```
# Install dependencies (Colab)
!pip install -q tensorflow scikit-learn matplotlib gradio
!pip install adjustText

# Imports
import io, sys, math, random
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import gradio as gr
from google.colab import files
from adjustText import adjust_text

# Global variables to store data and models
global_df = None
global_models = {}
global_scalers = {}
global_country_data = {}

# Helper functions for LSTM
def create_sequences(data, lookback):
    X, y = [], []
    for i in range(len(data) - lookback):
        X.append(data[i:(i + lookback)])
        y.append(data[i + lookback])
    return np.array(X), np.array(y)

def build_lstm(input_shape, units=32, dropout=0.2):
    model = Sequential()
    model.add(LSTM(units, activation='relu', input_shape=input_shape))
    model.add(Dropout(dropout))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')
    return model

def load_dataset(file):
    global global_df
    try:
        if file is None:
            return "No file provided.", "", gr.update(choices=[], value=None), gr.update(choices=[], value=None)

        if isinstance(file, str):
            df = pd.read_csv(file)
        else:
            if hasattr(file, "read"):
                file.seek(0)
                df = pd.read_csv(file)
            elif hasattr(file, "name"):
                df = pd.read_csv(file.name)
            else:
                df = pd.read_csv(file)

        df.columns = [c.strip() for c in df.columns]
        global_df = df.copy()

        countries = sorted(df['Country'].dropna().unique().tolist()) if 'Country' in df.columns else []

        html_preview = df.head(10).to_html(index=False)
        status = f"Dataset loaded successfully! Shape: {df.shape}"

        return status, html_preview, gr.update(choices=countries, value=countries[0] if countries else None), gr.update(choi
    except Exception as e:
        import traceback
        traceback.print_exc()
        return f"Error loading file: {e}", "", gr.update(choices=[], value=None), gr.update(choices=[], value=None)

def preprocess_energy(
    df,
    country_col='Country',
    year_col='Year'
```

```
    year_col='Year',
    zero_as_missing_cols=None,
    min_points_for_interpolation=3,
    inplace=False
):
    if not inplace:
        df = df.copy()

    df.columns = [c.strip() for c in df.columns]
    df.dropna(axis=1, how='all', inplace=True)

    missing_placeholders = ['', ' ', 'NA', 'N/A', 'na', 'n/a', 'NULL', 'null', '--', '?', '-', 'nan', 'NaN']
    df.replace(missing_placeholders, np.nan, inplace=True)

    if country_col not in df.columns:
        raise KeyError(f"Country column '{country_col}' not found in dataframe")
    if year_col not in df.columns:
        date_candidates = [c for c in df.columns if 'date' in c.lower() or 'time' in c.lower()]
        if date_candidates:
            try:
                df[year_col] = pd.to_datetime(df[date_candidates[0]], errors='coerce').dt.year
            except Exception:
                df[year_col] = np.nan
        else:
            df[year_col] = np.nan

    df[year_col] = pd.to_numeric(df[year_col], errors='coerce')
    df = df[~df[year_col].isna()].copy()
    df[year_col] = df[year_col].astype(int)

    df[country_col] = df[country_col].astype(str).str.strip()
    df = df[df[country_col] != ''].reset_index(drop=True)

    exclude = {country_col, year_col}
    candidate_cols = [c for c in df.columns if c not in exclude]
    numeric_cols = []
    for c in candidate_cols:
        coerced = pd.to_numeric(df[c], errors='coerce')
        parseable_ratio = coerced.notna().sum() / max(1, len(coerced))
        if pd.api.types.is_numeric_dtype(df[c]) or parseable_ratio >= 0.5:
            df[c] = coerced
            numeric_cols.append(c)

    total_rows = len(df)
    countries = sorted(df[country_col].dropna().unique().tolist())
    year_min = int(df[year_col].min()) if total_rows>0 else None
    year_max = int(df[year_col].max()) if total_rows>0 else None
    initial_missing = df[numeric_cols].isna().sum().sum()
    initial_zeros = (df[numeric_cols] == 0).sum().sum()

    dup_mask = df.duplicated(subset=[country_col, year_col], keep=False)
    dup_count = dup_mask.sum()
    if dup_count > 0:
        agg_cols = numeric_cols
        df = df.groupby([country_col, year_col], as_index=False).agg({**{c: 'mean' for c in agg_cols}})
    else:
        df = df.sort_values([country_col, year_col]).reset_index(drop=True)

    problematic_zero_count = 0
    if zero_as_missing_cols:
        for col in zero_as_missing_cols:
            if col in df.columns:
                zeros = (df[col] == 0)
                problematic_zero_count += zeros.sum()
                df.loc[zeros, col] = np.nan

    interpolation_filled = 0
    median_filled = 0
    missing_report = []

    for country in df[country_col].unique():
        mask = df[country_col] == country
        sub = df.loc[mask].sort_values(year_col).reset_index()
        if len(sub) == 0:
            continue
        numeric_sub = sub[numeric_cols]
        missing_before = numeric_sub.isna().sum().sum()
        if len(sub) >= min_points_for_interpolation:
            filled = numeric_sub.interpolate(method='linear', limit_direction='both', axis=0)
            missing_after = filled.isna().sum().sum()
            interpolation_filled += (missing_before - missing_after)
            df.loc[mask, numeric_cols] = filled.values
            if missing_before > 0:
```

```
                missing_report.append(f"{country}: {missing_before} -> {missing_after} missing (interpolation)")
            else:
                filled = numeric_sub.ffill().bfill()
                missing_after = filled.isna().sum().sum()
                df.loc[mask, numeric_cols] = filled.values
                if missing_before > 0:
                    missing_report.append(f"{country}: {missing_before} -> {missing_after} missing (ffill/bfill)")

        remaining_missing = df[numeric_cols].isna().sum().sum()
        if remaining_missing > 0:
            for col in numeric_cols:
                cnt = df[col].isna().sum()
                if cnt > 0:
                    med = df[col].median(skipna=True)
                    if np.isnan(med):
                        med = 0.0
                    df[col].fillna(med, inplace=True)
                    median_filled += cnt

        final_missing = df[numeric_cols].isna().sum().sum()
        final_zeros = (df[numeric_cols] == 0).sum().sum()

        df = df.sort_values([country_col, year_col]).reset_index(drop=True)

        report_lines = []
        report_lines.append(f"Processed {total_rows:,} rows; Countries: {len(countries)}; Years: {year_min} - {year_max}")
        report_lines.append(f"Missing initially: {int(initial_missing)}")
        report_lines.append(f"Filled by interpolation: {int(interpolation_filled)}")
        report_lines.append(f"Filled by median: {int(median_filled)}")
        report_lines.append(f"Zeros replaced in specified columns: {int(problematic_zero_count)}")
        report_lines.append("")
        for col in numeric_cols:
          col_initial_missing = df[col].isna().sum()
          col_final_missing = df[col].isna().sum()
          col_zeros = int((df[col] == 0).sum())

          report_lines.append(
              f" * {col}: {col_initial_missing} -> {col_final_missing} missing | {col_zeros} zeros"
          )

        report_text = "\n".join(report_lines)

        fig = None
        target_col = None
        for candidate in ['Total Energy Consumption (TWh)', 'Total Energy Consumption', 'Total Energy', 'Energy Consumption (TWh
            if candidate in df.columns:
                target_col = candidate
                break

        if target_col:
            try:
                agg = df.groupby(year_col)[target_col].sum().dropna()
                fig, ax = plt.subplots(figsize=(10, 4))
                ax.plot(agg.index.values, agg.values, marker='o', linewidth=2)
                ax.set_title("Global Total Energy Consumption Over Years")
                ax.set_xlabel("Year")
                ax.set_ylabel(target_col)
                ax.grid(True, alpha=0.3)
                plt.tight_layout()
            except Exception:
                fig = None

        return df, report_text, fig

    def preprocess_wrapper():
        global global_df
        if global_df is None:
            return "Error: No dataset loaded. Please load a CSV file first.", "", None, gr.update(choices=[], value=None), gr.up

        try:
            zero_as_missing_cols = [
                'Per Capita Energy Use (kWh)', 'Renewable Energy Share (%)',
                'Fossil Fuel Dependency (%)', 'Industrial Energy Use (%)',
                'Household Energy Use (%)', 'Carbon Emissions (Million Tons)',
                'Total Energy Consumption (TWh)'
            ]

            preprocessed_df, report_text, fig = preprocess_energy(
                global_df.copy(),
                zero_as_missing_cols=zero_as_missing_cols,
                inplace=False
            )
```

```
            global_df = preprocessed_df

            countries = sorted(global_df['Country'].dropna().unique().tolist()) if 'Country' in global_df.columns else []

            status_message = f"Preprocessing complete! Data shape: {global_df.shape}"

            return (
                status_message,
                report_text,
                fig,
                gr.update(choices=countries, value=countries[0] if countries else None),
                gr.update(choices=countries, value=countries[0] if countries else None),
                gr.update(choices=countries, value=countries[0] if countries else None)
            )
        except Exception as e:
            import traceback
            traceback.print_exc()
            return f"Error during preprocessing: {e}", "", None, gr.update(choices=[], value=None), gr.update(choices=[], value=

    def train_models(country, lookback=5, lstm_units=32):
        global global_df, global_models, global_scalers, global_country_data

        if global_df is None:
            return "Please load and preprocess data first!", None, ""

        if not country:
            return "Please select a country!", None, ""

        try:
            country_df = global_df[global_df['Country'] == country].sort_values('Year').reset_index(drop=True)
            if len(country_df) < lookback + 3:
                return f"Not enough data for {country}. Need at least {lookback + 3} years.", None, ""

            target_col = 'Total Energy Consumption (TWh)'
            if target_col not in country_df.columns:
                return f"Target column '{target_col}' not found for {country}.", None, ""

            target_series = country_df[target_col].astype(float).values

            scaler_target = MinMaxScaler()
            scaled_target = scaler_target.fit_transform(target_series.reshape(-1, 1)).flatten()

            X_all, y_all = create_sequences(scaled_target, lookback=lookback)
            if X_all.size == 0:
                return "Not enough sequence samples after creating sequences.", None, ""

            n_samples = len(X_all)
            train_n = int(0.7 * n_samples)
            val_n = train_n + int(0.15 * n_samples)

            X_train = X_all[:train_n].reshape((train_n, lookback, 1))
            y_train = y_all[:train_n]
            X_test = X_all[val_n:].reshape((len(X_all[val_n:]), lookback, 1))
            y_test = y_all[val_n:]

            raw_series = target_series
            X_raw, y_raw = create_sequences(raw_series, lookback=lookback)
            if X_raw.size == 0:
                return "Not enough raw sequence samples for classical regressors.", None, ""
            X_raw_flat = np.array(X_raw).reshape(len(X_raw), lookback)
            train_X_raw = X_raw_flat[:train_n]
            test_X_raw = X_raw_flat[val_n:]
            y_test_raw = y_raw[val_n:]

            lr = LinearRegression()
            lr.fit(train_X_raw, y_raw[:train_n])
            y_pred_lr = lr.predict(test_X_raw) if len(test_X_raw) > 0 else np.array([])
            rmse_lr = math.sqrt(mean_squared_error(y_test_raw, y_pred_lr)) if len(y_pred_lr) > 0 else float('nan')
            mae_lr = mean_absolute_error(y_test_raw, y_pred_lr) if len(y_pred_lr) > 0 else float('nan')

            dt = DecisionTreeRegressor(random_state=42)
            rf = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1)

            dt.fit(train_X_raw, y_raw[:train_n])
            rf.fit(train_X_raw, y_raw[:train_n])

            y_pred_dt = dt.predict(test_X_raw) if len(test_X_raw) > 0 else np.array([])
            y_pred_rf = rf.predict(test_X_raw) if len(test_X_raw) > 0 else np.array([])

            rmse_dt = math.sqrt(mean_squared_error(y_test_raw, y_pred_dt)) if len(y_pred_dt) > 0 else float('nan')
            mae_dt = mean_absolute_error(y_test_raw, y_pred_dt) if len(y_pred_dt) > 0 else float('nan')

            rmse_rf = math.sqrt(mean_squared_error(y_test_raw, y_pred_rf)) if len(y_pred_rf) > 0 else float('nan')
```

```
            rmse_rf = math.sqrt(mean_squared_error(y_test_raw, y_pred_rf)) if len(y_pred_rf) > 0 else float('nan')
            mae_rf = mean_absolute_error(y_test_raw, y_pred_rf) if len(y_pred_rf) > 0 else float('nan')

            model = build_lstm((lookback, 1), units=lstm_units, dropout=0.2)
            early_stop = EarlyStopping(monitor='loss', patience=10, restore_best_weights=True)
            model.fit(X_train, y_train, epochs=100, batch_size=8, callbacks=[early_stop], verbose=0)

            y_pred_lstm_scaled = model.predict(X_test).flatten() if len(X_test) > 0 else np.array([])
            y_pred_lstm = scaler_target.inverse_transform(y_pred_lstm_scaled.reshape(-1, 1)).flatten() if len(y_pred_lstm_scaled
            y_test_unscaled = scaler_target.inverse_transform(y_test.reshape(-1, 1)).flatten() if len(y_test) > 0 else np.array(

            rmse_lstm = math.sqrt(mean_squared_error(y_test_unscaled, y_pred_lstm)) if len(y_test_unscaled) > 0 else float('nan'
            mae_lstm = mean_absolute_error(y_test_unscaled, y_pred_lstm) if len(y_pred_lstm) > 0 else float('nan')

            global_models[country] = {
                'lr': lr,
                'lstm': model,
                'dt': dt,
                'rf': rf,
                'scaler_target': scaler_target,
                'lookback': lookback
            }
            global_country_data[country] = country_df

            start_idx = lookback + val_n
            years_for_test = country_df['Year'].values[start_idx:start_idx + len(y_test_unscaled)]

            fig, ax = plt.subplots(figsize=(10, 5))

            if len(y_test_unscaled) > 0:
                ax.plot(years_for_test, y_test_unscaled, label='Actual', marker='o', linewidth=2)
            if len(y_pred_lstm) > 0:
                ax.plot(years_for_test, y_pred_lstm, label='LSTM Prediction', marker='s', linewidth=2)
            if len(y_pred_lr) > 0:
                ax.plot(years_for_test, y_pred_lr, label='Linear Regression', marker='^', linewidth=1.5, linestyle='--')
            if len(y_pred_dt) > 0:
                ax.plot(years_for_test, y_pred_dt, label='Decision Tree', marker='x', linewidth=1.5, linestyle=':')
            if len(y_pred_rf) > 0:
                ax.plot(years_for_test, y_pred_rf, label='Random Forest', marker='d', linewidth=1.5, linestyle='-.')

            ax.set_title(f"Energy Consumption Forecast - {country}")
            ax.set_xlabel("Year")
            ax.set_ylabel(target_col)

            if len(years_for_test) > 0:
                ax.set_xticks(np.arange(years_for_test.min(), years_for_test.max() + 1, 1))

            ax.legend()
            ax.grid(True)
            plt.tight_layout()

            results_text = (
                f"Model Results for {country}:\n\n"
                f"Linear Regression: RMSE={rmse_lr:.4f}, MAE={mae_lr:.4f}\n"
                f"Decision Tree:      RMSE={rmse_dt:.4f}, MAE={mae_dt:.4f}\n"
                f"Random Forest:      RMSE={rmse_rf:.4f}, MAE={mae_rf:.4f}\n"
                f"LSTM:               RMSE={rmse_lstm:.4f}, MAE={mae_lstm:.4f}\n\n"
                f"Data points={len(country_df)}, lookback={lookback}, test_samples={len(y_test_unscaled)}"
            )

            return f"Models trained successfully for {country}!", fig, results_text

        except Exception as e:
            import traceback
            traceback.print_exc()
            return f"Error training models: {e}", None, ""

    # ----------------- ENHANCED FORECASTING WITH MONTHLY BREAKDOWN -----------------
    def forecast_future_with_monthly(country, years_ahead=5):
        global global_models, global_country_data

        if not country:
            return None, None, "Please select a country first!"
        if country not in global_models:
            return None, None, "Please train models for this country first!"

        try:
            model_data = global_models[country]
            country_df = global_country_data[country]
            lookback = model_data['lookback']
            target_col = 'Total Energy Consumption (TWh)'
            target_series = country_df[target_col].astype(float).values
            scaler_target = model_data['scaler_target']
```

```
scaled_target = scaler_target.transform(target_series.reshape(-1,1)).flatten()
last_window = scaled_target[-lookback:].copy()

def recursive_forecast(last_window_scaled, steps, model, scaler_target):
    window = last_window_scaled.copy()
    preds = []
    for s in range(steps):
        inp = window.reshape(1, window.shape[0], 1)
        p_scaled = model.predict(inp, verbose=0).flatten()[0]
        preds.append(p_scaled)
        window = np.roll(window, -1)
        window[-1] = p_scaled
    preds_unscaled = scaler_target.inverse_transform(np.array(preds).reshape(-1,1)).flatten()
    return preds_unscaled

# Get yearly forecast
yearly_forecast = recursive_forecast(last_window, years_ahead, model_data['lstm'], scaler_target)

# Create monthly breakdown (seasonal pattern)
def create_monthly_breakdown(yearly_forecast):
    """
    Convert yearly forecast to monthly with seasonal patterns
    Assumes typical energy consumption patterns:
    - Higher in winter (heating) and summer (cooling)
    - Lower in spring/fall
    """
    # Typical monthly distribution (can be customized per country/climate)
    monthly_pattern = np.array([
        1.10,  # Jan - high (winter)
        1.05,  # Feb - high
        0.95,  # Mar - medium
        0.90,  # Apr - low
        0.85,  # May - low
        0.95,  # Jun - medium
        1.15,  # Jul - high (summer)
        1.10,  # Aug - high
        0.95,  # Sep - medium
        0.90,  # Oct - low
        0.95,  # Nov - medium
        1.05   # Dec - high
    ])

    # Normalize pattern to sum to 12 (so yearly total = original forecast)
    monthly_pattern = monthly_pattern * (12 / monthly_pattern.sum())

    monthly_forecast = {}
    monthly_dates = []

    last_hist_year = int(country_df['Year'].values[-1])

    for year_idx, yearly_value in enumerate(yearly_forecast):
        year = last_hist_year + year_idx + 1
        monthly_values = yearly_value * monthly_pattern / 12

        for month in range(1, 13):
            date_str = f"{year}-{month:02d}"
            monthly_dates.append(date_str)
            monthly_forecast[date_str] = monthly_values[month-1]

    return monthly_dates, monthly_forecast

monthly_dates, monthly_forecast_dict = create_monthly_breakdown(yearly_forecast)
monthly_values = [monthly_forecast_dict[date] for date in monthly_dates]

# Plot 1: Yearly Forecast
hist_years = country_df['Year'].values
hist_vals = target_series
last_hist_year = int(hist_years[-1])
forecast_years = list(range(last_hist_year + 1, last_hist_year + years_ahead + 1))

fig_yearly, ax_yearly = plt.subplots(figsize=(10, 5))
ax_yearly.plot(hist_years[-10:], hist_vals[-10:], label='Historical', marker='o', linewidth=2, color='cyan')
ax_yearly.plot(forecast_years, yearly_forecast, label='Forecast', marker='s', linewidth=2, linestyle='--', color='ye
ax_yearly.set_title(f"Yearly Energy Consumption Forecast - {country}", fontsize=14, color='white')
ax_yearly.set_xlabel("Year", color='white')
ax_yearly.set_ylabel(target_col, color='white')
ax_yearly.legend(facecolor='#1a1a2e', edgecolor='white', labelcolor='white')
ax_yearly.grid(True, alpha=0.3)
ax_yearly.tick_params(colors='white')
ax_yearly.set_facecolor('#1a1a2e')
fig_yearly.patch.set_facecolor('#1a1a2e')
plt.tight_layout()
```

```
            # Plot 2: Monthly Breakdown
            fig_monthly, ax_monthly = plt.subplots(figsize=(12, 6))

            # Plot first 2 years of monthly data for clarity
            display_months = min(24, len(monthly_dates))  # Show max 2 years
            months_to_show = monthly_dates[:display_months]
            values_to_show = monthly_values[:display_months]

            # Create x-axis labels (Year-Month)
            x_positions = np.arange(len(months_to_show))

            bars = ax_monthly.bar(x_positions, values_to_show,
                                  color=plt.cm.viridis(np.linspace(0, 1, len(months_to_show))),
                                  alpha=0.7, edgecolor='white', linewidth=1)

            # Add value labels on bars
            for i, (x, y) in enumerate(zip(x_positions, values_to_show)):
                if i % 3 == 0:  # Label every 3rd bar to avoid clutter
                    ax_monthly.text(x, y + np.max(values_to_show)*0.01, f'{y:.2f}',
                                    ha='center', va='bottom', fontsize=8, color='white', rotation=90)

            ax_monthly.set_title(f"Monthly Energy Consumption Breakdown - {country}\n(First {display_months//12} Years)",
                                 fontsize=14, color='white', pad=20)
            ax_monthly.set_xlabel("Month", color='white')
            ax_monthly.set_ylabel(f"Monthly {target_col}", color='white')

            # Set x-axis ticks for better readability
            tick_positions = [i*6 for i in range(len(months_to_show)//6 + 1) if i*6 < len(months_to_show)]
            ax_monthly.set_xticks(tick_positions)
            ax_monthly.set_xticklabels([months_to_show[i] for i in tick_positions], rotation=45, color='white')

            ax_monthly.grid(True, alpha=0.2, axis='y')
            ax_monthly.set_facecolor('#1a1a2e')
            fig_monthly.patch.set_facecolor('#1a1a2e')
            ax_monthly.tick_params(colors='white')
            plt.tight_layout()

            # Forecast text
            forecast_text = f"Forecast for {country} (next {years_ahead} years):\n"
            forecast_text += "Yearly Totals:\n"
            for i, val in enumerate(yearly_forecast, 1):
                forecast_text += f"Year {last_hist_year + i}: {val:.2f} TWh\n"

            forecast_text += f"\nMonthly Breakdown (Sample):\n"
            # Show first 6 months as sample
            for i in range(min(6, len(monthly_dates))):
                forecast_text += f"{monthly_dates[i]}: {monthly_values[i]:.3f} TWh\n"

            forecast_text += f"\nBased on {len(country_df)} years of historical data, lookback {lookback}"
            forecast_text += f"\nSeasonal pattern: High in Jan/Feb (winter) & Jul/Aug (summer)"

            return fig_yearly, fig_monthly, forecast_text

        except Exception as e:
            import traceback
            traceback.print_exc()
            return None, None, f"Error in forecasting: {e}"

    def perform_clustering(n_clusters=3):
        global global_df
        if global_df is None:
            return None, "Please load dataset first!"

        try:
            feature_cols = [
                'Per Capita Energy Use (kWh)', 'Renewable Energy Share (%)',
                'Fossil Fuel Dependency (%)', 'Industrial Energy Use (%)',
                'Household Energy Use (%)', 'Carbon Emissions (Million Tons)'
            ]

            df_sorted = global_df.sort_values('Year')
            agg_df = df_sorted.groupby('Country')[feature_cols].last()
            agg_df = agg_df.dropna()

            if len(agg_df) < n_clusters:
                return None, "Not enough complete country data to perform clustering."

            scaler = StandardScaler()
            X_scaled = scaler.fit_transform(agg_df.values)

            pca = PCA(n_components=2)
            X_pca = pca.fit_transform(X_scaled)
```

```
            kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
            clusters = kmeans.fit_predict(X_scaled)

            fig, ax = plt.subplots(figsize=(12, 8))
            scatter = ax.scatter(
                X_pca[:, 0], X_pca[:, 1],
                c=clusters, cmap='viridis', s=120, alpha=0.9, edgecolor='black'
            )

            from scipy.spatial import ConvexHull
            colors = plt.cm.viridis(np.linspace(0, 1, n_clusters))
            for cluster_id in range(n_clusters):
                pts = X_pca[clusters == cluster_id]
                if len(pts) >= 3:
                    hull = ConvexHull(pts)
                    hull_pts = pts[hull.vertices]
                    ax.fill(hull_pts[:, 0], hull_pts[:, 1], color=colors[cluster_id], alpha=0.18)
                    ax.plot(hull_pts[:, 0], hull_pts[:, 1], color=colors[cluster_id], linewidth=2)

            texts = [ax.text(X_pca[i, 0], X_pca[i, 1], country, fontsize=9)
                     for i, country in enumerate(agg_df.index)]
            adjust_text(texts, arrowprops=dict(arrowstyle='-', color='gray', lw=0.5), force_points=(0.2, 0.4))

            ax.set_title(f'Country Clustering (K-means, k={n_clusters})', fontsize=14)
            ax.set_xlabel('Principal Component 1 (Scaled)')
            ax.set_ylabel('Principal Component 2 (Scaled)')
            ax.grid(True, alpha=0.3)
            plt.tight_layout()

            cbar = plt.colorbar(scatter, ax=ax, label='Cluster')
            cbar.ax.tick_params(axis='y')

            agg_df['Cluster'] = clusters

            summary_text = "Cluster Summary (Mean Values):\n\n"
            cluster_summary = agg_df.groupby('Cluster')[feature_cols].mean().round(2)
            for cluster_id in sorted(cluster_summary.index):
                summary_text += f" Cluster {cluster_id}:\n"
                cluster_data = cluster_summary.loc[cluster_id]
                summary_text += f"  Countries: {len(agg_df[agg_df['Cluster'] == cluster_id])}\n"
                summary_text += f"  Per Capita Use: {cluster_data['Per Capita Energy Use (kWh)']} kWh\n"
                summary_text += f"  Renewable Share: {cluster_data['Renewable Energy Share (%)']}%\n"
                summary_text += f"  Fossil Fuel Share: {cluster_data['Fossil Fuel Dependency (%)']}%\n"
                summary_text += f"  Carbon Emissions: {cluster_data['Carbon Emissions (Million Tons)']} MT\n"
                summary_text += f"  Industrial Use: {cluster_data['Industrial Energy Use (%)']}%\n"
                summary_text += f"  Household Use: {cluster_data['Household Energy Use (%)']}%\n\n"

            return fig, summary_text

        except Exception as e:
            import traceback
            traceback.print_exc()
            return None, f"Error in clustering: {str(e)}"

    def detect_energy_alerts(country=None, sensitivity=2.0):
        global global_df

        if global_df is None:
            return None, "Please load dataset first!"

        try:
            target_col = 'Total Energy Consumption (TWh)'
            if target_col not in global_df.columns:
                return None, f"Target column '{target_col}' not found in dataset!"

            if country and country != "All Countries":
                df_filtered = global_df[global_df['Country'] == country].copy()
                if len(df_filtered) < 5:
                    return None, f"Not enough data for {country}. Need at least 5 years."
            else:
                df_filtered = global_df.copy()

            alerts = []

            countries_to_check = [country] if country and country != "All Countries" else df_filtered['Country'].unique()

            for current_country in countries_to_check:
                country_data = df_filtered[df_filtered['Country'] == current_country].sort_values('Year')

                if len(country_data) < 5:
                    continue
```

```
        energy_values = country_data[target_col].values
        years = country_data['Year'].values

        Q1 = np.percentile(energy_values, 25)
        Q3 = np.percentile(energy_values, 75)
        IQR = Q3 - Q1
        lower_bound = Q1 - sensitivity * IQR
        upper_bound = Q3 + sensitivity * IQR

        for i, (year, energy) in enumerate(zip(years, energy_values)):
            if energy > upper_bound:
                deviation = (energy - upper_bound) / IQR

                context = ""
                if i > 0:
                    prev_energy = energy_values[i-1]
                    change = ((energy - prev_energy) / prev_energy) * 100
                    context = f" (+{change:+.1f}% from previous year)"
                elif i < len(energy_values) - 1:
                    next_energy = energy_values[i+1]
                    change = ((energy - next_energy) / next_energy) * 100
                    context = f" (+{change:+.1f}% compared to next year)"

                alerts.append({
                    'country': current_country,
                    'year': int(year),
                    'energy': energy,
                    'upper_bound': upper_bound,
                    'deviation': deviation,
                    'context': context
                })

    alerts.sort(key=lambda x: x['deviation'], reverse=True)

    if not alerts:
        return None, f"No energy consumption alerts detected for {country if country != 'All Countries' else 'any countr

    fig, ax = plt.subplots(figsize=(12, 8))

    with plt.style.context('dark_background'):
        fig.set_facecolor('#1a1a2e')
        ax.set_facecolor('#1a1a2e')

        if country and country != "All Countries":
            country_data = df_filtered[df_filtered['Country'] == country].sort_values('Year')
            years = country_data['Year'].values
            energy = country_data[target_col].values

            ax.plot(years, energy, 'o-', color='cyan', linewidth=2, markersize=6, label=f'{country} Energy Consumption')

            alert_years = [alert['year'] for alert in alerts if alert['country'] == country]
            alert_energy = [alert['energy'] for alert in alerts if alert['country'] == country]
            ax.scatter(alert_years, alert_energy, color='red', s=150, zorder=5,
                       label='High Consumption Alerts', edgecolors='white', linewidth=2)

            if alerts:
                threshold = alerts[0]['upper_bound']
                ax.axhline(y=threshold, color='orange', linestyle='--', alpha=0.7,
                           label=f'Alert Threshold ({threshold:.1f} TWh)')

        else:
            top_alerts = alerts[:10]
            colors = plt.cm.Reds(np.linspace(0.5, 1, len(top_alerts)))

            for i, alert in enumerate(top_alerts):
                country_data = global_df[global_df['Country'] == alert['country']].sort_values('Year')
                years = country_data[target_col].values

                ax.plot(country_data['Year'].values, years, 'o-', alpha=0.3, linewidth=1, markersize=4)
                ax.scatter(alert['year'], alert['energy'], color=colors[i], s=120, zorder=5,
                           label=f"{alert['country']} ({alert['year']})")

    ax.set_title(f'Energy Consumption Alerts\n({country if country != "All Countries" else "Top 10 Countries"})',
                 fontsize=14, color='white', pad=20)
    ax.set_xlabel('Year', color='white', fontsize=12)
    ax.set_ylabel('Total Energy Consumption (TWh)', color='white', fontsize=12)
    ax.legend(facecolor='#1a1a2e', edgecolor='white', labelcolor='white')
    ax.grid(True, alpha=0.3)
    ax.tick_params(colors='white')

    plt.tight_layout()

    alert_text = f"ENERGY CONSUMPTION ALERTS\n\n"
```

```
            alert_text += f"Sensitivity: {sensitivity} (IQR multiplier)\n"
            alert_text += f"Total alerts detected: {len(alerts)}\n\n"

            for i, alert in enumerate(alerts[:15]):
                severity = "CRITICAL" if alert['deviation'] > 3 else "HIGH" if alert['deviation'] > 2 else "MEDIUM"
                alert_text += f"{i+1}. {alert['country']} ({alert['year']}): {alert['energy']:.1f} TWh {alert['context']}\n"
                alert_text += f"   Threshold: {alert['upper_bound']:.1f} TWh | Severity: {severity}\n\n"

            if len(alerts) > 15:
                alert_text += f"... and {len(alerts) - 15} more alerts\n"

            alert_text += f"\nRecommendation: Investigate economic growth, industrial activity, or data quality issues for these

            return fig, alert_text

        except Exception as e:
            import traceback
            traceback.print_exc()
            return None, f"Error detecting alerts: {str(e)}"

    def create_interface():
        custom_css = """
        .gradio-container {
            background: linear-gradient(135deg, #0c0c0c 0%, #1a1a2e 50%, #16213e 100%);
            color: white;
            font-family: 'Arial', sans-serif;
        }
        .gradio-container .tab-nav {
            background: rgba(255, 255, 255, 0.1) !important;
            backdrop-filter: blur(10px);
            border-radius: 10px;
            margin: 10px;
            padding: 10px;
        }
        .gradio-container .tab-nav button {
            background: transparent !important;
            color: white !important;
            border: none !important;
            padding: 12px 24px !important;
            margin: 5px !important;
            border-radius: 8px !important;
            transition: all 0.3s ease !important;
        }
        .gradio-container .tab-nav button:hover {
            background: rgba(0, 255, 255, 0.2) !important;
            transform: translateY(-2px);
        }
        .gradio-container .tab-nav button.selected {
            background: linear-gradient(45deg, #00ffff, #0080ff) !important;
            color: black !important;
            font-weight: bold;
            box-shadow: 0 0 20px #00ffff;
        }
        .gradio-button {
            background: linear-gradient(45deg, #ff00ff, #00ffff) !important;
            border: none !important;
            color: white !important;
            font-weight: bold !important;
            padding: 12px 30px !important;
            border-radius: 25px !important;
            transition: all 0.3s ease !important;
            box-shadow: 0 0 15px rgba(0, 255, 255, 0.5);
            margin: 10px 0px !important;
        }
        .gradio-button:hover {
            transform: translateY(-3px);
            box-shadow: 0 0 25px rgba(0, 255, 255, 0.8);
        }
        .gradio-plot {
            border-radius: 15px;
            background: rgba(255, 255, 255, 0.05) !important;
            backdrop-filter: blur(10px);
            padding: 20px;
            border: 1px solid rgba(255, 255, 255, 0.1) !important;
        }
        .gradio-textbox, .gradio-number, .gradio-dropdown {
            background: rgba(255, 255, 255, 0.1) !important;
            border: 1px solid rgba(255, 255, 255, 0.3) !important;
            color: white !important;
            border-radius: 10px !important;
            padding: 12px !important;
        }
        .gradio-textbox label, .gradio-number label, .gradio-dropdown label {
```

```
    color: #00ffff !important;
    font-weight: bold;
}
.gradio-markdown {
    color: white !important;
}
.gradio-container h1, .gradio-container h2, .gradio-container h3 {
    background: linear-gradient(45deg, #00ffff, #ff00ff);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
    text-align: center;
}
.plot-container {
    background: transparent !important;
}
"""

with gr.Blocks(theme=gr.themes.Soft(), css=custom_css) as demo:
    gr.Markdown(
        """
        # Energy Consumption Forecasting Dashboard
        ### Advanced Analytics with Machine Learning
        """
    )

    with gr.Tabs():
        with gr.TabItem("Data Loading"):
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 1: Upload Your Dataset")
                    file_input = gr.File(
                        label="Upload Energy Dataset (CSV)",
                        file_types=[".csv"],
                        type="filepath"
                    )
                    load_btn = gr.Button("Load Dataset", variant="primary", size="lg")
                with gr.Column():
                    gr.Markdown("### Status & Preview")
                    load_status = gr.Textbox(
                        label="Status",
                        interactive=False,
                        lines=2
                    )
                    data_preview = gr.HTML(
                        label="Data Preview",
                        value="<div style='color: white; text-align: center; padding: 20px;'>Upload a CSV file to begin<
                    )

            train_country_dropdown = gr.Dropdown(visible=False)
            forecast_country_dropdown = gr.Dropdown(visible=False)

            load_btn.click(
                load_dataset,
                inputs=[file_input],
                outputs=[load_status, data_preview, train_country_dropdown, forecast_country_dropdown]
            )

        with gr.TabItem("Preprocessing"):
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 2: Preprocess Data")
                    preprocess_btn = gr.Button("Preprocess Data", variant="primary", size="lg")
                with gr.Column():
                    gr.Markdown("### Processing Results")
                    preprocess_status = gr.Textbox(
                        label="Status",
                        interactive=False,
                        lines=2
                    )
                    missing_info = gr.Textbox(
                        label="Missing Values & Data Quality Report",
                        interactive=False,
                        lines=15
                    )

            with gr.Row():
                eda_plot = gr.Plot(
                    label="Global Energy Consumption Trend",
                    value=None
                )

        with gr.TabItem("Model Training"):
```

```
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 3: Train Prediction Models")
                    country_dropdown = gr.Dropdown(
                        label="Select Country",
                        choices=[],
                        interactive=True,
                        value=None
                    )
                    with gr.Row():
                        lookback_slider = gr.Slider(
                            minimum=3, maximum=10, value=5, step=1,
                            label="Lookback Window (years)"
                        )
                        units_slider = gr.Slider(
                            minimum=16, maximum=64, value=32, step=16,
                            label="LSTM Units"
                        )
                    train_btn = gr.Button("Train Models", variant="primary", size="lg")

                with gr.Column():
                    gr.Markdown("### Training Results")
                    train_status = gr.Textbox(
                        label="Status",
                        interactive=False,
                        lines=2
                    )
                    results_text = gr.Textbox(
                        label="Model Performance",
                        interactive=False,
                        lines=12
                    )

            with gr.Row():
                training_plot = gr.Plot(
                    label="Model Performance Comparison",
                    value=None
                )

        with gr.TabItem("Energy Alerts"):
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 4: Detect Energy Consumption Alerts")
                    alert_country_dropdown = gr.Dropdown(
                        label="Select Country (or 'All Countries' for overview)",
                        choices=[],
                        value="All Countries",
                        interactive=True
                    )
                    with gr.Row():
                        sensitivity_slider = gr.Slider(
                            minimum=1.0, maximum=3.0, value=2.0, step=0.1,
                            label="Alert Sensitivity (lower = more sensitive)"
                        )
                    alert_btn = gr.Button("Detect Alerts", variant="primary", size="lg")

                with gr.Column():
                    gr.Markdown("### Alert Results")
                    alert_status = gr.Textbox(
                        label="Alert Analysis",
                        interactive=False,
                        lines=15
                    )

            with gr.Row():
                alert_plot = gr.Plot(
                    label="Energy Consumption Alert Visualization",
                    value=None
                )

        # UPDATED FORECASTING TAB WITH MONTHLY BREAKDOWN
        with gr.TabItem("Forecasting"):
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 5: Generate Forecasts")
                    forecast_country = gr.Dropdown(
                        label="Select Country",
                        choices=[],
                        interactive=True,
                        value=None
                    )
                    forecast_years = gr.Slider(
                        minimum=1, maximum=10, value=5, step=1,
```

```
                        label="Years to Forecast"
                    )
                    forecast_btn = gr.Button("Generate Forecast", variant="primary", size="lg")

                with gr.Column():
                    gr.Markdown("### Forecast Results")
                    forecast_output = gr.Textbox(
                        label="Forecast Results",
                        interactive=False,
                        lines=8
                    )

            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Yearly Forecast")
                    forecast_plot_yearly = gr.Plot(
                        label="Yearly Energy Consumption Forecast",
                        value=None
                    )
                with gr.Column():
                    gr.Markdown("### Monthly Breakdown")
                    forecast_plot_monthly = gr.Plot(
                        label="Monthly Energy Consumption Forecast",
                        value=None
                    )

        with gr.TabItem("Country Clustering"):
            with gr.Row():
                with gr.Column():
                    gr.Markdown("### Step 6: Analyze Country Clusters")
                    cluster_slider = gr.Slider(
                        minimum=2, maximum=8, value=4, step=1,
                        label="Number of Clusters"
                    )
                    cluster_btn = gr.Button("Analyze Clusters", variant="primary", size="lg")

                with gr.Column():
                    gr.Markdown("### Cluster Analysis")
                    cluster_summary = gr.Textbox(
                        label="Cluster Summary",
                        interactive=False,
                        lines=15
                    )

            with gr.Row():
                cluster_plot = gr.Plot(
                    label="Country Clusters Visualization",
                    value=None
                )

    preprocess_btn.click(
        preprocess_wrapper,
        outputs=[preprocess_status, missing_info, eda_plot, country_dropdown, forecast_country, alert_country_dropdown]
    )

    train_btn.click(
        train_models,
        inputs=[country_dropdown, lookback_slider, units_slider],
        outputs=[train_status, training_plot, results_text]
    )

    alert_btn.click(
        detect_energy_alerts,
        inputs=[alert_country_dropdown, sensitivity_slider],
        outputs=[alert_plot, alert_status]
    )

    # UPDATED FORECAST BUTTON WITH MONTHLY BREAKDOWN
    forecast_btn.click(
        forecast_future_with_monthly,  # Using the new enhanced function
        inputs=[forecast_country, forecast_years],
        outputs=[forecast_plot_yearly, forecast_plot_monthly, forecast_output]  # Added monthly plot output
    )

    cluster_btn.click(
        perform_clustering,
        inputs=[cluster_slider],
        outputs=[cluster_plot, cluster_summary]
    )

    gr.Markdown(
        """
        ---
```

```
            ### Energy Forecasting System
            *Built with TensorFlow, Scikit-learn, and Gradio*
            *Features: LSTM Neural Networks, Linear Regression, K-means Clustering, PCA*

            ** Expected CSV Format:**
            - Country, Year, Total Energy Consumption (TWh), Per Capita Energy Use (kWh)
            - Renewable Energy Share (%), Fossil Fuel Dependency (%), Industrial Energy Use (%)
            - Household Energy Use (%), Carbon Emissions (Million Tons), Energy Price Index (USD/kWh)
            """
        )

    return demo

if __name__ == "__main__":
    print("Starting Energy Forecasting Dashboard...")
    print("Please upload your CSV file when prompted")
    print("The interface will open automatically")

    demo = create_interface()
    demo.launch(
        share=True,
        debug=True,
        show_error=True
    )
```

```
Collecting adjustText
  Downloading adjustText-1.3.0-py3-none-any.whl.metadata (3.1 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from adjustText) (2.0.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (from adjustText) (3.10.0)
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages (from adjustText) (1.16.3)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (1.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (0.12.1
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (4
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (1
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (25.
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText) (3.
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib->adjustText)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib->a
Downloading adjustText-1.3.0-py3-none-any.whl (13 kB)
Installing collected packages: adjustText
Successfully installed adjustText-1.3.0
```