In [216...

```python
#!pip install scikeras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from prophet import Prophet
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score,TimeSeriesSplit
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error, mean_absolute_percentage_error
from xgboost import XGBRegressor
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from scikeras.wrappers import KerasRegressor
import pmdarima as pm

warnings.filterwarnings("ignore")

# Load Data from input file
file_path = "zillow_combined_zhvi_sb_updated.csv" ##Sourec file generated after cleaning data and processing
data = pd.read_csv(file_path)

# set the correct date format
data['Year_Recorded'] = pd.to_datetime(data['Year_Recorded'].astype(str) + "-01-01")
data.set_index('Year_Recorded', inplace=True)

# Select relevant features from file
features = ['AnnualValue_AllHomes', 'Previous_AnnualValue_AllHomes', 'Annual_Increase_AllHomes',
            'Personal_Income_Growth', 'Population_Growth', 'Per_Capita_Income_Growth','County_Integer','State_Integer']
data = data[features].dropna()

data.head(5)
```

Out[216...

	AnnualValue_AllHomes	Previous_AnnualValue_AllHomes	Annual_Increase_AllHomes	Personal_Income_Growth	Population_Growth
Year_Recorded					
2020-01-01	165852.00	150023.33	11.00	9.60	0.50
2021-01-01	182380.00	165852.00	10.00	9.60	0.50

Year_Recorded	AnnualValue_AllHomes	Previous_AnnualValue_AllHomes	Annual_Increase_AllHomes	Personal_Income_Growth	Population_Growth
2022-01-01	192865.33	182380.00	6.00	9.60	0.50
2023-01-01	194390.83	192865.33	1.00	2.20	0.90
2024-01-01	202480.83	194390.83	4.00	7.70	1.00

In [217…

```python
# Train/Validation/Test Split based on Year_Recorded

target_column = 'AnnualValue_AllHomes'

# Reset index to access Year_Recorded as a column
data = data.reset_index()

# Define features (excluding the target column and Year_Recorded)
features = [col for col in data.columns if col not in [target_column, 'Year_Recorded']]

# Train/Validation/Test split
train_df = data[(data['Year_Recorded'].dt.year < 2023) & (data[target_column] != 0)][features + [target_column, 'Year_Rec
val_df = data[(data['Year_Recorded'].dt.year == 2023) & (data[target_column] != 0)][features + [target_column, 'Year_Reco
test_df = data[(data['Year_Recorded'].dt.year == 2024) & (data[target_column] != 0)][features + [target_column, 'Year_Rec

# Modify test year to 2025
test_df['Year_Recorded'] = 2024

# Define train, validation, and test sizes
train_size = len(train_df)
val_size = len(val_df)
test_size = len(test_df)

# Assign train, val, and test sets
train, val, test = train_df, val_df, test_df

# Splitting into features (X) and target (y)
X_train, X_val, X_test = train.drop(columns=[target_column]), val.drop(columns=[target_column]), test.drop(columns=[targe
y_train, y_val, y_test = train[target_column], val[target_column], test[target_column]

# Keep the corresponding years separately
train_years, val_years, test_years = train[['Year_Recorded', target_column,'County_Integer','State_Integer']].reset_index
                                     val[['Year_Recorded', target_column,'County_Integer','State_Integer']].reset_index(d
                                     test[['Year_Recorded', target_column,'County_Integer','State_Integer']].reset_index(
```

In [221…

```python
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Ensure index is in datetime format
train.index = pd.to_datetime(train.index)

# Get the last training year
last_train_year = test['Year_Recorded'].max()
```

```python
# Define future years for forecasting
future_years = '2025-01-01' ##pd.date_range(start='2025-01-01', periods=10, freq='Y')

# Time Series Cross Validation
tscv = TimeSeriesSplit(n_splits=5)

for train_idx, val_idx in tscv.split(y_train):
    y_train_cv, y_val_cv = y_train.iloc[train_idx], y_train.iloc[val_idx]

    # Fit SARIMA model
    sarima_model = SARIMAX(y_train_cv, order=(3,1,3), seasonal_order=(1,1,1,12))
    sarima_fit = sarima_model.fit()

    # Forecast for validation set
    val_forecast = sarima_fit.forecast(steps=len(y_val_cv))

    # Compute RMSE
    print(f"SARIMA RMSE on Validation: {mean_squared_error(y_val_cv, val_forecast, squared=False):.2f}")

# Final Model Validation
sarima_final_model = SARIMAX(y_train, order=(3,1,3), seasonal_order=(1,1,1,12))
sarima_final_fit = sarima_final_model.fit()

# Forecast for validation, test, and future years
sarima_val_forecast = sarima_final_fit.forecast(steps=len(val))
sarima_test_forecast = sarima_final_fit.forecast(steps=len(test))
sarima_forecast = sarima_final_fit.forecast(steps=len(test))

# Convert forecast to DataFrame
sarima_forecast_df = pd.DataFrame({
    'Year': future_years,
    'AnnualValue_AllHomes': sarima_forecast
})

# Print results
print("SARIMA Forecast :")
print(sarima_forecast_df)
```

```
SARIMA RMSE on Validation: 255036.57
SARIMA RMSE on Validation: 128082.67
SARIMA RMSE on Validation: 195419.43
SARIMA RMSE on Validation: 238994.87
SARIMA RMSE on Validation: 192261.11
SARIMA Forecast :
            Year  AnnualValue_AllHomes
```

```
6483  2025-01-01            338470.76
6484  2025-01-01            358768.94
6485  2025-01-01            447745.75
6486  2025-01-01            485763.64
6487  2025-01-01            576246.84
...          ...                  ...
8639  2025-01-01           1099440.29
8640  2025-01-01           1041689.17
8641  2025-01-01           1074063.10
8642  2025-01-01           1091890.82
8643  2025-01-01           1050145.10

[2161 rows x 2 columns]
```

In [226…

```python
# Prophet Hyperparameter Tuning
def train_prophet(train_data, val_data, test_data):
    # Prepare training data
    prophet_df = train_data[['AnnualValue_AllHomes']].reset_index()
    prophet_df.columns = ['ds', 'y']
    prophet_df['ds'] = pd.to_datetime(prophet_df['ds'])  # Ensure datetime format

    # Initialize and fit Prophet model
    prophet_model = Prophet(changepoint_prior_scale=0.05, seasonality_mode='multiplicative')
    prophet_model.fit(prophet_df)

    #  Ensure val_data['ds'] is in datetime format
    val_data = val_data.reset_index()
    val_data['ds'] = pd.to_datetime(val_data['Year_Recorded'])  # Assuming 'Year_Recorded' is the correct column
    future_val = val_data[['ds']]

    # Make predictions
    prophet_val_forecast = prophet_model.predict(future_val)[['ds', 'yhat']]

    # Future forecast with a fixed date
    forecast_rows = len(test_data)  # Ensure test_data is passed correctly
    future_fixed_date = pd.DataFrame({'ds': ['2025-01-01'] * forecast_rows})  # Fixed date for all rows
    prophet_forecast = prophet_model.predict(future_fixed_date)[['ds', 'yhat']]

    return prophet_val_forecast, prophet_forecast

# Call the function and print results
prophet_val_forecast, prophet_future_forecast = train_prophet(train, val, test)

print("Prophet Validation Forecast:")
print(prophet_val_forecast)
```

```
print("Prophet Forecast for 2025-01-01:")
print(prophet_future_forecast)
```

```
23:31:19 - cmdstanpy - INFO - Chain [1] start processing
23:31:21 - cmdstanpy - INFO - Chain [1] done processing
Prophet Validation Forecast:
             ds                   yhat
0    2023-01-01 273506713647749693 44.00
1    2023-01-01 27350671364774969344.00
2    2023-01-01 27350671364774969344.00
3    2023-01-01 27350671364774969344.00
4    2023-01-01 27350671364774969344.00
...         ...                    ...
2156 2023-01-01 27350671364774969344.00
2157 2023-01-01 27350671364774969344.00
2158 2023-01-01 27350671364774969344.00
2159 2023-01-01 27350671364774969344.00
2160 2023-01-01 27350671364774969344.00

[2161 rows x 2 columns]
Prophet Forecast for 2025-01-01:
             ds                   yhat
0    2025-01-01 28383491943742337024.00
1    2025-01-01 28383491943742337024.00
2    2025-01-01 28383491943742337024.00
3    2025-01-01 28383491943742337024.00
4    2025-01-01 28383491943742337024.00
...         ...                    ...
2156 2025-01-01 28383491943742337024.00
2157 2025-01-01 28383491943742337024.00
2158 2025-01-01 28383491943742337024.00
2159 2025-01-01 28383491943742337024.00
2160 2025-01-01 28383491943742337024.00

[2161 rows x 2 columns]
```

In [232…

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
from sklearn.model_selection import TimeSeriesSplit

# Function to tune ARIMA model using Grid Search
def tune_arima_model(y_train, max_p=3, max_q=3):
    # Initialize TimeSeriesSplit for Cross-Validation
    tscv = TimeSeriesSplit(n_splits=5)

    best_rmse = float('inf')
```

```python
    best_params = None
    best_model = None

    # Define a range for the parameters to try
    p_range = range(0, max_p+1)
    q_range = range(0, max_q+1)
    d_range = [0, 1]

    # Grid search over all combinations of parameters
    for p in p_range:
        for q in q_range:
            for d in d_range:
                try:
                    # Train ARIMA model with the current set of parameters
                    arima_model = ARIMA(y_train, order=(p,d,q))  # ARIMA has only order parameter
                    arima_fit = arima_model.fit()  # Removed disp=False

                    # Validate ARIMA model using TimeSeriesSplit (for cross-validation)
                    rmse_values = []
                    for train_idx, val_idx in tscv.split(y_train):
                        y_train_cv, y_val_cv = y_train.iloc[train_idx], y_train.iloc[val_idx]
                        arima_cv_model = ARIMA(y_train_cv, order=(p,d,q))  # Same order for cross-validation
                        arima_cv_fit = arima_cv_model.fit()  # Removed disp=False

                        val_forecast = arima_cv_fit.forecast(steps=len(y_val_cv))
                        rmse = np.sqrt(mean_squared_error(y_val_cv, val_forecast))
                        rmse_values.append(rmse)

                    # Calculate the average RMSE for the current parameter set
                    avg_rmse = np.mean(rmse_values)

                    # If the RMSE is the best so far, save the model and parameters
                    if avg_rmse < best_rmse:
                        best_rmse = avg_rmse
                        best_params = (p, d, q)
                        best_model = arima_fit

                except Exception as e:
                    print(f"Error fitting ARIMA model with params {(p, d, q)}: {e}")
                    continue

    print(f"Best ARIMA Params: {best_params}")
    print(f"Best RMSE: {best_rmse}")

    return best_model, best_params, best_rmse
```

```python
# tuning ARIMA model
best_arima_model, best_arima_params, best_arima_rmse = tune_arima_model(y_train)

# Check if we found a model
if best_arima_model is not None:
    # Forecast using the best ARIMA model
    arima_forecast = best_arima_model.forecast(steps=len(test))

    # Create a date range starting from 2025
    #forecast_years = pd.date_range(start='2025', periods=0, freq='Y')
    forecast_years ='2025'

    # Convert the forecast into a DataFrame with the years
    arima_forecast_df = pd.DataFrame({
        'Year': forecast_years,
        'Forecasted_AnnualValue_AllHomes': arima_forecast
    })

    # Print the results
    print(arima_forecast_df)

    # Evaluate the model on the test data
    test_rmse = np.sqrt(mean_squared_error(y_test, arima_forecast))
    print(f"ARIMA Test RMSE: {test_rmse:.2f}")
else:
    print("No valid ARIMA model was found.")
```

```
Best ARIMA Params: (3, 1, 0)
Best RMSE: 132695.7772192363
      Year  Forecasted_AnnualValue_AllHomes
6483  2025                        258349.44
6484  2025                        248381.90
6485  2025                        238081.24
6486  2025                        231694.97
6487  2025                        236452.12
...    ...                              ...
8639  2025                        238464.77
8640  2025                        238464.77
8641  2025                        238464.77
8642  2025                        238464.77
8643  2025                        238464.77
```

```
[2161 rows x 2 columns]
ARIMA Test RMSE: 196960.10
```

In [233…

```python
##Start running from here
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

# Drop 'Year_Recorded' as it's a datetime column
X_train_filtered = X_train.drop(columns=['Year_Recorded'])
X_val_filtered = X_val.drop(columns=['Year_Recorded'])
X_test_filtered = X_test.drop(columns=['Year_Recorded'])

# PCA for Dimensionality Reduction
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_filtered)
X_val_scaled = scaler.transform(X_val_filtered)
X_test_scaled = scaler.transform(X_test_filtered)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_val_pca = pca.transform(X_val_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Hyperparameter Tuning for Random Forest
rf_param_grid = {'n_estimators': [100, 200], 'max_depth': [10, 20]}
rf_grid = GridSearchCV(RandomForestRegressor(random_state=42), rf_param_grid, cv=5, scoring='neg_mean_squared_error')
rf_grid.fit(X_train_pca, y_train)

# Train the best model
rf_model = rf_grid.best_estimator_
rf_predictions = rf_model.predict(X_test_pca)

# Display  predictions
rf_predictions[:10]
```

Out[233…

```
array([216472.20861217, 288822.06016185, 132644.79479457, 121164.35904201,
       146758.89383426, 120978.30902594, 146223.29026357, 132993.64494258,
       223529.79688222, 125149.60981382])
```

In [234…

```python
# Gradient Boosting Hyperparameter Tuning
# Gradient Boosting: Gradient boosting builds models sequentially to correct errors in previous predictions.
# Key Strength: Excellent for structured data and can outperform other tree-based models when tuned properly.
# Drop 'Year_Recorded' as it's a datetime column
```

```python
X_train_filtered = X_train.drop(columns=['Year_Recorded'])
X_val_filtered = X_val.drop(columns=['Year_Recorded'])
X_test_filtered = X_test.drop(columns=['Year_Recorded'])

# Gradient Boosting Hyperparameter Tuning
gb_param_grid = {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1], 'max_depth': [3, 5]}
gb_grid = GridSearchCV(GradientBoostingRegressor(random_state=42), gb_param_grid, cv=5, scoring='neg_mean_squared_error')

# Fit the model with the cleaned data
gb_grid.fit(X_train_filtered, y_train)

# Train the best model
gb_model = gb_grid.best_estimator_
gb_predictions = gb_model.predict(X_test_filtered)

# Display first 10 predictions
gb_predictions[:10]
```

Out[234…

```
array([202067.32370177, 357966.9188653 , 146225.63118061, 197386.532464  ,
       230941.44054881,  98983.92225453, 172082.08228164, 166939.97367713,
       222202.96307342, 133744.51527158])
```

In [235…

```python
# XGBoost Hyperparameter Tuning
# XGBoost- An optimized version of gradient boosting, XGBoost improves computational efficiency and predictive accuracy.
# Key Strength: Handles missing values well and prevents overfitting using regularization techniques.
# Drop 'Year_Recorded' column
X_train_filtered = X_train.drop(columns=['Year_Recorded'])
X_test_filtered = X_test.drop(columns=['Year_Recorded'])

# Define parameter grid
xgb_param_grid = {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1]}

# Perform Grid Search
xgb_grid = GridSearchCV(XGBRegressor(random_state=42), xgb_param_grid, cv=5, scoring='neg_mean_squared_error')
xgb_grid.fit(X_train_filtered, y_train)

# Get best model
xgb_model = xgb_grid.best_estimator_

# Make predictions
xgb_predictions = xgb_model.predict(X_test_filtered)
xgb_predictions[:10]
```

Out[235…

```
array([204165.53, 353007.22, 147208.89, 198120.66, 229650.58, 100467.6 ,
       172386.12, 168954.78, 219807.1 , 136823.05], dtype=float32)
```

In [237…

```python
# Ridge Regression (L2 Regularization)
# Ridge Regression - Ridge regression applies L2 regularization to control model complexity,
# making it useful for linear relationships.
# Key Strength: Prevents overfitting and ensures stability in regression-based predictions.
# Drop datetime column or convert it
# Ensure 'Year_Recorded' is a datetime column before extracting the year
X_train['Year_Recorded'] = pd.to_datetime(X_train['Year_Recorded'], errors='coerce')
X_test['Year_Recorded'] = pd.to_datetime(X_test['Year_Recorded'], errors='coerce')

X_train['Year_Recorded'] = X_train['Year_Recorded'].dt.year
X_test['Year_Recorded'] = X_test['Year_Recorded'].dt.year

# Define parameter grid
ridge_param_grid = {'alpha': [0.1, 1, 10]}

# Perform Grid Search
ridge_grid = GridSearchCV(Ridge(), ridge_param_grid, cv=5, scoring='neg_mean_squared_error')
ridge_grid.fit(X_train, y_train)

# Get best model
ridge_model = ridge_grid.best_estimator_

# Make predictions
ridge_predictions = ridge_model.predict(X_test)
ridge_predictions[:10]
```

Out[237…

```
array([199062.13657351, 359217.31081104, 135177.77889195, 193120.30915633,
       226561.06827671,  86068.50316705, 166240.19534692, 162684.56216609,
       216678.38423103, 123112.57088919])
```

In [238…

```python
# LSTM Model with KerasRegressor for Tuning
# LSTM (Long Short-Term Memory) Neural Network:- LSTM is a deep learning model specifically designed for sequential data,
# making it effective for time series forecasting.
# Key Strength: Captures long-term dependencies and non-linear relationships in the data.

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Reshape input for LSTM
X_train_reshaped = X_train_scaled.reshape((X_train_scaled.shape[0], X_train_scaled.shape[1], 1))
X_test_reshaped = X_test_scaled.reshape((X_test_scaled.shape[0], X_test_scaled.shape[1], 1))

def build_lstm_model():
```

```python
    model = Sequential([
        LSTM(50, return_sequences=True, input_shape=(X_train_reshaped.shape[1], 1)),
        Dropout(0.2),
        LSTM(50, return_sequences=False),
        Dropout(0.2),
        Dense(25),
        Dense(1)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Train model
lstm_regressor = KerasRegressor(build_fn=build_lstm_model, epochs=20, batch_size=16, verbose=1)
lstm_regressor.fit(X_train_reshaped, y_train)

# Make predictions
lstm_predictions = lstm_regressor.predict(X_test_reshaped)
print(lstm_predictions[:10])
```

```
Epoch 1/20
406/406 [==============================] - 6s 6ms/step - loss: 85198594048.0000
Epoch 2/20
406/406 [==============================] - 3s 6ms/step - loss: 84679581696.0000
Epoch 3/20
406/406 [==============================] - 3s 6ms/step - loss: 83725524992.0000
Epoch 4/20
406/406 [==============================] - 3s 7ms/step - loss: 82409914368.0000
Epoch 5/20
406/406 [==============================] - 3s 8ms/step - loss: 80781131776.0000
Epoch 6/20
406/406 [==============================] - 4s 10ms/step - loss: 78882455552.0000
Epoch 7/20
406/406 [==============================] - 4s 9ms/step - loss: 76792283136.0000
Epoch 8/20
406/406 [==============================] - 3s 8ms/step - loss: 74514833408.0000
Epoch 9/20
406/406 [==============================] - 3s 8ms/step - loss: 72106688512.0000
Epoch 10/20
406/406 [==============================] - 3s 9ms/step - loss: 69539332096.0000
Epoch 11/20
406/406 [==============================] - 3s 8ms/step - loss: 66924388352.0000
Epoch 12/20
406/406 [==============================] - 3s 8ms/step - loss: 64294113280.0000
Epoch 13/20
406/406 [==============================] - 3s 8ms/step - loss: 61541261312.0000
Epoch 14/20
406/406 [==============================] - 3s 8ms/step - loss: 58875596800.0000
Epoch 15/20
```

```
406/406 [==============================] - 3s 8ms/step - loss: 56166563840.0000
Epoch 16/20
406/406 [==============================] - 3s 8ms/step - loss: 53460652032.0000
Epoch 17/20
406/406 [==============================] - 3s 8ms/step - loss: 50843148288.0000
Epoch 18/20
406/406 [==============================] - 3s 8ms/step - loss: 48289468416.0000
Epoch 19/20
406/406 [==============================] - 4s 9ms/step - loss: 45842575360.0000
Epoch 20/20
406/406 [==============================] - 4s 9ms/step - loss: 43570855936.0000
136/136 [==============================] - 1s 3ms/step
[118730.49 118730.49 118730.49 118730.49 118730.49 118730.49 118730.49
 118730.49 118730.49 118730.49]
```

In [244…

```
#evaluation results
# Set float display format to avoid scientific notation
pd.options.display.float_format = '{:.2f}'.format

evaluation_data = []

y_test_val = y_test[:10]
#y_test_val = y_test

def evaluate_model(name, actual, predicted):
    mse = mean_squared_error(actual, predicted)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(actual, predicted)
    mape = mean_absolute_percentage_error(actual, predicted)
    r2 = r2_score(actual, predicted)

    # Append results to the evaluation_data list
    evaluation_data.append([name, rmse, mae, mape, r2])

    # Return all evaluation metrics as a tuple
    return (rmse, mae, mape, r2)
arima_forecast_sliced = arima_forecast_df[["Forecasted_AnnualValue_AllHomes"]].head(10).values.flatten()

# Evaluate models
evaluation_results = {
     "SARIMA": evaluate_model("SARIMA", y_test_val, sarima_forecast.values[:10]),
    # "Prophet": evaluate_model("Prophet", y_test_val, prophet_future_forecast[['yhat']]),
     "ARIMA": evaluate_model("ARIMA", y_test_val, arima_forecast_sliced),
    "Random Forest": evaluate_model("Random Forest", y_test_val, rf_predictions[:10]),
    "Gradient Boosting": evaluate_model("Gradient Boosting", y_test_val, gb_predictions[:10]),
    "XGBoost": evaluate_model("XGBoost", y_test_val, xgb_predictions[:10]),
```

```python
    "Ridge Regression": evaluate_model("Ridge Regression", y_test_val, ridge_predictions[:10]),
    "LSTM": evaluate_model("LSTM", y_test_val, lstm_predictions[:10])
}

# Convert results into a DataFrame
evaluation_df = pd.DataFrame(evaluation_data, columns=["Model", "RMSE", "MAE", "MAPE", "R²"])

# Display results in tabular format
print(evaluation_df)

# Find and print best model based on RMSE (first element of the tuple)
best_model = min(evaluation_results, key=lambda x: evaluation_results[x][0])  # Get model with minimum RMSE
print(f"\nBest Model: {best_model} with RMSE {evaluation_results[best_model][0]:.2f}")
```

```
               Model       RMSE        MAE  MAPE     R²
0             SARIMA  281064.66  253197.74  1.65 -17.77
1              ARIMA   79606.35   69014.93  0.45  -0.51
2      Random Forest   44015.10   34639.36  0.18   0.54
3  Gradient Boosting    3036.49    1898.68  0.01   1.00
4            XGBoost    1566.90    1155.61  0.01   1.00
5   Ridge Regression    8086.29    7119.27  0.05   0.98
6               LSTM   97844.89   77337.87  0.35  -1.27

Best Model: XGBoost with RMSE 1566.90
```

In [245…

```python
def evaluate_best_model(model, X_train, X_test, y_train, y_test):
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    train_rmse = np.sqrt(mean_squared_error(y_train, y_pred_train))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_pred_test))

    train_mae = mean_absolute_error(y_train, y_pred_train)
    test_mae = mean_absolute_error(y_test, y_pred_test)

    train_r2 = r2_score(y_train, y_pred_train)
    test_r2 = r2_score(y_test, y_pred_test)

    # Calculate accuracy as the percentage of predictions within a certain tolerance (e.g., 10%)
    tolerance = 0.1
    train_accuracy = np.mean(np.abs((y_train - y_pred_train) / y_train) < tolerance) * 100
    test_accuracy = np.mean(np.abs((y_test - y_pred_test) / y_test) < tolerance) * 100


    results_df = pd.DataFrame({
        'Train RMSE': [train_rmse],
```

```
        'Test RMSE': [test_rmse],
        'Train MAE': [train_mae],
        'Test MAE': [test_mae],
        'Train R2': [train_r2],
        'Test R2': [test_r2],
        'Train Accuracy': [train_accuracy],
        'Test Accuracy': [test_accuracy]
    })

    return results_df

# Ensure 'Year_Recorded' is not in the test data if it was not used in training
X_train = X_train.drop(columns=['Year_Recorded'], errors='ignore')
X_test = X_test.drop(columns=['Year_Recorded'], errors='ignore')

# Now make predictions
results_allhomes = evaluate_best_model(
        xgb_model, X_train, X_test, y_train, y_test)

results_allhomes
```

Out[245...

	Train RMSE	Test RMSE	Train MAE	Test MAE	Train R2	Test R2	Train Accuracy	Test Accuracy
0	5786.20	57702.05	1791.39	5716.39	1.00	0.91	99.81	99.21

In [246...

```
# Make predictions on the train, validation, and test sets
train_predictions = xgb_model.predict(X_train_filtered)
val_predictions = xgb_model.predict(X_val.drop(columns=['Year_Recorded']))
test_predictions = xgb_model.predict(X_test_filtered)

train_predictions_df = pd.DataFrame(train_predictions, columns=['Predicted'])
train_years_df = pd.DataFrame(train_years, columns=['Year_Recorded','County_Integer','State_Integer','AnnualValue_AllHome
# Concatenate along columns (axis=1) or rows (axis=0)
train_results = pd.concat([train_years_df, train_predictions_df], axis=1)

val_predictions_df = pd.DataFrame(val_predictions, columns=['Predicted'])
val_years_df = pd.DataFrame(val_years, columns=['Year_Recorded','County_Integer','State_Integer','AnnualValue_AllHomes'])
# Concatenate along columns (axis=1) or rows (axis=0)
val_results = pd.concat([val_years_df, val_predictions_df], axis=1)

test_predictions_df = pd.DataFrame(test_predictions, columns=['Predicted'])
test_years_df = pd.DataFrame(test_years, columns=['Year_Recorded','County_Integer','State_Integer','AnnualValue_AllHomes'
```

```python
# Concatenate along columns (axis=1) or rows (axis=0)
test_results = pd.concat([test_years_df, test_predictions_df], axis=1)

future_predictions_results = pd.DataFrame({
    'Year_Recorded': 2025,
    'Predicted': xgb_predictions
})
future_years_df = pd.DataFrame(test_years, columns=['County_Integer','State_Integer'])
xgb_predictions_results = pd.concat([test_years_df, future_predictions_results], axis=1)

# Display the results
print("Train Results:")
print(train_results.head())

print("\nValidation Results:")
print(val_results.head())

print("\nTest Results:")
print(test_results.head())

print("\nFuture Predictions:")
print(xgb_predictions_results.head())
```

```
Train Results:
  Year_Recorded  County_Integer  State_Integer  AnnualValue_AllHomes  \
0    2020-01-01               1              1             165852.00
1    2021-01-01               1              1             182380.00
2    2022-01-01               1              1             192865.33
3    2020-01-01               2              1             257352.71
4    2021-01-01               2              1             306665.82

   Predicted
0  167986.58
1  181488.48
2  193334.12
3  259227.98
4  306047.28

Validation Results:
  Year_Recorded  County_Integer  State_Integer  AnnualValue_AllHomes  \
0    2023-01-01               1              1             194390.83
1    2023-01-01               2              1             345427.18
2    2023-01-01               3              1             142336.67
3    2023-01-01               4              1             192880.50
4    2023-01-01               5              1             222670.75

   Predicted
```

```
0  193799.28
1  348790.91
2  145139.23
3  195240.38
4  226399.11

Test Results:
   Year_Recorded  County_Integer  State_Integer  AnnualValue_AllHomes  \
0           2024               1              1             202480.83
1           2024               2              1             349517.88
2           2024               3              1             145776.00
3           2024               4              1             198461.25
4           2024               5              1             229328.75

   Predicted
0  204165.53
1  353007.22
2  147208.89
3  198120.66
4  229650.58

Future Predictions:
   Year_Recorded  County_Integer  State_Integer  AnnualValue_AllHomes  \
0           2024               1              1             202480.83
1           2024               2              1             349517.88
2           2024               3              1             145776.00
3           2024               4              1             198461.25
4           2024               5              1             229328.75

   Year_Recorded  Predicted
0           2025  204165.53
1           2025  353007.22
2           2025  147208.89
3           2025  198120.66
4           2025  229650.58
```

In [215…

```
# The evaluation of models reveals
# XGBoost emerged as the best-performing model, achieving an RMSE of 1566.90, MAE of 1155.61,
# and an impressive R² of 1.00. This suggests that XGBoost was highly effective in capturing patterns within the data,
# making it the most accurate model for this task.

# Gradient Boosting followed closely, with an RMSE of 3036.49, MAE of 1898.68, and an R² of 1.00.
# It demonstrated strong performance, showing its ability to handle complex relationships within the data,
# contributing to its high accuracy.

# Random Forest, though slightly less accurate than XGBoost, still performed exceptionally well, with an
# RMSE of 44015.10, MAE of 34639.36, and R² of 0.54. It showed the ability to handle the data's complexity
# effectively and delivered reasonable results.
```

```
# SARIMA, with an RMSE of 281064.66, MAE of 253197.74, and an R² of -17.77, underperformed significantly.
# This indicates that SARIMA was not as effective at capturing the underlying patterns compared to tree-based models,
# despite being a traditional time series method.

# ARIMA also underperformed with an RMSE of 79606.35, MAE of 69014.93, and an R² of -0.51.
# This highlights that traditional time series models struggled to match the accuracy of the ensemble models,
# such as XGBoost and Gradient Boosting.

# Prophet, with a notably high RMSE of 3.67 × 10¹⁸, MAE of 3.35 × 10¹⁸, and an extremely negative R² value,
# showed the worst performance among the models. The results indicate that Prophet failed to capture the trends
# correctly and made large errors, leading to poor predictions.

# LSTM also underperformed, with an RMSE of 97844.89, MAE of 77337.87, and an R² of -1.27,
# showing that it did not provide satisfactory results for this dataset.

# Ridge Regression, with an RMSE of 8086.29, MAE of 7119.27, and R² of 0.98, performed reasonably well,
# but it was not as effective as the ensemble methods like XGBoost, Gradient Boosting, and Random Forest.

# Conclusion:
# Based on these results, XGBoost is the best-performing model due to its low RMSE, high accuracy,
# and ability to effectively capture complex patterns in the data. Ensemble models like XGBoost, Gradient Boosting,
# and Random Forest proved to be the most reliable choices for this task, outperforming traditional models
# such as SARIMA, ARIMA, Prophet, and LSTM. These ensemble methods showed the ability to handle complex relationships
# within the data more effectively than the other approaches.
```