import pandas as pd

import numpy as np

import os

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

from sklearn.neighbors import KNeighborsRegressor

from sklearn.cluster import KMeans

from sklearn.metrics import mean\_squared\_error, r2\_score, mean\_absolute\_error

import joblib

from io import BytesIO

class LandPricePredictor:

    def \_\_init\_\_(self, data\_path):

        self.data\_path = data\_path

        self.raw\_data = pd.read\_csv(data\_path)

        self.model\_dir = 'models'

        os.makedirs(self.model\_dir, exist\_ok=True)

        self.model\_paths = {

            'Linear Regression': os.path.join(self.model\_dir, 'linear\_regression.joblib'),

            'Random Forest': os.path.join(self.model\_dir, 'random\_forest.joblib'),

            'KNN Regressor': os.path.join(self.model\_dir, 'knn\_regressor.joblib')

        }

        self.columns\_path = os.path.join(self.model\_dir, 'train\_columns.joblib')

        self.linear\_model = None

        self.rf\_model = None

        self.knn\_model = None

        self.train\_columns = None

        self.kmeans\_model = None

        self.knn\_similarity\_model = None

        self.ensure\_model\_exists()

        self.load\_models()

    def preprocess\_data(self):

        """Prepare data for training"""

        data = self.raw\_data.copy()

        # Clean column names

        data.columns = data.columns.str.strip().str.lower().str.replace(' ', '\_')

        # Convert categorical variables

        data = pd.get\_dummies(data, columns=['Area', 'Land Use Type', 'Development Potential'])

        # Ensure target column name is consistent

        target\_col = 'Land Price (?/sq.ft)'

        if target\_col not in data.columns:

            target\_col = 'Land Price (?/sq.ft)'

        X = data.drop(target\_col, axis=1)

        y = data[target\_col]

        return train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    def train\_model(self):

        print("Training models...")

        X\_train, X\_test, y\_train, y\_test = self.preprocess\_data()

        models = {

            'Linear Regression': LinearRegression(),

            'Random Forest': RandomForestRegressor(n\_estimators=100, random\_state=42),

            'KNN Regressor': KNeighborsRegressor(n\_neighbors=5)

        }

        self.results = {}

        for model\_name, model in models.items():

            model.fit(X\_train, y\_train)

            y\_pred = model.predict(X\_test)

            mse = mean\_squared\_error(y\_test, y\_pred)

            r2 = r2\_score(y\_test, y\_pred)

            mae = mean\_absolute\_error(y\_test, y\_pred)

            self.results[model\_name] = {

                'MSE': mse,

                'R2 Score': r2,

                'MAE': mae

            }

            joblib.dump(model, self.model\_paths[model\_name])

            print(f"Trained {model\_name}: MSE={mse:.2f}, R2={r2:.4f}, MAE={mae:.2f}")

        joblib.dump(X\_train.columns.tolist(), self.columns\_path)

        print("Model training complete!")

    def ensure\_model\_exists(self):

        if not all(os.path.exists(path) for path in self.model\_paths.values()) or not os.path.exists(self.columns\_path):

            print("Models not found. Training new models...")

            self.train\_model()

    def load\_models(self):

        try:

            self.linear\_model = joblib.load(self.model\_paths['Linear Regression'])

            self.rf\_model = joblib.load(self.model\_paths['Random Forest'])

            self.knn\_model = joblib.load(self.model\_paths['KNN Regressor'])

            self.train\_columns = joblib.load(self.columns\_path)

        except Exception as e:

            raise RuntimeError(f"Error loading models: {str(e)}")

    def get\_target\_column(self):

        possible\_names = ['Land Price (?/sq.ft)', 'Land Price (?/sq.ft)']

        for name in possible\_names:

            if name in self.raw\_data.columns:

                return name

        raise ValueError("Could not find land price column in dataset")

    def predict\_for\_area(self, area\_name, prediction\_years=12):

      try:

        area\_name = area\_name.strip().title()

        if area\_name not in self.raw\_data['Area'].unique():

            raise ValueError(f"Area '{area\_name}' not found in dataset.")

        area\_row = self.raw\_data[self.raw\_data['Area'] == area\_name].iloc[0]

        current\_price = float(area\_row[self.get\_target\_column()])

        sample\_data = self.raw\_data[self.raw\_data['Area'] == area\_name].iloc[[0]].copy()

        # Preprocess the sample data

        sample\_processed = pd.get\_dummies(

            sample\_data,

            columns=['Area', 'Land Use Type', 'Development Potential']

        )

        # Ensure all training columns are present

        missing\_cols = set(self.train\_columns) - set(sample\_processed.columns)

        for col in missing\_cols:

            sample\_processed[col] = 0

        sample\_processed = sample\_processed[self.train\_columns]

        # Get predictions from each model

        lr\_pred = self.linear\_model.predict(sample\_processed)

        rf\_pred = self.rf\_model.predict(sample\_processed)

        knn\_pred = self.knn\_model.predict(sample\_processed)

        # Create predictions for all years

        lr\_preds = [round(float(lr\_pred[0]), 2)] \* prediction\_years

        rf\_preds = [round(float(rf\_pred[0]), 2)] \* prediction\_years

        knn\_preds = [round(float(knn\_pred[0]), 2)] \* prediction\_years

        # Calculate future predictions with inflation (4.8%)

        inflation\_rate = 0.048

        future\_predictions = [round(current\_price, 2)]

        for \_ in range(1, prediction\_years):

            future\_predictions.append(round(future\_predictions[-1] \* (1 + inflation\_rate), 2))

        return {

            'Linear Regression': lr\_preds,

            'Random Forest': rf\_preds,

            'KNN Regressor': knn\_preds,

            'Corrected Average with Inflation': future\_predictions,

            'Current Price': round(current\_price, 2)

        }

      except Exception as e:

        print(f"Prediction error: {str(e)}")

        raise

    def plot\_price\_comparison(self, years, current\_prices, future\_prices, area\_name):

     try:

        # Convert all inputs to numpy arrays and validate

        years = np.array(years, dtype=int)

        current\_prices = np.array(current\_prices, dtype=float)

        future\_prices = np.array(future\_prices, dtype=float)

        # Create current price array (constant value)

        current\_price\_array = np.full\_like(years, current\_prices[0], dtype=float)

        # Debugging: Print detailed array info

        print("\nDetailed Array Validation:")

        print(f"Years: {years.shape}, min={years.min()}, max={years.max()}, dtype={years.dtype}")

        print(f"Current Prices: {current\_prices.shape}, min={current\_prices.min()}, max={current\_prices.max()}, dtype={current\_prices.dtype}")

        print(f"Future Prices: {future\_prices.shape}, min={future\_prices.min()}, max={future\_prices.max()}, dtype={future\_prices.dtype}")

        print(f"Current Price Array: {current\_price\_array.shape}, min={current\_price\_array.min()}, max={current\_price\_array.max()}, dtype={current\_price\_array.dtype}")

        # Check for NaN/inf values

        for arr, name in zip([years, current\_prices, future\_prices],

                           ['years', 'current\_prices', 'future\_prices']):

            if np.any(np.isnan(arr)):

                raise ValueError(f"Array {name} contains NaN values")

            if np.any(np.isinf(arr)):

                raise ValueError(f"Array {name} contains infinite values")

        # Verify all arrays have exactly the same length

        lengths = {len(years), len(current\_prices), len(future\_prices), len(current\_price\_array)}

        if len(lengths) != 1:

            min\_len = min(len(years), len(current\_prices), len(future\_prices), len(current\_price\_array))

            print(f"Warning: Array length mismatch. Truncating to {min\_len} elements")

            years = years[:min\_len]

            current\_prices = current\_prices[:min\_len]

            future\_prices = future\_prices[:min\_len]

            current\_price\_array = current\_price\_array[:min\_len]

        # Create plot with tight layout

        plt.figure(figsize=(12, 7))

        # Plot data with explicit x and y values

        plt.plot(years, current\_prices, 'r--o', label='Current prices')

        plt.plot(years, future\_prices, 'g-^', label='With Inflation (4.8%)')

        # Plot formatting

        plt.xlabel('Year', fontsize=12)

        plt.ylabel('Price (₹/sq.ft)', fontsize=12)

        plt.title(f'Price Trend Prediction for {area\_name} ({years[0]}-{years[-1]})',

                 fontsize=14, pad=20)

        plt.legend()

        plt.grid(True, linestyle='--', alpha=0.7)

        # Add value annotations

        for i, year in enumerate(years):

            if year % 2 == 0 or year == years[-1]:  # Label every 2 years + last year

                plt.annotate(f'₹{current\_prices[i]:.0f}',

                             (year, current\_prices[i]),

                             textcoords="offset points",

                             xytext=(0, 10),

                             ha='center')

                plt.annotate(f'₹{future\_prices[i]:.0f}',

                             (year, future\_prices[i]),

                             textcoords="offset points",

                             xytext=(0, -15),

                             ha='center')

        plt.tight\_layout()

        # Save plot to bytes buffer

        buf = BytesIO()

        plt.savefig(buf, format='png', dpi=150, bbox\_inches='tight')

        buf.seek(0)

        plt.close()

        return buf

     except Exception as e:

        print(f"Detailed plotting error: {str(e)}")

        print(f"Error occurred with array shapes: years={len(years)}, current={len(current\_prices)}, future={len(future\_prices)}")

        raise

    def analyze\_data(self, area\_name):

       if area\_name not in self.raw\_data['Area'].unique():

         raise ValueError("Invalid area name! Choose from available areas in the dataset.")

       try:

        # Get the trained Random Forest model

        model = self.rf\_model

        # Create a sample row for the specified area

        area\_data = self.raw\_data[self.raw\_data['Area'] == area\_name].iloc[[0]].copy()

        # Preprocess the data exactly like during training

        processed\_data = pd.get\_dummies(

            area\_data,

            columns=['Area', 'Land Use Type', 'Development Potential']

        )

        # Ensure all training columns are present

        missing\_cols = set(self.train\_columns) - set(processed\_data.columns)

        for col in missing\_cols:

            processed\_data[col] = 0

        processed\_data = processed\_data[self.train\_columns]

        # Get feature importances

        feature\_importance = model.feature\_importances\_

        # Create DataFrame with feature names and importances

        feature\_importance\_df = pd.DataFrame({

            'Feature': self.train\_columns,

            'Importance': feature\_importance

        }).sort\_values(by='Importance', ascending=False)

        # Filter out zero-importance features and dummy variables

        feature\_importance\_df = feature\_importance\_df[

            (feature\_importance\_df['Importance'] > 0) &

            (~feature\_importance\_df['Feature'].str.startswith(('Area\_', 'Land Use Type\_', 'Development Potential\_')))

        ]

        # Get top 5 most important features

        top\_factors = feature\_importance\_df.head(5).to\_dict('records')

        print("\nFeature Importance Analysis:")

        print(f"Total features: {len(self.train\_columns)}")

        print(f"Non-zero features: {len(feature\_importance\_df)}")

        return sorted(top\_factors, key=lambda x: x['Importance'], reverse=True)

       except Exception as e:

        print(f"\nError in analyze\_data: {str(e)}")

        print(f"Features: {self.train\_columns}")

        print(f"Importances length: {len(model.feature\_importances\_)}")

        raise ValueError(f"Analysis failed: {str(e)}")

    def cluster\_areas(self, n\_clusters=5):

        """Cluster areas based on features"""

        X = pd.get\_dummies(self.raw\_data.drop(self.get\_target\_column(), axis=1), drop\_first=False)

        self.kmeans\_model = KMeans(n\_clusters=n\_clusters, random\_state=42, n\_init=10)

        clusters = self.kmeans\_model.fit\_predict(X)

        clustered\_df = self.raw\_data.copy()

        clustered\_df['Cluster'] = clusters

        print("\nClustering completed. Areas grouped into clusters:")

        for i in range(n\_clusters):

            cluster\_areas = clustered\_df[clustered\_df['Cluster'] == i]['Area'].unique()

            print(f"Cluster {i + 1}: {', '.join(cluster\_areas)}")

        return clustered\_df[['Area', 'Cluster']].drop\_duplicates()

    def find\_similar\_areas\_knn(self, area\_name, n\_neighbors=5):

     try:

        # Standardize column names

        df = self.raw\_data.copy()

        df.columns = df.columns.str.strip().str.lower().str.replace(' ', '\_')

        # Print unique area names for debugging

        print(f"Unique areas in dataset: {df['area'].unique()}")

        # Normalize the area name for comparison

        normalized\_area\_name = area\_name.strip().lower()

        # Check if area exists (case insensitive)

        if normalized\_area\_name not in df['area'].str.lower().unique():

            raise ValueError(f"Area '{area\_name}' not found in dataset.")

        # Get target column name (properly formatted)

        target\_col = 'land\_price\_(?/sq.ft)'.lower().replace(' ', '\_')

        if target\_col not in df.columns:

            raise ValueError("Could not find land price column in dataset")

        # Convert categorical columns to dummies

        categorical\_cols = ['area', 'land\_use\_type', 'development\_potential']

        # Create processed dataframe with dummy variables

        df\_processed = pd.get\_dummies(df, columns=categorical\_cols)

        # Verify target column exists before dropping

        if target\_col not in df\_processed.columns:

            raise ValueError(f"Target column '{target\_col}' not found in processed data")

        features = df\_processed.drop(target\_col, axis=1)

        # Initialize KNN model if not already done

        if self.knn\_similarity\_model is None:

            print("\nInitializing KNN similarity model...")

            self.knn\_similarity\_model = KNeighborsRegressor(n\_neighbors=n\_neighbors+1)

            self.knn\_similarity\_model.fit(features, df\_processed[target\_col])

            print("KNN similarity model trained successfully")

        # Get the row for the specified area

        area\_rows = df[df['area'].str.lower() == normalized\_area\_name]

        if area\_rows.empty:

            raise ValueError(f"No data found for area '{area\_name}'.")

        area\_row = area\_rows.iloc[[0]]  # Get the first matching row

        area\_processed = pd.get\_dummies(area\_row, columns=categorical\_cols)

        # Align columns with training features

        missing\_cols = set(features.columns) - set(area\_processed.columns)

        for col in missing\_cols:

            area\_processed[col] = 0

        area\_processed = area\_processed[features.columns]

        # Find similar areas

        distances, indices = self.knn\_similarity\_model.kneighbors(area\_processed)

        # Get similar areas (excluding the query area itself)

        similar\_indices = indices[0][1:]  # Skip first result (self)

        similar\_areas = df.iloc[similar\_indices]['area'].str.title().unique().tolist()

        return similar\_areas[:n\_neighbors]  # Return at most n\_neighbors areas

     except Exception as e:

        print(f"\nError finding similar areas: {str(e)}")

        print(f"Available columns in raw data: {list(df.columns)}")

        print(f"Available columns in processed data: {list(df\_processed.columns) if 'df\_processed' in locals() else 'Not processed'}")

        print(f"Target column used: {target\_col if 'target\_col' in locals() else 'Not determined'}")

        raise ValueError(f"Could not find similar areas: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Initialize predictor

    predictor = LandPricePredictor('data/land\_prediction\_Dataset.csv')

    # User input

    area = input("Enter the area to analyze: ").strip()

    try:

        # Get predictions for the specified area

        predictions = predictor.predict\_for\_area(area, prediction\_years=12)

        print(f"\nPredicted land prices for '{area.title()}':")

        for model, values in predictions.items():

            if model != 'Current Price':

                print(f"{model}: {values}")

        # Generate years for plotting

        years = list(range(2025, 2037))

        # Debugging: Print lengths of predictions

        print(f"\nLengths of predictions:")

        print(f"Random Forest Predictions Length: {len(predictions['Random Forest'])}")

        print(f"Corrected Average with Inflation Length: {len(predictions['Corrected Average with Inflation'])}")

        print(f"Years Length: {len(years)}")

        # Ensure all arrays have the same length before plotting

        if len(predictions['Random Forest']) != len(years) or len(predictions['Corrected Average with Inflation']) != len(years):

            raise ValueError("Prediction arrays and years array must be of the same length.")

        # Generate plot

        plot\_buf = predictor.plot\_price\_comparison(

            years,

            predictions['Random Forest'],

            predictions['Corrected Average with Inflation'],

            area.title()

        )

        # Save plot to file

        with open('price\_prediction.png', 'wb') as f:

            f.write(plot\_buf.getbuffer())

        print("\nSaved prediction plot as 'price\_prediction.png'")

        # Display key factors influencing land price

        analysis = predictor.analyze\_data(area.title())

        print("\nKey factors influencing land price:")

        for factor in analysis:

            print(f"- {factor['Feature']}: {factor['Importance']:.4f}")

        # Cluster analysis

        print("\nArea clustering:")

        predictor.cluster\_areas(n\_clusters=4)

        # Find similar areas

        similar = predictor.find\_similar\_areas\_knn(area)

        print(f"\nAreas similar to {area.title()}: {', '.join(similar)}")

    except Exception as e:

        print(f"\nError: {str(e)}")

\*\*  
@app.route('/predict', methods=['GET', 'POST'])

def predict():

    if 'user' not in session:

        flash("Please log in first", "danger")

        return redirect(url\_for('login'))

    if request.method == 'POST':

        area = request.form.get('area', '').strip()

        if not area:

            flash("Please enter an area name", "danger")

            return redirect(url\_for('predict'))

        try:

            predictions = predictor.predict\_for\_area(area, prediction\_years=12)

            current\_price = predictions.get('Current Price')

            logging.info("Predictions output: %s", predictions)

            # Validate the predictions output

            if not isinstance(predictions, dict):

                raise ValueError("Predictions must be a dictionary.")

            if current\_price is None:

                raise ValueError("Current Price not found in predictions.")

            required\_keys = ['Random Forest', 'Linear Regression', 'KNN Regressor', 'Corrected Average with Inflation']

            for key in required\_keys:

                if key not in predictions:

                    raise ValueError(f"{key} not found in predictions.")

            # Process factors

            factors = predictor.analyze\_data(area)

            descriptive\_factors = []

            for factor in factors:

                name = factor['Feature']

                desc = name  # Default description

                if name == '1 prev land price':

                    desc = 'Previous Year (2024) Price'

                elif name == '2 prev land price':

                    desc = 'Two Years Prior (2023) Price'

                elif name.isdigit() and len(name) == 4:

                    desc = f'{name} Price'

                descriptive\_factors.append({

                    'name': desc,

                    'importance': float(factor['Importance'])

                })

            # Create years list matching the prediction length

            prediction\_length = len(predictions['Corrected Average with Inflation'])

            years = list(range(2025, 2025 + prediction\_length))

            # Adjust lengths if necessary

            min\_length = min(len(years),

                             len(predictions['Random Forest']),

                             len(predictions['Corrected Average with Inflation']),

                             len(predictions['Linear Regression']),

                             len(predictions['KNN Regressor']))

            # Adjust all arrays to the minimum length

            years = years[:min\_length]

            predictions['Random Forest'] = predictions['Random Forest'][:min\_length]

            predictions['Corrected Average with Inflation'] = predictions['Corrected Average with Inflation'][:min\_length]

            predictions['Linear Regression'] = predictions['Linear Regression'][:min\_length]

            predictions['KNN Regressor'] = predictions['KNN Regressor'][:min\_length]

            # Debugging: Check for NaN values

            if any(np.isnan(predictions['Random Forest'])) or any(np.isnan(predictions['Corrected Average with Inflation'])):

                raise ValueError("One or more prediction arrays contain NaN values.")

            # Generate plot

            plot\_buf = None

            try:

                plot\_buf = predictor.plot\_price\_comparison(

                    years,

                    predictions['Random Forest'],

                    predictions['Corrected Average with Inflation'],

                    area

                )

            except Exception as plot\_error:

                logging.error("Plot generation failed: %s", plot\_error)

            # Prepare data for template, including Current Price

            prediction\_data = {

                'area': area,

                'years': years,

                'current\_price': current\_price,

                'predictions': [

                    {

                        'name': 'Linear Regression',

                        'values': [float(x) for x in predictions['Linear Regression']],

                        'color': '#3498db'

                    },

                    {

                        'name': 'Random Forest',

                        'values': [float(x) for x in predictions['Random Forest']],

                        'color': '#e74c3c'

                    },

                    {

                        'name': 'KNN Regressor',

                        'values': [float(x) for x in predictions['KNN Regressor']],

                        'color': '#f39c12'

                    },

                    {

                        'name': 'Corrected Average with Inflation',

                        'values': [float(x) for x in predictions['Corrected Average with Inflation']],

                        'color': '#27ae60'

                    }

                ],

                'factors': descriptive\_factors,

                'plot\_data': None

            }

            if plot\_buf:

                try:

                    plot\_data = base64.b64encode(plot\_buf.getvalue()).decode('utf-8')

                    prediction\_data['plot\_data'] = plot\_data

                except Exception as e:

                    logging.error("Plot encoding failed: %s", e)

            return render\_template('result.html', data=prediction\_data)

        except Exception as e:

            logging.error("Prediction error: %s", e)

            flash(f"Error generating predictions: {str(e)}", "danger")

            return redirect(url\_for('predict'))

    return render\_template('predict.html')

\*\*\*\*search  
@app.route('/search', methods=['GET', 'POST'])

def search():

if 'user' not in session:

return redirect(url\_for('login'))

if request.method == 'POST':

area = request.form.get("area", "").strip()

land\_type = request.form.get("land\_type", "").strip()

min\_price = request.form.get("min\_price", type=float, default=0)

max\_price = request.form.get("max\_price", type=float, default=float("inf"))

water\_table\_level = request.form.get("water\_table", type=float, default=None)

filtered\_results = [

land for land in land\_data

if (not area or land["Area"].lower() == area.lower()) and

(not land\_type or land["Land Use Type"].lower() == land\_type.lower()) and

(min\_price <= land["Land Price (?/sq.ft)"] <= max\_price) and

(water\_table\_level is None or land["Water Table Depth (m)"] <= water\_table\_level)

return render\_template('search\_results.html', results=filtered\_results)

return render\_template('search.html')

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*predict.html  
<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <meta name="viewport" content="width=device-width, initial-scale=1.0">

   <title>Predict Land Prices</title>

   <link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}">

   <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.2/css/all.min.css">

   <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

   <style>

       body {

           background-image: url('https://png.pngtree.com/thumb\_back/fh260/background/20190221/ourmid/pngtree-light-green-green-light-green-bloom-image\_11709.jpg');

           background-size: cover;

           font-family: 'Roboto', sans-serif;

           color: #f0fff0;

           margin: 0;

           padding: 0;

       }

       header {

           background-color: #00703C;

           padding: 20px;

           text-align: center;

           font-size: 24px;

           color: #f0fff0;

       }

       nav a {

           color: #f0fff0;

           text-decoration: none;

           margin: 0 10px;

           font-size: 18px;

       }

       main {

           text-align: center;

           padding: 50px 20px;

       }

       form {

           background-color: rgba(0, 112, 60, 0.8);

           padding: 20px;

           border-radius: 10px;

           display: inline-block;

       }

       label, input {

           font-size: 18px;

           color: #fff;

           margin-bottom: 10px;

           display: block;

       }

       input {

           padding: 10px;

           width: 80%;

           border: none;

           border-radius: 5px;

           color: black;

           background-color: white;

       }

       button {

           background-color: rgb(97, 219, 152);

           color: #fff;

           border: none;

           padding: 10px 20px;

           font-size: 16px;

           cursor: pointer;

           border-radius: 5px;

       }

       button:hover {

           background-color: rgb(15, 175, 100);

       }

       ul {

           list-style: none;

           padding: 0;

       }

       ul li {

           background-color: rgba(0, 112, 60, 0.8);

           padding: 10px;

           margin: 5px;

           border-radius: 5px;

           font-size: 18px;

       }

       h3.prediction-title {

           color: rgb(15, 175, 100);

           font-size: 22px;

           margin-top: 20px;

       }

       canvas {

           background-color: white;

           border-radius: 10px;

           margin-top: 30px;

           padding: 10px;

       }

       footer {

           background-color: #00703C;

           padding: 10px;

           text-align: center;

           color: #f0fff0;

       }

   </style>

</head>

<body>

   <header>

       <h1>Explore Future Land Price Trends</h1>

       <nav>

           <a href="{{ url\_for('home') }}"><i class="fas fa-home"></i> Home</a>

       </nav>

   </header>

   <main>

    <form method="POST" action="{{ url\_for('predict') }}">

        <label for="area">Enter Area Name:</label>

        <input type="text" id="area" name="area" required>

        <button type="submit">Predict</button>

    </form>

       {% if error %}

           <p style="color: red;">{{ error }}</p>

       {% endif %}

       {% if inflated\_prices %}

           <h3 class="prediction-title">Predicted Prices for {{ area }} (Next 5 Years)</h3>

           <ul>

               {% set current\_year = 2025 %}

               {% for price in inflated\_prices %}

                   <li>Year {{ current\_year }}: ₹{{ price | round(2) }} /sq.ft</li>

                   {% set current\_year = current\_year + 1 %}

               {% endfor %}

           </ul>

           <canvas id="priceChart" width="600" height="300"></canvas>

           <script>

               const years = {{ years | tojson }};

               const values = {{ inflated\_prices | tojson }};

               const ctx = document.getElementById('priceChart').getContext('2d');

               new Chart(ctx, {

                   type: 'line',

                   data: {

                       labels: years.map(y => 'Year ' + y),

                       datasets: [{

                           label: 'Predicted Price (₹)',

                           data: values,

                           borderColor: 'rgba(75, 192, 192, 1)',

                           backgroundColor: 'rgba(75, 192, 192, 0.1)',

                           fill: true,

                           tension: 0.3

                       }]

                   },

                   options: {

                       responsive: true,

                       scales: {

                           y: {

                               beginAtZero: false,

                               title: { display: true, text: 'Price in ₹' }

                           },

                           x: {

                               title: { display: true, text: 'Years' }

                           }

                       }

                   }

               });

           </script>

       {% endif %}

       {% if lr\_predictions and rf\_predictions and knn\_predictions %}

           <h3 class="prediction-title">Model-wise Predictions</h3>

           <ul>

               <li><strong>Linear Regression:</strong> ₹{{ lr\_predictions | round(2) }} /sq.ft</li>

               <li><strong>Random Forest:</strong> ₹{{ rf\_predictions | round(2) }} /sq.ft</li>

               <li><strong>KNN Regressor:</strong> ₹{{ knn\_predictions | round(2) }} /sq.ft</li>

           </ul>

       {% endif %}

       {% if predictions %}

           <h3 class="prediction-title">Detailed Prediction for {{ area }}</h3>

           <p><strong>Predicted Price:</strong> ₹{{ predictions.predicted\_price | round(2) }} /sq.ft</p>

           <p><strong>Cluster:</strong> {{ predictions.cluster }}</p>

           {% if area %}

               <img src="{{ url\_for('static', filename=area + '\_trend.png') }}" alt="Trend Plot" style="margin-top: 20px; max-width: 80%; border-radius: 10px;">

           {% endif %}

       {% endif %}

   </main>

   <script>

    document.querySelector('form').addEventListener('submit', function(e) {

        const areaInput = document.getElementById('area');

        if (!areaInput.value.trim()) {

            e.preventDefault();

            alert('Please enter an area name');

            areaInput.focus();

        }

    });

</script>

   <footer>

       <p>&copy; 2025 Land Price Prediction. All rights reserved.</p>

   </footer>

</body>

</html>