# Predictive Model

## 1. Introduction

In [1]:
```
import sys
import json
import pandas as pd
import itertools
import numpy as np
import seaborn as sns
import Orange
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from scipy.stats import friedmanchisquare, rankdata
from Orange.evaluation import compute_CD, graph_ranks
import pmdarima as pm
from pmdarima.arima.utils import ndiffs
```

## 2. Data Generation

- Calculate the total debts from all lenders over time
- Generating a dataset with new month feature such as 'month'. The resulting DataFrame is saved as a CSV file named 'month_smedebtsu.csv'
- Generating a dataset with new date features such as 'day', 'month', 'year', 'quarter', 'dayofweek', and 'dayofyear'. The resulting DataFrame is saved as a CSV file named 'date_smedebtsu.csv'
- Generating a dataset with lagged features such as 'total_debts_lag1' and 'total_debts_lag2'. The resulting DataFrame is saved as a CSV file named 'lag_smedebtsu.csv'

In [2]:
```
%run -i "../src/data_generation.py"
```

```
File saved successfully!
File path:  C:\Users\Admin\Desktop\Code\Coding_Interview\data/features\date_s
medebtsu.csv
File saved successfully!
File path:  C:\Users\Admin\Desktop\Code\Coding_Interview\data/features\lag_2_
smedebtsu.csv
File saved successfully!
File path:  C:\Users\Admin\Desktop\Code\Coding_Interview\data/features\lag_4_
smedebtsu.csv
```

In [3]:
```python
# Show the generated dataset
date_debts = pd.read_csv("../data/features/date_smedebtsu.csv")
date_debts.head()
```

Out[3]:

| | date | total_debts | day | month | year | quarter | dayofweek | dayofyear |
|---|---|---|---|---|---|---|---|---|
| **0** | 2013-10-13 | 228007.01 | 13 | 10 | 2013 | 4 | 6 | 286 |
| **1** | 2013-11-13 | 227988.31 | 13 | 11 | 2013 | 4 | 2 | 317 |
| **2** | 2013-12-10 | 265199.00 | 10 | 12 | 2013 | 4 | 1 | 344 |
| **3** | 2014-01-23 | 299453.00 | 23 | 1 | 2014 | 1 | 3 | 23 |
| **4** | 2014-03-05 | 290103.00 | 5 | 3 | 2014 | 1 | 2 | 64 |

In [4]:
```python
# Show the generated dataset
lag_debts = pd.read_csv("../data/features/lag_2_smedebtsu.csv")
lag_debts.head()
```

Out[4]:

| | date | total_debts | total_debts_lag1 | total_debts_lag2 |
|---|---|---|---|---|
| **0** | 2013-12-10 | 265199.0 | 227988.31 | 228007.01 |
| **1** | 2014-01-23 | 299453.0 | 265199.00 | 227988.31 |
| **2** | 2014-03-05 | 290103.0 | 299453.00 | 265199.00 |
| **3** | 2014-04-05 | 304337.0 | 290103.00 | 299453.00 |
| **4** | 2014-05-05 | 293623.0 | 304337.00 | 290103.00 |

In [5]:
```python
# Show the generated dataset
lag_debts = pd.read_csv("../data/features/lag_4_smedebtsu.csv")
lag_debts.head()
```

Out[5]:

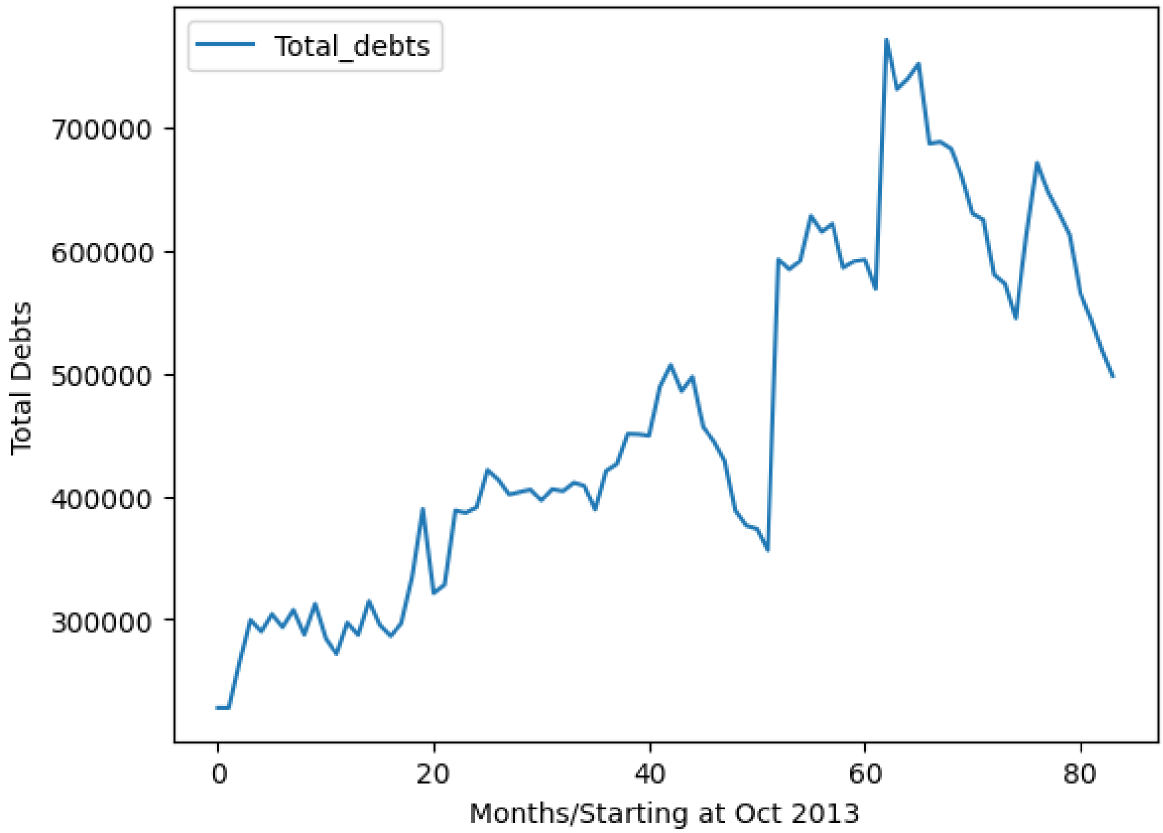
| | date | total_debts | total_debts_lag1 | total_debts_lag2 | total_debts_lag3 | total_debts_lag4 |
|---|---|---|---|---|---|---|
| **0** | 2014-03-05 | 290103.0 | 299453.0 | 265199.0 | 227988.31 | 228007.01 |
| **1** | 2014-04-05 | 304337.0 | 290103.0 | 299453.0 | 265199.00 | 227988.31 |
| **2** | 2014-05-05 | 293623.0 | 304337.0 | 290103.0 | 299453.00 | 265199.00 |
| **3** | 2014-06-06 | 307582.0 | 293623.0 | 304337.0 | 290103.00 | 299453.00 |
| **4** | 2014-07-07 | 287573.0 | 307582.0 | 293623.0 | 304337.00 | 290103.00 |

# 3. Forecasting using Statistical Models

- Statistical models, such as Autoregressive (AR), Moving Average (MA), and Autoregressive Integrated Moving Average (ARIMA), are commonly employed for

In [6]:

```
# We focus on the month of the year and the total debts. Plot a graph
processed_data_path = "../data/processed/processed_smedebtsu.csv"
df = pd.read_csv(processed_data_path)
# df['Date_time'] = pd.to_datetime(df['Date_time'])
df['Total_debts'] = df.sum(axis=1, numeric_only=True)
df_total_debts = df[['Date_time', 'Total_debts']]
df_total_debts.plot()

plt.xlabel("Months/Starting at Oct 2013")
plt.ylabel("Total Debts")
plt.show()
```




## Time-series decomposition

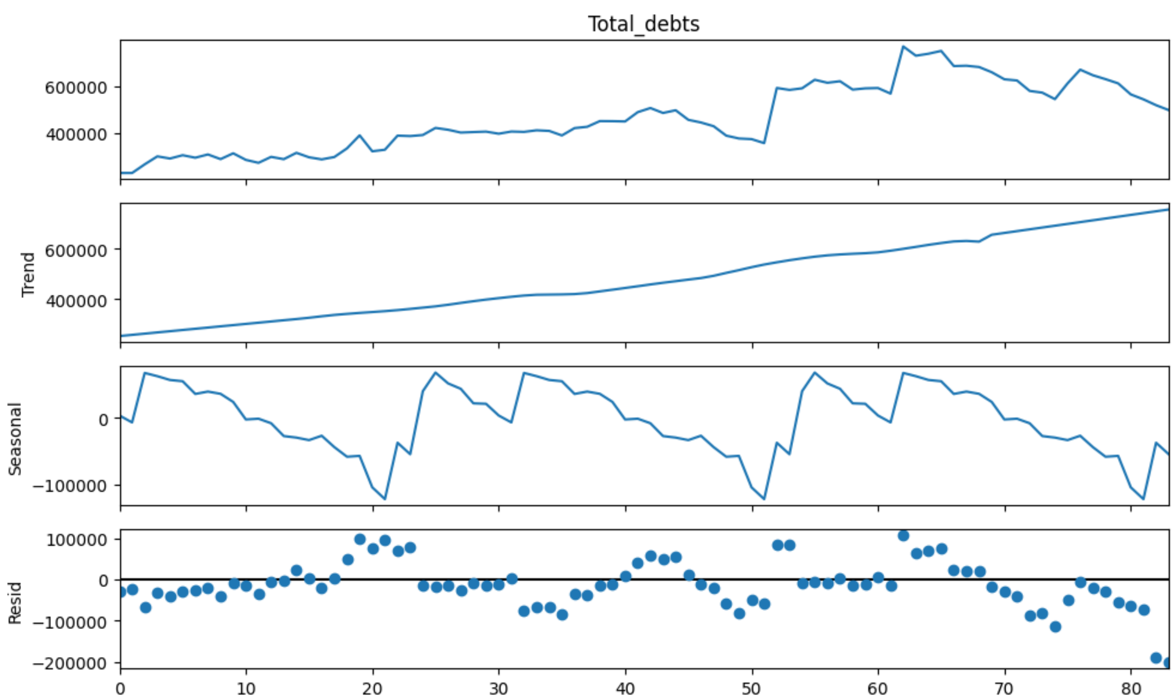
- Time series decomposition involves thinking of a series as a combination of level, trend, seasonality, and noise components.
- Decomposition provides a useful abstract model for thinking about time series generally and for better understanding problems during time series analysis and forecasting.

In [7]:

```python
# Let's find seasonal decomposition of time-series models
def decomposition(df, period):
    # decompistion instance
    result_decom = seasonal_decompose(df['Total_debts'], model="additive",
                                      period=period, extrapolate_trend='fr
    # plot the components
    fig = result_decom.plot()
    fig.set_size_inches((10, 6))
    # Tight layout to realign things
    fig.tight_layout()
    plt.show()

    # capture the components
    debts_trend = result_decom.trend
    debts_season = result_decom.seasonal
    debts_resid = result_decom.resid
    return debts_trend, debts_season, debts_resid
```

In [8]:

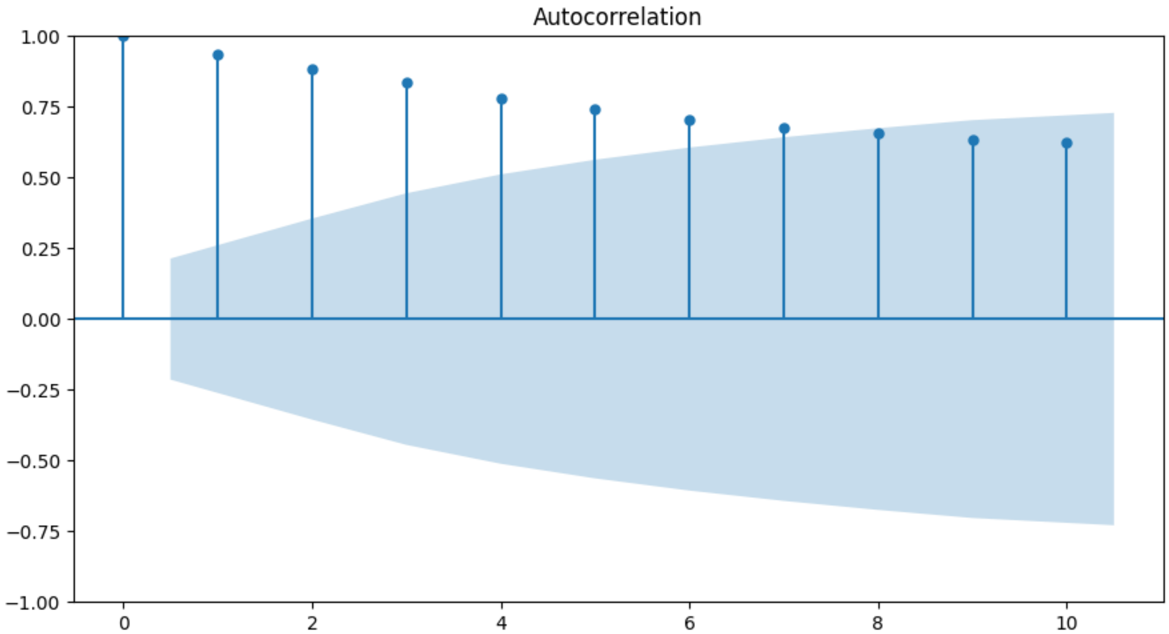
```python
# Let's find the components for Total_debts
# Period = 30 months
debts_trend, debts_season, debts_resid = decomposition(df_total_debts, per
```

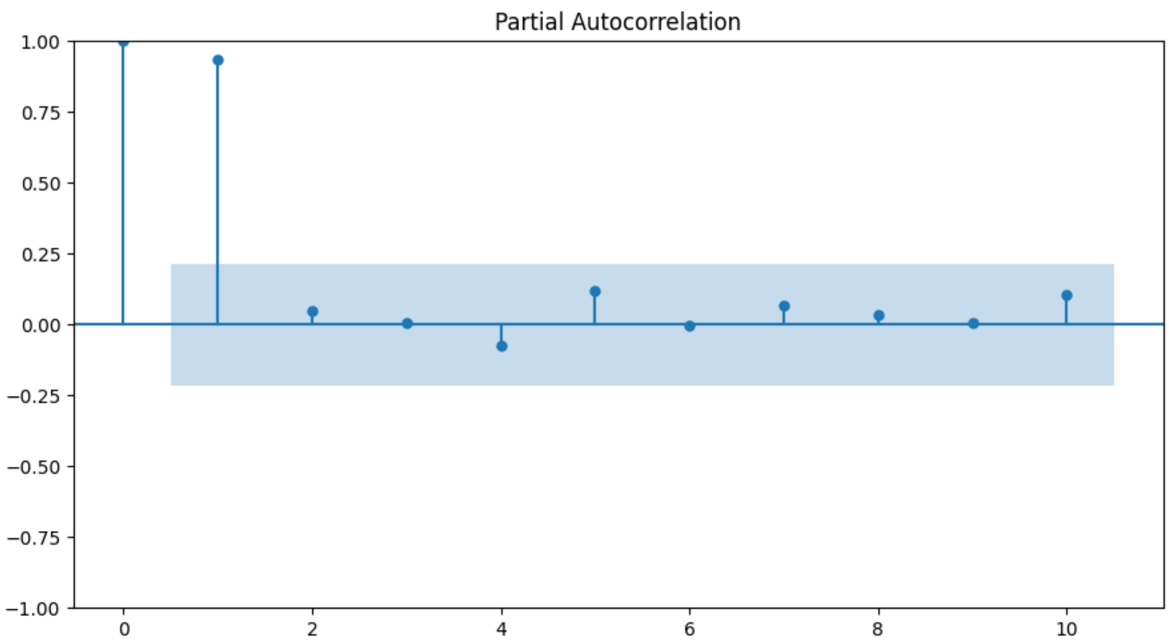


## Plot the autocorrelation and partial auto-correlation

In [9]:
```
# function to return ACF and PACF plots
def plot_acf_pacf(df, lags):
    var = df['Total_debts']
    # plot the ACF
    fig = plot_acf(var, lags=lags)
    fig.set_size_inches((9, 5))
    fig.tight_layout()
    plt.show()

    # plot the PACF
    fig = plot_pacf(var, lags=lags)
    fig.set_size_inches((9,5))
    fig.tight_layout()
    plt.show()
```

In [10]:
```
# Plot ACF and PACF of Total_debts
plot_acf_pacf(df_total_debts, lags=10)
```




- **Auto-correlation interpretation**: A slow decline in auto-correlation indicates time-series not stationary, we can prove the stationary of time series by **Dicky-fuller test**
- **Partical auto-correlation interpretation interpretation**: The graph indicates that the correlation between consecutive data points can only be effectively measured with a lag of one, as it is more significant compared to other lagged time-series.

## Dicky-Fuller Test (Stationary test)

- We will use "Dickey-Fuller test" to determine stationary.
- Hypothesis to prove Dicky-Fuller Test:

- H0 - The time-series data is non-stationary
- H1 - The time-series data is stationary

In [11]:
```
def ADF_test(df):
    print("Results of Dickey-Fuller Test:")
    dftest = adfuller(df['Total_debts'], autolag="AIC")
    dfoutput = pd.Series(
        dftest[0:4],
        index=[
            "Test Statistic",
            "p-value",
            "Lags Used",
            "Number of Observations Used",
        ],
    )
    for key, value in dftest[4].items():
        dfoutput["Critical Value (%s)" % key] = value
    print(dfoutput)
```

In [12]:
```
ADF_test(df_total_debts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                 -1.892313
p-value                         0.335686
Lags Used                       0.000000
Number of Observations Used    83.000000
Critical Value (1%)            -3.511712
Critical Value (5%)            -2.897048
Critical Value (10%)           -2.585713
dtype: float64
```

- **Dicky-Fuller Test interpretation**: We can see p-value are much greater than 0.05, so the time-series data cannot reject H0. Therefore, the time-series data is not stationary

## Finding degree of differencing

In [13]:
```
def find_degree_of_differencing(df):
    # Perform ADF test to determine stationarity
    result = adfuller(df['Total_debts'])
    p_value = result[1]

    if p_value < 0.05:
        # Data is stationary, no differencing needed
        debts_ndiffs = 0
    else:
        # Increment differencing until data becomes stationary
        debts_ndiffs = 0
        while p_value >= 0.05:
            differenced_data = df['Total_debts'].diff().dropna()
            result = adfuller(differenced_data)
            p_value = result[1]
            debts_ndiffs += 1

    print(f"The degree of differencing is {debts_ndiffs} for 'Total_debts
```

In [14]:
```
find_degree_of_differencing(df_total_debts)
```

```
The degree of differencing is 1 for 'Total_debts'.
```

## Train forecasting models using Auto-ARIMA

- **ARIMA (AutoRegressive Integrated Moving Average)** is a forecasting model used to predict future values of a time series, such as total debts, by considering the relationship between the observations, differencing the data, and incorporating the impact of past forecast errors. Here's how you can apply ARIMA for forecasting total debts:
- **Auto Regression (AR)**: In the AR component of ARIMA, you analyze how the total debts at a given time depend on its own past values. You can identify the appropriate lag order, denoted as 'p', by examining the autocorrelation function (ACF) plot of the total debts time series. This helps determine the number of lagged values to include in the model.
- **Integration (I)**: The integration component of ARIMA focuses on making the total debts time series stationary. Stationarity implies that the statistical properties of the time series, such as mean and variance, remain constant over time. You can achieve stationarity by differencing the raw total debts observations. The differencing order, denoted as 'd', represents the number of times you need to difference the data to achieve stationarity.
- **Moving Average (MA)**: The MA component of ARIMA takes into account the dependency between an observation and the residual errors from a moving average model applied to lagged observations. The order of the moving average component, denoted as 'q', determines the number of lagged forecast errors to include in the model.

In [15]:
```
# Split data into train and test sets using a specific day '2020-12-10'.
# This ensures that the results can be compared with other methods using t

df_total_debts['Date_time'] = pd.to_datetime(df_total_debts['Date_time'])

def split_arima(df):
    boundary_idx_test = '2020-12-10'
    train_df = df[df['Date_time'] < boundary_idx_test]
    test_df = df[df['Date_time'] >= boundary_idx_test]

    print("--------------Total Debts-----------------")
    print(f"Train Size: {len(train_df)}, Test Size: {len(test_df)}")

    return train_df, test_df

train_df, test_df = split_arima(df_total_debts)
```

```
--------------Total Debts-----------------
Train Size: 64, Test Size: 20
```

- We perform a grid search over a range of parameters for ARIMA models. We iterate through all combinations of parameters and fit an SARIMAX model using statsmodels. We keep track of the model with the lowest AIC (Akaike Information Criterion) as the best model.

In [16]:
```
def find_best_fit_arima(df):
    model = pm.auto_arima(df, test = 'adf',
                          start_p = 1, start_q = 1,
                          max_p = 3, max_q = 3,
                          d = None, seasonal = True,
                          start_P = 0, m = 3,
                          trace = True, error_action = 'ignore',
                          suppress_warnings = True, stepwise = True,
                          D = 1, information_criterion = 'aic')

    print(model.summary())
    print('\n')

    return model

model_arima_debts = find_best_fit_arima(train_df['Total_debts'])
```

```
Performing stepwise search to minimize aic
 ARIMA(1,0,1)(0,1,1)[3] intercept   : AIC=1498.428, Time=0.07 sec
 ARIMA(0,0,0)(0,1,0)[3] intercept   : AIC=1526.757, Time=0.01 sec
 ARIMA(1,0,0)(1,1,0)[3] intercept   : AIC=1502.418, Time=0.03 sec
 ARIMA(0,0,1)(0,1,1)[3] intercept   : AIC=1507.945, Time=0.03 sec
 ARIMA(0,0,0)(0,1,0)[3]             : AIC=1531.776, Time=0.01 sec
 ARIMA(1,0,1)(0,1,0)[3] intercept   : AIC=1505.027, Time=0.03 sec
 ARIMA(1,0,1)(1,1,1)[3] intercept   : AIC=1498.627, Time=0.10 sec
 ARIMA(1,0,1)(0,1,2)[3] intercept   : AIC=1497.044, Time=0.06 sec
 ARIMA(1,0,1)(1,1,2)[3] intercept   : AIC=1498.979, Time=0.13 sec
 ARIMA(0,0,1)(0,1,2)[3] intercept   : AIC=1506.742, Time=0.05 sec
 ARIMA(1,0,0)(0,1,2)[3] intercept   : AIC=1494.604, Time=0.05 sec
 ARIMA(1,0,0)(0,1,1)[3] intercept   : AIC=1495.942, Time=0.03 sec
 ARIMA(1,0,0)(1,1,2)[3] intercept   : AIC=1496.596, Time=0.11 sec
 ARIMA(1,0,0)(1,1,1)[3] intercept   : AIC=1496.531, Time=0.07 sec
 ARIMA(0,0,0)(0,1,2)[3] intercept   : AIC=1520.242, Time=0.04 sec
 ARIMA(2,0,0)(0,1,2)[3] intercept   : AIC=1496.558, Time=0.06 sec
 ARIMA(2,0,1)(0,1,2)[3] intercept   : AIC=1497.549, Time=0.11 sec
 ARIMA(1,0,0)(0,1,2)[3]             : AIC=1500.668, Time=0.04 sec

Best model:  ARIMA(1,0,0)(0,1,2)[3] intercept
Total fit time: 1.032 seconds
                                      SARIMAX Results
==============================================================================
=================
Dep. Variable:                                     y   No. Observations:
64
Model:             SARIMAX(1, 0, 0)x(0, 1, [1, 2], 3)   Log Likelihood
-742.302
Date:                               Thu, 18 May 2023   AIC
1494.604
Time:                                       01:43:59   BIC
1505.158
Sample:                                            0   HQIC
1498.740
                                                - 64
Covariance Type:                                 opg
==============================================================================
=
                 coef    std err          z      P>|z|      [0.025      0.97
5]
------------------------------------------------------------------------------
-
intercept   8953.1582   3267.678      2.740      0.006    2548.627    1.54e+0
4
ar.L1          0.6141      0.153      4.019      0.000       0.315       0.91
4
ma.S.L3       -0.5524      0.262     -2.111      0.035      -1.065      -0.04
0
ma.S.L6       -0.2733      0.261     -1.049      0.294      -0.784       0.23
7
sigma2      2.628e+09      0.016   1.61e+11      0.000    2.63e+09    2.63e+0
9
==============================================================================
======
Ljung-Box (L1) (Q):                    0.20   Jarque-Bera (JB):
145.14
```

```
Prob(Q):                              0.65   Prob(JB):
0.00
Heteroskedasticity (H):               4.58   Skew:
2.02
Prob(H) (two-sided):                  0.00   Kurtosis:
9.38
=============================================================================
======

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (comple
x-step).
[2] Covariance matrix is singular or near-singular, with condition number 9.7
3e+25. Standard errors may be unstable.
```

In [17]:

```
model_arima_debts
```

Out[17]:

```
ARIMA
ARIMA(1,0,0)(0,1,2)[3] intercept
```

- **Model ARIMA interpretation**: We will choose the best model ARIMA with the lowest AIC (Akaike Information Criterion) ARIMA (1,0,0)(0,1,2)[3]

## Forecasting on test data/ Calculating RMSE/ Visulization true values and predicted values

In [18]:

```
def plot_predictions(train_values, test_values, predicted_values, lower_co
    plt.figure(figsize=(10, 6))
    plt.plot(train_values.index, train_values, label='Train Values')
    plt.plot(test_values.index, test_values, label='Test Values')
    plt.plot(predicted_values.index, predicted_values, color='red', label=
    plt.fill_between(lower_confidence.index, lower_confidence, upper_confi
    plt.xlabel('Time')
    plt.ylabel('Total Debts')
    plt.title('Total Debts: Train, Predicted, and True Values with Confide
    plt.legend()
    plt.show()
```

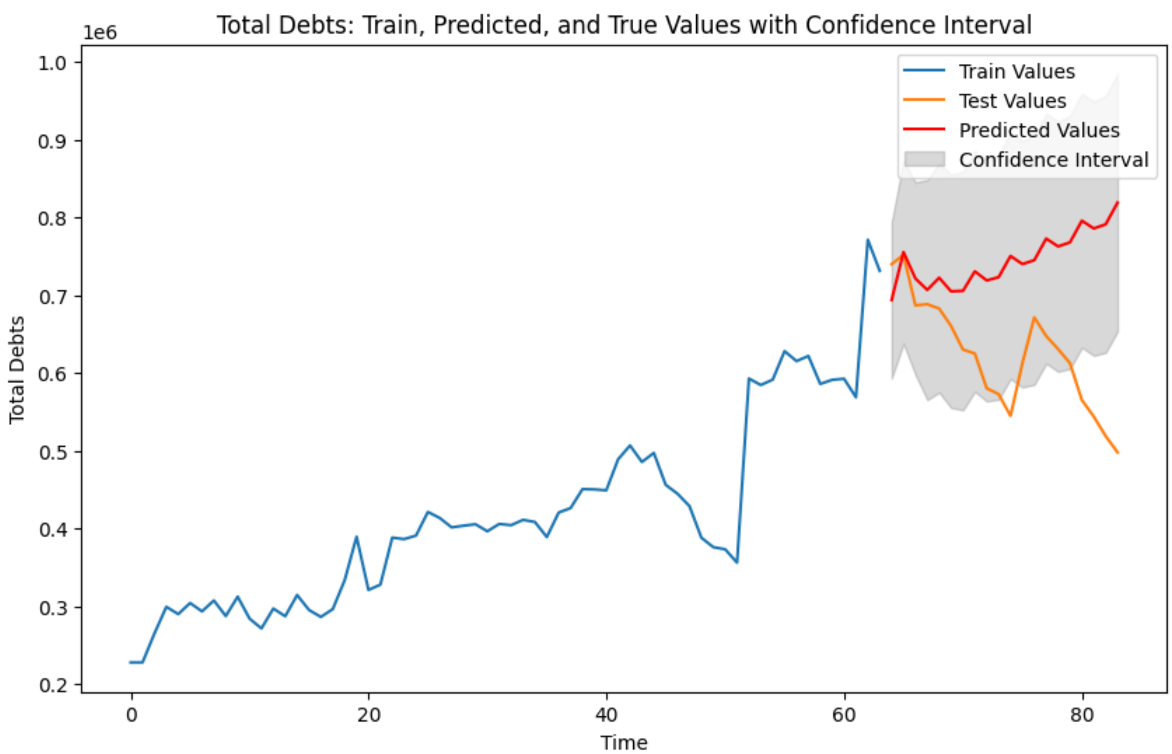
In [19]:

```
def make_predictions_and_print_rmse(model, test_df):
    print(f"forecasting and RMSE of total debts")

    forecast, confidence_interval = model.predict(X=test_df, n_periods = l
    forecasts = pd.Series(forecast, index = test_df[:len(test_df)].index)
    lower = pd.Series(confidence_interval[:, 0], index = test_df[:len(tes
    upper = pd.Series(confidence_interval[:, 1], index = test_df[:len(tes

    rmse = np.sqrt(np.mean((forecast.values - test_df.values) ** 2))

    print("RMSE is: ", rmse)

    return forecasts, lower, upper

forecast_values, lower_confidence, upper_confidence = make_predictions_and
```

```
forecasting and RMSE of total debts
RMSE is:  154249.19718203467
```

In [20]:

```
# Assuming you have the true values in a DataFrame named 'true_values'
train_values = train_df['Total_debts']
test_values = test_df['Total_debts']
# Plotting the predicted values, true values, and confidence interval
plot_predictions(train_values, test_values, forecast_values, lower_confid
```




- **Result interpretation**: It is observed that the forecasting of ARIMA model tends to deviate from the actual values. The difference can be attributed to the underlying trend of the data. As the historical data exhibits an upward trend, the ARIMA model tends to project a continuation of this trend into the future.

# 4. Run Base Machine Learning Regressor Models

- This study aims to predict total debts using machine learning regressor models, including Linear Regression, XGBoost, and LightGBM. By employing these models, we seek to enhance the accuracy of total debt predictions and identify the most effective approach. Through the analysis of historical total debt data and the utilization of advanced machine learning techniques, we anticipate providing valuable insights into the prediction of future total debts, contributing to improved financial decision-making and risk management strategies.
- **The steps are implemented:**

1. **Split data use time-series cross validation**: In time series cross-validation, the dataset is split into multiple folds based on time. I use the rolling window approach, a fixed-size training window is moved forward in time, and at each step, a model is trained on the data within the window and evaluated on the subsequent period (test set).

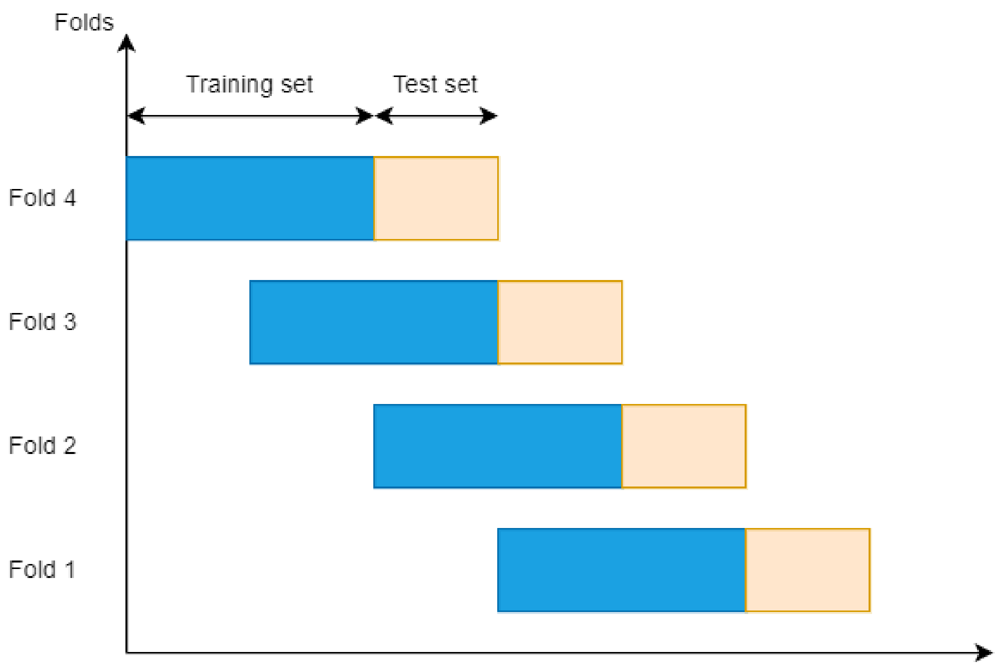



**Figure 1: Time series Cross-Validation**

2. **Run Machine Learning Regressor Models**: In time series cross-validation, each fold represents a distinct period in the time series data. When running machine learning regressors on each fold, the goal is to train the model on the historical data within the training set and evaluate its performance on the subsequent period (test set).
3. **Model evaluation**: Once the model is trained, it is evaluated on the validation set. The performance metrics, such as mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), and R2 score (R2_score).

4. **Results Structure**: The results from each fold are collected and calculated the mean of metrics across all folds for each regression algorithms, which analyzes to understand the

In [21]:
```
# Run the main model
%run -i "../src/demo_run_base_regressor_models.py"
```

```
2023-05-18 01:43:59.886688  File 0: date_smedebtsu.csv
2023-05-18 01:44:00.999421  File 1: lag_2_smedebtsu.csv
2023-05-18 01:44:02.133255  File 2: lag_4_smedebtsu.csv
```

## Visualize predicted results

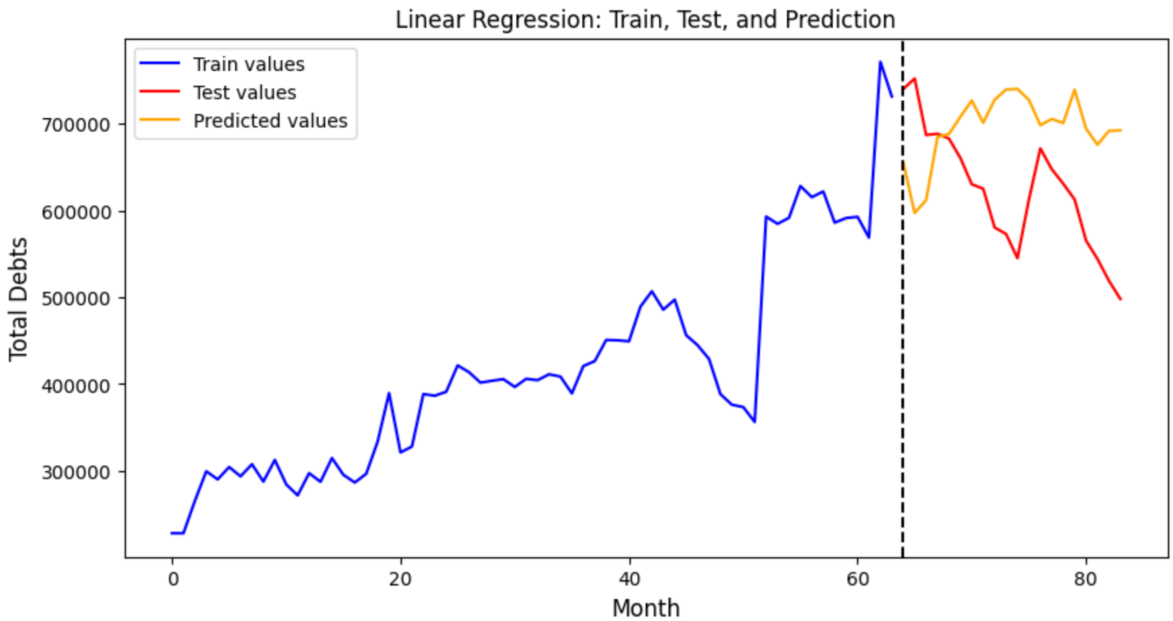
#### Dataset: "date_smedebtsu"

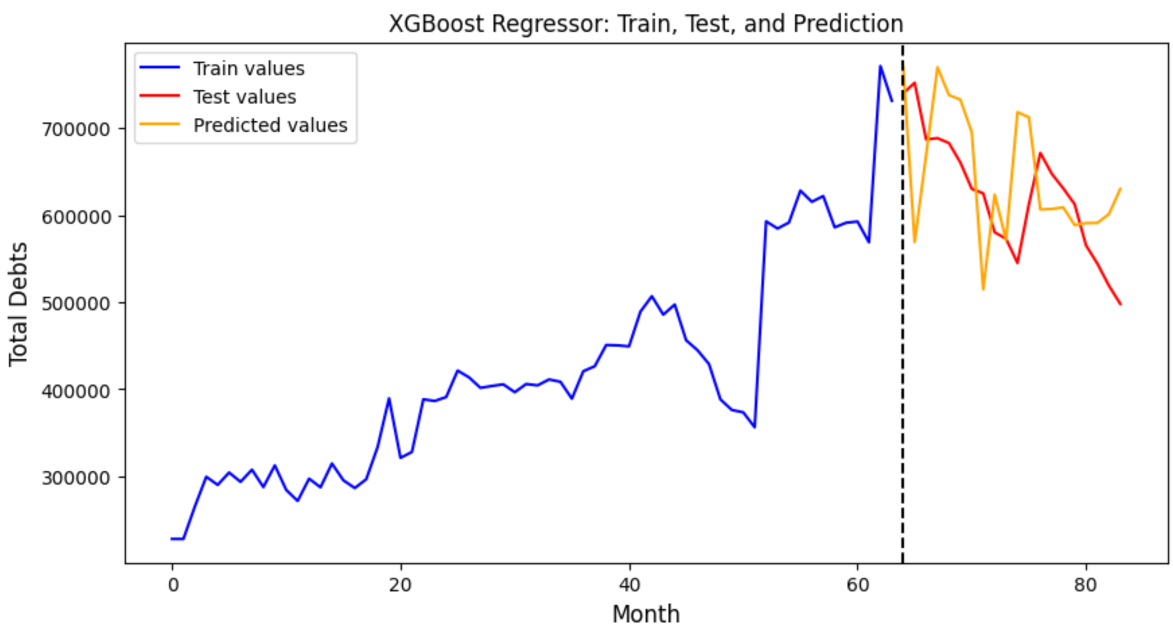
In [22]:
```
date_results_path = "../results/date_smedebtsu/ML_regression_models/averag
with open(date_results_path) as f:
    results = json.load(f)
```

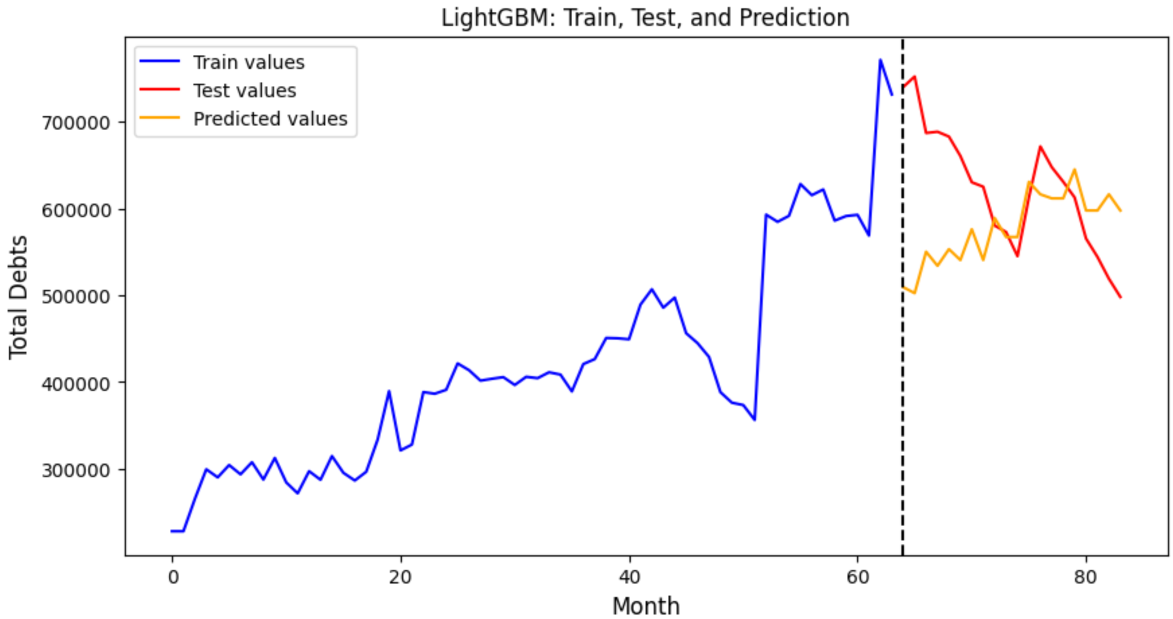
In [23]:

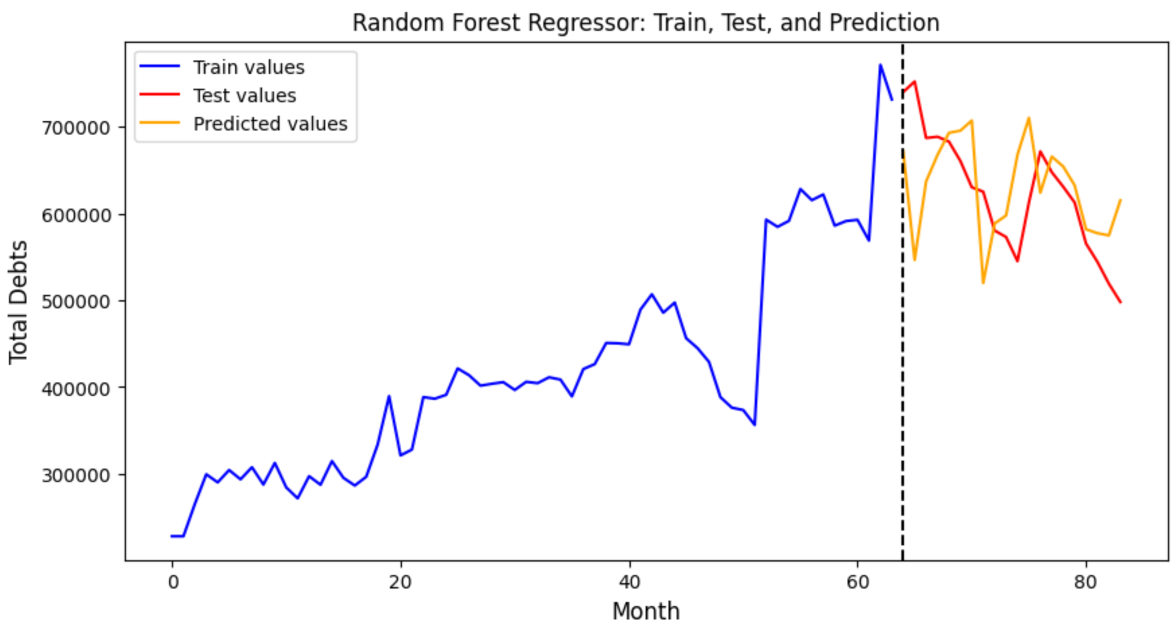
```
correct_name_algs = {
    'linear_regression': 'Linear Regression',
    'xgboost_regressor': 'XGBoost Regressor',
    'light_GBM': 'LightGBM',
    'random_forest_regressor': 'Random Forest Regressor'
}
def plot_train_test_prediction(result):
    train = result["train"]
    test = result["test"]
    algorithm = result["model"]
    correct_algorithm = correct_name_algs[algorithm]
    predictions = result["prediction"]

    concatenated_data = train + test
    split_index = len(train)

    df = pd.DataFrame(concatenated_data)

    train_data = df[:split_index]
    test_data = df[split_index:]

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

    ax.plot(train_data.index, train_data[0], color="blue")
    ax.plot(test_data.index, test_data[0], color="red")
    ax.plot(test_data.index, predictions, color="orange")

    ax.set_xlabel("Month", fontsize=12)
    ax.set_ylabel("Total Debts", fontsize=12)
    ax.axvline(test_data.index[0], color='black', ls='--')
    ax.legend(["Train values", "Test values", "Predicted values"])

    ax.title.set_text(f"{correct_algorithm}: Train, Test, and Prediction")
    plt.show()
```

In [24]:
```
# Iterate over each algorithm's results
for result in results:
    plot_train_test_prediction(result)
```





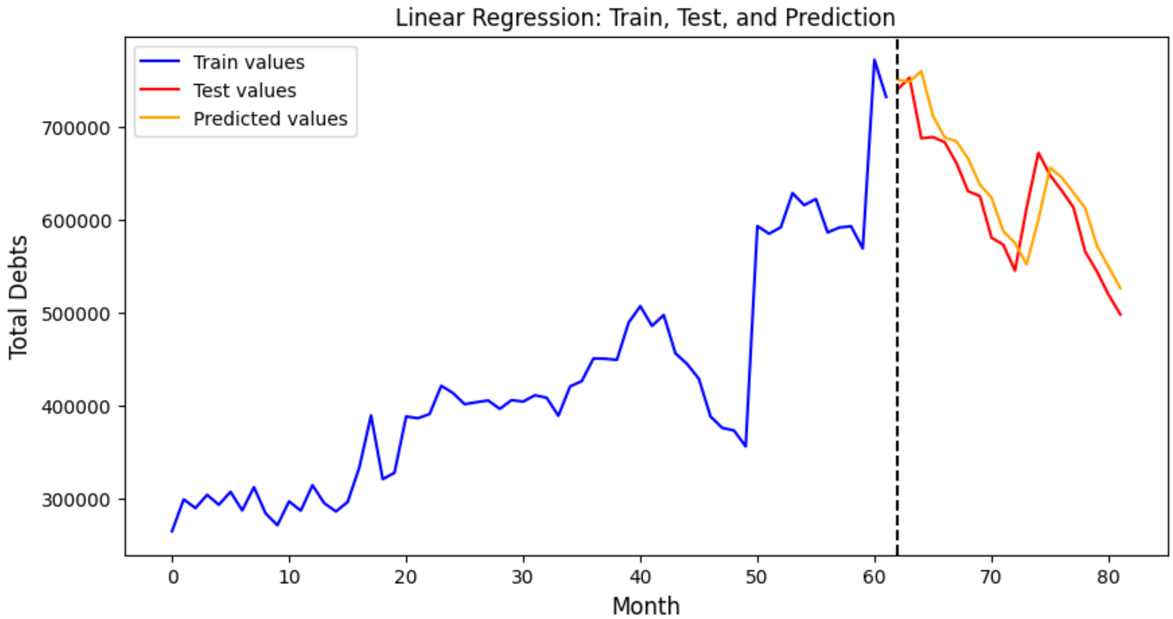



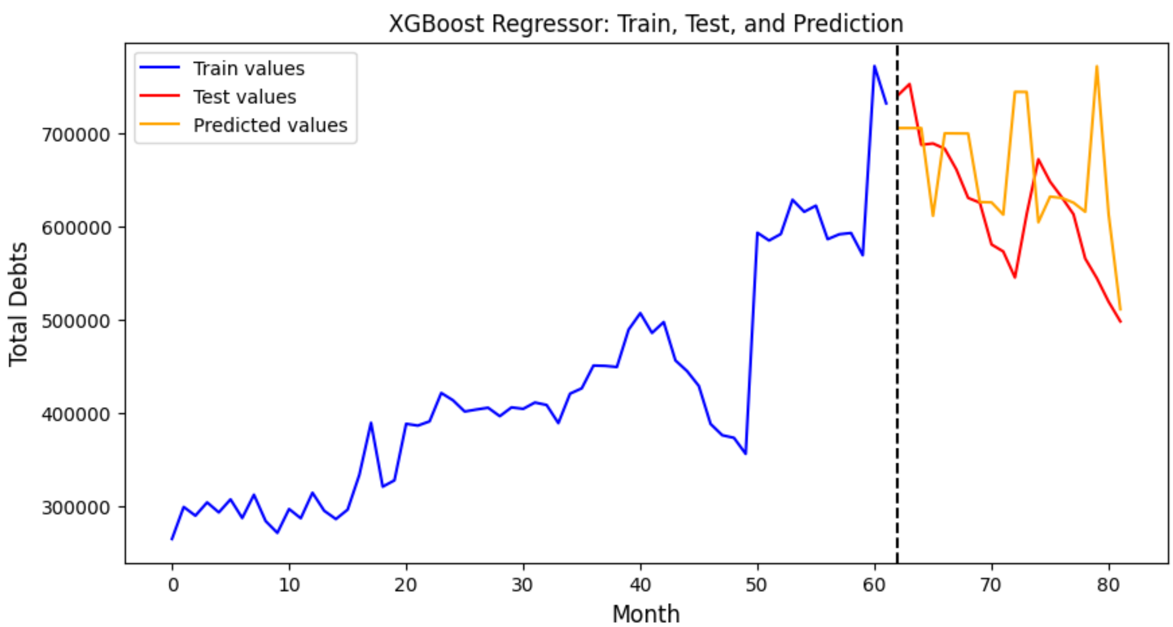

**Result interpretation**: Based on the analysis of the test and prediction results, we observe that the Linear Regression and Light GBM algorithms show a significant deviation from the trend of the "total_debts" label. However, the Random Forest Regressor and XGBoost Regressor algorithms demonstrate a closer approximation to the underlying trend. These findings suggest that the Random Forest Regressor and XGBoost Regressor models may be more suitable for predicting the "total_debts" based on the available features

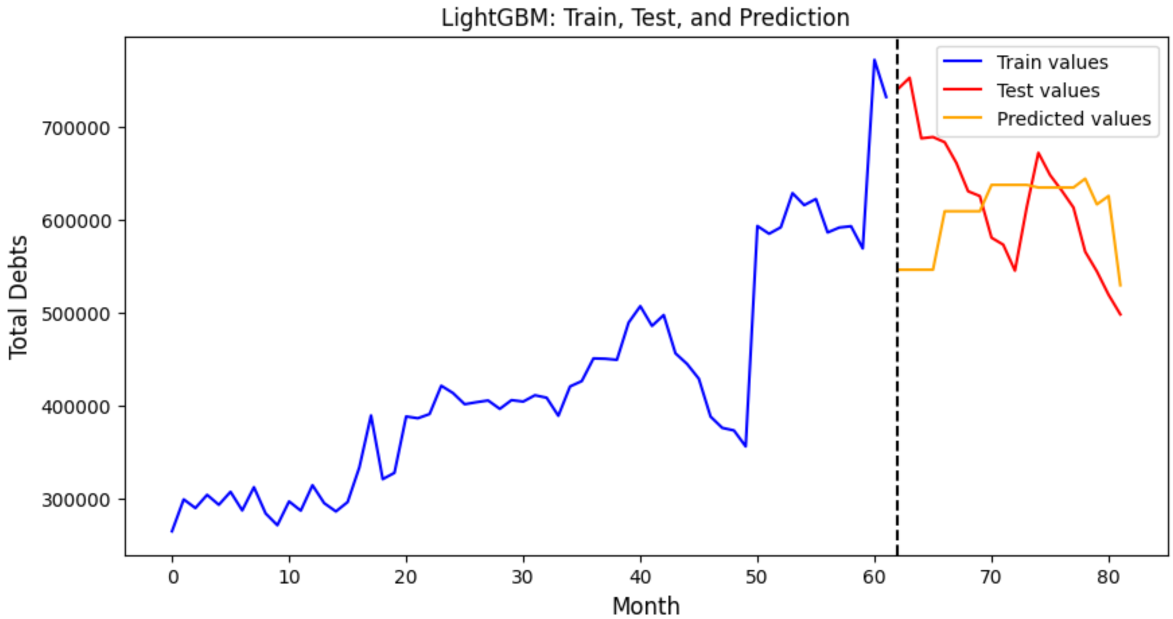
**Dataset: "lag_2_smedebtsu"**

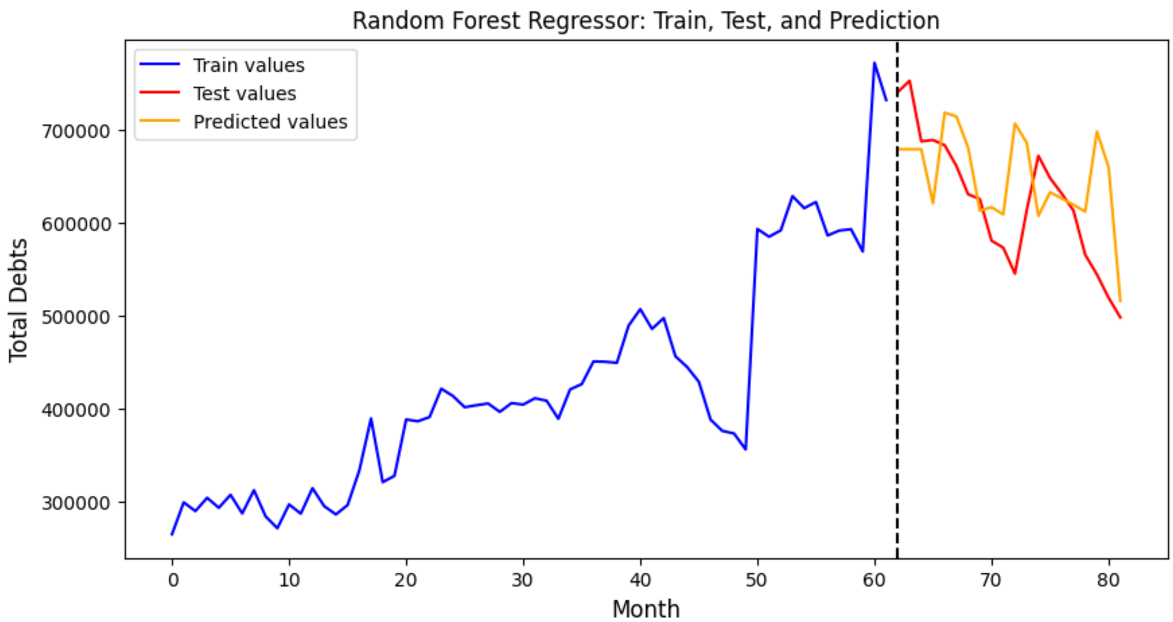
In [25]:
```
date_results_path = "../results/lag_2_smedebtsu/ML_regression_models/avera
with open(date_results_path) as f:
    results = json.load(f)
```

In [26]:
```python
# Iterate over each algorithm's results
for result in results:
    plot_train_test_prediction(result)
```





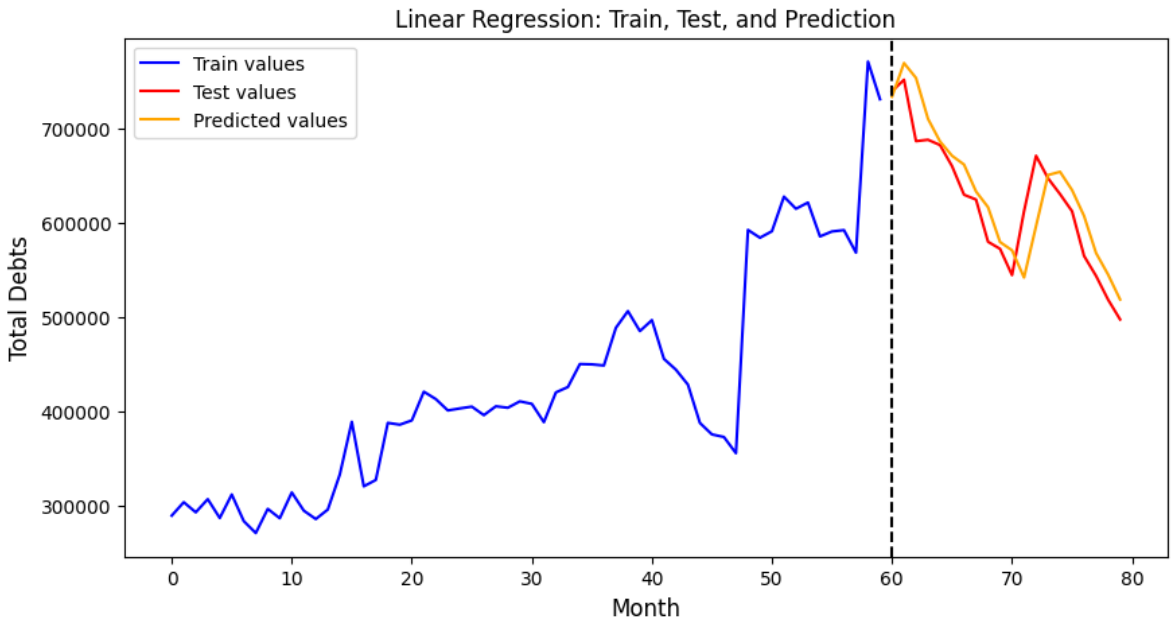



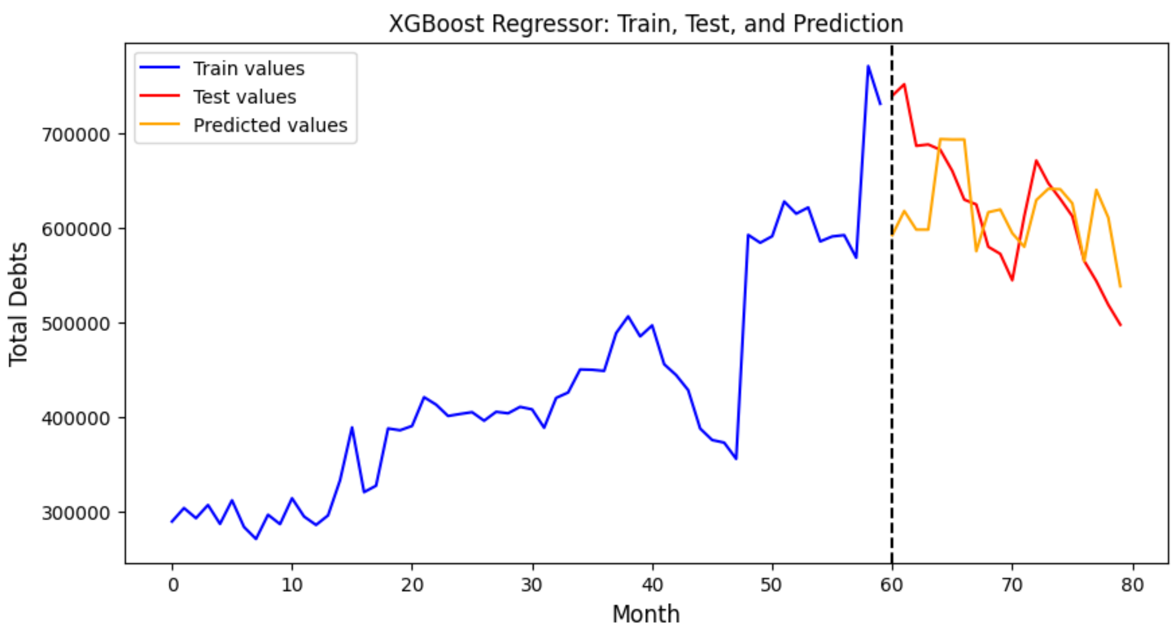

**Result interpretation**: Regarding the dataset with a lag of 1 day and 2 days in the 'total_debts' variable, the majority of the algorithms exhibited accurate trend predictions, with the exception of LightGBM.

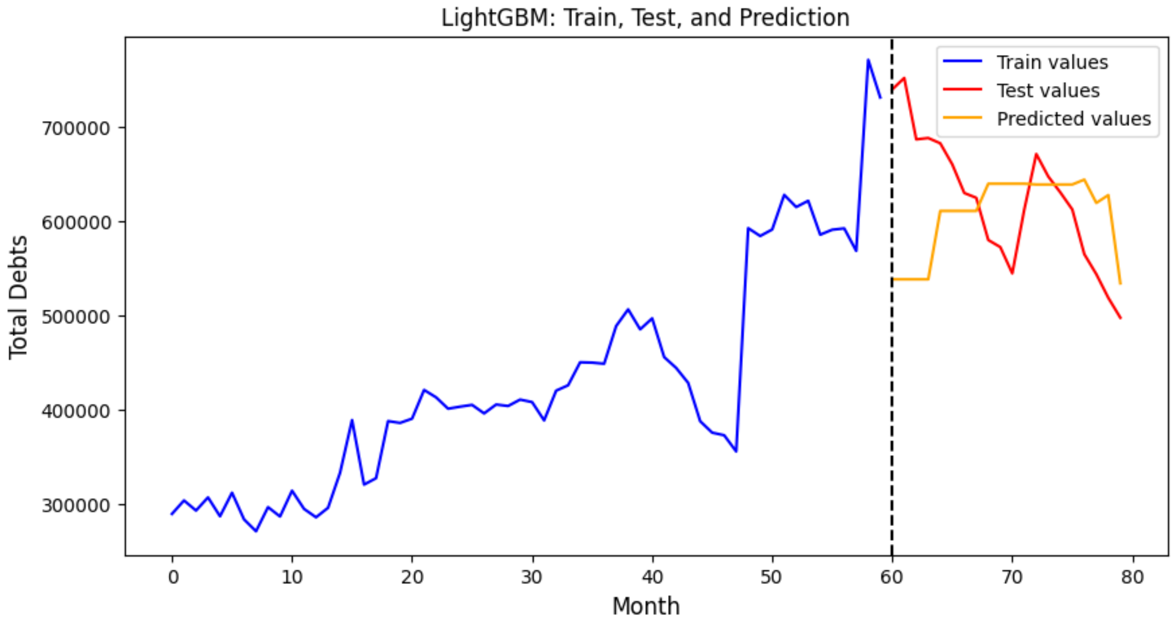
**Dataset: "lag_4_smedebtsu"**

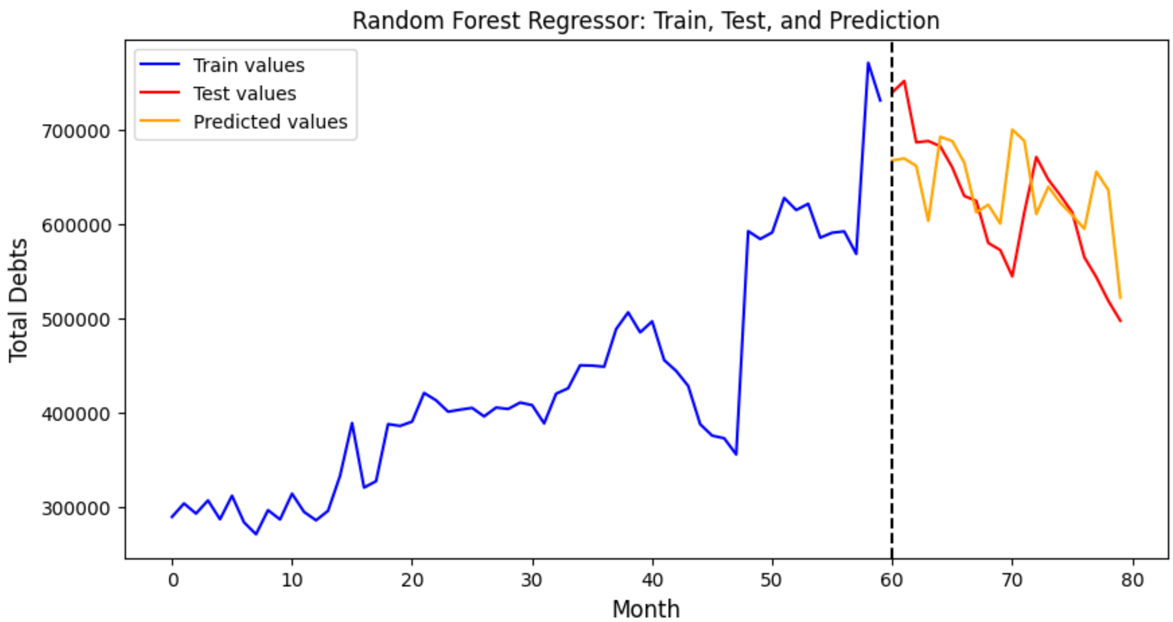
In [27]:
```
date_results_path = "../results/lag_4_smedebtsu/ML_regression_models/avera
with open(date_results_path) as f:
    results = json.load(f)
```

In [28]:
```
# Iterate over each algorithm's results
for result in results:
    plot_train_test_prediction(result)
```






# 5. Train forecasting models using LSTM model

**The steps are implemented:**

1. Split data into train/ validation/ test using a specific day '2020-12-10'.
2. Building model
3. Evaluation test sets use MAE, MSE, RMSE, R2_score metrics and then save results into **"DL_models"**
4. Visualization the predictions and ground truth

```
# Results structure in 2 datasets "date_smedebtsu" and "lag_smedebtsu", which were created by 2. Data Generation
├──date_smedebtsu
│    ├──DL_models
```

```
│     └───ML_regression_models
│
└───lag_smedebtsu
    ├───DL_models
    └───ML_regression_models
```

In [29]:

```
# Run the main model
%run -i "../src/demo_run_DL_model.py"
```

```
2023-05-18 01:44:08.457683  File 0: date_smedebtsu.csv
The number of training samples:  51
The number of validation samples:  13
The number of testing samples:  20
Starting training...
Epoch 1/10
4/4 [==============================] - 3s 185ms/step - loss: 0.0684 - mean
_absolute_error: 0.2291 - val_loss: 0.3624 - val_mean_absolute_error: 0.57
84
Epoch 2/10
4/4 [==============================] - 0s 16ms/step - loss: 0.0320 - mean_
absolute_error: 0.1454 - val_loss: 0.2412 - val_mean_absolute_error: 0.462
2
Epoch 3/10
4/4 [==============================] - 0s 15ms/step - loss: 0.0182 - mean_
absolute_error: 0.1108 - val_loss: 0.1477 - val_mean_absolute_error: 0.360
8
Epoch 4/10
4/4 [==============================] - 0s 13ms/step - loss: 0.0223 - mean_
```

## Visualize predicted results

**Dataset: "date_smedebtsu"**

In [30]:

```
date_results_path = "../results/date_smedebtsu/DL_models/average_results.j
with open(date_results_path) as f:
    results = json.load(f)
```

In [31]:

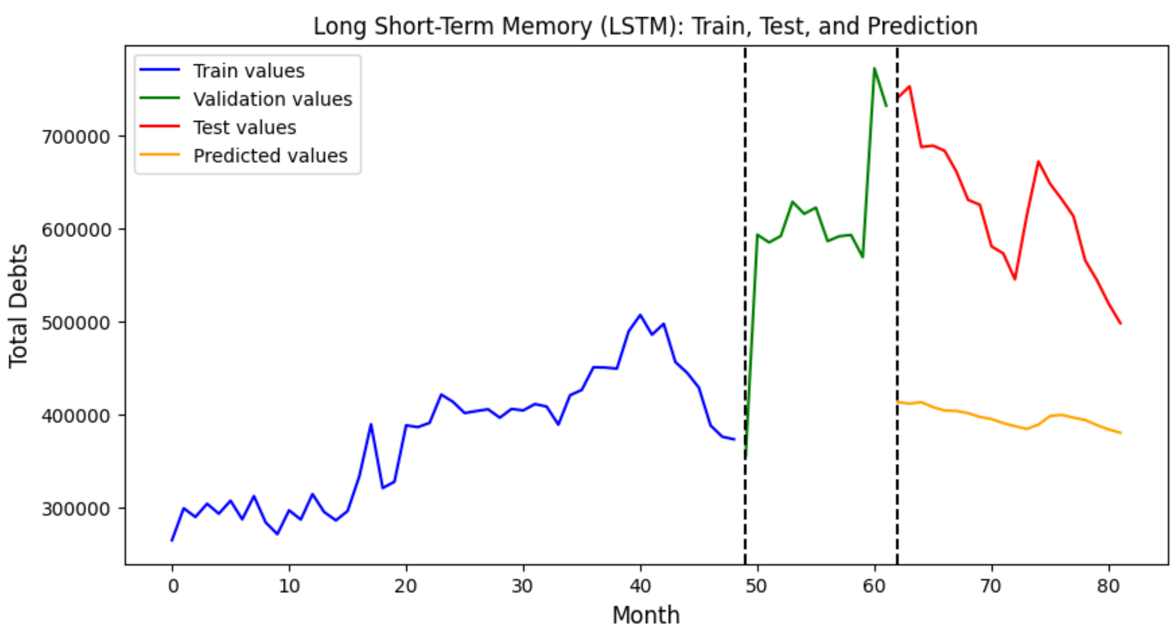
```
correct_name_algs = {
    "LSTM": "Long Short-Term Memory (LSTM)"
}
def plot_train_test_prediction(result):
    train = result["train"]
    val = result["val"]
    test = result["test"]
    algorithm = result["model"]
    correct_algorithm = correct_name_algs[algorithm]
    predictions = result["prediction"]

    concatenated_data = train + val + test
    split_train_index = len(train)
    split_val_index = len(train) + len(val)

    df = pd.DataFrame(concatenated_data)

    train_data = df[:split_train_index]
    val_data = df[split_train_index: split_val_index]
    test_data = df[split_val_index:]

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

    ax.plot(train_data.index, train_data[0], color="blue")
    ax.plot(val_data.index, val_data[0], color="green")
    ax.plot(test_data.index, test_data[0], color="red")
    ax.plot(test_data.index, predictions, color="orange")

    ax.set_xlabel("Month", fontsize=12)
    ax.set_ylabel("Total Debts", fontsize=12)

    ax.axvline(val_data.index[0], color='black', ls='--')
    ax.axvline(test_data.index[0], color='black', ls='--')

    ax.legend(["Train values", "Validation values", "Test values", "Predi

    ax.title.set_text(f"{correct_algorithm}: Train, Test, and Prediction"
    plt.show()
```

In [32]:
```
plot_train_test_prediction(results)
```


**Dataset: "lag_2_smedebtsu"**

In [33]:
```
date_results_path = "../results/lag_2_smedebtsu/DL_models/average_results
with open(date_results_path) as f:
    results = json.load(f)
```

In [34]:
```
plot_train_test_prediction(results)
```

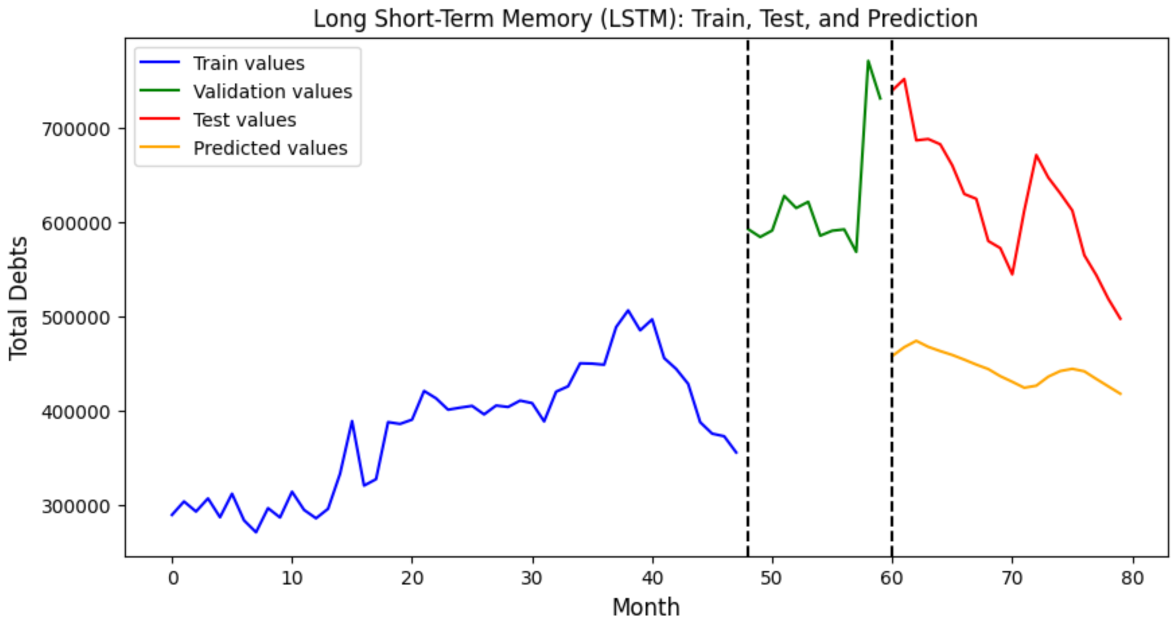



**Result interpretation**: It was observed that the LSTM models did not perform well overall. The limited availability of training data seemed to hinder the models' ability to effectively capture and leverage the underlying patterns in the time series. As a result, the predictions made by the

**Dataset: "lag_4_smedebtsu"**

In [35]:
```
date_results_path = "../results/lag_4_smedebtsu/DL_models/average_results
with open(date_results_path) as f:
    results = json.load(f)
```

In [36]:
```
plot_train_test_prediction(results)
```




# 6. Hypothesis test

- We will analyze of multiple base regressors and LSTM over 3 datasets "date_smedebtsu", "lag_2_smedebtsu", and "lag_4_smedebtsu"

In [37]:

```
def get_metric_files(result_path, method):
    rmse_benchmark = {
        'dataset': [],
        'LSTM': [],
        'linear_regression': [],
        'xgboost_regressor': [],
        'light_GBM': [],
        'random_forest_regressor': []
    }

    for dataset in os.listdir(result_path):
        dataset_path = os.path.join(result_path, dataset)
        for model_dir in os.listdir(dataset_path):
            file_path = os.path.join(dataset_path, model_dir, 'average_res
            with open(file_path) as f:
                data = json.load(f)
                if model_dir == 'DL_models':
                    rmse_benchmark['dataset'].append(data['dataset'])
                    rmse_benchmark['LSTM'].append(data[method])
                elif model_dir == 'ML_regression_models':
                    for element in data:
                        model_name = element["model"]
                        if model_name in rmse_benchmark:
                            rmse_benchmark[model_name].append(element[f'm

    return rmse_benchmark
```

## RMSE

In [38]:

```
result_path = "../results/"
rmse_benchmark = get_metric_files(result_path, method="RMSE")
# Get all RMSE metrics from other algorithms
rmse_benchmark_df = pd.DataFrame(rmse_benchmark)
rmse_benchmark_df
```

Out[38]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_ |
|---|---|---|---|---|---|---|
| **0** | date_smedebtsu | 246333.393263 | 111773.503881 | 81297.727978 | 85993.442301 | |
| **1** | lag_2_smedebtsu | 234095.810116 | 34555.416457 | 74626.367183 | 76812.045530 | |
| **2** | lag_4_smedebtsu | 186976.766552 | 33279.585977 | 59248.939071 | 78702.524397 | |

## MAE

In [39]:
```
# Get all MAE metrics from other algorithms
mae_benchmark = get_metric_files(result_path, method="MAE")
mae_benchmark_df = pd.DataFrame(mae_benchmark)
mae_benchmark_df
```

Out[39]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_ |
|---|---|---|---|---|---|---|
| 0 | date_smedebtsu | 239475.446375 | 103811.241691 | 68599.656438 | 82018.569004 | |
| 1 | lag_2_smedebtsu | 226236.612000 | 28757.928412 | 59833.640812 | 72410.788245 | |
| 2 | lag_4_smedebtsu | 178026.187000 | 27351.387495 | 54190.376125 | 74574.032671 | |

## MSE

In [40]:
```
# Get all MSE metrics from other algorithms
mse_benchmark = get_metric_files(result_path, method="MSE")
mse_benchmark_df = pd.DataFrame(mse_benchmark)
mse_benchmark_df
```

Out[40]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_f |
|---|---|---|---|---|---|---|
| 0 | date_smedebtsu | 6.068014e+10 | 1.407074e+10 | 7.099038e+09 | 1.147568e+10 | |
| 1 | lag_2_smedebtsu | 5.480085e+10 | 1.237683e+09 | 7.220648e+09 | 8.564133e+09 | |
| 2 | lag_4_smedebtsu | 3.496031e+10 | 1.187563e+09 | 4.592698e+09 | 9.175307e+09 | |

## R2 score

In [41]:
```
# Get all R2_score metrics from other algorithms
R2_score_benchmark = get_metric_files(result_path, method="R2_score")
R2_score_benchmark_df = pd.DataFrame(R2_score_benchmark)
R2_score_benchmark_df
```

Out[41]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_forest_ |
|---|---|---|---|---|---|---|
| 0 | date_smedebtsu | -11.990665 | -21.891185 | -10.063880 | -14.356732 | |
| 1 | lag_2_smedebtsu | -10.732000 | -1.048521 | -10.742697 | -10.193474 | |
| 2 | lag_4_smedebtsu | -6.484453 | -1.003279 | -5.233603 | -10.936199 | |

## Friedmen test

So we begin by checking if the Friedman test is significant: Are there any significant differences in the rankings of the learners over all datasets?

Hypothesis:

- H0 - There are not differences
- H1 - There are significant differences

In [42]:
```
mae_benchmark_df
```

Out[42]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_ |
|---|---|---|---|---|---|---|
| **0** | date_smedebtsu | 239475.446375 | 103811.241691 | 68599.656438 | 82018.569004 | |
| **1** | lag_2_smedebtsu | 226236.612000 | 28757.928412 | 59833.640812 | 72410.788245 | |
| **2** | lag_4_smedebtsu | 178026.187000 | 27351.387495 | 54190.376125 | 74574.032671 | |

In [43]:
```
algorithms_names = mae_benchmark_df.drop('dataset', axis=1).columns
performances_array = mae_benchmark_df[algorithms_names].values

print(friedmanchisquare(*performances_array))
```

```
FriedmanchisquareResult(statistic=8.400000000000006, pvalue=0.014995576820477
66)
```

**Friedmen test interpretation**: Based on the observed p-values, which are smaller than 0.05, we can reject the null hypothesis (H0) and conclude that there are significant differences among the results. Therefore, we proceeded to conduct a Nemenyi test to compare pairs of regression algorithms.

## Nemenyi test

- The Nemenyi test is a statistical test used to compare multiple treatments or algorithms simultaneously. It determines whether there are significant differences between the rankings or performance of the treatments, helping to identify the superior ones.
- We can use the **MAE benchmark** to conduct the Hypothesis test

In [44]:
```
mae_benchmark_df
```

Out[44]:

| | dataset | LSTM | linear_regression | xgboost_regressor | light_GBM | random_ |
|---|---|---|---|---|---|---|
| **0** | date_smedebtsu | 239475.446375 | 103811.241691 | 68599.656438 | 82018.569004 | |
| **1** | lag_2_smedebtsu | 226236.612000 | 28757.928412 | 59833.640812 | 72410.788245 | |
| **2** | lag_4_smedebtsu | 178026.187000 | 27351.387495 | 54190.376125 | 74574.032671 | |

In [45]:
```
correct_algorithms_names = {
    'linear_regression': 'Linear Regression',
    'xgboost_regressor': 'XGBoost Regressor',
    'light_GBM': 'LightGBM',
    'random_forest_regressor': 'Random Forest Regressor',
    'LSTM': 'Long Short-Term Memory (LSTM)'
}

# First, we extract the algorithms names.
algorithms_names = mae_benchmark_df.drop('dataset', axis=1).columns
correct_algorithms_names = [correct_algorithms_names[name] for name in al
correct_algorithms_names
```

Out[45]:
```
['Long Short-Term Memory (LSTM)',
 'Linear Regression',
 'XGBoost Regressor',
 'LightGBM',
 'Random Forest Regressor']
```

In [46]:
```
# Then, we extract the performances as a numpy.ndarray.
performances_array = rmse_benchmark_df[algorithms_names].values
performances_array
```

Out[46]:
```
array([[246333.39326337, 111773.50388123,  81297.72797762,
         85993.44230133,  71194.2623387 ],
       [234095.8101163 ,  34555.41645723,  74626.36718286,
         76812.04553    ,  66442.1610892 ],
       [186976.76655182,  33279.58597666,  59248.93907111,
         78702.52439711,  59635.77691899]])
```

In [47]:
```
# Ranking of algorithms
ranks = np.array([rankdata(p) for p in performances_array])
pd.DataFrame(ranks, columns=algorithms_names, index=mae_benchmark_df["data
```

Out[47]:

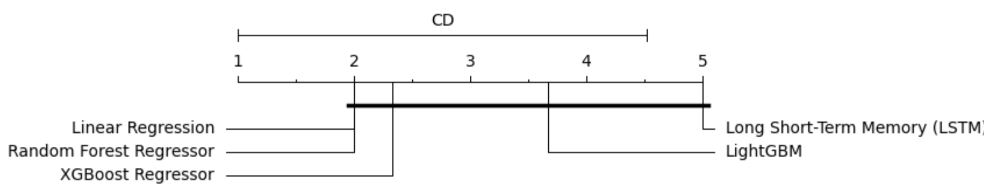
| | LSTM | linear_regression | xgboost_regressor | light_GBM | random_forest_regress |
|---|---|---|---|---|---|
| **dataset** | | | | | |
| **date_smedebtsu** | 5.0 | 4.0 | 2.0 | 3.0 | |
| **lag_2_smedebtsu** | 5.0 | 1.0 | 3.0 | 4.0 | : |
| **lag_4_smedebtsu** | 5.0 | 1.0 | 2.0 | 4.0 | : |

In [48]:
```
# Calculating the average ranks.
average_ranks=np.mean(ranks,axis=0)
print('\n'.join('{}: average rank: {}'.format(a,r) for a,r in zip(correct_
```

```
Long Short-Term Memory (LSTM): average rank: 5.0
Linear Regression: average rank: 2.0
XGBoost Regressor: average rank: 2.3333333333333335
LightGBM: average rank: 3.6666666666666665
Random Forest Regressor: average rank: 2.0
```

## Plot CD

In [49]:
```
cd = compute_CD(average_ranks, n=len(mae_benchmark_df), alpha='0.05', tes
graph_ranks(average_ranks, names=correct_algorithms_names, cd=cd, width=1(
```




**Result interpretation**: You can utilize Linear Regression and Random Forest Regressor for analyzing this data. However, it is important to note that these models have limitations due to the small size of the dataset.