app.py

```python
import streamlit as st
import pandas as pd
import os
import sys

# Add the current directory to path to allow imports from src
sys.path.append(os.getcwd())

from src.eda import load_data, show_stats, plot_correlation, plot_distribution, show_missing_values, plot_pairplot, impute_missing_values, impute_all_missing_values, convert_to_numeric
from src.model import train_linear_regression, train_polynomial_regression, evaluate_model, plot_regression_results, train_knn_regression, train_random_forest_regression, generate_model_explanation, plot_actual_vs_predicted

# Set page config
st.set_page_config(page_title="EDA & Regression Analysis", layout="wide")

st.title("Nuclear Energy Insights Dashboard & EDA")

# --- Sidebar: Data Loading ---
st.sidebar.header("1. Upload Data")
uploaded_file = st.sidebar.file_uploader("Upload your CSV file", type=["csv"])

# Default file path
default_file_path = "archive/us_nuclear_generating_statistics_1971_2021.csv"

# Initialize session state for dataframe if not exists
if 'df' not in st.session_state:
    st.session_state.df = None

# Load data logic
if uploaded_file is not None:
    # Check if we need to reload (e.g. new file uploaded)
    # Simple check: just reload. For optimization, could check file name.
    # For now, if uploaded_file changes, Streamlit re-runs script, so we reload.
    st.session_state.df = load_data(uploaded_file)
    st.sidebar.success("File uploaded successfully!")
elif st.session_state.df is None and os.path.exists(default_file_path):
    st.sidebar.info(f"Using default dataset: {os.path.basename(default_file_path)}")
    st.session_state.df = load_data(default_file_path)
elif st.session_state.df is None:
    st.sidebar.warning("Please upload a CSV file to proceed.")

# --- Main App Logic ---
if st.session_state.df is not None:
    df = st.session_state.df # Local alias for convenience
    # Sidebar Navigation
```

```python
    page = st.sidebar.radio("Navigate", ["Exploratory Data Analysis (EDA)", "Regression Modeling"])

    if page == "Exploratory Data Analysis (EDA)":
        st.header("🔎 Exploratory Data Analysis")

        # Data Overview
        st.subheader("Dataset Overview")
        st.write(f"Shape: {df.shape[0]} rows, {df.shape[1]} columns")
        st.dataframe(df.head())

        # Stats
        st.subheader("Descriptive Statistics")
        st.write(df.describe())

        # Missing Values
        st.subheader("Missing Values")
        missing_vals = show_missing_values(df)
        st.write(missing_vals)

        # Imputation
        if missing_vals.sum() > 0:
            st.markdown("### Impute Missing Values")
            cols_with_missing = missing_vals[missing_vals > 0].index.tolist()

            if cols_with_missing:
                c1, c2, c3 = st.columns([2, 1, 1])
                with c1:
                    col_to_impute = st.selectbox("Select Column to Impute", cols_with_missing)
                with c2:
                    imp_strategy = st.selectbox("Strategy", ["Mean", "Median", "Mode"])
                with c3:
                    if st.button("Apply Imputation"):
                        st.session_state.df = impute_missing_values(st.session_state.df, col_to_impute, imp_strategy)
                        st.success(f"Imputed {col_to_impute} with {imp_strategy}")
                        st.rerun()

            st.markdown("#### Bulk Imputation")
            c_bulk1, c_bulk2 = st.columns([2, 1])
            with c_bulk1:
                bulk_strategy = st.selectbox("Bulk Strategy (All Columns)", ["Mean", "Median", "Mode"])
            with c_bulk2:
                if st.button("Impute All"):
                    st.session_state.df = impute_all_missing_values(st.session_state.df, bulk_strategy)
                    st.success(f"Imputed all valid columns with {bulk_strategy}")
                    st.rerun()

```

```
        # Visualizations
        st.subheader("Visualizations")

        col1, col2 = st.columns(2)

        with col1:
            st.markdown("### Correlation Heatmap")
            fig_corr = plot_correlation(df)
            if fig_corr:
                st.pyplot(fig_corr)

        with col2:
            st.markdown("### Distribution Plot")
            numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()
            if numeric_cols:
                selected_col = st.selectbox("Select column for distribution", numeric_cols)
                fig_dist = plot_distribution(df, selected_col)
                st.pyplot(fig_dist)
            else:
                st.write("No numeric colums for distribution plot.")

    elif page == "Regression Modeling":
        st.header("📈 Regression Modeling")

        # Column Selection
        numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()

        # Check if we have enough numeric columns; if not, try to convert
        if len(numeric_cols) < 2:
            with st.spinner("Attempting to convert text columns to numbers..."):
                st.session_state.df = convert_to_numeric(st.session_state.df)
                df = st.session_state.df # Refresh local alias
                numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()

            if len(numeric_cols) >= 2:
                st.success(f"Successfully converted data! Found {len(numeric_cols)} numeric columns.")

        if len(numeric_cols) < 2:
            st.error("Dataset needs at least 2 numeric columns for regression.")
            st.write("Current Numeric Columns:", numeric_cols)
            st.write("All Columns & Types:", df.dtypes)
        else:
            col1, col2 = st.columns(2)
            with col1:
                target_col = st.selectbox("Select Target Variable (Y)", numeric_cols, index=len(numeric_cols)-1)
            with col2:
                feature_options = [c for c in numeric_cols if c != target_col]
                # Auto-select the first feature by default to avoid empty state error
```

```python
                default_feat = [feature_options[0]] if feature_options else None
                feature_col = st.multiselect("Select Feature Variable(s) (X)", feature_options, default=default_feat)

            if not feature_col:
                st.warning("Please select at least one feature variable.")
            else:
                model_type = st.radio("Select Model Type", ["Linear Regression", "Polynomial Regression", "KNN Regression", "Random Forest Regression"])

                degree = 2
                k_neighbors = 5
                n_estimators = 100

                if model_type == "Polynomial Regression":
                    degree = st.slider("Select Polynomial Degree", 2, 5, 2)
                elif model_type == "KNN Regression":
                    k_neighbors = st.slider("Select K Neighbors", 1, 20, 5)
                elif model_type == "Random Forest Regression":
                    n_estimators = st.slider("Select Number of Trees (Estimators)", 10, 500, 100, step=10)

                if st.button("Train Model"):
                    # Create a subset for training
                    train_df = df.dropna(subset=feature_col + [target_col])

                    if len(train_df) == 0:
                        st.error("No data left after removing missing values. Please check your data.")
                    else:
                        if len(df) != len(train_df):
                            st.warning(f"Dropped {len(df) - len(train_df)} rows containing missing values.")

                        X = train_df[feature_col].values # Multiselect returns list, so this works for both single and multi
                        y = train_df[target_col].values

                    # Ensure X is 2D
                    if len(X.shape) == 1:
                        X = X.reshape(-1, 1)

                    poly_features = None # Default

                    if model_type == "Linear Regression":
                        model = train_linear_regression(X, y)
                    elif model_type == "Polynomial Regression":
                        model, poly_features = train_polynomial_regression(X, y, degree)
                    elif model_type == "KNN Regression":
                        model = train_knn_regression(X, y, k_neighbors)
                    elif model_type == "Random Forest Regression":
```

```
                        model = train_random_forest_regression(X, y, n_estimators)

                    # Evaluate happens here for all because logic is shared except for poly transform
                    mse, r2, y_pred = evaluate_model(model, X, y, poly_features) # evaluate_model generates predictions too

                    # metrics
                    st.success("Model Trained!")
                    m_col1, m_col2 = st.columns(2)
                    m_col1.metric("R2 Score", f"{r2:.4f}")
                    m_col2.metric("MSE", f"{mse:.4f}")

                    with st.expander("ℹ️ How to interpret these results?"):
                        st.write("""
                        **1. R2 Score (0 to 1):**
                        - Represents accuracy. **1.0 (100%)** is perfect.
                        - **< 0.3**: Weak prediction.
                        - **0.3 - 0.7**: Moderate.
                        - **> 0.7**: Strong.

                        **2. Mean Squared Error (MSE):**
                        - The average squared difference between actual and predicted values.
                        - **Lower is better**. 0 means no error.

                        **3. Regression Plot:**
                        - **Blue Dots**: The model's predictions.
                        - **Red Line**: Perfect prediction (Actual = Predicted).
                        - **Goal**: Points should be as close to the red line as possible.
                        """)

                    # Plot
                    st.subheader("Regression Plot")
                    if len(feature_col) > 1:
                        # Multi-feature: Plot Actual vs Predicted
                        fig_reg = plot_actual_vs_predicted(y, y_pred, title=f"{model_type} (Actual vs Predicted)")
                        st.pyplot(fig_reg)
                        st.info("Note: When using multiple features, we plot 'Actual vs Predicted' because we cannot easily visualize >3 dimensions.")
                    else:
                        # Single feature: Standard regression plot
                        title = f"{model_type}"
                        if model_type == "Polynomial Regression":
                            title += f" (Degree: {degree})"
                        elif model_type == "KNN Regression":
                            title += f" (K: {k_neighbors})"
                        elif model_type == "Random Forest Regression":
                            title += f" (Trees: {n_estimators})"

                        fig_reg = plot_regression_results(X, y, y_pred, title=title)
```

```
                        st.pyplot(fig_reg)

                    # Explanation
                    st.subheader("Model Insights")
                    explanation = generate_model_explanation(model, model_type, feature_col, target_col)
                    st.markdown(explanation)

            # Suggestion for non-linear check
            st.info("Tip: If R2 is low for Linear Regression, try Polynomial Regression to capture non-linear relationships.")

```