FUTURE SALES PREDICTION PROJECTS USING MACHINE LEARING

BATCH MEMBER:

N.P.SUSHMITHA

K.SARANYA

S.PRIYA DHARSHINI

A.NEHA

INTRODUTION:

Machine learning is a powerful tool that can be used to predict sales and improve business outcomes. In this article, we will discuss how machine learning can be used to predict sales and the different methods that can be used to do so.

One of the most common methods used to predict sales is **regression analysis.** This method involves using historical sales data to train a model that can predict future sales. The model can take into account factors such as **past sales, marketing campaigns, and economic indicators** to make its predictions.

Another popular method for predicting sales is **time series analysis**. This method involves using historical sales data to identify patterns and trends in sales over time. The model can then use these patterns to make predictions about future sales. This method is particularly useful for predicting sales in seasonal industries, such as retail and tourism.

Another approach is using **decision tree-based algorithms** like **Random Forest, Gradient Boosting** etc. These algorithms are particularly useful when there are many factors that can influence sales, such as product features, customer demographics, and market conditions. The algorithm can help identify the most important factors and use them to make predictions.

In addition to these methods, machine learning can also be used to predict sales through the use of **neural networks.** Neural networks are a type of machine learning algorithm

that can learn to recognize patterns in data. They can be trained on large amounts of sales data and can make predictions about future sales.

Machine learning can also be used to predict sales by using **clustering algorithms,** which can help identify groups of similar customers. This information can then be used to create targeted marketing campaigns and improve sales strategies.

Sales Prediciton Using Python:

So, now we will try to predict sales using various machine learning techniques.

Code:

1. Importing Libraries

```
# EDA Libraries:

import pandas as pd
import numpy as np

import matplotlib.colors as col
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import datetime
from pathlib import Path
import random

# Scikit-Learn models:

from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.ensemble import RandomForestRegressor
from xgboost.sklearn import XGBRegressor
from sklearn.model_selection import KFold, cross_val_score, train_test_split

# LSTM:

import keras
from keras.layers import Dense
from keras.models import Sequential
from keras.callbacks import EarlyStopping
```

```
from keras.utils import np_utils
from keras.layers import LSTM


# ARIMA Model:

import statsmodels.tsa.api as smt
import statsmodels.api as sm
from statsmodels.tools.eval_measures import rmse


import pickle
import warnings
```

2. Loading and Exploration of the Data

The data must first be loaded before being transformed into a structure that will be used by each of our models. Each row of data reflects a single day's worth of sales at one of 10 stores in its most basic form. Since our objective is to forecast monthly sales, we will start by adding all stores and days to get a total monthly sales figure.

```
warnings.filterwarnings("ignore", category=FutureWarning)
dataset = pd.read_csv('../input/demand-forecasting-kernels-only/sample_submission.csv')
df = dataset.copy()
df.head()
```

Output:

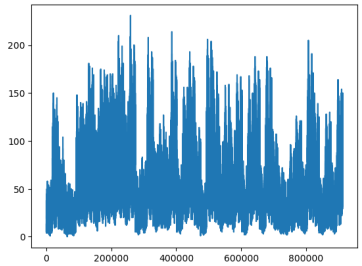
Out[18]:

	id	sales
0	0	52
1	1	52
2	2	52
3	3	52
4	4	52

Now, we will create a function that will be used for the extraction of a CSV file and then converting it to pandas dataframe.

```
def load_data(file_name):
    """Returns a pandas dataframe from a csv file."""
    return pd.read_csv(file_name)



df_s.tail()

# To view basic statistical details about dataset:

df_s['sales'].describe()

df_s['sales'].plot()
```

Output:




```
def monthlyORyears_sales(data,time=['monthly','years']):
    datadata = data.copy()
    if time == "monthly":
        # Drop the day indicator from the date column:
        datadata.date = data.date.apply(lambda x: str(x)[:-3])
    else:
        datadata.date = data.date.apply(lambda x: str(x)[:4])

   # Sum sales per month:
    datadata = data.groupby('date')['sales'].sum().reset_index()
    data.date = pd.to_datetime(data.date)

    return data
```

The above function returns a dataframe where each row represents total sales for a given month. Columns include 'date' by month and 'sales'.

```
m_df = monthlyORyears_sales(df_s,"monthly")
```

```
2.
3. m_df.to_csv('./monthly_data.csv')
4.
5. m_df.head(10)
```

Output:

Out[26]:

	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
5	2013-06-01	795597
6	2013-07-01	855922
7	2013-08-01	766761
8	2013-09-01	689907
9	2013-10-01	656587

In the above data frame, each row now represents the total sales for a given month across stores.

```
1. y_df = monthlyORyears_sales(df_s,"years")
2. y_df
```

Output:

Out[27]:

	date	sales
0	2013-01-01	7941243
1	2014-01-01	9135482
2	2015-01-01	9536887
3	2016-01-01	10357160
4	2017-01-01	10733740

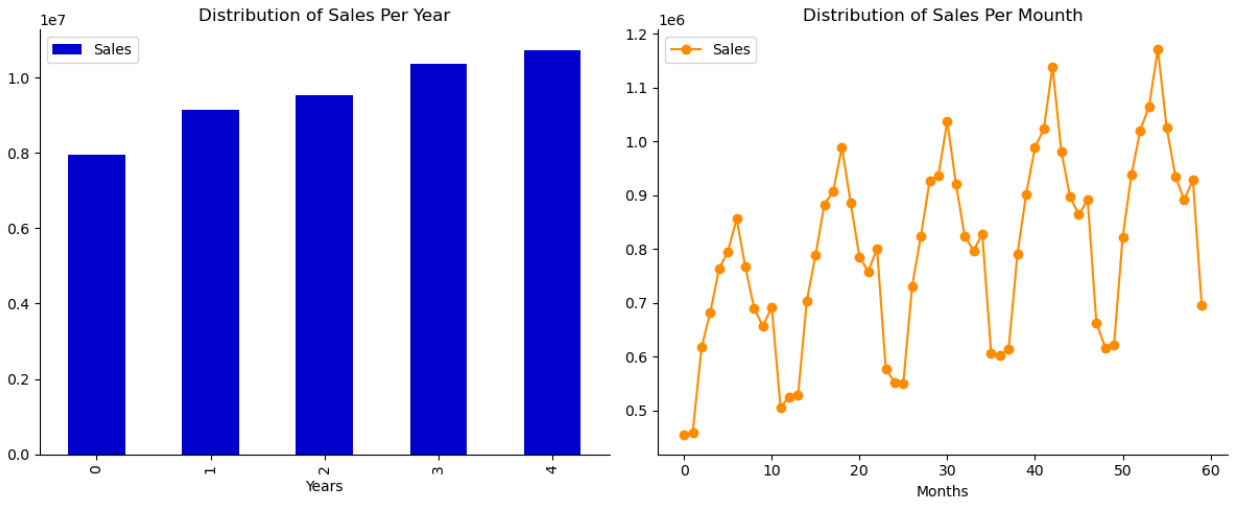
In the above data frame, each row now represents the total sales for a given year across stores.

```
1. layout = (1, 2)
2.
3. raw = plt.subplot2grid(layout, (0 ,0))
4. law = plt.subplot2grid(layout, (0 ,1))
5.
6. years = y_df['sales'].plot(kind = "bar",color = 'mediumblue', label="Sales",ax=raw, figsize=(12,5))
7. months = m_df['sales'].plot(marker = 'o',color = 'darkorange', label="Sales", ax=law)
```

```
8.
9.  years.set(xlabel = "Years",title = "Distribution of Sales Per Year")
10. months.set(xlabel = "Months", title = "Distribution of Sales Per Mounth")
11.
12. sns.despine()
13. plt.tight_layout()
14.
15. years.legend()
16. months.legend()
```

Output:

<matplotlib.legend.Legend at 0x27280058fa0>



A number of alternative models, including weighted moving average models and autoregressive integrated moving average (ARIMA) models, can be used to forecast time series. Some of them need the trend and seasonality removed first. For instance, you would have to exclude this trend from the time series if you were analyzing the number of active visitors on your website, and it was increasing by 10% each month. To obtain the final forecasts, you would need to add the trend back after the model has been trained and has begun to make predictions. Similarly to this, if you were attempting to forecast the monthly sales of sunscreen lotion, you would likely see considerable seasonality: since it sells well during the summer, the same pattern would be repeated every year.

By computing the difference between the value at each time step and the value from one year earlier, for instance, you would be able to eliminate this seasonality from the time series (this technique is called differencing). Again, to obtain the final forecasts, you would need to re-add the seasonal pattern once the model has been trained and has made several predictions.

3. EDA(Exploratory Data Analysis)

We will compute the difference between each month's sales and add it as a new column to our data frame to make it stationary.

The sales_time() function will print the total time taken for stores in days, years and months.

```
def sales_time(data):
    """Time interval of dataset:"""

    data.date = pd.to_datetime(data.date)
    n_of_days = data.date.max() - data.date.min()
    n_of_years = int(n_of_days.days / 365)

    print(f"Days: {n_of_days.days}\nYears: {n_of_years}\nMonth: {12 * n_of_years}")

sales_time(df_s)
```

Output:

```
Days: 1825
Years: 5
Month: 60
```

```


# Let's sell it per store:

def sales_per_store(data):
    sales_by_store = data.groupby('store')['sales'].sum().reset_index()

    fig, ax = plt.subplots(figsize=(8,6))
    sns.barplot(sales_by_store.store, sales_by_store.sales, color='darkred')

    ax.set(xlabel = "Store Id", ylabel = "Sum of Sales", title = "Total Sales Per Store")

    return sales_by_store
```

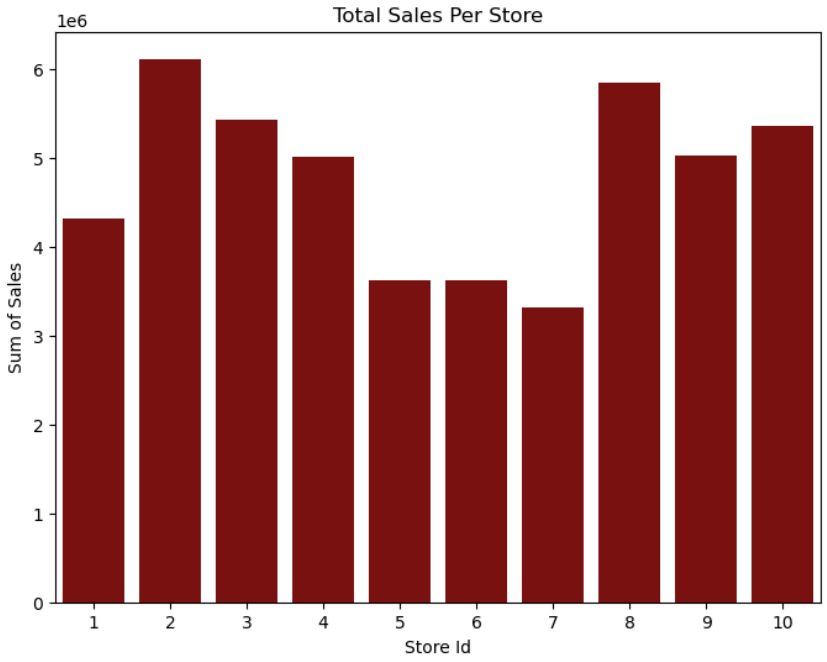
The above function represents the sales in each store.

```
sales_per_store(df_s)
```

Output:

Out[33]:

	store	sales
0	1	4315603
1	2	6120128
2	3	5435144
3	4	5012639
4	5	3631016
5	6	3627670
6	7	3320009
7	8	5856169
8	9	5025976
9	10	5360158



The above graph represents the total sale from each store.

From the above graph, we can interpret that Store Id 2 has the highest sales of **6120128** and the lowest sales is Store Id 7 of **5856169**.

```
# Overall for five years:

average_m_sales = m_df.sales.mean()
print(f"Overall Average Monthly Sales: ${average_m_sales}")

def avarage_12months():
# Last one year (this will be the forecasted sales):
    average_m_sales_1y = m_df.sales[-12:].mean()
    print(f"Last 12 months average monthly sales: ${average_m_sales_1y}")
avarage_12months()
```

Output:

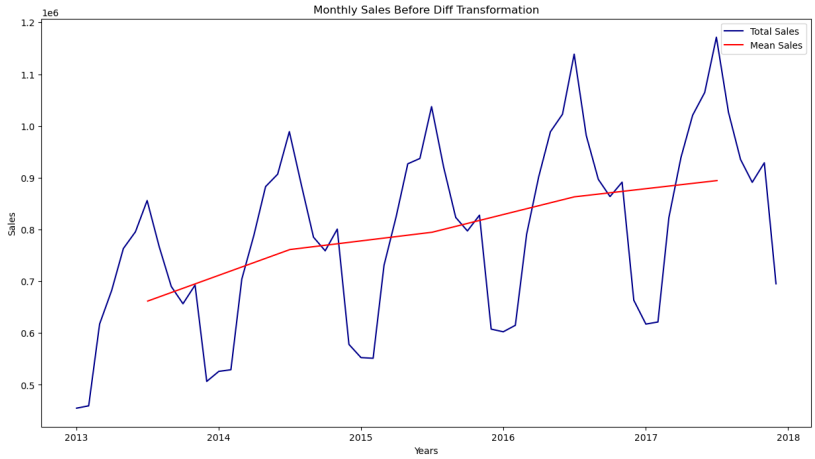
```
Overall Avarage Monthly Sales: $795075.2
Last 12 months average monthly sales: $894478.3333333334
```

4. Determining Time Series Stationary

The fundamental idea is to simulate or estimate the trend and seasonality present in the series and then subtract these to get a stationary series. Then, this series can use statistical forecasting techniques. By putting trend and seasonality restrictions back, the anticipated values would then be converted into the original scale.

```
1.  def time_plot(data, x_col, y_col, title):
2.      fig, ax = plt.subplots(figsize = (15,8))
3.      sns.lineplot(x_col, y_col, datadata = data, axax = ax, color = 'darkblue', label='Total Sales')
4.
5.      s_mean = data.groupby(data.date.dt.year)[y_col].mean().reset_index()
6.      s_mean.date = pd.to_datetime(s_mean.date, format='%Y')
7.      sns.lineplot((s_mean.date + datetime.timedelta(6*365/12)), y_col, data=s_mean, axax=ax, color='red', label='Mean S
    ales')
8.
9.      ax.set(xlabel = "Years",
10.            ylabel = "Sales",
11.            titletitle = title)
12.
13.
14. time_plot(m_df, 'date', 'sales', 'Monthly Sales Before Diff Transformation' )
```

OUTPUT:

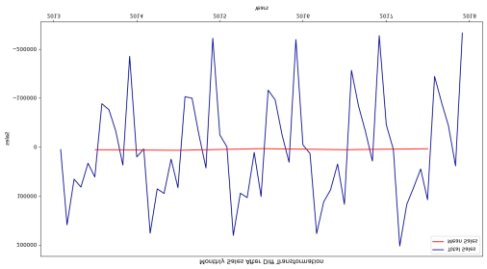


5. Differencing

We will calculate the difference between subsequent words in the series using this way. The changing mean is often eliminated using differencing.

```
def get_diff(data):
    """Calculate the difference in sales month over month:"""

    data['sales_diff'] = data.sales.diff()
    datadata = data.dropna()

    data.to_csv('./stationary_df.csv')

    return data

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

Output:



Now, we will set up the data for our various model types, that it represents monthly sales and has been modified to be stationary.

To do this, we'll define two distinct structures:

1. One is going to be used for ARIMA modelling.
2. The remaining models will utilize the other.

ARIMA Modeling

ARIMA (AutoRegressive Integrated Moving Average) is a popular time series forecasting model used for univariate time series data.

ARIMA models are fit to time series data to make predictions about future values. The process of fitting an ARIMA model involves selecting the order of the **AR, I, and MA component**s, as well as the coefficients of each component. These coefficients are estimated using optimization algorithms like maximum likelihood estimation or numerical optimization. The resulting model can then be used to generate predictions for future values of the time series.

```
def build_arima_data(data):
    """Generates a CSV file with a datetime index and a dependent sales column for ARIMA modelling."""

    da_data = data.set_index('date').drop('sales', axis=1)
```

```
    da_data.dropna(axis=0)

    da_data.to_csv('./arima_df.csv')

    return da_data

datatime_df = build_arima_data(stationary_df)
datatime_df # ARIMA Dataframe
```

Output:

Out[45]:

date	sales_diff
2013-02-01	4513.0
2013-03-01	157965.0
2013-04-01	64892.0
2013-05-01	80968.0
2013-06-01	32355.0
2013-07-01	60325.0
2013-08-01	-89161.0
2013-09-01	-76854.0
2013-10-01	-33320.0
2013-11-01	36056.0
2013-12-01	-186036.0
2014-01-01	19380.0
2014-02-01	3130.0
2014-03-01	175184.0
2014-04-01	84613.0
2014-05-01	93963.0
2014-06-01	23965.0
2014-07-01	82168.0
2014-08-01	-103414.0
2014-09-01	-100472.0
2014-10-01	-26241.0
2014-11-01	41900.0

Observing Lags

Observing lags is an important step in the ARIMA modelling process. The goal of observing lags is to determine the order of the autoregressive (AR) component in the ARIMA model. The autoregressive component is based on past values of the time series, and the order of the AR component determines the number of past values that are used as predictors.

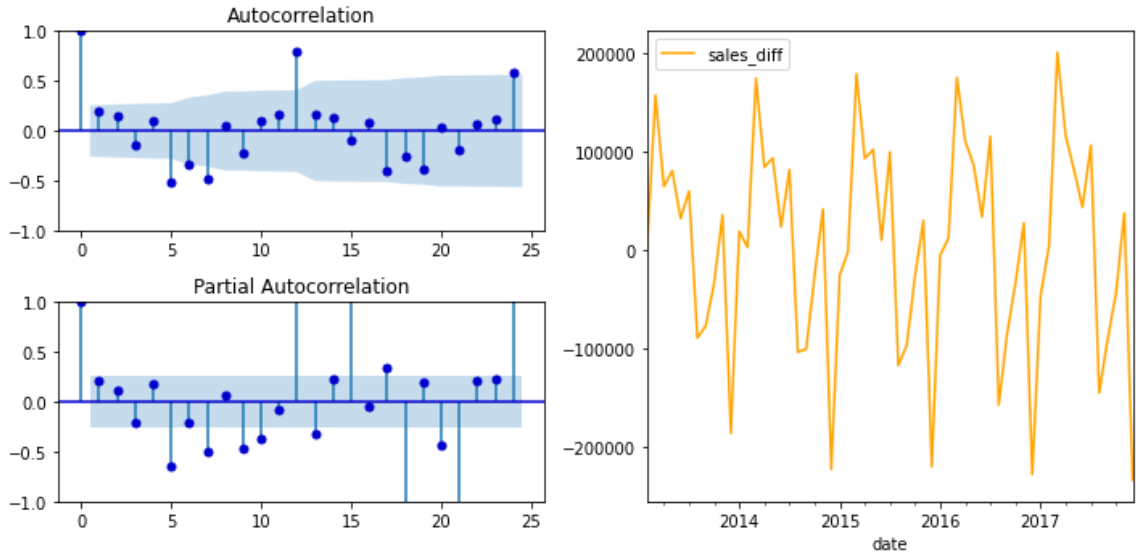
To observe lags, you typically plot the autocorrelation function (ACF) and partial autocorrelation function (PACF) of the time series. The ACF is a plot of the correlation between the time series and lagged versions of itself, while the PACF is a plot of the correlation between the time series and its lagged values, controlling for the effects of any intermediate lags.

To build a new data frame for our other models, we will assign each character to a prior month's sales. We will look at the autocorrelation and partial autocorrelation plots and use the guidelines for selecting lags in ARIMA modelling to decide how many months to include in our feature set. We can maintain a constant look-back time for both our ARIMA and regressive models in this way.

```
def plots_lag(data, lags=None):
    """Convert dataframe to datetime index"""
```

```
3.      dt_data = data.set_index('date').drop('sales', axis=1)
4.      dt_data.dropna(axis=0)
5.
6.
7.      law  = plt.subplot(122)
8.      acf  = plt.subplot(221)
9.      pacf = plt.subplot(223)
10.
11.     dt_data.plot(ax=law, figsize=(10, 5), color='orange')
12.     # Plot the autocorrelation function:
13.     smt.graphics.plot_acf(dt_data, lagslags=lags, ax=acf, color='mediumblue')
14.     smt.graphics.plot_pacf(dt_data, lagslags=lags, ax=pacf, color='mediumblue')
15.
16.     # Will also adjust the spacing between subplots to minimize the overlaps:
17.     plt.tight_layout()
18.
19. plots_lag(stationary_df, lags=24);
```

OUTPUT:



Regressive Modeling

Regression modelling is a statistical method used to model the relationship between a dependent variable and one or more independent variables. The goal of regression modelling is to identify the relationship between the independent variables and the dependent variable and to use this relationship to make predictions about the dependent variable.

Let's make a CSV file with columns for sales, dependent variables, and prior sales for each delay, and rows for each month. The EDA is used to construct the 12 delay characteristics. Regression modelling uses data.

```
# Let's create a data frame for transformation from time series to supervised:

def built_supervised(data):
    supervised_df = data.copy()

    # Create a column for each lag:
    for i in range(1, 13):
        col_name = 'lag_' + str(i)
        supervised_df[col_name] = supervised_df['sales_diff'].shift(i)

    # Drop null values:
    supervised_dfsupervised_df = supervised_df.dropna().reset_index(drop=True)

    supervised_df.to_csv('./model_df.csv', index=False)

    return supervised_df


model_df = built_supervised(stationary_df)
model_df
```

Output:

Out[27]:

	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
4	2014-06-01	906842	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0	32355.0
5	2014-07-01	989010	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0	60325.0
6	2014-08-01	885596	-103414.0	82168.0	23965.0	93963.0	84613.0	175184.0	3130.0	19380.0	-186036.0	36056.0	-33320.0	-76854.0	-89161.0
	2014-														

```
model_df.info() # Supervised Dataframe
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 47 entries, 0 to 46
Data columns (total 15 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   date        47 non-null     datetime64[ns]
 1   sales       47 non-null     int64
 2   sales_diff  47 non-null     float64
 3   lag_1       47 non-null     float64
 4   lag_2       47 non-null     float64
 5   lag_3       47 non-null     float64
 6   lag_4       47 non-null     float64
 7   lag_5       47 non-null     float64
 8   lag_6       47 non-null     float64
 9   lag_7       47 non-null     float64
 10  lag_8       47 non-null     float64
 11  lag_9       47 non-null     float64
 12  lag_10      47 non-null     float64
 13  lag_11      47 non-null     float64
 14  lag_12      47 non-null     float64
dtypes: datetime64[ns](1), float64(13), int64(1)
memory usage: 5.6 KB
```

We detach our data so that the last 12 months are part of the test set, and the rest of the data is used to train our model.

Train and test Data

```
1. def train_test_split(data):
2.     datadata = data.drop(['sales','date'], axis=1)
3.     train , test = data[:-12].values, data[-12:].values
4.
5.     return train, test
6.
7. train, test = train_test_split(model_df)
8. print(f"Shape of  Train: {train.shape}\nShape of  Test: {test.shape}")
```

Output:

```
Shape of  Train: (35, 13)
Shape of  Test: (12, 13)
```

6. Scaling Data

Scaling data is the process of transforming the values of the variables in a dataset so that they are in a similar range. This is often done to prevent some variables from having an undue influence on the model due to their large scale.

```
1. def scale_data(train_set,test_set):
2.     """Scales data using MinMaxScaler and separates data into X_train, y_train,
```

```
    X_test, and y_test."""

    # Apply Min Max Scaler:
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scalerscaler = scaler.fit(train_set)

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

    # Reshape test set:
    test_settest_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() # returns the array, flattened!
    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)
print(f"Shape of X Train: {X_train.shape}\nShape of y Train: {y_train.shape}\nShape of X Test: {X_test.shape}\nShape of y Test: {y_test.shape}")
```

Output:

```
Shape of X Train: (35, 12)
Shape of y Train: (35,)
Shape of X Test: (12, 12)
Shape of y Test: (12,)
```

7. Reverse Scaling

Reverse scaling is the process of transforming a set of scaled variables back to their original scale. This can be necessary when you want to interpret the results of a modelling analysis in terms of the original variables rather than the scaled variables. The process of reverse scaling depends on the method used to scale the data.

```
def re_scaling(y_pred, x_test, scaler_obj, lstm=False):
    """For visualizing and comparing results, undoes the scaling effect on predictions."""
    # y_pred: model predictions
    # x_test: features from the test set used for predictions
```

```
    # scaler_obj: the scalar objects used for min-max scaling
    # lstm: indicate if the model run is the lstm. If True, additional transformation occurs

     # Reshape y_pred:
     y_predy_pred = y_pred.reshape(y_pred.shape[0],
                                   1,
                                   1)

     if not lstm:
         x_testx_test = x_test.reshape(x_test.shape[0],
                                       1,
                                       x_test.shape[1])

     # Rebuild test set for inverse transform:
     pred_test_set = []
     for index in range(0, len(y_pred)):
         pred_test_set.append(np.concatenate([y_pred[index],
                                              x_test[index]],
                                             axis=1) )

     # Reshape pred_test_set:
     pred_test_set = np.array(pred_test_set)
     pred_test_setpred_test_set = pred_test_set.reshape(pred_test_set.shape[0],
                                                        pred_test_set.shape[2])

     # Inverse transform:
     pred_test_set_inverted = scaler_obj.inverse_transform(pred_test_set)

     return pred_test_set_inverted
```

We now have two distinct data structures:

- We have a DataTime index in our ARIMA structure.
- Lags are characteristics of our supervised structure.

8. Predictions DataFrame

```
def prediction_df(unscale_predictions, origin_df):
    """Generates a dataframe that shows the predicted sales for each month
    for plotting results."""

    # unscale_predictions: the model predictions that do not have min-max or other scaling applied
    # origin_df: the original monthly sales dataframe

    # Create a dataframe that shows the predicted sales:
    result_list = []
```

```
    sales_dates = list(origin_df[-13:].date)
    act_sales = list(origin_df[-13:].sales)

    for index in range(0, len(unscale_predictions)):
        result_dict = {}
        result_dict['pred_value'] = int(unscale_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
```

Model Score

A model score function is a function that measures the accuracy or performance of a predictive model. The score function provides a quantitative measure of the model's ability to make accurate predictions, and it is used to compare different models and select the best model for a particular task.

This helper function will save the root mean squared error (RMSE) and mean absolute error (MAE) of our predictions to compare the performance of our models.

```
model_scores = {}

def get_scores(unscale_df, origin_df, model_name):
    """Prints the root mean squared error, mean absolute error, and r2 scores
    for each model. Saves all results in a model_scores dictionary for
    comparison."""

    rmse = np.sqrt(mean_squared_error(origin_df.sales[-12:],
                                      unscale_df.pred_value[-12:]))

    mae = mean_absolute_error(origin_df.sales[-12:],
                              unscale_df.pred_value[-12:])

    r2 = r2_score(origin_df.sales[-12:],
                  unscale_df.pred_value[-12:])

    model_scores[model_name] = [rmse, mae, r2]

    print(f"RMSE: {rmse}\nMAE: {mae}\nR2 Score: {r2}")
```

Graph

With this plot_results() function, it will plot a line graph of the model.

```
def plot_results(results, origin_df, model_name):
```

```
    # results: a dataframe with unscaled predictions

    fig, ax = plt.subplots(figsize=(15,5))
    sns.lineplot(origin_df.date, origin_df.sales, data=origin_df, axax=ax,
                 label='Original', color='blue')
    sns.lineplot(results.date, results.pred_value, data=results, axax=ax,
                 label='Predicted', color='red')


    ax.set(xlabel = "Date",
           ylabel = "Sales",
           title = f"{model_name} Sales Forecasting Prediction")

    ax.legend(loc='best')

    filepath = Path('./model_output/{model_name}_forecasting.svg')
    filepath.parent.mkdir(parents=True, exist_ok=True)
    plt.savefig(f'./model_output/{model_name}_forecasting.svg')



def regressive_model(train_data, test_data, model, model_name):
    """Runs regressive models in SKlearn framework. First, calls scale_data
    to split into X and y and scale the data. Then fits and predicts. Finally,
    predictions are unscaled, scores are printed, and results are plotted and
    saved."""

    # Split into X & y and scale data:
    X_train, y_train, X_test, y_test, scaler_object = scale_data(train_data,
                                                 test_data)

    # Run sklearn models:
    mod = model
    mod.fit(X_train, y_train)
    predictions = mod.predict(X_test) # y_pred=predictions

    # Undo scaling to compare predictions against original data:
    origin_df = m_df
    unscaled = re_scaling(predictions, X_test, scaler_object) # unscaled_predictions
    unscaled_df = prediction_df(unscaled, origin_df)

    # Print scores and plot results:
    get_scores(unscaled_df, origin_df, model_name)
    plot_results(unscaled_df, origin_df, model_name)
```

Modelling

We will use the underlying regression model for our task:

- **Linear Regression**
- **Random Forest Regressor**
- **XGBoost**
- **LSTM**

Now we will try to find the RMSE, MAE and R2 Score through each model.

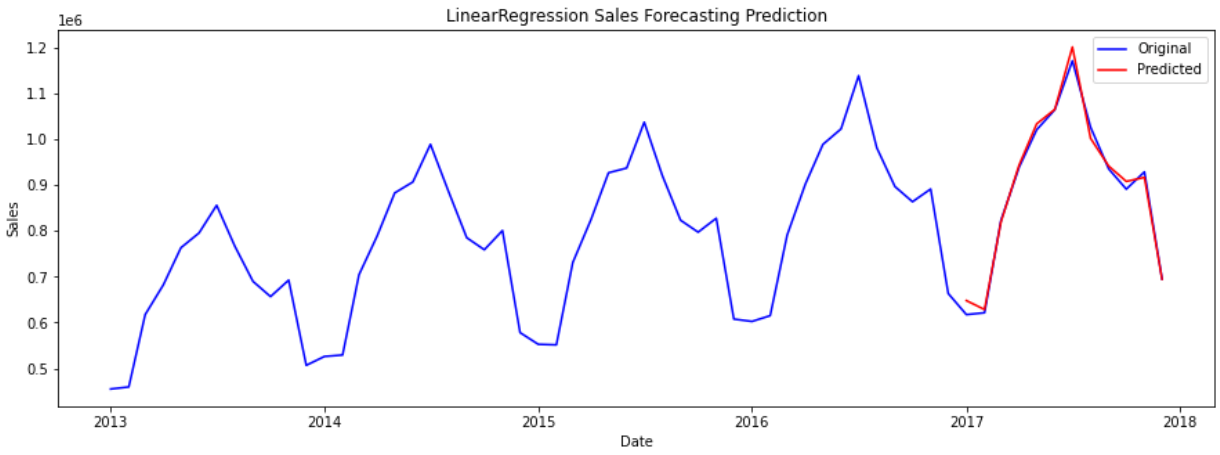
1. Linear Regression

Linear Regression is a statistical method used for modelling the linear relationship between a dependent variable and one or more independent variables. It is a type of supervised learning, which means that it is used for making predictions based on input variables.

1. regressive_model(train, test, LinearRegression(), 'LinearRegression')

Output:

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



Random Forest Regressor

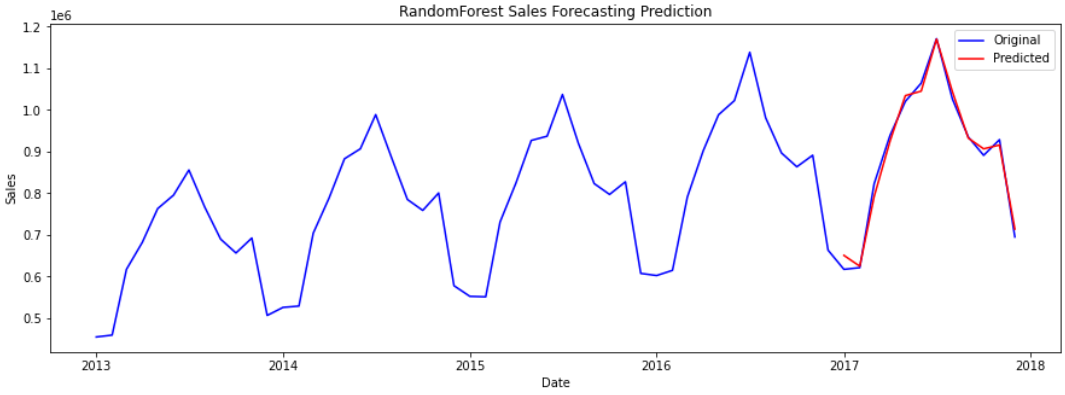
Random Forest Regressor is a type of ensemble learning method used for regression problems. It is an extension of the decision tree algorithm, where multiple decision trees are combined to form a forest.

1. regressive_model(train, test, RandomForestRegressor(n_estimators=100, max_depth=20),

2. 'RandomForest')

Output:

```
RMSE: 18075.47800428341
MAE: 15285.75
R2 Score: 0.9884714004311924
```



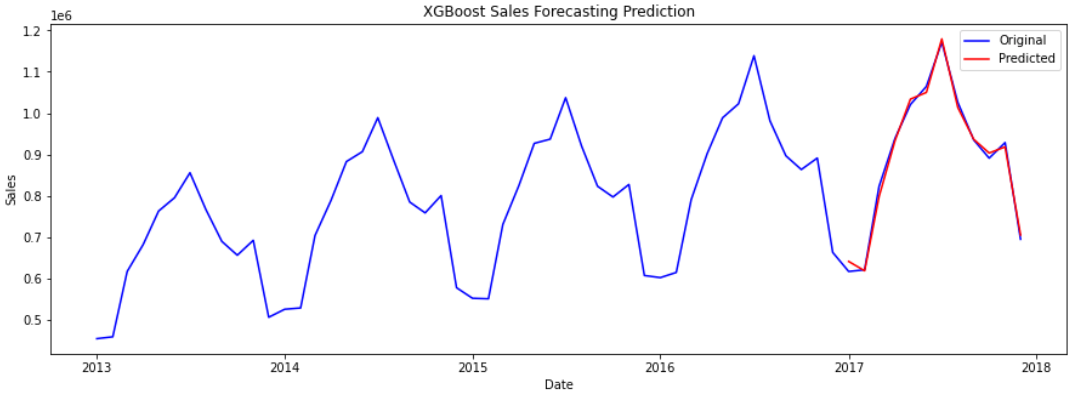
3. XGBOOST

XGBoost Regression is a specific implementation of the XGBoost algorithm for regression problems, where the goal is to predict a continuous target variable. It can handle both linear and non-linear relationships between the independent and dependent variables, as well as handle large datasets and missing data.

```
1. regressive_model(train, test, XGBRegressor(n_estimators=100,max_depth=3,
2.                                 learning_rate=0.2,objective='reg:squarederror'), 'XGBoost')
```

Output:

```
RMSE: 13574.854581787116
MAE: 11649.75
R2 Score: 0.993497694881536
```



LSTM

LSTM is a type of recurrent neural network that is especially useful for predicting sequential data.

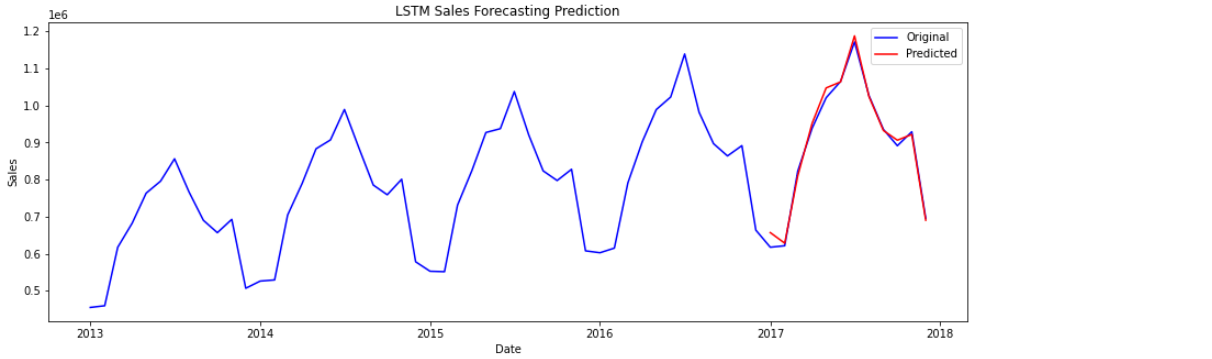
```
def lstm_model(train_data, test_data):
    """Runs a long-short-term-memory neural net with two dense layers.
    Generates predictions that are then unscaled.
    Scores are printed, and the results are plotted and saved."""
    # train_data: dataset used to train the model
    # test_data: dataset used to test the model


    # Split into X & y and scale data:
    X_train, y_train, X_test, y_test, scaler_object = scale_data(train_data, test_data)

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


    # Build LSTM:
    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='mse', optimizer='adam', metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=50, batch_size=1, verbose=1,
              shuffle=False)
```

```
    predictions = model.predict(X_test,batch_size=1)

    # Undo scaling to compare predictions against original data:
    origin_df = m_df
    unscaled = re_scaling(predictions, X_test, scaler_object, lstm=True)
    unscaled_df = prediction_df(unscaled, origin_df)

    get_scores(unscaled_df, origin_df, 'LSTM')
    plot_results(unscaled_df, origin_df, 'LSTM')


lstm_model(train,test)
```

OUTPUT:

```
Epoch 1/50
35/35 [==============================] - 2s 2ms/step - loss: 0.6237 - accuracy: 0.0000e+00
Epoch 2/50
35/35 [==============================] - 0s 2ms/step - loss: 0.5171 - accuracy: 0.0000e+00
Epoch 3/50
35/35 [==============================] - 0s 2ms/step - loss: 0.4395 - accuracy: 0.0000e+00
Epoch 4/50
35/35 [==============================] - 0s 2ms/step - loss: 0.3770 - accuracy: 0.0000e+00
Epoch 5/50
35/35 [==============================] - 0s 2ms/step - loss: 0.3259 - accuracy: 0.0000e+00
Epoch 6/50
35/35 [==============================] - 0s 2ms/step - loss: 0.2836 - accuracy: 0.0000e+00
Epoch 7/50
35/35 [==============================] - 0s 2ms/step - loss: 0.2481 - accuracy: 0.0000e+00
Epoch 8/50
35/35 [==============================] - 0s 2ms/step - loss: 0.2181 - accuracy: 0.0000e+00
Epoch 9/50
35/35 [==============================] - 0s 2ms/step - loss: 0.1925 - accuracy: 0.0000e+00
Epoch 10/50
```

```
Epoch 43/50
35/35 [==============================] - 0s 2ms/step - loss: 0.0078 - accuracy: 0.0286
Epoch 44/50
35/35 [==============================] - 0s 2ms/step - loss: 0.0075 - accuracy: 0.0286
Epoch 45/50
35/35 [==============================] - 0s 2ms/step - loss: 0.0072 - accuracy: 0.0286
Epoch 46/50
35/35 [==============================] - 0s 3ms/step - loss: 0.0070 - accuracy: 0.0286
Epoch 47/50
35/35 [==============================] - 0s 3ms/step - loss: 0.0067 - accuracy: 0.0286
Epoch 48/50
35/35 [==============================] - 0s 3ms/step - loss: 0.0065 - accuracy: 0.0286
Epoch 49/50
35/35 [==============================] - 0s 2ms/step - loss: 0.0064 - accuracy: 0.0286
Epoch 50/50
35/35 [==============================] - 0s 3ms/step - loss: 0.0062 - accuracy: 0.0286
RMSE: 16375.912860051498
MAE: 12375.833333333334
R2 Score: 0.9905374538601524
```



ARIMA MODELING

```
datatime_df.index = pd.to_datetime(datatime_df.index)


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

    # Generate predictions:
    start, end, dynamic = 40, 100, 7
    data['pred_value'] = sar.predict(startstart=start, endend=end, dynamicdynamic=dynamic)
    pred_df = data.pred_value[start+dynamic:end]

    data[["sales_diff","pred_value"]].plot(color=['blue', 'Red'])
    plt.legend(loc='upper left')

    model_score = {}

    rmse = np.sqrt(mean_squared_error(data.sales_diff[-12:], data.pred_value[-12:]))
    mae = mean_absolute_error(data.sales_diff[-12:], data.pred_value[-12:])
```

```
    r2 = r2_score(data.sales_diff[-12:], data.pred_value[-12:])
    model_scores['ARIMA'] = [rmse, mae, r2]

    print(f"RMSE: {rmse}\nMAE: {mae}\nR2 Score: {r2}")

    return sar, data, pred_df

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

Compare Model

Comparing different machine learning models is an important step in the process of building a predictive model. When comparing models, several factors should be considered, that includes; **Accuracy, Training time, Scalability, Model Complexity, Overfitting, Interpretability, Flexibility, Prediction time** etc.

But in our case, we will consider the RMSE, MAE and R2 Score.

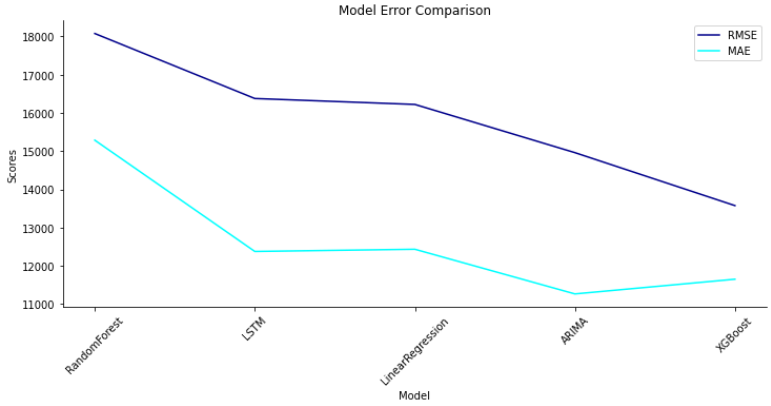
```
def create_results_df():
    results_dict = pickle.load(open("model_scores.p", "rb"))

    results_dict.update(pickle.load(open("ARIMAmodel_scores.p", "rb")))

    results_df = pd.DataFrame.from_dict(results_dict, orient='index',
                                        columns=['RMSE', 'MAE','R2'])

    results_dfresults_df = results_df.sort_values(by='RMSE', ascending=False).reset_index()

    results_df.to_csv('./results.csv')

    fig, ax = plt.subplots(figsize=(12, 5))
    sns.lineplot(np.arange(len(results_df)), 'RMSE', data=results_df, axax=ax,
                 label='RMSE', color='darkblue')
    sns.lineplot(np.arange(len(results_df)), 'MAE', data=results_df, axax=ax,
                 label='MAE', color='Cyan')

    plt.xticks(np.arange(len(results_df)),rotation=45)
    ax.set_xticklabels(results_df['index'])
    ax.set(xlabel = "Model",
           ylabel = "Scores",
           title = "Model Error Comparison")
    sns.despine()

    plt.savefig(f'./model_output/compare_models.png')

    return results_df

```

```
30.
31. results = create_results_df()
32. results
```

OUTPUT:

Out[48]:

	index	RMSE	MAE	R2
0	RandomForest	18075.478004	15285.750000	0.988471
1	LSTM	16375.912860	12375.833333	0.990537
2	LinearRegression	16221.040791	12433.000000	0.990716
3	ARIMA	14959.893468	11265.335749	0.983564
4	XGBoost	13574.854582	11649.750000	0.993498




```
1. avarage_12months()
```

Output:

```
Last 12 months average monthly sales: $894478.3333333334
```

```
1. average = 894478.3333333334
2. XGBoost = results.MAE.values[4]
3. percentage_off = round(XGBoost/average*100,2)
4.
5. print(f"With XGBoost, prediction is within {percentage_off}% of the actual.")
```

While comparing the model, **we find that XGBoost has the lowest RMSE Score of 13574.854582, which concludes that it has the highest accuracy among the other models.**

Through the percentage_off test, we find that XGBoost has the predictions that are actually in the percentage of 1.3% considering the actual prediction.

Overall, machine learning can be a powerful tool for predicting sales and improving business outcomes. Whether you are using regression analysis, time series analysis, decision tree-based algorithms or neural networks, machine learning can help you make more accurate predictions and take action to improve your sales.

Conclusion

In conclusion, Machine Learning can be a powerful tool in the hands of businesses to predict sales and make informed decisions. With a combination of various algorithms, historical data and neural networks, businesses can improve their sales and make better decisions for their future.