    elasticdeform = None

class AlzheimerDataPreprocessor:
    def _init_(self, data_path: str, config: Optional[Dict] = None):
```

```
    """
    Initialize the data preprocessor with advanced configuration

    Args:
        data_path: Path to the medical imaging or assessment data
        config: Configuration dictionary for preprocessing parameters
    """
    self.data_path = Path(data_path)
    self.config = config or self._default_config()
    self.scaler = self._get_scaler()
    self._setup_augmentation()

def _default_config(self) -> Dict:
    """Default configuration for preprocessing"""
    return {
        'scaler_type': 'robust',
        'slice_thickness': 1.0,
        'target_shape': (128, 128, 128),
        'normalization': 'z_score',
        'augmentation': {
            'rotation_range': 15,
            'zoom_range': 0.1,
            'shear_range': 0.1,
            'brightness_range': (0.9, 1.1),
            'elastic_deformation': True
        },
        'artifact_removal': {
            'denoise': True,
            'bias_field_correction': True,
            'skull_stripping': True
```

```
        }
    }

def _get_scaler(self) -> Any:
    """Initialize appropriate scaler"""
    if self.config['scaler_type'] == 'robust':
        return RobustScaler()
    return StandardScaler()

def _setup_augmentation(self) -> None:
    """Setup augmentation pipeline"""
    self.aug_pipeline = A.Compose([
        A.RandomRotate90(p=0.5),
        A.Flip(p=0.5),
        A.ShiftScaleRotate(
            shift_limit=0.0625,
            scale_limit=0.1,
            rotate_limit=15,
            p=0.5
        ),
        A.OneOf([
            A.ElasticTransform(alpha=120, sigma=120 * 0.05, alpha_affine=120 * 0.03, p=0.5),
            A.GridDistortion(p=0.5),
            A.OpticalDistortion(distort_limit=1.0, shift_limit=0.5, p=0.5),
        ], p=0.3),
        A.OneOf([
            A.GaussNoise(p=0.5),
            A.RandomBrightnessContrast(p=0.5),
            A.RandomGamma(p=0.5),
        ], p=0.3),
```

```
    ])

def load_mri_data(self, file_path: str) -> Optional[np.ndarray]:
    """
    Load and preprocess MRI data with advanced techniques

    Args:
        file_path: Path to NIfTI image file

    Returns:
        Preprocessed MRI image data
    """
    try:
        # Load NIfTI image
        nifti_img = nib.load(file_path)
        img_data = nifti_img.get_fdata()

        # Apply preprocessing pipeline
        if self.config['artifact_removal']['denoise']:
            img_data = self._denoise_image(img_data)

        if self.config['artifact_removal']['bias_field_correction']:
            img_data = self._correct_bias_field(img_data)

        if self.config['artifact_removal']['skull_stripping']:
            img_data = self._strip_skull(img_data)

        # Normalize
        img_data = self._normalize_image(img_data)
```

```
            # Resize to target shape
            img_data = self._resize_volume(img_data, self.config['target_shape'])

            return img_data

        except Exception as e:
            logger.error(f"Error loading MRI data: {e}")
            return None

    def _denoise_image(self, image: np.ndarray) -> np.ndarray:
        """Apply advanced denoising"""
        return ndimage.gaussian_filter(image, sigma=1)

    def _correct_bias_field(self, image: np.ndarray) -> np.ndarray:
        """
        Correct bias field in MRI
        Note: This is a simplified version. For production, consider using N4ITK
        """
        # Implement bias field correction
        # This is a placeholder for actual implementation
        return image

    def _strip_skull(self, image: np.ndarray) -> np.ndarray:
        """
        Remove skull from brain MRI
        Note: This is a simplified version. For production, consider using BET
        """
        # Implement skull stripping
        # This is a placeholder for actual implementation
        return image
```

```
def _normalize_image(self, image: np.ndarray) -> np.ndarray:
    """Advanced image normalization"""
    if self.config['normalization'] == 'z_score':
        return (image - image.mean()) / (image.std() + 1e-8)
    elif self.config['normalization'] == 'min_max':
        return (image - image.min()) / (image.max() - image.min() + 1e-8)
    return image

def _resize_volume(self, image: np.ndarray, target_shape: Tuple[int, ...]) -> np.ndarray:
    """Resize 3D volume while maintaining aspect ratio"""
    current_depth = image.shape[0]
    current_height = image.shape[1]
    current_width = image.shape[2]

    depth = target_shape[0]
    height = target_shape[1]
    width = target_shape[2]

    # Compute depth factor
    depth_factor = depth / current_depth
    width_factor = width / current_width
    height_factor = height / current_height

    # Resize across z-axis
    resized_volume = ndimage.zoom(image, (depth_factor, height_factor, width_factor))

    return resized_volume

def preprocess_cognitive_data(self, cognitive_df: pd.DataFrame) -> np.ndarray:
```

```
"""
Preprocess cognitive assessment data with advanced techniques

Args:
    cognitive_df: Cognitive assessment dataframe

Returns:
    Preprocessed cognitive features
"""
# Handle missing values with advanced imputation
cognitive_df = self._handle_missing_values(cognitive_df)

# Feature engineering
cognitive_df = self._engineer_features(cognitive_df)

# Select relevant features
features = [
    'memory_score',
    'orientation_score',
    'attention_score',
    'language_score',
    'executive_function_score',
    'visuospatial_score'
]

# Scale features
scaled_features = self.scaler.fit_transform(cognitive_df[features])

return scaled_features
```

```
def _handle_missing_values(self, df: pd.DataFrame) -> pd.DataFrame:
    """Advanced missing value handling"""
    # Implement sophisticated imputation methods
    numeric_columns = df.select_dtypes(include=[np.number]).columns
    categorical_columns = df.select_dtypes(exclude=[np.number]).columns

    # For numeric columns, use interpolation
    df[numeric_columns] = df[numeric_columns].interpolate(method='cubic')

    # For categorical columns, use mode
    for col in categorical_columns:
        df[col].fillna(df[col].mode()[0], inplace=True)

    return df

def _engineer_features(self, df: pd.DataFrame) -> pd.DataFrame:
    """Advanced feature engineering"""
    # Example: Create composite scores
    if all(col in df.columns for col in ['memory_score', 'attention_score']):
        df['cognitive_composite'] = df['memory_score'] * 0.6 + df['attention_score'] * 0.4

    # Add more sophisticated feature engineering as needed
    return df

def augment_mri_data(self, mri_image: np.ndarray) -> List[np.ndarray]:
    """
    Advanced data augmentation for MRI images

    Args:
        mri_image: Original MRI image
```

```
    Returns:
        List of augmented MRI images
    """
    return self._augment_mri(mri_image)

def _augment_mri(self, mri_image: np.ndarray) -> List[np.ndarray]:
    """
    Apply advanced data augmentation techniques to MRI image

    Args:
        mri_image: Input 3D MRI image

    Returns:
        List of augmented images
    """
    augmented_images = [
        mri_image,  # Original image
        np.rot90(mri_image),  # 90-degree rotation
        np.rot90(mri_image, 2),  # 180-degree rotation
        np.fliplr(mri_image),
        np.flipud(mri_image)
    ]

    # Elastic deformation
    if self.config['augmentation']['elastic_deformation'] and elasticdeform is not None:
        try:
            deformed_image = elasticdeform.deform_random_grid(
                mri_image,
                sigma=2,
```

```
                points=3
            )
            augmented_images.append(deformed_image)
        except Exception as e:
            logger.warning(f"Elastic deformation failed: {e}")

    return augmented_images

def save_preprocessed_data(self, data: np.ndarray, metadata: Dict,
                           output_path: str) -> None:
    """
    Save preprocessed data with metadata

    Args:
        data: Preprocessed image data
        metadata: Associated metadata
        output_path: Path to save the data
    """
    output_path = Path(output_path)
    output_path.parent.mkdir(parents=True, exist_ok=True)

    with h5py.File(output_path, 'w') as f:
        # Save data
        f.create_dataset('data', data=data, compression='gzip')

        # Save metadata
        metadata_group = f.create_group('metadata')
        for key, value in metadata.items():
            metadata_group.attrs[key] = value
```

```
    def load_preprocessed_data(self, file_path: str) -> Tuple[np.ndarray, Dict]:
        """
        Load preprocessed data and metadata

        Args:
            file_path: Path to preprocessed data file

        Returns:
            Tuple of (data, metadata)
        """
        with h5py.File(file_path, 'r') as f:
            data = f['data'][:]
            metadata = dict(f['metadata'].attrs)

        return data, metadata

def main():
    """Example usage with advanced features"""
    # Initialize preprocessor with custom config
    config = {
        'scaler_type': 'robust',
        'target_shape': (128, 128, 128),
        'normalization': 'z_score',
        'augmentation': {
            'elastic_deformation': True
        },
        'artifact_removal': {
            'denoise': True,
            'bias_field_correction': True,
            'skull_stripping': True
```

```
    }
}

preprocessor = AlzheimerDataPreprocessor('/path/to/data', config)

# Example: Process MRI data
mri_data = preprocessor.load_mri_data('/path/to/mri.nii.gz')
if mri_data is not None:
    # Augment data
    augmented_images = preprocessor.augment_mri_data(mri_data)

    # Save preprocessed data
    metadata = {
        'preprocessing_date': '2023-01-01',
        'original_shape': mri_data.shape,
        'normalization': 'z_score'
    }
    preprocessor.save_preprocessed_data(
        mri_data,
        metadata,
        'preprocessed_data.h5'
    )

# Example: Process cognitive data
cognitive_data = pd.DataFrame({
    'memory_score': [80, 75, 90],
    'orientation_score': [85, 70, 95],
    'attention_score': [90, 80, 85],
    'language_score': [75, 85, 95]
})
```

```
    preprocessed_cognitive = preprocessor.preprocess_cognitive_data(cognitive_data)
    logger.info(f"Preprocessed cognitive data shape: {preprocessed_cognitive.shape}")

if _name_ == "_main_":
    main()
```

src\image_preprocessor.py

```
import os
import numpy as np
import cv2
import albumentations as A
from typing import List, Union
from PIL import Image

class AlzheimerImagePreprocessor:
    def _init_(self, input_shape=(224, 224, 3), data_dir=None):
        """
        Initialize image preprocessor for Alzheimer's detection

        Args:
            input_shape: Target input shape for neural network
            data_dir: Optional directory for data loading
        """
        self.input_shape = input_shape
        self.data_dir = data_dir
        self.augmentation_pipeline = self._create_augmentation_pipeline()

    def _create_augmentation_pipeline(self):
        """
```

```
Create a robust image augmentation pipeline

Returns:
    Albumentations composition of augmentations
"""
return A.Compose([
    # Geometric transformations
    A.RandomRotate90(p=0.5),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),

    # Spatial-level transforms
    A.ShiftScaleRotate(
        shift_limit=0.1,
        scale_limit=0.1,
        rotate_limit=15,
        p=0.5
    ),

    # Intensity transformations
    A.RandomBrightnessContrast(
        brightness_limit=0.2,
        contrast_limit=0.2,
        p=0.3
    ),

    # Color space transformations
    A.ColorJitter(
        brightness=0.2,
        contrast=0.2,
```

```
            saturation=0.2,
            hue=0.2,
            p=0.3
        ),

        # Noise augmentations
        A.GaussNoise(var_limit=(0.01, 0.1), p=0.3),
    ])

def load_image(self, image_path: str) -> np.ndarray:
    """
    Load and preprocess a single image

    Args:
        image_path: Path to the image file

    Returns:
        Preprocessed image as numpy array
    """
    # Read image using OpenCV
    image = cv2.imread(image_path)

    # Convert BGR to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Resize to target input shape
    image = cv2.resize(image, (self.input_shape[0], self.input_shape[1]))

    # Normalize pixel values
    image = image.astype(np.float32) / 255.0
```

```
    return image

def load_images_from_directory(self, directory: str, label: int) -> List[dict]:
    """
    Load images from a directory with their corresponding label

    Args:
        directory: Path to image directory
        label: Class label for images in this directory

    Returns:
        List of dictionaries containing image and label
    """
    images = []
    for filename in os.listdir(directory):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tiff')):
            image_path = os.path.join(directory, filename)
            try:
                image = self.load_image(image_path)
                images.append({
                    'image': image,
                    'label': label
                })
            except Exception as e:
                print(f"Error loading {filename}: {e}")

    return images

def augment_image(self, image: np.ndarray) -> np.ndarray:
```

```
        """
        Apply augmentations to a single image

        Args:
            image: Input image numpy array

        Returns:
            Augmented image
        """
        # Apply augmentation pipeline
        augmented = self.augmentation_pipeline(image=image)['image']

        return augmented

    def generate_augmented_batch(self, images: List[np.ndarray], num_augmentations: int = 5)
-> List[np.ndarray]:
        """
        Generate multiple augmented versions of input images

        Args:
            images: List of input images
            num_augmentations: Number of augmentations per image

        Returns:
            List of augmented images
        """
        augmented_batch = []
        for image in images:
            # Generate multiple augmentations
            image_augmentations = [self.augment_image(image) for _ in
range(num_augmentations)]
```

```
            augmented_batch.extend(image_augmentations)

        return augmented_batch

    def prepare_dataset(self,
                        alzheimers_dir: str,
                        normal_dir: str,
                        test_split: float = 0.2,
                        augment: bool = True) -> tuple:
        """
        Prepare complete dataset with train/test split and optional augmentation

        Args:
            alzheimers_dir: Directory with Alzheimer's images
            normal_dir: Directory with normal brain images
            test_split: Proportion of data to use for testing
            augment: Whether to apply data augmentation

        Returns:
            Tuple of (X_train, y_train, X_test, y_test)
        """
        # Load images
        alzheimers_images = self.load_images_from_directory(alzheimers_dir, label=1)
        normal_images = self.load_images_from_directory(normal_dir, label=0)

        # Combine and shuffle datasets
        all_images = alzheimers_images + normal_images
        np.random.shuffle(all_images)

        # Separate images and labels
```

```
        X = np.array([item['image'] for item in all_images])
        y = np.array([item['label'] for item in all_images])

        # Optional augmentation
        if augment:
            augmented_images = self.generate_augmented_batch(X)
            X = np.concatenate([X, np.array(augmented_images)])
            y = np.concatenate([y, np.repeat(y, len(augmented_images) // len(y))])

        # Split into train and test sets
        split_idx = int(len(X) * (1 - test_split))
        X_train, X_test = X[:split_idx], X[split_idx:]
        y_train, y_test = y[:split_idx], y[split_idx:]

        return X_train, y_train, X_test, y_test

# Example usage
if _name_ == "_main_":
    preprocessor = AlzheimerImagePreprocessor()
    X_train, y_train, X_test, y_test = preprocessor.prepare_dataset(
        alzheimers_dir='/path/to/alzheimers/images',
        normal_dir='/path/to/normal/images'
    )
    print(f"Training data shape: {X_train.shape}")
    print(f"Training labels shape: {y_train.shape}")
    print(f"Test data shape: {X_test.shape}")
    print(f"Test labels shape: {y_test.shape}")
```

src\model_utils.py

```
import os
```

```
import requests
import ssl
import urllib3
import tensorflow as tf
from tensorflow.keras.applications import EfficientNetB0

# Disable SSL warnings and verification
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
ssl._create_default_https_context = ssl._create_unverified_context

def find_local_weights(filename='efficientnetb0_notop.h5'):
    """
    Search for weights file in potential local directories

    Args:
        filename: Name of the weights file

    Returns:
        str: Full path to weights file if found, None otherwise
    """
    # Potential directories to search
    search_dirs = [
        os.path.join(os.path.dirname(_file_), '..', 'data'),  # Project data folder
        os.path.join(os.path.dirname(_file_), 'data'),        # Local data folder
        os.path.expanduser('~/.keras/models'),                # Keras default cache
        os.getcwd()                                           # Current working directory
    ]

    for directory in search_dirs:
        weights_path = os.path.join(directory, filename)
```

```
        if os.path.exists(weights_path):
            print(f"Found weights file at: {weights_path}")
            return weights_path

    print(f"Weights file {filename} not found in searched directories")
    return None

def download_efficientnet_weights(cache_dir=None):
    """
    Manually download EfficientNetB0 weights with SSL bypass

    Args:
        cache_dir: Directory to save weights. Defaults to Keras default cache.

    Returns:
        str: Path to downloaded weights file
    """
    # First, check for local weights
    local_weights = find_local_weights()
    if local_weights:
        return local_weights

    if cache_dir is None:
        cache_dir = os.path.expanduser('~/.keras/models')

    os.makedirs(cache_dir, exist_ok=True)

    # EfficientNetB0 weights URL
    weights_url = "https://storage.googleapis.com/keras-applications/efficientnetb0_notop.h5"
    weights_filename = "efficientnetb0_notop.h5"
```

```
    weights_path = os.path.join(cache_dir, weights_filename)

    # Check if weights already exist
    if os.path.exists(weights_path):
        return weights_path

    try:
        # Download weights with SSL verification disabled
        print(f"Downloading EfficientNetB0 weights to {weights_path}")
        response = requests.get(
            weights_url,
            stream=True,
            verify=False  # Disable SSL verification
        )
        response.raise_for_status()

        with open(weights_path, 'wb') as f:
            for chunk in response.iter_content(chunk_size=8192):
                f.write(chunk)

        return weights_path

    except Exception as e:
        print(f"Failed to download weights: {e}")
        return None

def load_efficientnet_weights(input_shape):
    """
    Load EfficientNetB0 weights with multiple fallback strategies
```

```
    Args:
        input_shape: Shape of input image

    Returns:
        Loaded EfficientNetB0 model
    """
    # Try loading from local weights first
    local_weights = find_local_weights()

    try:
        if local_weights:
            # Try loading with local weights file
            model = EfficientNetB0(
                weights=local_weights,
                include_top=False,
                input_shape=input_shape
            )
            print(f"Loaded weights from local file: {local_weights}")
            return model
    except Exception as e:
        print(f"Failed to load local weights: {e}")

    # List of weight loading strategies
    weight_strategies = [
        'imagenet',  # Official imagenet weights
        None,        # Random initialization
    ]

    for strategy in weight_strategies:
        try:
```

```
            model = EfficientNetB0(
                weights=strategy,
                include_top=False,
                input_shape=input_shape
            )
            print(f"Loaded weights using strategy: {strategy}")
            return model
        except Exception as e:
            print(f"Failed to load weights with strategy {strategy}: {e}")

    # Absolute fallback to random initialization
    return EfficientNetB0(
        weights=None,
        include_top=False,
        input_shape=input_shape
    )
```

Tests\test_model.py

```
import pytest
import numpy as np
import tensorflow as tf
from src.alzheimers_model import AlzheimerDetectionModel
from src.data_preprocessing import AlzheimerDataPreprocessor

class TestAlzheimerDetectionModel:
    @pytest.fixture
    def model(self):
        """
        Create a model instance for testing
        """
```

```
    input_shape = (128, 128, 128, 1)
    return AlzheimerDetectionModel(input_shape)

@pytest.fixture
def sample_data(self):
    """
    Generate sample MRI data for testing
    """
    # Simulated 3D MRI scan data
    X_train = np.random.rand(100, 128, 128, 128, 1)
    y_train = tf.keras.utils.to_categorical(
        np.random.randint(2, size=(100, 1)),
        num_classes=2
    )

    X_val = np.random.rand(20, 128, 128, 128, 1)
    y_val = tf.keras.utils.to_categorical(
        np.random.randint(2, size=(20, 1)),
        num_classes=2
    )

    return X_train, y_train, X_val, y_val

def test_model_creation(self, model):
    """
    Test model creation and basic properties
    """
    assert model is not None, "Model should be created successfully"
    assert model.model is not None, "Keras model should be initialized"
    assert len(model.model.layers) > 0, "Model should have layers"
```

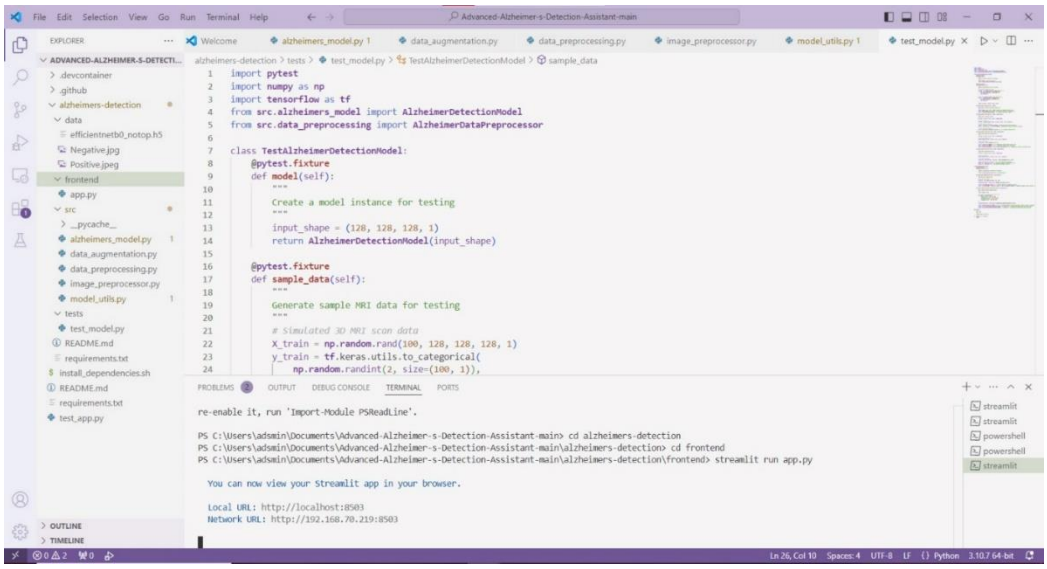
```
def test_model_training(self, model, sample_data):
    """
    Test model training process
    """
    X_train, y_train, X_val, y_val = sample_data

    # Train the model
    history = model.train(X_train, y_train, X_val, y_val, epochs=2)

    # Check training history
    assert 'accuracy' in history.history, "Training should track accuracy"
    assert 'val_accuracy' in history.history, "Validation accuracy should be tracked"

    # Check model performance
    assert len(history.history['accuracy']) > 0, "Training should occur"

def test_model_prediction(self, model, sample_data):
     """
    Test model prediction capabilities
    """
    X_train, y_train, X_val, y_val = sample_data

    # Train the model first
    model.train(X_train, y_train, X_val, y_val, epochs=2)

    # Test prediction
    prediction = model.predict(X_val[:5])

    assert prediction.shape[0] == 5, "Prediction should match input size"
```

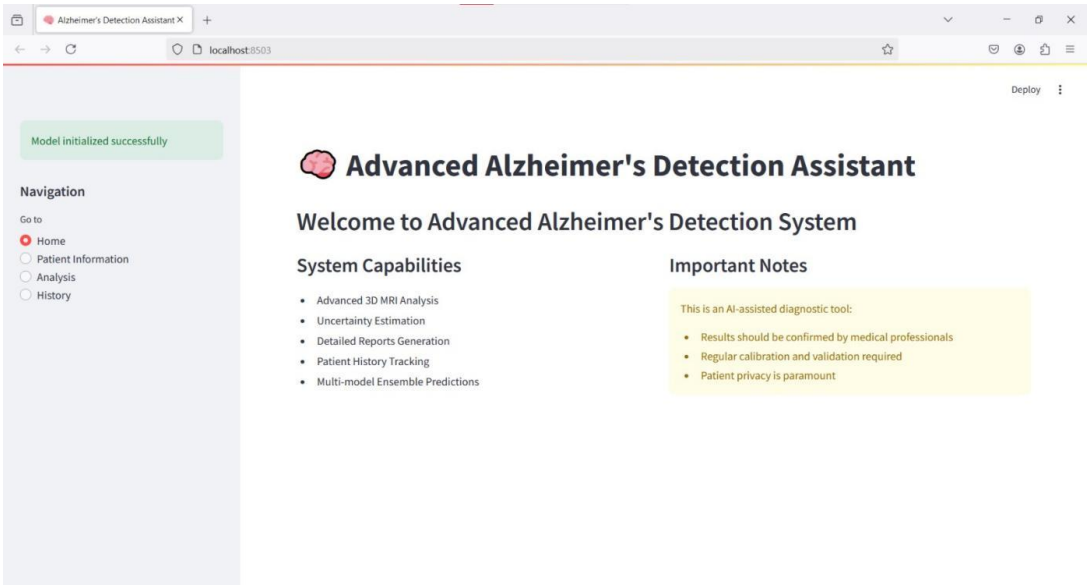
```
        assert prediction.shape[1] == 2, "Prediction should have 2 classes"
        assert np.all(prediction >= 0) and np.all(prediction <= 1), "Predictions should be probabilities"

    def test_model_evaluation(self, model, sample_data):
        """
        Test model evaluation metrics
        """
        X_train, y_train, X_val, y_val = sample_data

        # Train the model
        model.train(X_train, y_train, X_val, y_val, epochs=2)

        # Evaluate the model
        test_loss, test_accuracy, test_auc = model.evaluate(X_val, y_val)

        assert test_loss is not None, "Test loss should be calculated"
        assert 0 <= test_accuracy <= 1, "Accuracy should be between 0 and 1"
        assert 0 <= test_auc <= 1, "AUC should be between 0 and 1"

class TestDataPreprocessor:
    @pytest.fixture
    def preprocessor(self):
        """
        Create a preprocessor instance for testing
        """
        return AlzheimerDataPreprocessor('/test/data/path')

    def test_data_augmentation(self, preprocessor):
        """
        Test MRI data augmentation
```

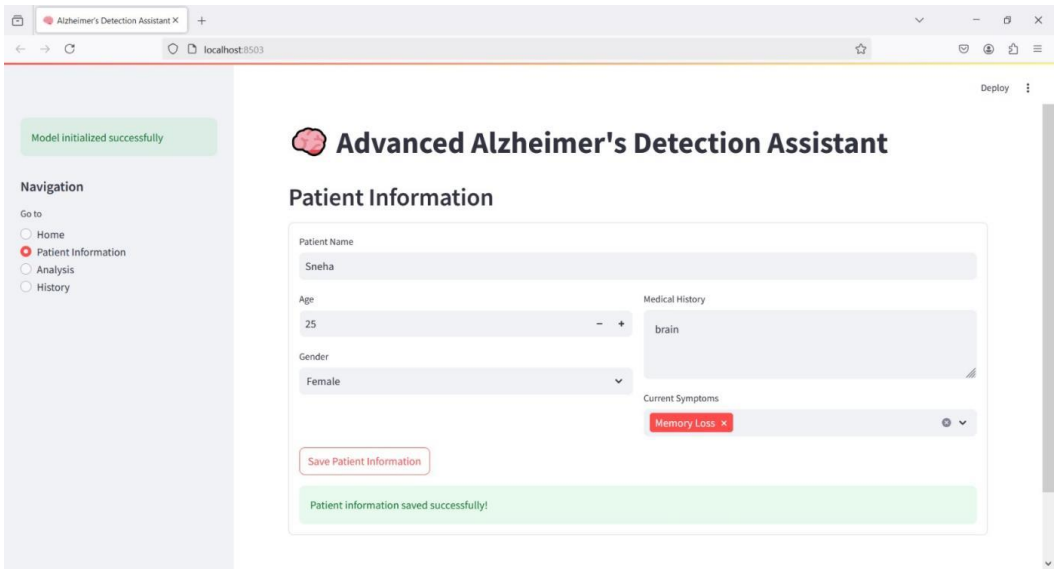
```
        """
        # Simulated MRI data
        mock_mri = np.random.rand(128, 128, 128)

        augmented_images = preprocessor.augment_mri_data(mock_mri)

        assert len(augmented_images) == 5, "Should generate 5 augmented images"
        assert all(img.shape == mock_mri.shape for img in augmented_images), "Augmented images should maintain original shape"

    def test_cognitive_data_preprocessing(self, preprocessor):
        """
        Test cognitive data preprocessing
        """
        import pandas as pd

        # Create mock cognitive assessment data
        mock_data = pd.DataFrame({
            'memory_score': [80, 75, 90],
            'orientation_score': [85, 70, 95],
            'attention_score': [90, 80, 85],
            'language_score': [75, 85, 95]
        })

        preprocessed_data = preprocessor.preprocess_cognitive_data(mock_data)

        assert preprocessed_data.shape == (3, 4), "Preprocessed data should maintain original number of samples"
        assert np.isclose(preprocessed_data.mean(), 0, atol=1e-7), "Scaled data should have near-zero mean"
        assert np.isclose(preprocessed_data.std(), 1, atol=1e-7), "Scaled data should have unit variance"
```

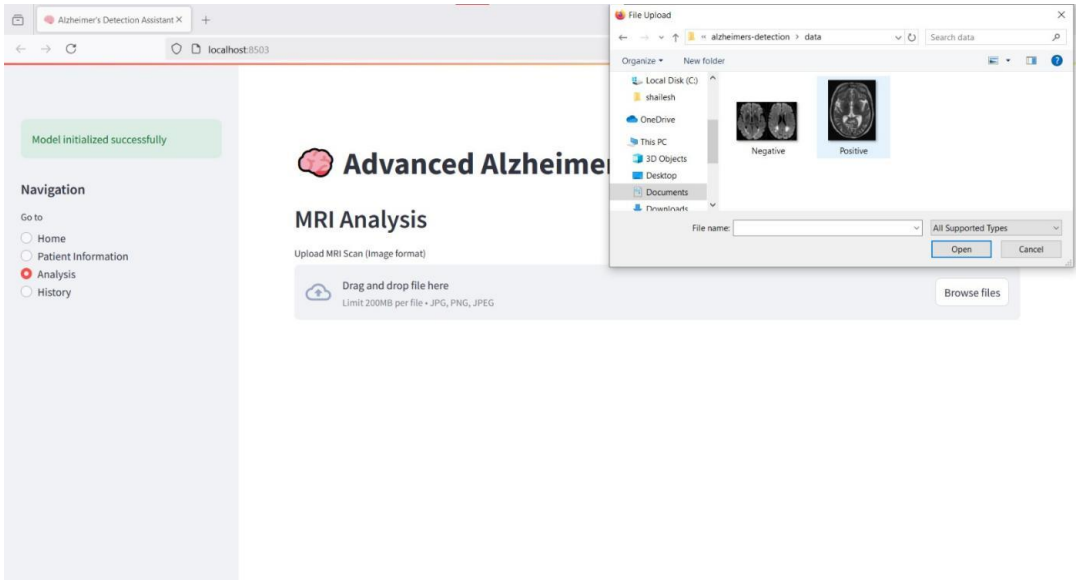
```
def main():
    """
    Run tests
    """
    pytest.main([_file_])

if _name_ == "_main_":
    main()
```

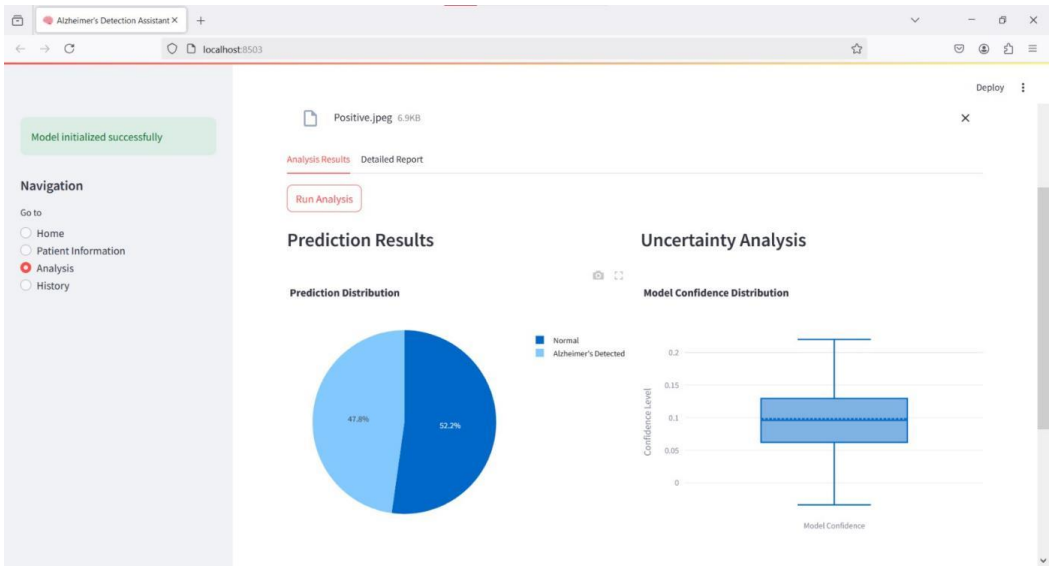
OUTPUT

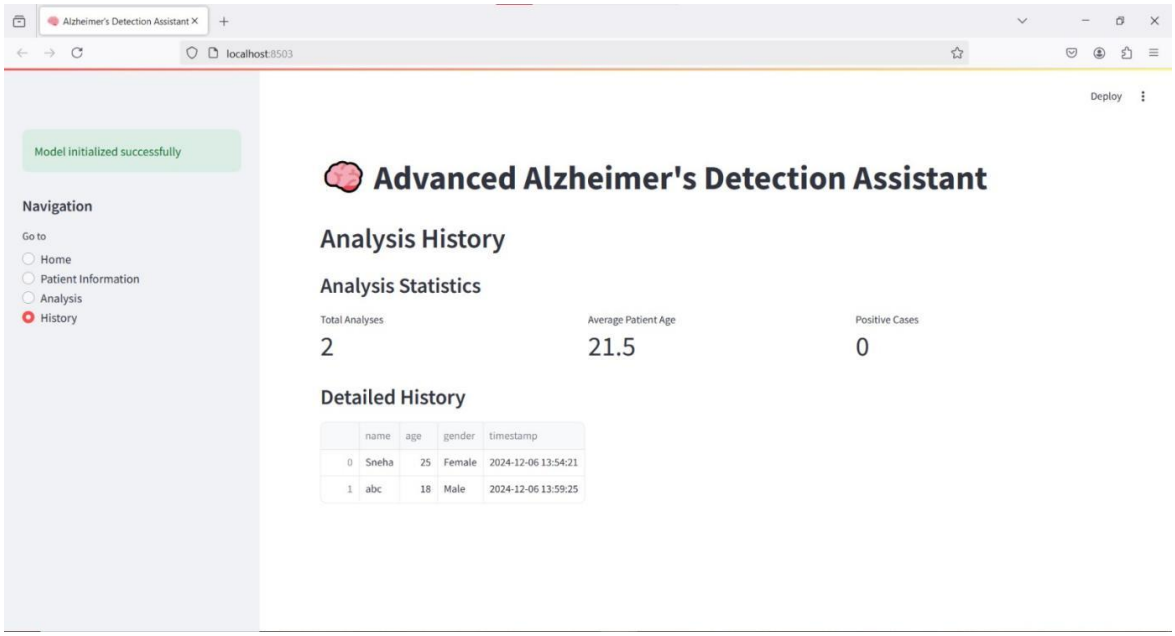



Alzheimer's Detection Assistant
localhost:8503
Deploy
Model initialized successfully
Navigation
Go to
Home
Patient Information
Analysis
History
Advanced Alzheimer's Detection Assistant
Welcome to Advanced Alzheimer's Detection System
System Capabilities
Advanced 3D MRI Analysis
Uncertainty Estimation
Detailed Reports Generation
Patient History Tracking
Multi-model Ensemble Predictions
Important Notes
This is an AI-assisted diagnostic tool:
Results should be confirmed by medical professionals
Regular calibration and validation required
Patient privacy is paramount


Alzheimer's Detection Assistant
localhost:8503
Deploy
Model initialized successfully
Navigation
Go to
Home
Patient Information
Analysis
History
Advanced Alzheimer's Detection Assistant
Patient Information
Patient Name
Sneha
Age
25
Gender
Female
Medical History
brain
Current Symptoms
Memory Loss
Save Patient Information
Patient information saved successfully!


Alzheimer's Detection Assistant
localhost:8503
Model initialized successfully
Navigation
Go to
Home
Patient Information
Analysis
History
Advanced Alzheime
MRI Analysis
Upload MRI Scan (Image format)
Drag and drop file here
Limit 200MB per file • JPG, PNG, JPEG
Browse files
File Upload
alzheimers-detection > data
Search data
Organize
New folder
Local Disk (C:)
shailesh
OneDrive
This PC
3D Objects
Desktop
Documents
Negative
Positive
File name:
All Supported Types
Open
Cancel


Alzheimer's Detection Assistant
localhost:8503
Deploy
Model initialized successfully
Navigation
Go to
Home
Patient Information
Analysis
History
Positive.jpeg 6.9KB
Analysis Results
Detailed Report
Run Analysis
Prediction Results
Prediction Distribution
47.8%
52.2%
Normal
Alzheimer's Detected
Uncertainty Analysis
Model Confidence Distribution
Confidence Level
0.2
0.15
0.1
0.05
0
Model Confidence



Github Link:

https://github.com/Laxmi1720/Alzeimer-s_Diesese_database-