

Land_price_prediction_16072025

July 16, 2025

1 AI-Driven Agents for Fair and Efficient Property Valuation

- **Student Name:** Fahad Abubaker Bahashwan
- **Advisor:** Dr. Ashraf Elsayed
- **Institution:** MidOcean University
- **Date:** 16, July 2025

What we'll do: 1. Load and explore your data 2. Create geographic and time-based features 3. Train different ML models 4. Compare which model works best 5. Save the best model for future use

1.1 Step 1: Import Libraries

First, let's import all the tools we need:

```
[10]: # Data handling
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# XGBoost (advanced model)
try:
    from xgboost import XGBRegressor
    print(" XGBoost available")
except ImportError:
    print(" XGBoost not available, will skip this model")
```

```

# Model saving
import joblib

# Settings
plt.style.use('default')
plt.rcParams['figure.figsize'] = (10, 6)
np.random.seed(42)
plt.rcParams['font.family'] = 'Segoe UI Emoji'
print(" All libraries loaded successfully!")

```

XGBoost available
All libraries loaded successfully!

1.2 Step 2: Load Data

```

[11]: # Load the Excel file
print(" Loading data...")
df = pd.read_excel('WorkedData3.xlsx')

# Focus on geographic and time data only
df_geo = df[['latitude', 'longitude', 'transactionDate', 'MeterPrice']].copy()

print(f" Data loaded successfully!")
print(f" We have {len(df_geo):,} land transactions")
print(f" Date range: {df_geo['transactionDate'].min().strftime('%Y-%m-%d')} to {df_geo['transactionDate'].max().strftime('%Y-%m-%d')}")
print(f" Price range: {df_geo['MeterPrice'].min():,} to {df_geo['MeterPrice'].max():,} per m²")

# Show first few rows
print("\n First 5 transactions:")
display(df_geo.head())

# Basic info about the data
print("\n Data Summary:")
display(df_geo.describe())

```

Loading data..
Data loaded successfully!
We have 2,133 land transactions
Date range: 2024-06-11 to 2025-06-02
Price range: 56 to 33,907 per m²

First 5 transactions:

	latitude	longitude	transactionDate	MeterPrice
0	21.513147	39.229104	2024-06-11	8000
1	21.521520	39.225639	2024-06-11	1070

2	21.819456	39.083658	2024-06-11	2116
3	21.574636	39.154524	2024-06-11	8215
4	21.768730	39.157569	2024-06-11	5833

Data Summary:

	latitude	longitude	transactionDate	MeterPrice
count	2133.000000	2133.000000	2133	2133.000000
mean	21.703143	39.154693	2024-12-12 01:08:11.139240448	3900.627286
min	21.305824	39.039992	2024-06-11 00:00:00	56.000000
25%	21.581512	39.100728	2024-09-25 00:00:00	1800.000000
50%	21.761534	39.150505	2024-12-11 00:00:00	2950.000000
75%	21.827729	39.194508	2025-02-24 00:00:00	5335.000000
max	21.990680	39.342914	2025-06-02 00:00:00	33907.000000
std	0.137826	0.059448	NaN	2941.222128

1.2.1 Step 3: Create Geographic & Time Features

Transform location coordinates and dates into meaningful features for ML:

```
[12]: def create_geographic_features(data, fitted_kmeans=None, reference_stats=None):
    """
    Create features from geographic coordinates and time data only

    Parameters:
    - data: DataFrame with latitude, longitude, transactionDate, MeterPrice
    - fitted_kmeans: Pre-fitted KMeans model (for prediction)
    - reference_stats: Dictionary with reference statistics (for prediction)
    """
    print(" Creating geographic and time features...")

    df_features = data.copy()

    # === TIME FEATURES ===
    print(" Creating time features...")
    df_features['year'] = df_features['transactionDate'].dt.year
    df_features['month'] = df_features['transactionDate'].dt.month
    df_features['quarter'] = df_features['transactionDate'].dt.quarter
    df_features['day_of_year'] = df_features['transactionDate'].dt.dayofyear

    # Continuous time features
    if reference_stats is None:
        start_date = df_features['transactionDate'].min()
    else:
        start_date = reference_stats['start_date']

    df_features['days_since_start'] = (df_features['transactionDate'] -
    ↪ start_date).dt.days
```

```

df_features['years_since_start'] = df_features['days_since_start'] / 365.25

# Cyclical time features (capture seasonality)
df_features['month_sin'] = np.sin(2 * np.pi * df_features['month'] / 12)
df_features['month_cos'] = np.cos(2 * np.pi * df_features['month'] / 12)

# === GEOGRAPHIC FEATURES ===
print(" Creating geographic features...")

# Basic coordinate features # 21.538796889565933, 39.197197060899846 -␣
↪Jeddah Center Point
if reference_stats is None:
    lat_center = 21.538796889565933
    lon_center = 39.197197060899846
    lat_min, lat_max = df_features['latitude'].min(),␣
↪df_features['latitude'].max()
    lon_min, lon_max = df_features['longitude'].min(),␣
↪df_features['longitude'].max()
else:
    lat_center = reference_stats['lat_center']
    lon_center = reference_stats['lon_center']
    lat_min, lat_max = reference_stats['lat_min'],␣
↪reference_stats['lat_max']
    lon_min, lon_max = reference_stats['lon_min'],␣
↪reference_stats['lon_max']

# Distance from geographic center -- Euclidean Distance -- formula
df_features['distance_from_center'] = np.sqrt(
    (df_features['latitude'] - lat_center)**2 +
    (df_features['longitude'] - lon_center)**2
)

# Normalized coordinates (0-1 scale)
df_features['lat_normalized'] = (df_features['latitude'] - lat_min) /␣
↪(lat_max - lat_min)
df_features['lon_normalized'] = (df_features['longitude'] - lon_min) /␣
↪(lon_max - lon_min)

# Distance from corners (captures edge effects)
df_features['distance_from_ne'] = np.sqrt(
    (df_features['latitude'] - lat_max)**2 + (df_features['longitude'] -␣
↪lon_max)**2
)
df_features['distance_from_sw'] = np.sqrt(
    (df_features['latitude'] - lat_min)**2 + (df_features['longitude'] -␣
↪lon_min)**2

```

```

)

# === LOCATION CLUSTERS ===
print(" Creating location clusters...")

if fitted_kmeans is None:
    # Create geographic clusters
    kmeans = KMeans(n_clusters=8, random_state=42, n_init=10)
    df_features['geo_cluster'] = kmeans.fit_predict(
        df_features[['latitude', 'longitude']]
    )
else:
    # Use pre-fitted clusters
    df_features['geo_cluster'] = fitted_kmeans.predict(
        df_features[['latitude', 'longitude']]
    )
    kmeans = fitted_kmeans

# Distance to cluster center
cluster_centers = kmeans.cluster_centers_
df_features['distance_to_cluster_center'] = [
    np.sqrt((lat - cluster_centers[cluster, 0])**2 +
            (lon - cluster_centers[cluster, 1])**2)
    for lat, lon, cluster in zip(
        df_features['latitude'],
        df_features['longitude'],
        df_features['geo_cluster']
    )
]

# === INTERACTION FEATURES ===
print(" Creating interaction features...")

# Time-location interactions
df_features['year_lat_interaction'] = df_features['year'] *
↳ df_features['lat_normalized']
df_features['year_lon_interaction'] = df_features['year'] *
↳ df_features['lon_normalized']

# Geographic ratios
df_features['lat_lon_ratio'] = df_features['lat_normalized'] /
↳ (df_features['lon_normalized'] + 0.001)

# === MARKET TREND FEATURES ===
print(" Creating market features...")

if reference_stats is None:

```

```

        # Calculate moving averages for training data
        df_sorted = df_features.sort_values('transactionDate')
        df_sorted['price_ma_30'] = df_sorted['MeterPrice'].rolling(window=30,
↪min_periods=1).mean()
        df_sorted['price_ma_90'] = df_sorted['MeterPrice'].rolling(window=90,
↪min_periods=1).mean()

        # Merge back
        df_features = df_features.merge(
            df_sorted[['transactionDate', 'latitude', 'longitude',
↪'price_ma_30', 'price_ma_90']],
            on=['transactionDate', 'latitude', 'longitude'],
            how='left'
        )

        overall_avg_price = df_features['MeterPrice'].mean()
    else:
        # For prediction, use overall average
        overall_avg_price = reference_stats['overall_avg_price']
        df_features['price_ma_30'] = overall_avg_price
        df_features['price_ma_90'] = overall_avg_price

    # === CLEAN UP ===
    # Remove original columns
    columns_to_drop = ['transactionDate', 'latitude', 'longitude']
    df_final = df_features.drop(columns=columns_to_drop)

    # One-hot encode clusters
    df_final = pd.get_dummies(df_final, columns=['geo_cluster'],
↪prefix='cluster', drop_first=True)

    print(f" Feature creation complete!")
    print(f" Features created: {len(df_final.columns) - 1}")

    # Return reference stats for prediction
    if reference_stats is None:
        reference_stats = {
            'start_date': start_date,
            'lat_center': lat_center,
            'lon_center': lon_center,
            'lat_min': lat_min,
            'lat_max': lat_max,
            'lon_min': lon_min,
            'lon_max': lon_max,
            'overall_avg_price': overall_avg_price
        }
    return df_final, kmeans, reference_stats

```

```

    else:
        return df_final

# Create features from your geographic data
df_with_features, fitted_kmeans, reference_stats = _
    ↪ create_geographic_features(df_geo)

# Show the new features
print("\n Your data now has these geographic features:")
feature_names = [col for col in df_with_features.columns if col != 'MeterPrice']
for i, feature in enumerate(feature_names, 1):
    print(f"{i:2d}. {feature}")

print("\n Sample of enhanced data:")
display(df_with_features.head())

```

Creating geographic and time features...

Creating time features...

Creating geographic features...

Creating location clusters...

Creating interaction features...

Creating market features...

Feature creation complete!

Features created: 26

Your data now has these geographic features:

1. year
2. month
3. quarter
4. day_of_year
5. days_since_start
6. years_since_start
7. month_sin
8. month_cos
9. distance_from_center
10. lat_normalized
11. lon_normalized
12. distance_from_ne
13. distance_from_sw
14. distance_to_cluster_center
15. year_lat_interaction
16. year_lon_interaction
17. lat_lon_ratio
18. price_ma_30
19. price_ma_90
20. cluster_1
21. cluster_2
22. cluster_3

```

23. cluster_4
24. cluster_5
25. cluster_6
26. cluster_7

```

Sample of enhanced data:

	MeterPrice	year	month	quarter	day_of_year	days_since_start	\
0	8000	2024	6	2	163	0	
1	1070	2024	6	2	163	0	
2	2116	2024	6	2	163	0	
3	8215	2024	6	2	163	0	
4	5833	2024	6	2	163	0	

	years_since_start	month_sin	month_cos	distance_from_center	...	\
0	0.0	1.224647e-16	-1.0	0.040939	...	
1	0.0	1.224647e-16	-1.0	0.033278	...	
2	0.0	1.224647e-16	-1.0	0.302755	...	
3	0.0	1.224647e-16	-1.0	0.055727	...	
4	0.0	1.224647e-16	-1.0	0.233323	...	

	lat_lon_ratio	price_ma_30	price_ma_90	cluster_1	cluster_2	cluster_3	\
0	0.484133	8000.000000	8000.000000	False	False	False	
1	0.513071	4535.000000	4535.000000	False	False	False	
2	5.166970	3728.666667	3728.666667	True	False	False	
3	1.035398	4850.250000	4850.250000	False	False	False	
4	1.736935	5046.800000	5046.800000	False	True	False	

	cluster_4	cluster_5	cluster_6	cluster_7
0	False	True	False	False
1	False	True	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False

[5 rows x 27 columns]

1.3 Step 4: Prepare Data for Machine Learning

Split the data into training and testing sets:

```

[17]: # Separate features (X) and target (y)
X = df_with_features.drop('MeterPrice', axis=1) # Features
y = df_with_features['MeterPrice']             # Target

print(f" Target variable (MeterPrice):")
print(f" Average: {y.mean():.0f}")
print(f" Range: {y.min():,} to {y.max():,}")

```

```

# Split data: 80% for training, 20% for testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(f"\n Data split:")
print(f"    Training: {len(X_train):,} transactions ({len(X_train)/len(X)*100:.
    ↪0f}%)")
print(f"    Testing:  {len(X_test):,} transactions ({len(X_test)/len(X)*100:.
    ↪0f}%)")

# Scale features
print("\n Scaling features...")
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Convert back to DataFrames
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns,
    ↪index=X_train.index)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns,
    ↪index=X_test.index)

print(" Data preparation complete!")

```

Target variable (MeterPrice):

Average: 3898

Range: 56 to 33,907

Data split:

Training: 1,714 transactions (80%)

Testing: 429 transactions (20%)

Scaling features...

Data preparation complete!

1.4 Step 5: Train Machine Learning Models

Train different models using only geographic and time features:

```

[26]: # Define models to test
models_to_test = {
    "Linear Regression": {
        "model": LinearRegression(),
        "use_scaled": True,
        "description": "Simple linear model (baseline)"
    },

```

```

    "Random Forest": {
        "model": RandomForestRegressor(n_estimators=100, random_state=42,
↪n_jobs=-1),
        "use_scaled": False,
        "description": "Ensemble of decision trees"
    }
}

# Add XGBoost if available
try:
    models_to_test["XGBoost"] = {
        "model": XGBRegressor(n_estimators=100, random_state=42, n_jobs=-1),
        "use_scaled": False,
        "description": "Advanced gradient boosting"
    }
except:
    print(" XGBoost not available, skipping...")

# Train and evaluate models
trained_models = {}
model_results = []

print(" Training models on geographic data only...\n")

for name, config in models_to_test.items():
    print(f" Training {name}...")
    print(f"    {config['description']}")

    # Choose scaled or original data
    if config['use_scaled']:
        X_train_model = X_train_scaled
        X_test_model = X_test_scaled
        print(f"    Using: Scaled features")
    else:
        X_train_model = X_train
        X_test_model = X_test
        print(f"    Using: Original features")

    # Train the model
    model = config['model']
    model.fit(X_train_model, y_train)

    # Make predictions
    y_pred = model.predict(X_test_model)

    # Calculate performance metrics
    mae = mean_absolute_error(y_test, y_pred)

```

```

mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Store results
model_results.append({
    'Model': name,
    'MAE': mae,
    'RMSE': rmse,
    'R²': r2,
    'Description': config['description']
})

trained_models[name] = {
    'model': model,
    'use_scaled': config['use_scaled'],
    'predictions': y_pred
}

print(f"    R² Score: {r2:.4f} (higher is better)")
print(f"    MAE: {mae:.0f} (lower is better)")
print()

print(" All models trained using geographic features only!")

```

Training models on geographic data only...

Training Linear Regression...

Simple linear model (baseline)

Using: Scaled features

R² Score: 0.4825 (higher is better)

MAE: 1307 (lower is better)

Training Random Forest...

Ensemble of decision trees

Using: Original features

R² Score: 0.5960 (higher is better)

MAE: 1022 (lower is better)

Training XGBoost...

Advanced gradient boosting

Using: Original features

R² Score: 0.5025 (higher is better)

MAE: 1118 (lower is better)

All models trained using geographic features only!

1.5 Step 6: Compare Model Performance

See which model performs best with geographic features:

```
[27]: # Create results table
results_df = pd.DataFrame(model_results)
results_df = results_df.sort_values('R2', ascending=False)

print(" MODEL PERFORMANCE (Geographic Features Only)")
print("=" * 60)
print("\n Results (sorted by R2 score):")

# Display results nicely
for idx, row in results_df.iterrows():
    print(f"\n{idx+1}. {row['Model']}")
    print(f"      R2 Score: {row['R2']:.4f} (explains {row['R2']*100:.1f}% of ↪
    price variation)")
    print(f"      Average Error: {row['MAE']:.0f} per m2")
    print(f"      RMSE: {row['RMSE']:.0f}")
    print(f"      {row['Description']}")

# Find best model
best_model_name = results_df.iloc[0]['Model']
best_r2 = results_df.iloc[0]['R2']
best_mae = results_df.iloc[0]['MAE']

print(f"\n WINNER: {best_model_name}")
print(f"      Accuracy: {best_r2:.1%} (using only coordinates & time)")
print(f"      Typical error: {best_mae:.0f} per m2")

# Visualize model comparison
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# R2 comparison
bars1 = ax1.bar(results_df['Model'], results_df['R2'],
                color=['gold' if x == best_model_name else 'lightblue' for x in ↪
                results_df['Model']])
ax1.set_title(' Model Accuracy (Geographic Features Only)')
ax1.set_ylabel('R2 Score (higher = better)')
ax1.set_ylim(0, 1)
ax1.grid(True, alpha=0.3)

# Add value labels
for bar, value in zip(bars1, results_df['R2']):
    ax1.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
             f'{value:.3f}', ha='center', va='bottom', fontweight='bold')

# MAE comparison
```

```

bars2 = ax2.bar(results_df['Model'], results_df['MAE'],
                color=['gold' if x == best_model_name else 'lightcoral' for x in
                ↪results_df['Model']])
ax2.set_title(' Prediction Error (Geographic Features Only)')
ax2.set_ylabel('Mean Absolute Error (lower = better)')
ax2.grid(True, alpha=0.3)

# Add value labels
for bar, value in zip(bars2, results_df['MAE']):
    ax2.text(bar.get_x() + bar.get_width()/2, bar.get_height() +
    ↪max(results_df['MAE'])*0.01,
            f'{value:.0f}', ha='center', va='bottom', fontweight='bold')

plt.tight_layout()
plt.show()

```

MODEL PERFORMANCE (Geographic Features Only)

=====

Results (sorted by R^2 score):

2. Random Forest

R^2 Score: 0.5960 (explains 59.6% of price variation)

Average Error: 1022 per m^2

RMSE: 1874

Ensemble of decision trees

3. XGBoost

R^2 Score: 0.5025 (explains 50.2% of price variation)

Average Error: 1118 per m^2

RMSE: 2079

Advanced gradient boosting

1. Linear Regression

R^2 Score: 0.4825 (explains 48.2% of price variation)

Average Error: 1307 per m^2

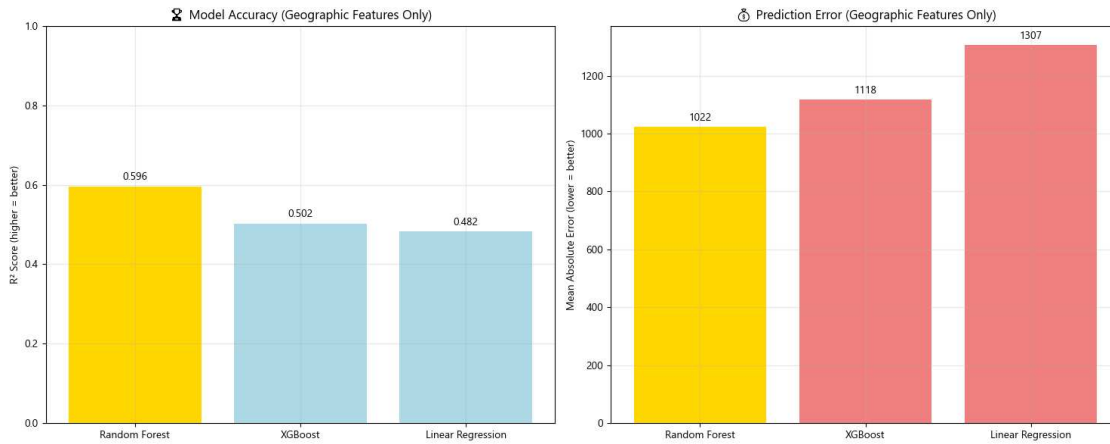
RMSE: 2121

Simple linear model (baseline)

WINNER: Random Forest

Accuracy: 59.6% (using only coordinates & time)

Typical error: 1022 per m^2



1.6 Step 7: Save the Geographic Model

Save model for predicting prices using only coordinates and time:

```
[28]: # Get predictions from best model
best_predictions = trained_models[best_model_name]['predictions']

# Create a complete model package
model_package = {
    'model': trained_models[best_model_name]['model'],
    'scaler': scaler if trained_models[best_model_name]['use_scaled'] else None,
    'feature_names': X_train.columns.tolist(),
    'model_name': best_model_name,
    'fitted_kmeans': fitted_kmeans,
    'reference_stats': reference_stats,
    'performance': {
        'r2_score': best_r2,
        'mae': best_mae,
        'rmse': results_df[results_df['Model'] == best_model_name]['RMSE'].
        iloc[0]
    },
    'training_info': {
        'training_samples': len(X_train),
        'test_samples': len(X_test),
        'total_features': len(X_train.columns),
        'date_trained': pd.Timestamp.now().strftime('%Y-%m-%d %H:%M:%S'),
        'features_used': 'Geographic coordinates and time only (no district_id)'
    }
}

# Save the model
```

```

model_filename = f'geographic_land_price_model_{best_model_name.lower()}.
↳replace(" ", "_").pkl'
joblib.dump(model_package, model_filename)

print(f" Geographic model saved as: {model_filename}")
print(f"\n Model Package Contains:")
print(f"     • Trained {best_model_name} model")
print(f"     • Feature scaler {'(included)' if model_package['scaler'] is not_
↳None else '(not needed)'}")
print(f"     • List of {len(model_package['feature_names'])} feature names")
print(f"     • Pre-fitted geographic clustering model")
print(f"     • Reference statistics for feature engineering")
print(f"     • Performance metrics (R2: {best_r2:.3f}, MAE: {best_mae:.0f})")
print(f"     • Training metadata")

# Create prediction function for geographic coordinates only
def predict_land_price_geographic(latitude, longitude, transaction_date,
                                  model_file=model_filename):
    """
    Predict land price using ONLY geographic coordinates and time

    Parameters:
    - latitude: Latitude coordinate
    - longitude: Longitude coordinate
    - transaction_date: Date as string 'YYYY-MM-DD'

    Returns:
    - Predicted price per square meter
    """

    # Load the saved model package
    package = joblib.load(model_file)

    # Create input data (no district_id needed!)
    input_data = pd.DataFrame({
        'latitude': [latitude],
        'longitude': [longitude],
        'transactionDate': [pd.to_datetime(transaction_date)],
        'MeterPrice': [0] # Placeholder
    })

    # Apply same feature engineering
    input_features = create_geographic_features(
        input_data,
        fitted_kmeans=package['fitted_kmeans'],
        reference_stats=package['reference_stats']
    )

```

```

input_features = input_features.drop('MeterPrice', axis=1)

# Ensure all features are present and in correct order
for feature in package['feature_names']:
    if feature not in input_features.columns:
        input_features[feature] = 0

input_features = input_features[package['feature_names']]

# Apply scaling if needed
if package['scaler'] is not None:
    input_features = package['scaler'].transform(input_features)

#print(input_features.head())

# Make prediction
prediction = package['model'].predict(input_features)[0]

return prediction

# Test the geographic prediction function
print("\n Testing geographic prediction function...")
test_lat = df_geo['latitude'].iloc[0]
test_lon = df_geo['longitude'].iloc[0]
test_date = '2024-12-01'

try:
    predicted_price = predict_land_price_geographic(test_lat, test_lon,
↪test_date)
    print(f" Test prediction (coordinates only):")
    print(f" Location: ({test_lat:.4f}, {test_lon:.4f})")
    print(f" Date: {test_date}")
    print(f" Predicted price: {predicted_price:.0f} per m²")
    print("\n Geographic model successfully saved and tested!")
except Exception as e:
    print(f" Error in prediction: {e}")
    print("Please check the model file and try again.")

```

Geographic model saved as: geographic_land_price_model_random_forest.pkl

Model Package Contains:

- Trained Random Forest model
- Feature scaler (not needed)
- List of 26 feature names
- Pre-fitted geographic clustering model
- Reference statistics for feature engineering
- Performance metrics (R^2 : 0.596, MAE: 1022)
- Training metadata

```

Testing geographic prediction function...
Creating geographic and time features...
Creating time features...
Creating geographic features...
Creating location clusters...
Creating interaction features...
Creating market features...
Feature creation complete!
Features created: 19
Test prediction (coordinates only):
  Location: (21.5131, 39.2291)
  Date: 2024-12-01
  Predicted price: 6443 per m2

Geographic model successfully saved and tested!

```

```

[30]: price = predict_land_price_geographic(
        latitude=21.82400071,
        longitude=39.07395258 ,
        transaction_date='2024-08-28'
    )

# 21.82400071 39.07395258 28-08-24 1611

print(f'Predicted price: {price:.0f} per m2')
# 2019

```

```

Creating geographic and time features...
Creating time features...
Creating geographic features...
Creating location clusters...
Creating interaction features...
Creating market features...
Feature creation complete!
Features created: 19
Predicted price: 1858 per m2

```

```
[ ]:
```