Demand Prediction

Comparing demand prediction Project Report

Instructor: Dr Murali Shankar

Authored by Sivarangareddy Pondugula

Reg No: 811197074

Introduction:

Machine learning and deep learning are becoming more popular in technological advancement. It is used to build up the application features and capabilities so that, eventually, it won't need human involvement or a predefined set of instructions to carry out most of its operations. This research aims to learn about the potential of machine learning Deep Learning models for predicting sales demand and to document the exploratory data analysis of this process.

Predicting potential buyers' interest in a product or service in the future is known as demand forecasting or sales forecasting. Looking at the company's sales data from the past and seasonal patterns might help you predict demand.
We have explored different machine-learning techniques in this project and compared those results. We can also extend this project by creating the ensemble method.

Problem statement:

We have five years of store-item sales data, and we predict next month's sales for 50 different items at 10 other stores.

Data fields

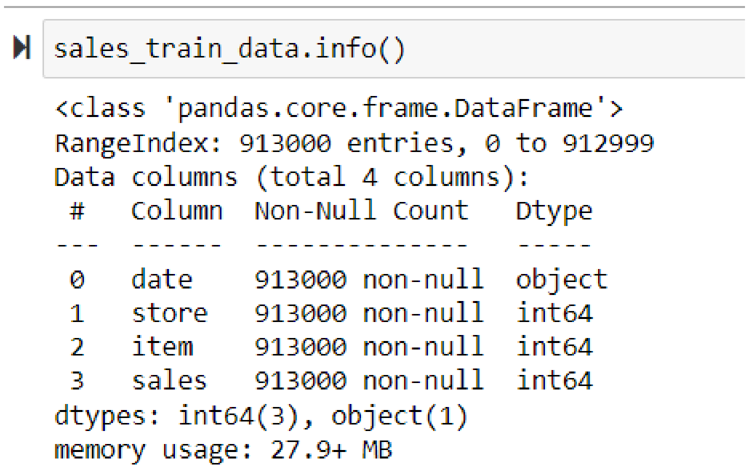
- **date** - Date of the sale date. There are no holiday effects or store closures.
- **store** - Store ID
- **item** - Item ID
- **sales** - Number of items sold at a particular store on a particular date.

Business Goal

We must make a model to forecast the demand with the available variables. The management will use it to understand the market for the product. They can accordingly manipulate the manufacturing supply and storage and use it as a business strategy. To meet certain demand levels. Further, the model will be a good way for management to understand the product dynamics of a new market.

Overview of your data set

The data set contains various factors affecting a particular demand for the store. There are four variables: date, store items, and sales. There is a total of 913000 observations.

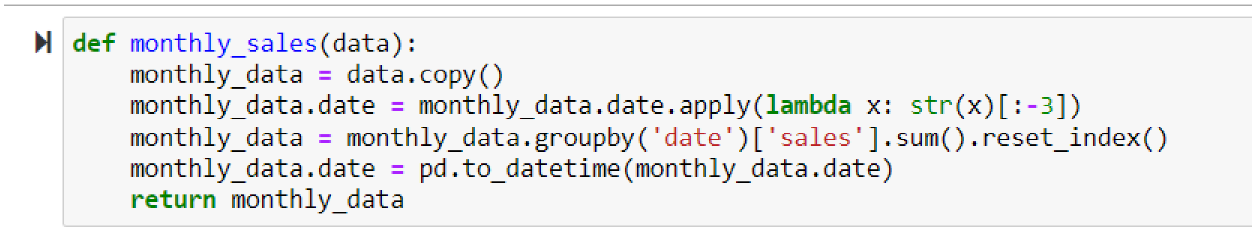


```
sales_train_data.info()
```

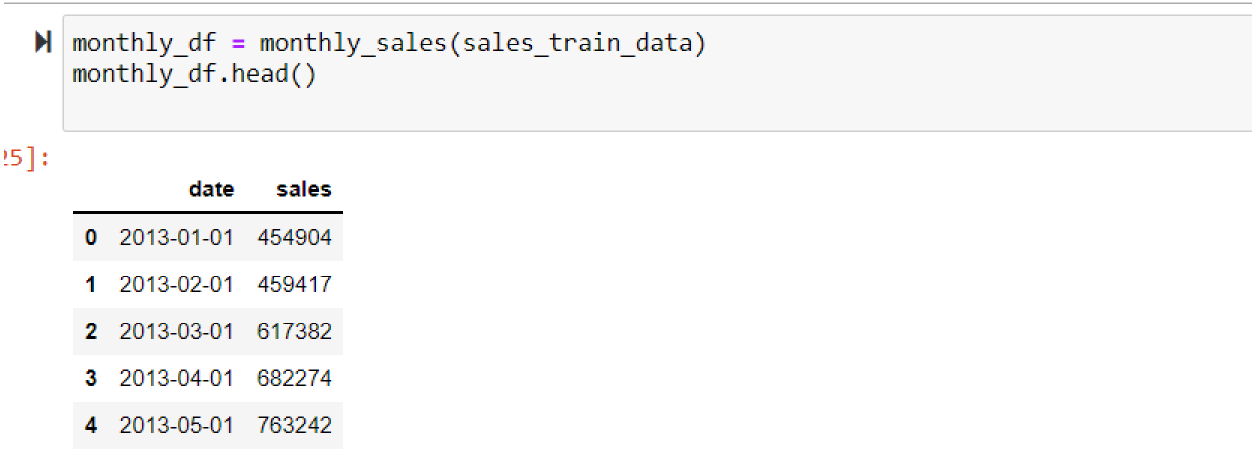
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 913000 entries, 0 to 912999
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   date    913000 non-null  object
 1   store   913000 non-null  int64
 2   item    913000 non-null  int64
 3   sales   913000 non-null  int64
dtypes: int64(3), object(1)
memory usage: 27.9+ MB
```

To begin, we must import the data and transform it into a format all models can read. We can go on to the next phase once that's done. Each data row represents the sales made at one of the 10 stores on a specific day. Since we aim to make monthly sales predictions, we will begin by consolidating sales data from all stores and days.

Each row in our new data frame reflects the combined monthly sales of all our retail locations.



```
def monthly_sales(data):
    monthly_data = data.copy()
    monthly_data.date = monthly_data.date.apply(lambda x: str(x)[:-3])
    monthly_data = monthly_data.groupby('date')['sales'].sum().reset_index()
    monthly_data.date = pd.to_datetime(monthly_data.date)
    return monthly_data
```



```
monthly_df = monthly_sales(sales_train_data)
monthly_df.head()
```

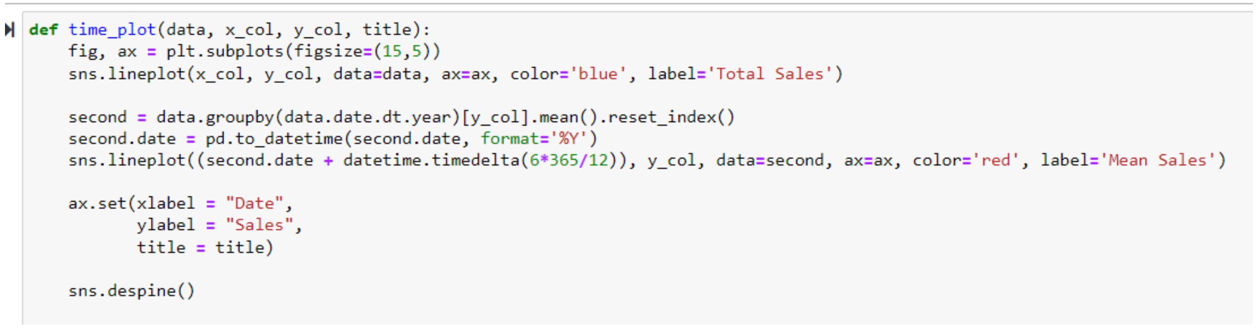
5]:

	date	sales
0	2013-01-01	454904
1	2013-02-01	459417
2	2013-03-01	617382
3	2013-04-01	682274
4	2013-05-01	763242

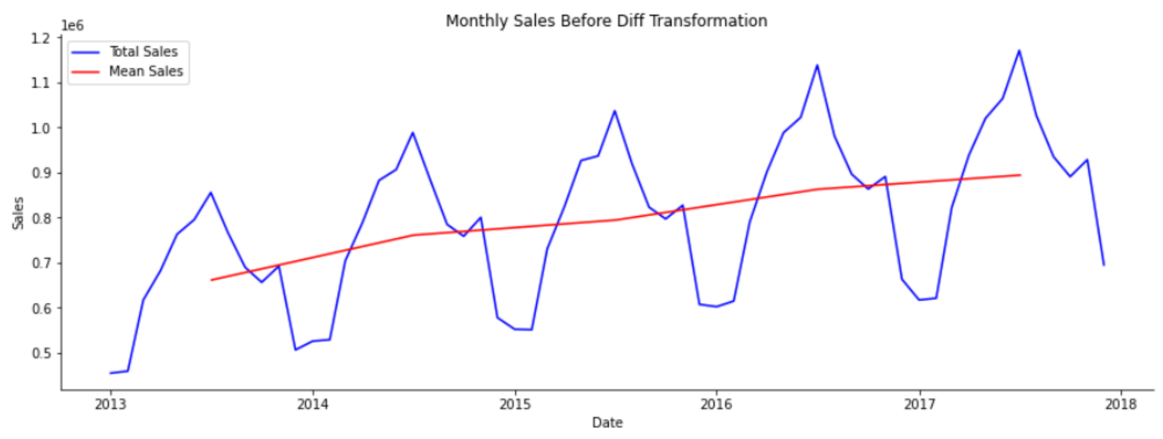
DATA PREPARATION

Stationary

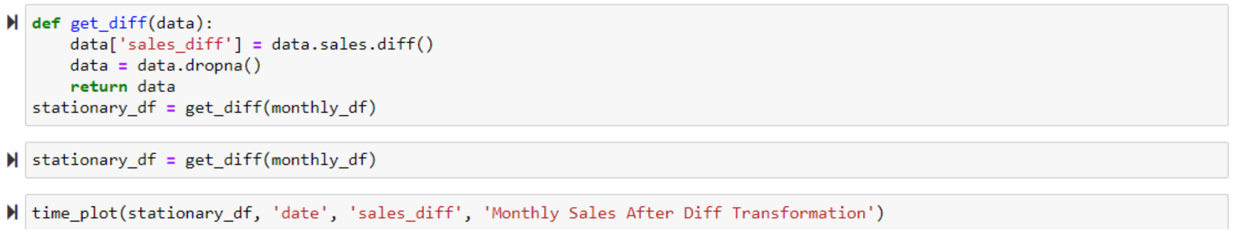
If we plot the total monthly sales against time, we will see a rise in average monthly sales. This indicates that our data is not stationary. To keep it consistent, we will first compute the difference in sales between each month, then insert this number into our data frame as a new column.



```
def time_plot(data, x_col, y_col, title):
    fig, ax = plt.subplots(figsize=(15,5))
    sns.lineplot(x_col, y_col, data=data, ax=ax, color='blue', label='Total Sales')

    second = data.groupby(data.date.dt.year)[y_col].mean().reset_index()
    second.date = pd.to_datetime(second.date, format='%Y')
    sns.lineplot((second.date + datetime.timedelta(6*365/12)), y_col, data=second, ax=ax, color='red', label='Mean Sales')

    ax.set(xlabel = "Date",
           ylabel = "Sales",
           title = title)

    sns.despine()
```



As we can see that their data is not stationary, we have also computed the ADF test, which indicates that it is not stationary. To make the stationary data, we have used different Stationary strategies.

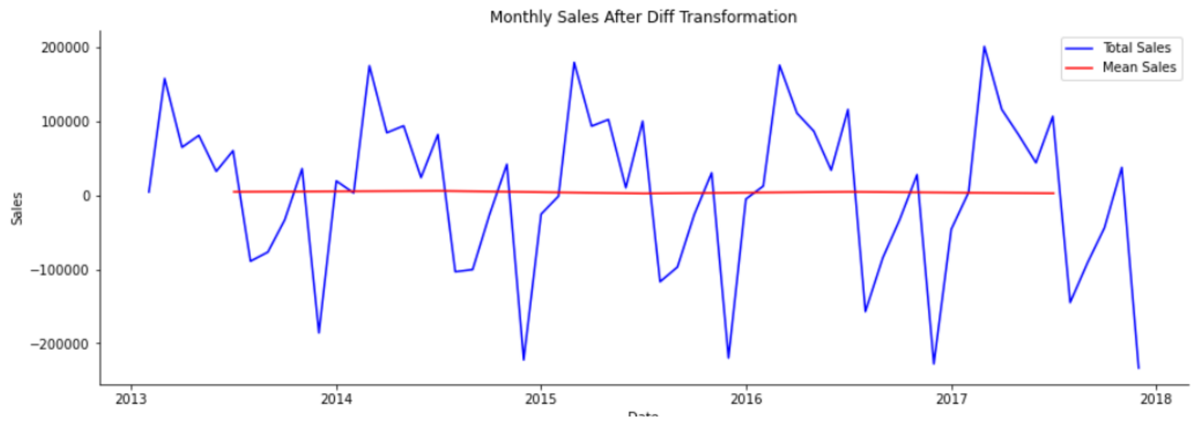


```
def get_diff(data):
    data['sales_diff'] = data.sales.diff()
    data = data.dropna()
    return data
stationary_df = get_diff(monthly_df)
```

```
stationary_df = get_diff(monthly_df)
```

```
time_plot(stationary_df, 'date', 'sales_diff', 'Monthly Sales After Diff Transformation')
```

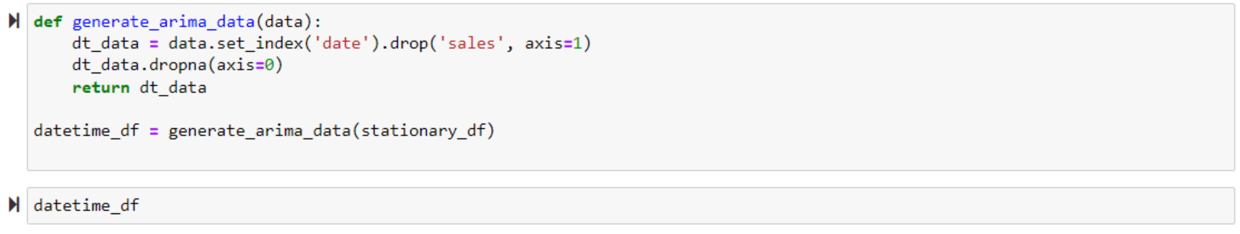
Now that our data represent monthly sales, we have transformed it into stationery



Supervised Data

We organised the data following the many sorts of models we use. To do this, we will first establish two distinct structures. The first structure will be used for ARIMA modelling, while the second structure will be used for modelling everything else.

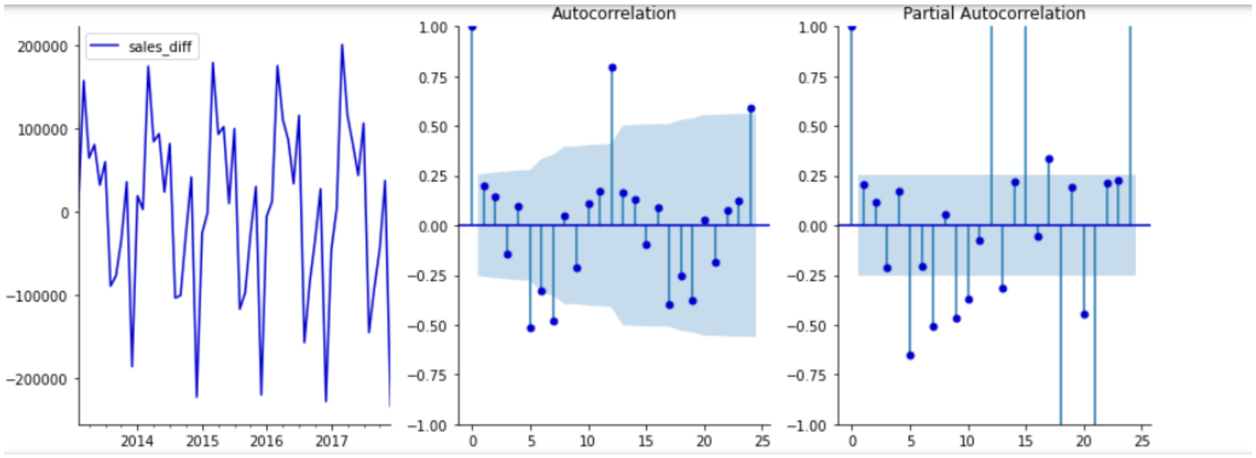
To make with the ARIMA model, we consider only a date time index and sales difference column.



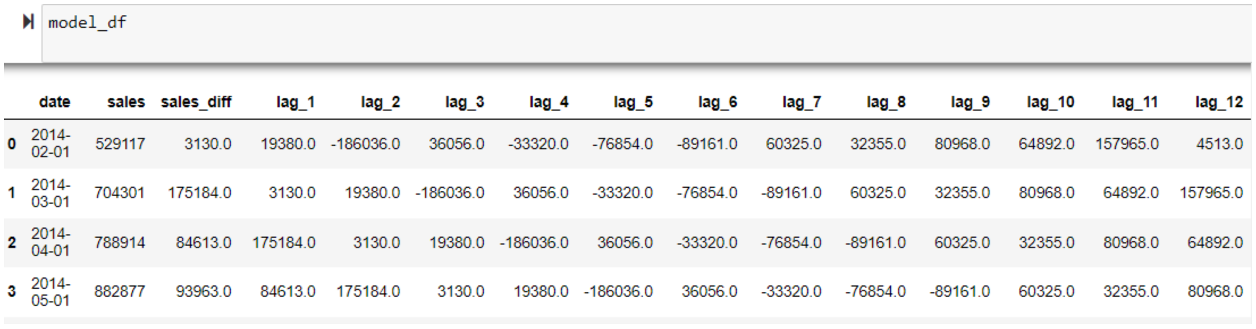
```
def generate_arima_data(data):
    dt_data = data.set_index('date').drop('sales', axis=1)
    dt_data.dropna(axis=0)
    return dt_data

datetime_df = generate_arima_data(stationary_df)
```

```
datetime_df
```

To use with the other models, like Linear regression, Random Forest Regression, XGBoost, and LSTMs, we have generated a new data frame where each feature will be a month's worth of sales data. Autocorrelation and partial autocorrelation plots, as well as the principles for picking lags in ARIMA modelling, will be used to settle on a final number of months to include in our feature set. Using this method, we can maintain a uniform period for the ARIMA and regressive models to use for their historical analysis.



Considering the above, we have decided that the look-back time we will use will be one year. As a result, we will construct a data frame with thirteen columns, one for each of the twelve months and a column for the dependent variable that we will be analysing, which will be the difference in sales.



```
model_df
```

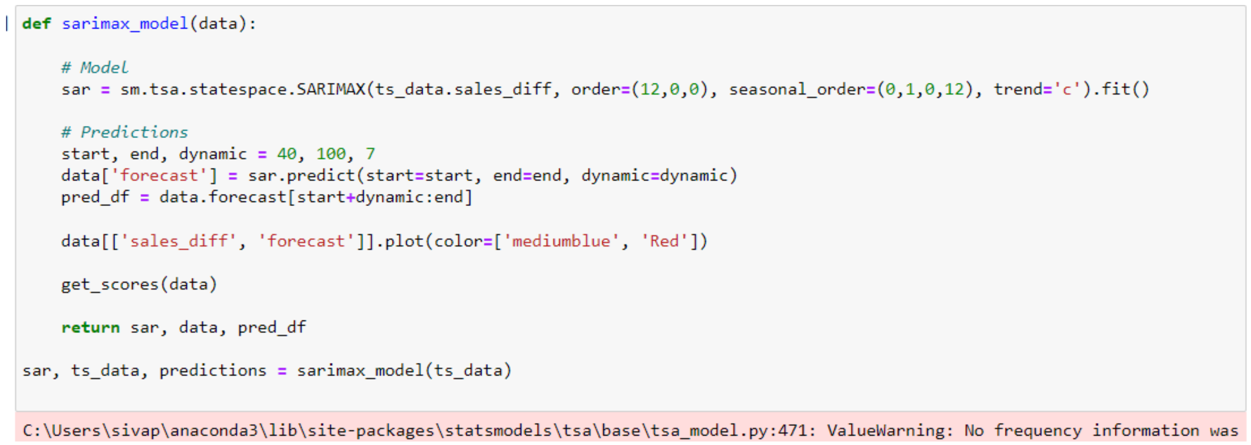
	date	sales	sales_diff	lag_1	lag_2	lag_3	lag_4	lag_5	lag_6	lag_7	lag_8	lag_9	lag_10	lag_11	lag_12
0	2014-02-01	529117	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0	157965.0	4513.0
1	2014-03-01	704301	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0	157965.0
2	2014-04-01	788914	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0	64892.0
3	2014-05-01	882877	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0	80968.0

Modelling

1. To create and assess our models, we use a series of helper functions that perform the following procedures.
2. We divide our data such that the most recent twelve months are included in the test set, and the remainder is used to train our model.
3. We scale the data using a min-max scaler so that all of our variables will be contained inside the range of -1 to 1.
4. Reverse scaling: After running our models, we will use this helper function to reverse the scaling above.
5. We have constructed a data frame that combines the actual sales collected in our test set and the expected outcomes from our model to evaluate how successful our efforts have been.
6. We have used the RMSE and use to evaluate the model prediction and compare the results

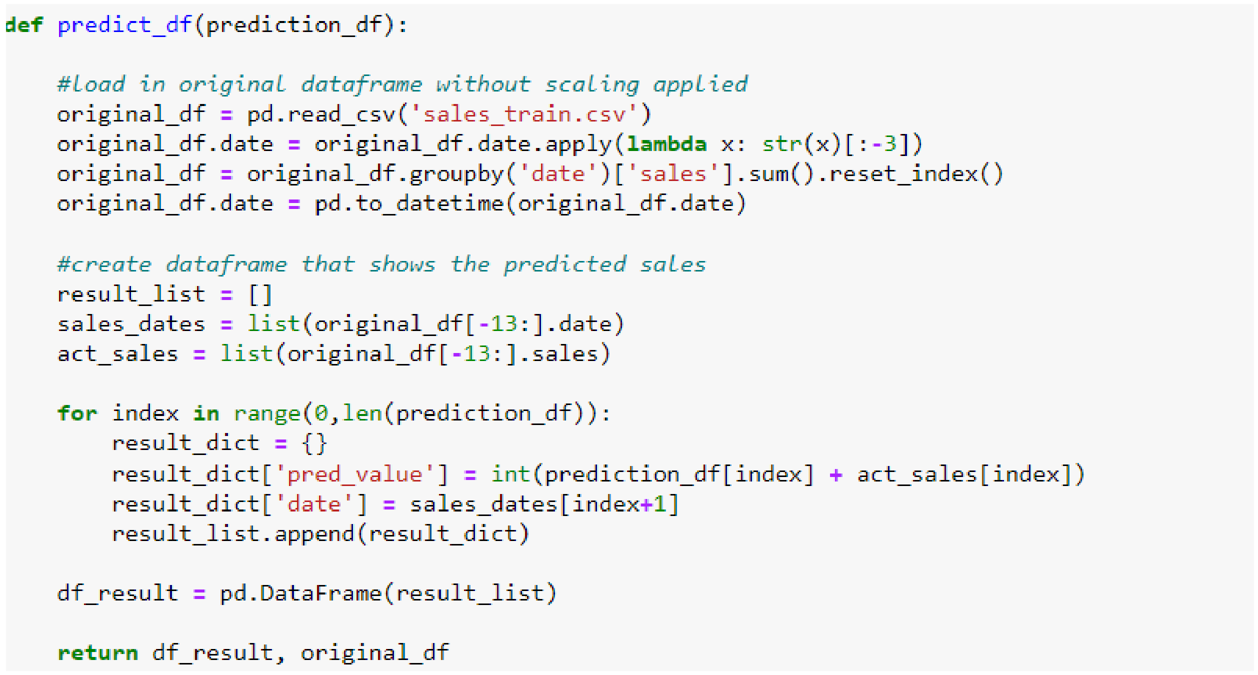
ARIMA

The ARIMA model looks quite different from the other models shown. We have used the SARIMAX package from stats models to train the model and provide dynamic predictions. The SARIMA model may be broken down into many different parts. Those are p as the autoregressive model d, the differencing term q, and the moving average model S, enabling us to add a seasonal component.



```
def sarimax_model(data):

    # Model
    sar = sm.tsa.statespace.SARIMAX(ts_data.sales_diff, order=(12,0,0), seasonal_order=(0,1,0,12), trend='c').fit()

    # Predictions
    start, end, dynamic = 40, 100, 7
    data['forecast'] = sar.predict(start=start, end=end, dynamic=dynamic)
    pred_df = data.forecast[start+dynamic:end]

    data[['sales_diff', 'forecast']].plot(color=['mediumblue', 'Red'])

    get_scores(data)

    return sar, data, pred_df

sar, ts_data, predictions = sarimax_model(ts_data)
```

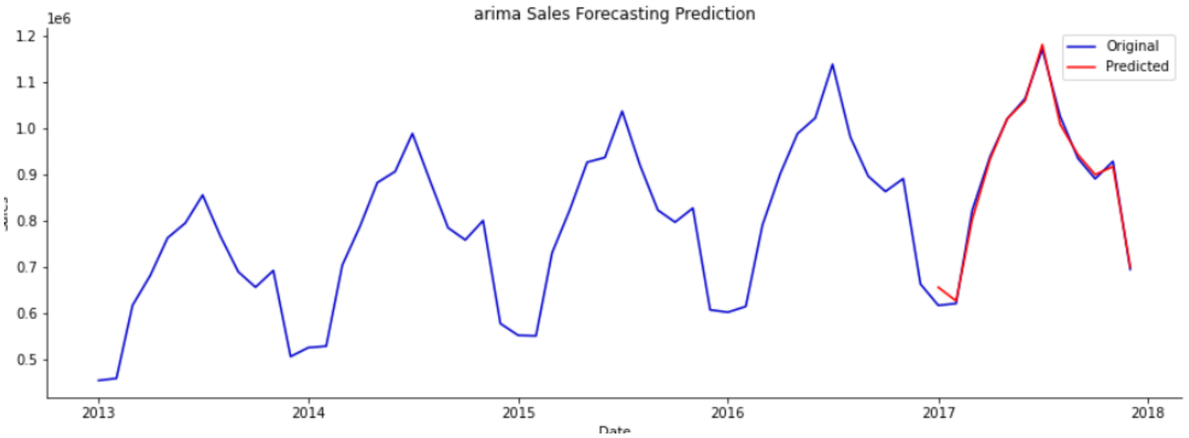
```
C:\Users\sivap\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: No frequency information was
```



```
def predict_df(prediction_df):

    #load in original dataframe without scaling applied
    original_df = pd.read_csv('sales_train.csv')
    original_df.date = original_df.date.apply(lambda x: str(x)[:-3])
    original_df = original_df.groupby('date')['sales'].sum().reset_index()
    original_df.date = pd.to_datetime(original_df.date)

    #create dataframe that shows the predicted sales
    result_list = []
    sales_dates = list(original_df[-13:].date)
    act_sales = list(original_df[-13:].sales)

    for index in range(0,len(prediction_df)):
        result_dict = {}
        result_dict['pred_value'] = int(prediction_df[index] + act_sales[index])
        result_dict['date'] = sales_dates[index+1]
        result_list.append(result_dict)

    df_result = pd.DataFrame(result_list)

    return df_result, original_df
```

In the above code, we can see that our model and then dynamically estimate the outcomes for the most recent 12 months of data. The actual sales from the previous month are used to project the next month's sales from static forecasts. In contrast, dynamic forecasting relies on past sales forecasts to inform projections for the next month.

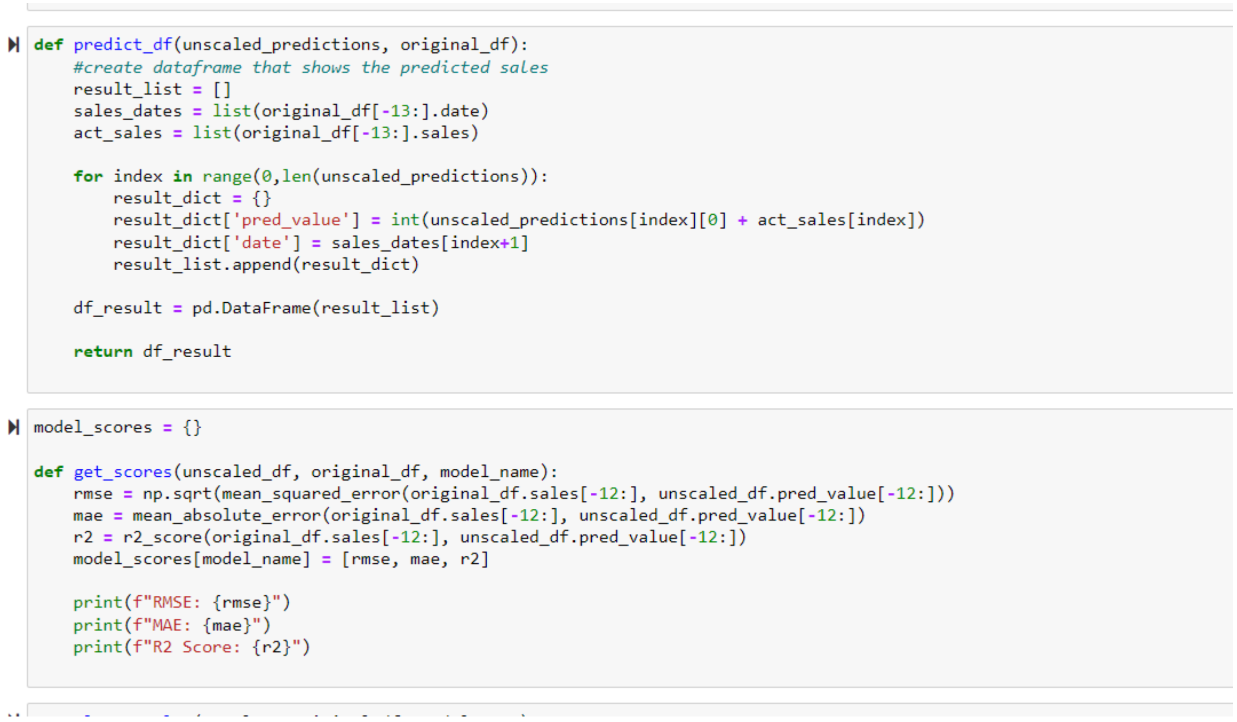
When we look at the below diagram we can predict very good results with help of ARIMA



Data Preparation for Machine learning models

The fit-predict framework provided by sci-kit-learn allows us to create regressive models. This will enable us to establish a common modelling framework from which we can build all subsequent models. The following function uses many of the utilities mentioned earlier, including data partitioning, model execution, and the presentation of root mean squared error and mean absolute error.

```
def scale_data(train_set, test_set):
    #apply Min Max Scaler
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaler = scaler.fit(train_set)

    # reshape training set
    train_set = train_set.reshape(train_set.shape[0], train_set.shape[1])
    train_set_scaled = scaler.transform(train_set)

    # reshape test set
    test_set = test_set.reshape(test_set.shape[0], test_set.shape[1])
    test_set_scaled = scaler.transform(test_set)

    X_train, y_train = train_set_scaled[:, 1:], train_set_scaled[:, 0:1].ravel()
    X_test, y_test = test_set_scaled[:, 1:], test_set_scaled[:, 0:1].ravel()

    return X_train, y_train, X_test, y_test, scaler

X_train, y_train, X_test, y_test, scaler_object = scale_data(train, test)
```

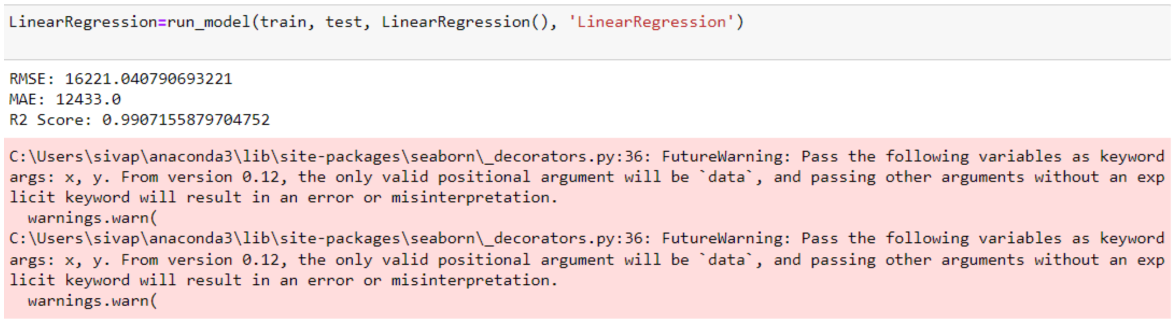


```python
def predict_df(unscaled_predictions, original_df):
    #create dataframe that shows the predicted sales
    result_list = []
    sales_dates = list(original_df[-13:].date)
    act_sales = list(original_df[-13:].sales)

    for index in range(0,len(unscaled_predictions)):
        result_dict = {}
        result_dict['pred_value'] = int(unscaled_predictions[index][0] + act_sales[index])
        result_dict['date'] = sales_dates[index+1]
        result_list.append(result_dict)

    df_result = pd.DataFrame(result_list)

    return df_result
```

```python
model_scores = {}

def get_scores(unscaled_df, original_df, model_name):
    rmse = np.sqrt(mean_squared_error(original_df.sales[-12:], unscaled_df.pred_value[-12:]))
    mae = mean_absolute_error(original_df.sales[-12:], unscaled_df.pred_value[-12:])
    r2 = r2_score(original_df.sales[-12:], unscaled_df.pred_value[-12:])
    model_scores[model_name] = [rmse, mae, r2]

    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
    print(f"R2 Score: {r2}")
```

Linear Regression

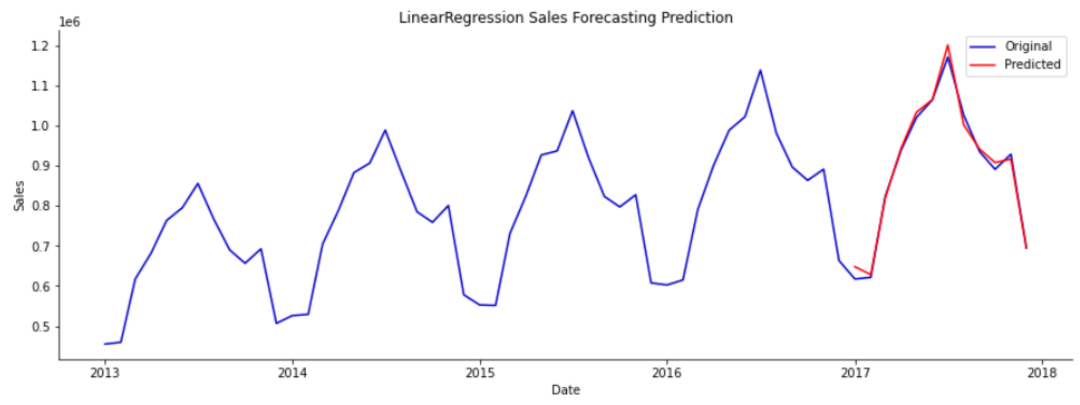
Initially, we have made linear regression by scaling data we can predict original data



```python
LinearRegression=run_model(train, test, LinearRegression(), 'LinearRegression')
```

```
RMSE: 16221.040790693221
MAE: 12433.0
R2 Score: 0.9907155879704752
```

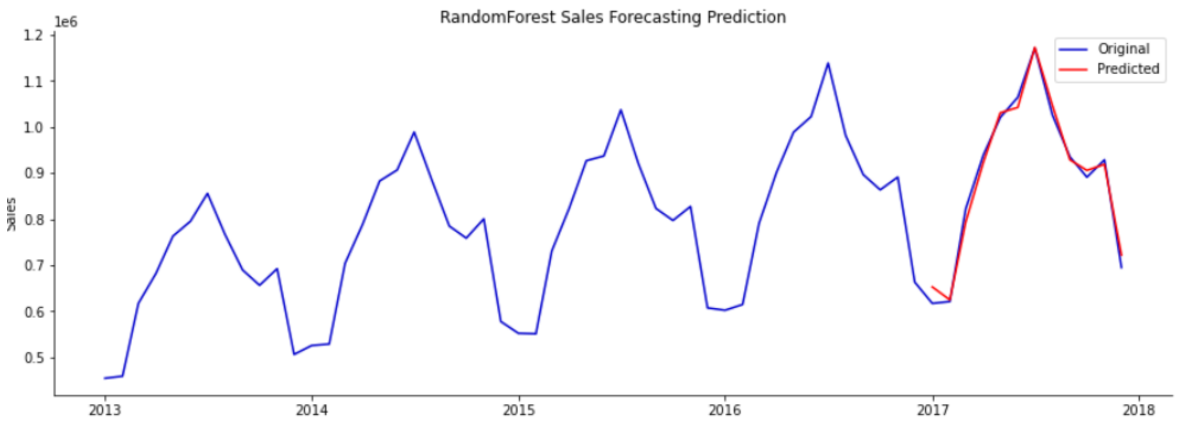
```
C:\Users\sivap\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variables as keyword
args: x, y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an exp
licit keyword will result in an error or misinterpretation.
  warnings.warn(
C:\Users\sivap\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variables as keyword
args: x, y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an exp
licit keyword will result in an error or misinterpretation.
  warnings.warn(
```



From the above result, linear regression can accurately predict the original data.

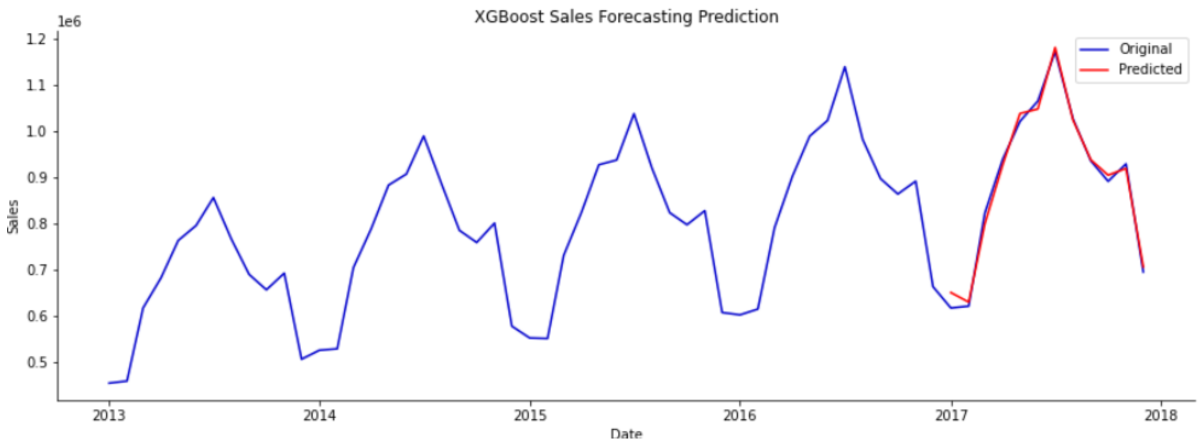
Random forest

We also used the scaled data for the random forest with n_estimators=100, and Max_depth=20. Since the data has been converted monthly, we have not used the bagging or boosting technique.



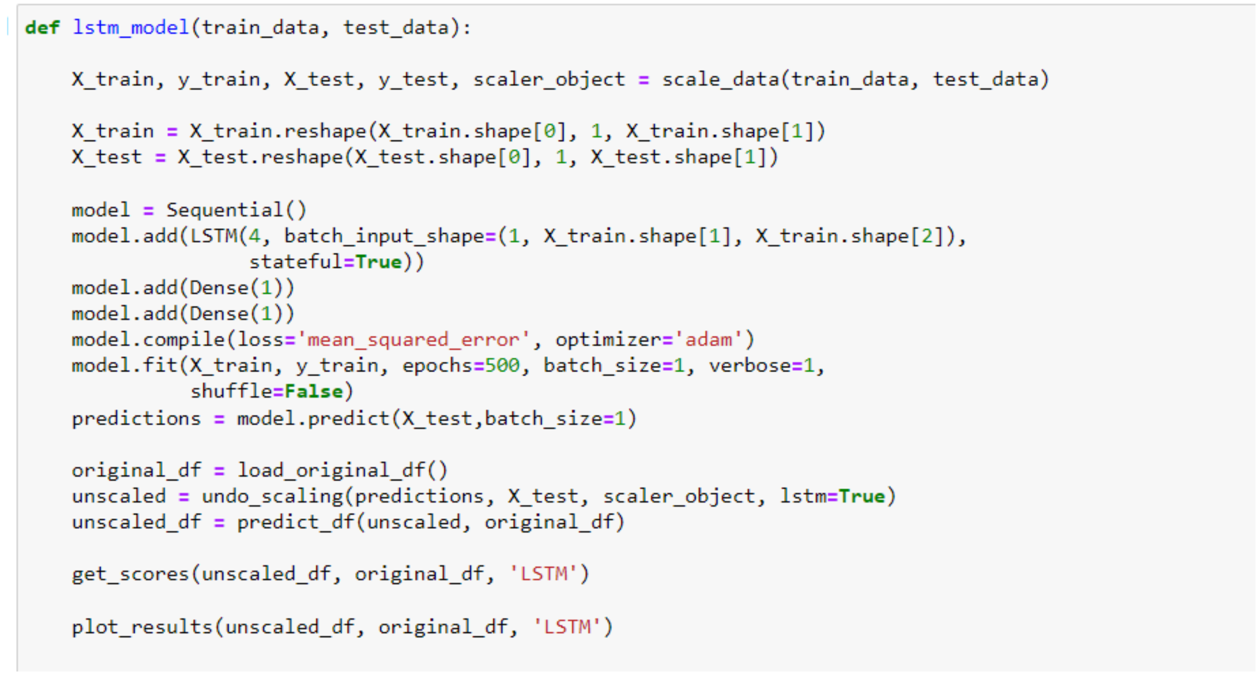
XGBOOST

Similarly, for the XGboost we have used the estimators as 100 and learning rate =0.2. When compared to all other models, RMSE and MAE are good.



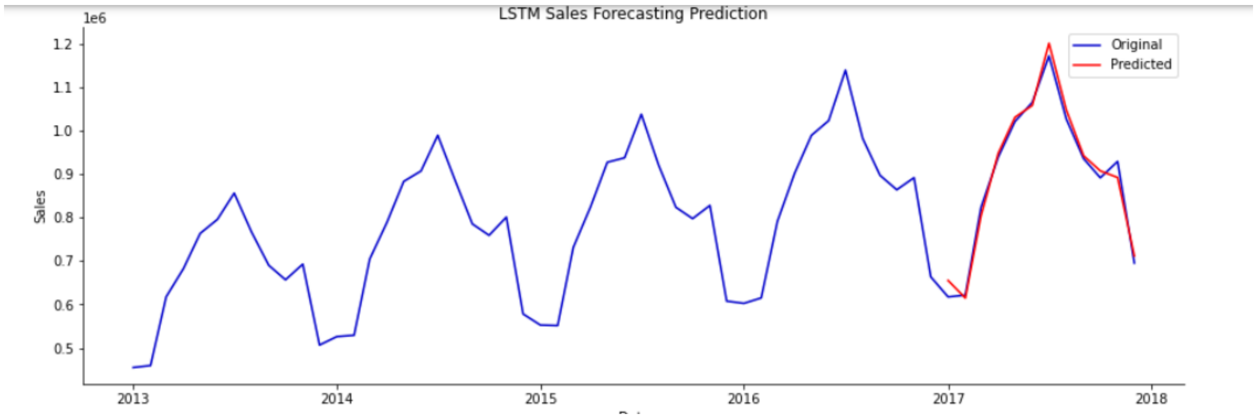
LSTM

The recurrent neural network, also known as the RNN long short-term memory (LSTM), is a powerful technique for formulating predictions using sequential input. To accomplish this objective, we will use a very basic LSTM. Seasonal details and increased model complexity may be included to achieve a higher level of accuracy. In this case, we have used the two hidden layers with a loss function of MSE, optimiser as Adam and epochs 500. Since we are implementing deep learning, there is no need to use masked data. Interestingly, we have used the unscaled data or not converted (log data). In this case, we have used a helper function to unscaled_df the scaled data.



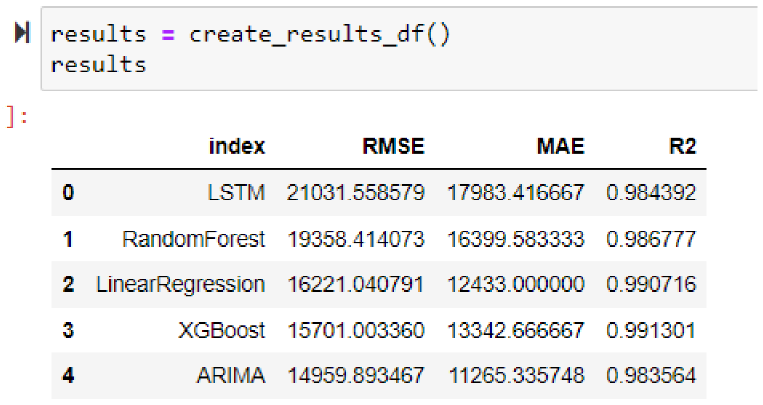
```
def lstm_model(train_data, test_data):

    X_train, y_train, X_test, y_test, scaler_object = scale_data(train_data, test_data)

    X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
    X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

    model = Sequential()
    model.add(LSTM(4, batch_input_shape=(1, X_train.shape[1], X_train.shape[2]),
                   stateful=True))
    model.add(Dense(1))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(X_train, y_train, epochs=500, batch_size=1, verbose=1,
              shuffle=False)
    predictions = model.predict(X_test,batch_size=1)

    original_df = load_original_df()
    unscaled = undo_scaling(predictions, X_test, scaler_object, lstm=True)
    unscaled_df = predict_df(unscaled, original_df)

    get_scores(unscaled_df, original_df, 'LSTM')

    plot_results(unscaled_df, original_df, 'LSTM')
```

The below diagram shows the LSTM prediction with actuals



Comparing Models

To evaluate the accuracy of each model, we will calculate the root mean squared error (RMSE) and the mean absolute error (MAE). Despite their somewhat dissimilar intuitive and mathematical meanings, these measures are often used to assess a model's efficacy.

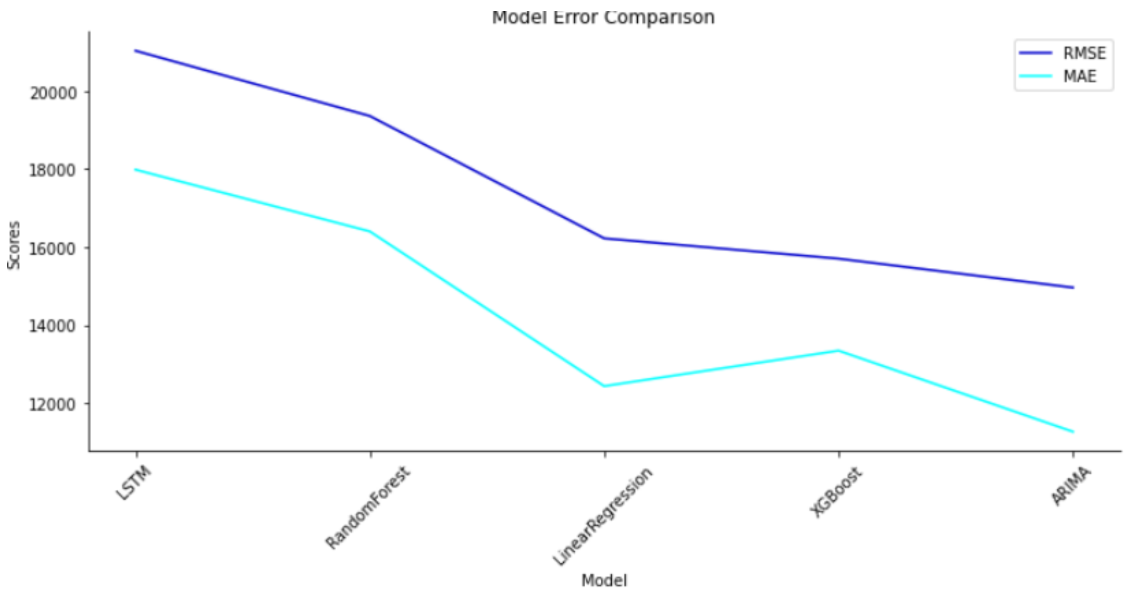


```
results = create_results_df()
results
```

]:

	index	RMSE	MAE	R2
0	LSTM	21031.558579	17983.416667	0.984392
1	RandomForest	19358.414073	16399.583333	0.986777
2	LinearRegression	16221.040791	12433.000000	0.990716
3	XGBoost	15701.003360	13342.666667	0.991301
4	ARIMA	14959.893467	11265.335748	0.983564

It is clear to us that the accuracy of our model outputs varies widely, even though they seemed comparable in the graphs that were just shown. The comparison between the two may be seen in the following image.



The ARIMA model had the most outstanding overall performance, followed by the linear regression and XGboost models in second and third place.The models have very little fine-tuning done to them to reduce complexity. For example, the LSTM might benefit from having many extra nodes and layers to improve its performance.

To get a better result with more accuracy.

- We can increase the degree of complexity in the case of Arima. It is possible to modify the model and create features for seasonal information, holidays, weekends, etc.
- Tune models using cross-validation or similar approaches to prevent overfitting data. We can also implement bagging or boosting techniques to avoid overfitting.
- Lastly, we can make the ensemble method to make better accuracy. Eventually, it leads to avoiding overfitting. In my opinion, an ensemble can make look even better than weak learners or base learners

Overall, understanding the seasonality, trends, cyclicality, unpredictability in sales and distribution and other features is a top priority for any significant firm, making time series analysis a crucial part of data analytics. These considerations aid businesses in making vital decisions with complete information. Every method has their strength and weakness in predicting times series, but when we combine or create an ensemble of the data, we can cancel out the fault (overfit). We can further work on this project by making the ensemble method.

Contribution

1. As we know, feature engineering is a vital task in the case of machine learning. In this project, we have organised data-converted stationery, log form and scaling, which is not required for Deep Learning to perform feature engineering. Even if we look at the code, we have unscaled the data. Still, it can compete with all other time series models.
2. No tuning was done to the model. Instead, the optimal setup arrived via rapid iteration. Tuning at least the number of neurons and the number of training epochs would provide far better outcomes. A callback for immediate termination of training may be helpful.
3. Creating a stacking ensemble model will be tremendously helpful in avoiding overfitting. It will cancel out the overfitting features and make the model more robust.
4. ARIMA model has excellent predictive power compared to all other base models but on the other side tuning parameters will make the more predictive power to every model

Reference

https://www.udemy.com/course/time-series-analysis/

https://www.kaggle.com/code/drindeng/store-item-dem-forecast-with-deep-neural-network

https://www.kaggle.com/competitions/demand-forecasting-kernels-only

https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-to-time-series-analysis/

https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-to-time-series-analysis/