SUPERVISED LEARNING WITH PROPHET & LSTM NETWORK

Time Series Analysis & Predictive Modeling Using Supervised Machine Learning

Stock price prediction using machine learning

Sarit Maitra
Jul 4, 2020 · 9 min read ★

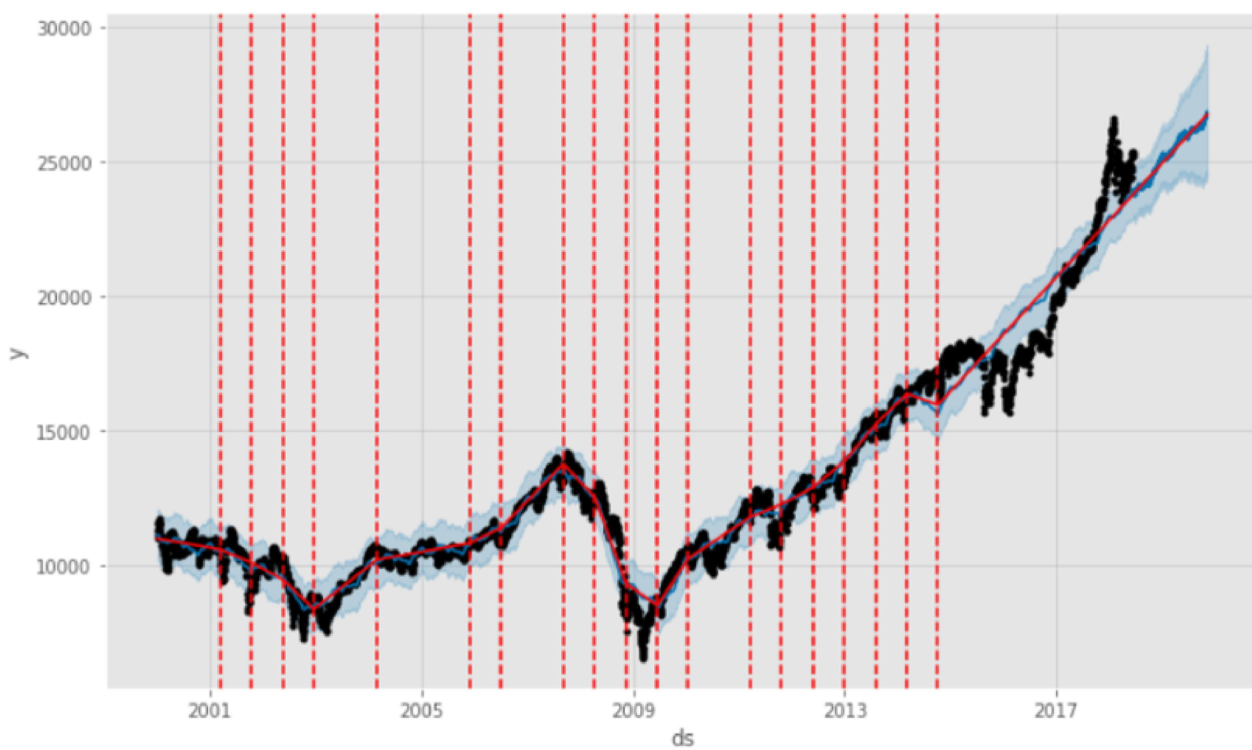


Image by author

Time-Series involves temporal datasets that change over a period of time and time-based attributes are of paramount importance in these datasets. The trading prices of stocks change constantly over time, and reflect various unmeasured factors such as market confidence, external influences, and other driving forces that may be hard to identify or measure. There are hypothesis like the Efficient Market Hypothesis, which says that it is almost impossible to beat the market consistently and there are others which disagree with it.

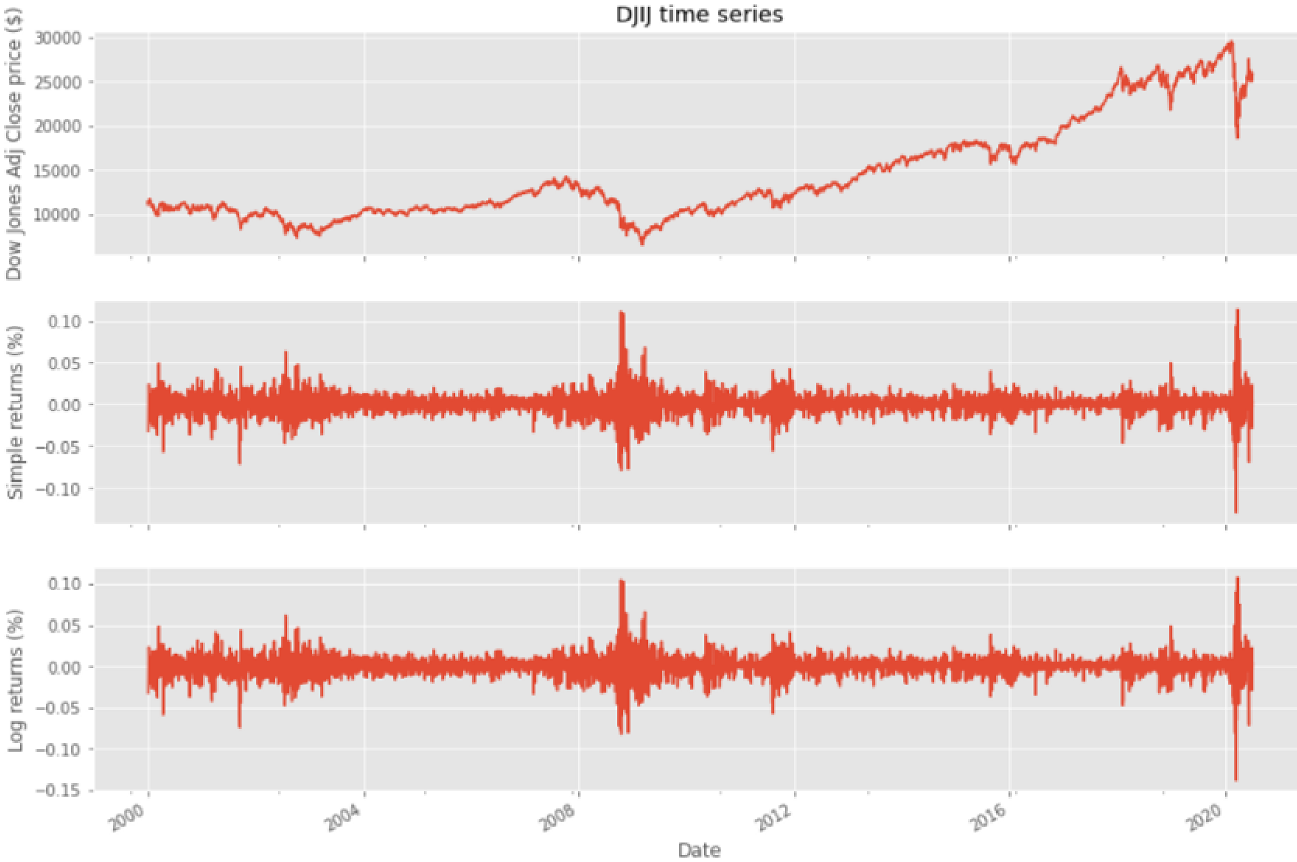
Problem Statement

Forecasting the future value of a given stock is a crucial task as investing in stock market involves higher risk.. Here, given the historical daily close price for Dow-Jones Index, we would like to prepare and compare forecasting models.

```
dji = web.DataReader('^DJI', data_source = 'yahoo', start = '2000-01-
01')
print(dji.head())
print('\n')
print(dji.shape)
```

```
                    High           Low          Open         Close     Volume     Adj Close
Date
2000-01-03  11522.009766  11305.690430  11501.849609  11357.509766  169750000  11357.509766
2000-01-04  11350.059570  10986.450195  11349.750000  10997.929688  178420000  10997.929688
2000-01-05  11215.099609  10938.669922  10989.370117  11122.650391  203190000  11122.650391
2000-01-06  11313.450195  11098.450195  11113.370117  11253.259766  176550000  11253.259766
2000-01-07  11528.139648  11239.919922  11247.059570  11522.559570  184900000  11522.559570


(5158, 6)
```

```
dji_series = dji['Adj Close']

#Calculate the simple and log returns using the adj close prices:
dji['simple_rtn'] = dji_series.pct_change()
dji['log_rtn'] = np.log(dji_series/dji_series.shift(1))

fig, ax = plt.subplots(3, 1, figsize=(14, 10), sharex=True)
dji_series.plot(ax=ax[0])
ax[0].set(title = 'DJIJ time series', ylabel = 'Dow Jones Adj Close
price ($)')
dji.simple_rtn.plot(ax=ax[1])
ax[1].set(ylabel = 'Simple returns (%)')
dji.log_rtn.plot(ax=ax[2])
ax[2].set(xlabel = 'Date', ylabel = 'Log returns (%)')
plt.show()
```




```
#Calculate the rolling mean and standard deviation:
df_rolling = dji[['simple_rtn']].rolling(window=21) .agg(['mean',
'std'])
df_rolling.columns = df_rolling.columns.droplevel()

#Join the rolling metrics to the original data:
df_outliers = dji.join(df_rolling)

#Define a function for detecting outliers:
def indentify_outliers(row, n_sigmas=3):
x = row['simple_rtn']
mu = row['mean']
sigma = row['std']
if (x > mu + 3 * sigma) | (x < mu - 3 * sigma):
return 1
else:
return 0

#Identify the outliers and extract their values for later use
df_outliers['outlier'] = df_outliers.apply(indentify_outliers,axis=1)
outliers = df_outliers.loc[df_outliers['outlier'] == 1,
['simple_rtn']]

#Plot the results:
fig, ax = plt.subplots()
ax.plot(df_outliers.index, df_outliers.simple_rtn, color='gray',
```

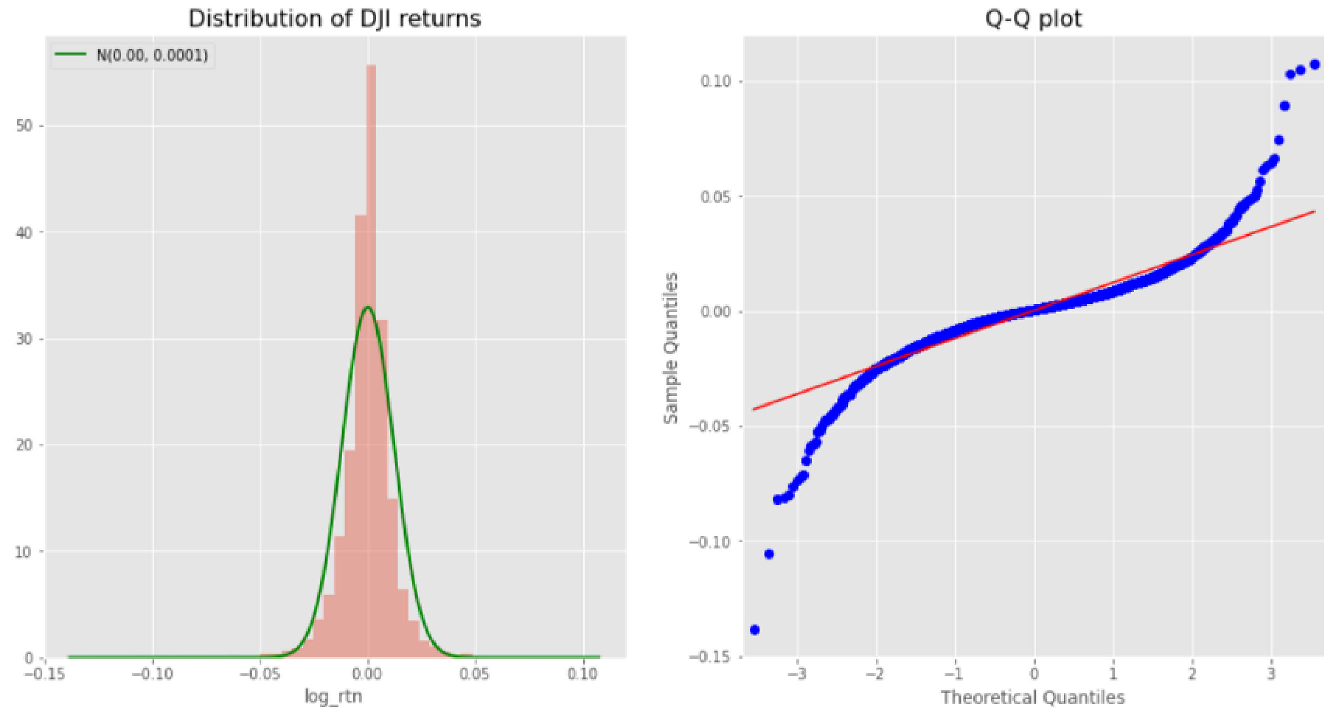
```
label='Normal')
ax.scatter(outliers.index, outliers.simple_rtn, color='red',
label='Anomaly')
ax.set_title("DJI returns")
ax.legend(loc='lower right')
plt.show()
```

Outlier identification

The black swan theory, which predicts that anomalous events, such as a stock market crash, are much more likely to occur than would be predicted by the normal distribution. A good example to illustrate the long-tailed nature of data is stock returns. Below shows the QQ-Plot for the daily DJI returns.

```
r_range = np.linspace(min(dji['log_rtn'].dropna()),
max(dji['log_rtn'].dropna()), num=1000)
mu = dji['log_rtn'].dropna().mean()
sigma = dji['log_rtn'].dropna().std()
norm_pdf = scs.norm.pdf(r_range, loc=mu, scale=sigma)

#Plot the histogram and the Q-Q plot
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

# histogram
sns.distplot(dji['log_rtn'].dropna(), kde=False, norm_hist=True,
ax=ax[0])
ax[0].set_title('Distribution of DJI returns', fontsize=16)
ax[0].plot(r_range, norm_pdf, 'g', lw=2, label=f'N({mu:.2f},
```

```
{sigma**2:.4f})')
ax[0].legend(loc='upper left');

# Q-Q plot
qq = sm.qqplot(dji['log_rtn'].dropna().values, line='s', ax=ax[1])
ax[1].set_title('Q-Q plot', fontsize = 16)
plt.show()
```



Q-Q plot

The points are close to the line for the data within one standard deviation of the mean. Tukey refers to this phenomenon as data being normal in the middle, but having much longer tails

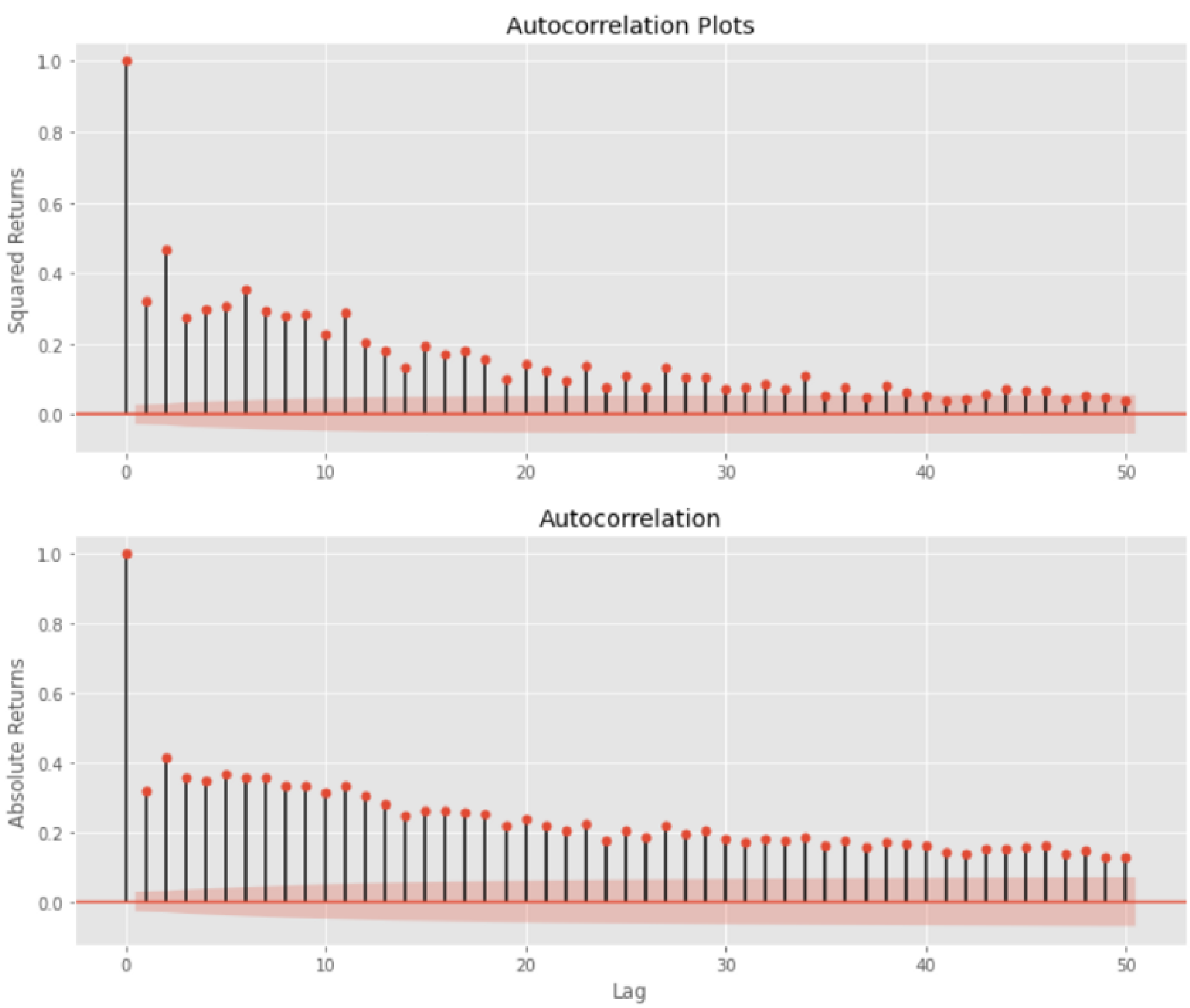
```
print("kurtosis:", scipy.stats.kurtosis(dji['log_rtn'].dropna(),bias=False))
print("skewness:", scipy.stats.skew(dji['log_rtn'].dropna(),bias=False))
print("JB:", scipy.stats.jarque_bera(dji['log_rtn'].dropna()))
```

```
kurtosis: 13.178200149240837
skewness: -0.374123198090276
JB: (37357.60391773232, 0.0)
```

Small and decreasing auto-correlation in squared/absolute returns

Motivation for modeling volatility by means of nonstationary processes is related to high persistence commonly observed in squared or absolute returns. This shows typical pattern if we draw ACF plots of the squared/absolute returns that are positive and slowly decreasing.

```
N_LAGS = 50
SIGNIFICANCE_LEVEL = 0.05
fig, ax = plt.subplots(2, 1, figsize=(12, 10))
smt.graphics.plot_acf(dji['log_rtn'].dropna() ** 2, lags=N_LAGS,
alpha=SIGNIFICANCE_LEVEL, ax = ax[0])
ax[0].set(title='Autocorrelation Plots', ylabel='Squared Returns')
smt.graphics.plot_acf(np.abs(dji['log_rtn'].dropna()), lags=N_LAGS,
alpha=SIGNIFICANCE_LEVEL, ax = ax[1])
ax[1].set(ylabel='Absolute Returns',xlabel='Lag')
plt.show()
```



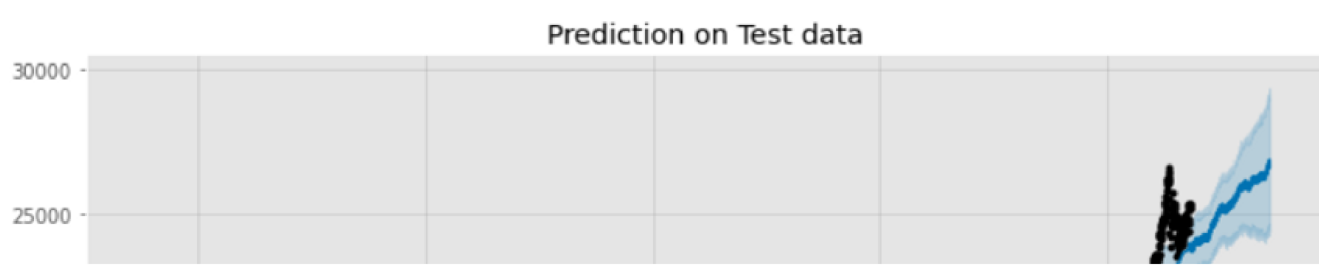
Auto-correlation plot

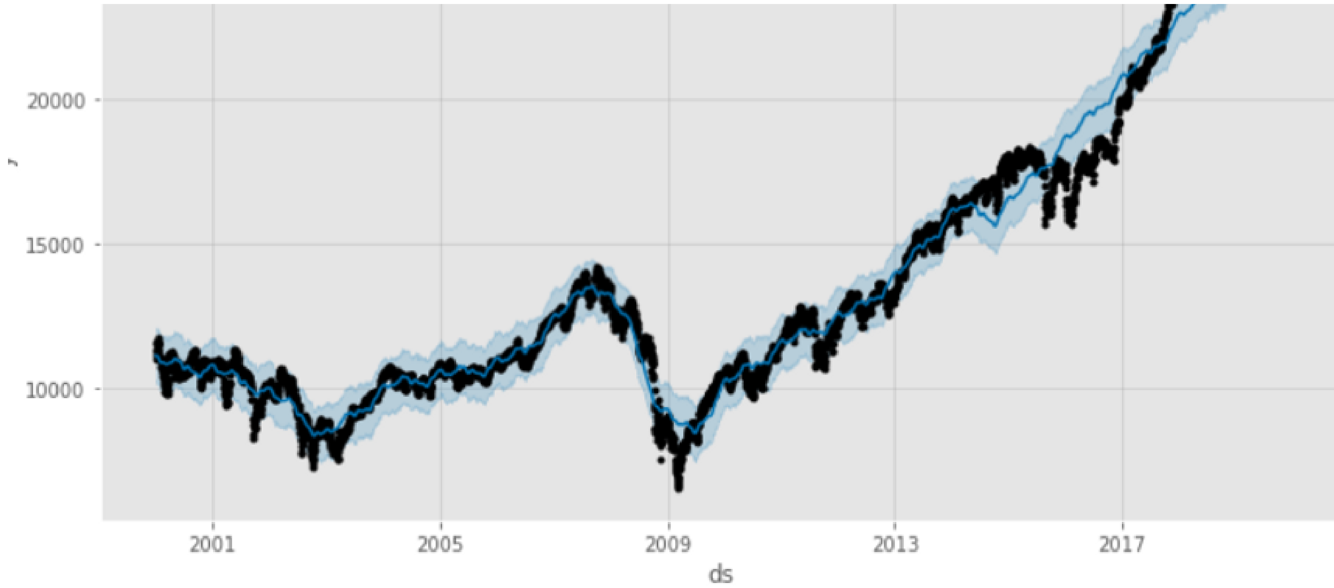
Prediction with Prophet

At its core, the Prophet procedure is an additive regression model with three main components:

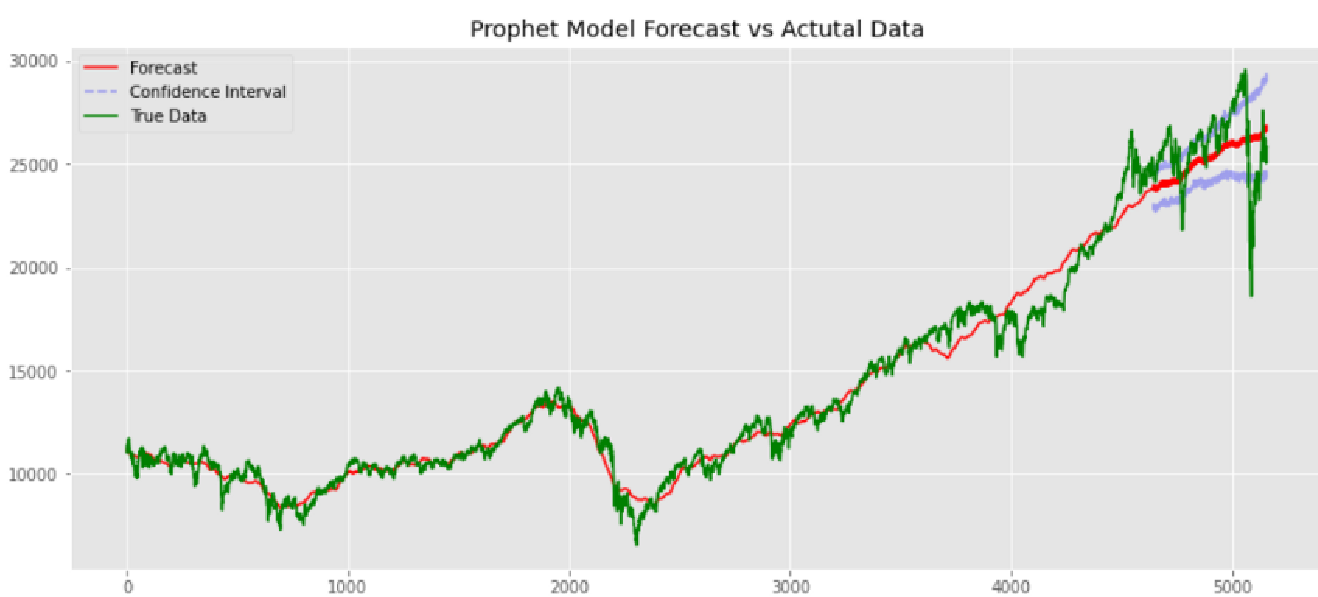
1. A piece-wise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting change-points from the data.
2. A yearly seasonal component modeled using fourier series.
3. A weekly seasonal component using dummy variables.

```
prophet = dji_series.reset_index() # reset index to get date_time as
a column
# prepare the required data-frame
prophet.rename(columns={'Date':'ds','Adj Close':'y'},inplace=True)
prophet = prophet[['ds','y']]

train_percent = 0.90

# prepare train and test sets
train_size = int(prophet.shape[0]*train_percent)
train = prophet.iloc[:train_size]
test = prophet.iloc[train_size+1:]

model_prophet = Prophet()# build a prophet model
model_prophet.fit(train) # fit the model

# prepare a future dataframe
future_dates =
model_prophet.make_future_dataframe(periods=test.shape[0])
# test.shape[0] = 515
forecast = model_prophet.predict(future_dates) # forecast values

model_prophet.plot(forecast) # plot for prediction
plt.title('Prediction on Test data')
```






```
lower_bound = (forecast.yhat_lower.iloc[train_size+1:])
upper_bound = (forecast.yhat_upper.iloc[train_size+1:])

# plot against true data
plt.plot(forecast['yhat'],c='r',label='Forecast')
plt.plot(lower_bound, linestyle='--',c='b',alpha=0.3,
label='Confidence Interval')
plt.plot(upper_bound, linestyle='--',c='b',alpha=0.3)
plt.plot(prophet['y'],c='g',label='True Data')
plt.legend()
plt.title('Prophet Model Forecast vs Actutal Data')
```

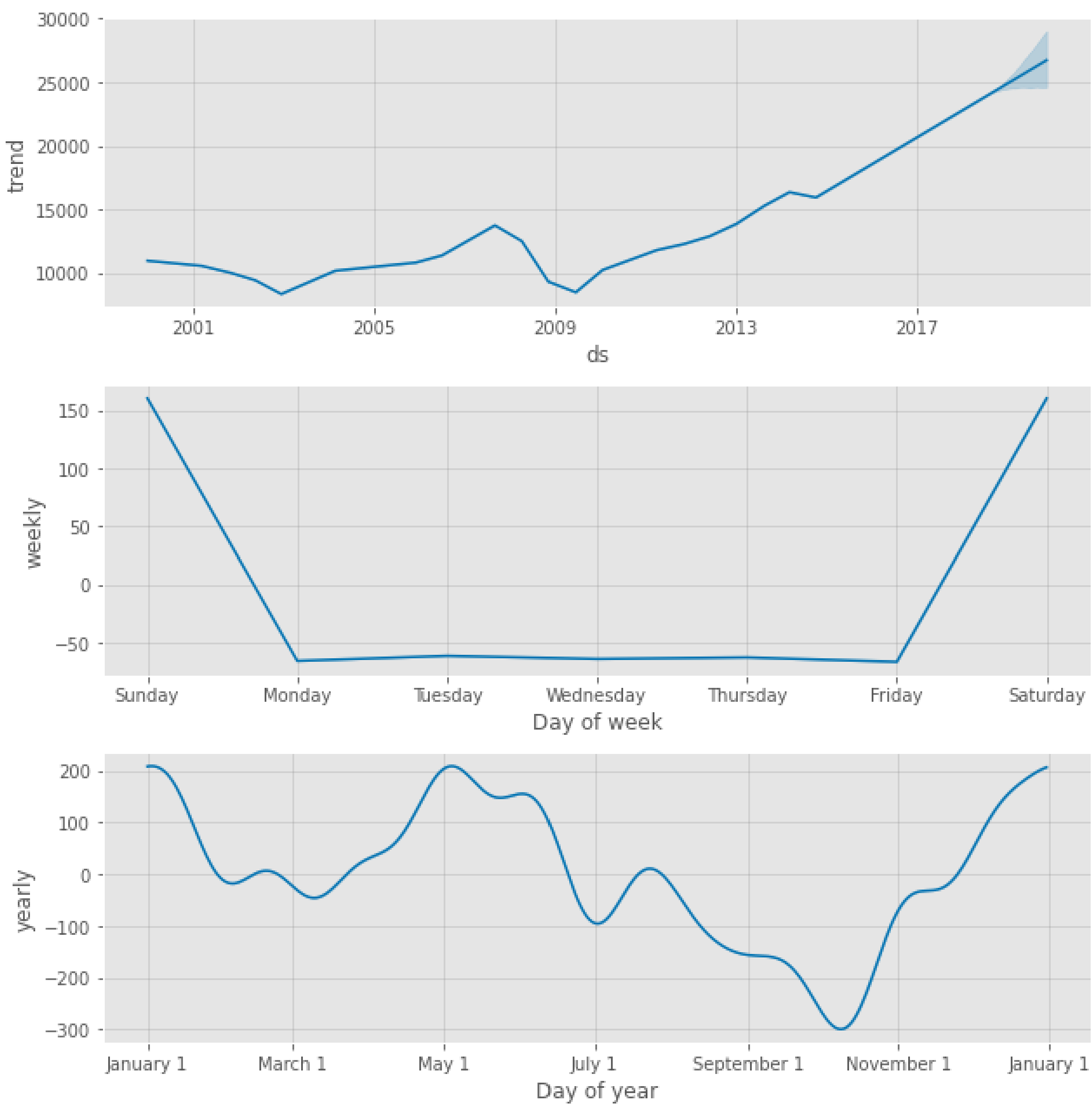


Probably business wants to see the report in below format:

```
combine = pd.concat([test.tail(), pd.DataFrame(forecast.yhat.tail())], axis=1).dropna()
combine['accuracy'] = round(combine.apply(lambda row: row.yhat /
                                          row.y *100, axis = 1),2)
combine['accuracy'] = pd.Series(["{0:.2f}%".format(val) for val in combine['accuracy']],
                                index = combine.index)
```

	ds	y	yhat	accuracy
5153	2020-06-26	25015.550781	26609.526318	106.37%
5154	2020-06-29	25595.800781	26844.013101	104.88%
5155	2020-06-30	25812.880859	26851.057124	104.02%
5156	2020-07-01	25734.970703	26631.613507	103.48%

> *model_prophet.plot_components(forecast)*



We will explore the application of LSTMs to this use case and generate forecasts. There are a number of ways this problem can be modeled to forecast values.

LSTM windowed architecture

To model our current use case as a regression problem, we state that the stock price at timestamp t+1 (dependent variable) is a function of stock price at timestamps t, t -1, t -2, …, t -n. Where n is the past window of stock prices.

Here a window size of 6 days is used. The value at time t+1 is forecasted using past six values. However, smaller or larger windows can be used to experiment the difference. We have data since 2000, so, multiple such sequences are created applying this windowed transformation in a rolling fashion.

```
# prepare training and testing data sets for LSTM based regression
modeling
def get_reg_train_test(timeseries,sequence_length= 51, train_size =
0.9,roll_mean_window=5, normalize=True,scale=False):
# rolling mean is used to smoothen the time series
if roll_mean_window:
timeseries = timeseries.rolling(roll_mean_window).mean().dropna()

# create windows
result = []
for index in range(len(timeseries) - sequence_length):
result.append(timeseries[index: index + sequence_length])

# normalize data using every time step is the % change from the 1st
# value in that window
if normalize:
normalised_data = []
for window in result:
normalised_window = [((float(p) / float(window[0])) - 1) for p in
window]
normalised_data.append(normalised_window)
result = normalised_data

# identify train-test splits
result = np.array(result)
row = round(train_size * result.shape[0])

# split train and test sets
train = result[:int(row), :]
test = result[int(row):, :]
```

```
# scale data in 0-1 range
scaler = None
if scale:
scaler=MinMaxScaler(feature_range=(0, 1))
train = scaler.fit_transform(train)
test = scaler.transform(test)

# split independent and dependent variables
x_train = train[:, :-1]
y_train = train[:, -1]
x_test = test[:, :-1]
y_test = test[:, -1]

# Transforms for LSTM input
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1],
1))
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
return x_train,y_train,x_test,y_test,scaler
```

```
# training and testing data sets for LSTM based sequence modeling
def get_seq_train_test(time_series, scaling=True,train_size=0.9):
    scaler = None
    if scaling:
        scaler = MinMaxScaler(feature_range=(0, 1))
        time_series = np.array(time_series).reshape(-1,1)
        scaled_stock_series = scaler.fit_transform(time_series)
    else:
        scaled_stock_series = time_series

    train_size = int(len(scaled_stock_series) * train_size)

    train = scaled_stock_series[0:train_size]
    test = scaled_stock_series[train_size:len(scaled_stock_series)]

    return train,test,scaler
```

```
# LSTM model for sequence modeling
def get_seq_model(hidden_units=4,input_shape=(1,1),verbose=False):
    # create and fit the LSTM network
    model = Sequential()
    # samples*timesteps*featuress

    model.add(LSTM(input_shape=input_shape,
                   units = hidden_units,
```

```
                  units = hidden_units,
                  return_sequences=True))

    # readout layer. TimeDistributedDense uses the same weights for all
    # time steps.
    model.add(TimeDistributed(Dense(1)))
    start = time.time()

    model.compile(loss="mse", optimizer="rmsprop")

    if verbose:
        print("> Compilation Time : ", time.time() - start)
        print(model.summary())

    return model
```

```
# Window wise prediction function
def predict_reg_multiple(model, data, window_size=6, prediction_len=3):
    prediction_list = []

    # loop for every sequence in the dataset
    for window in range(int(len(data)/prediction_len)):
        _seq = data[window*prediction_len]
        predicted = []
        # loop till required prediction length is achieved
        for j in range(prediction_len):
            predicted.append(model.predict(_seq[np.newaxis,:,:])[0,0])
            _seq = _seq[1:]
            _seq = np.insert(_seq, [window_size-1], predicted[-1], axis=0)
        prediction_list.append(predicted)
    return prediction_list
```

```
# Plot
def plot_reg_results(predicted_data, true_data, prediction_len=3):
    fig = plt.figure(facecolor='white')
    ax = fig.add_subplot(111)

    # plot actual data
    ax.plot(true_data,
            label='True Data',
            c='black',alpha=0.3)

    # plot flattened data
    plt.plot(np.array(predicted_data).flatten(),
             label='Prediction_full',
```

```
                    c='g',linestyle='--')

    #plot each window in the prediction list
    for i, data in enumerate(predicted_data):
        padding = [None for p in range(i * prediction_len)]
        plt.plot(padding + data, label='Prediction',c='black')

    plt.title("Forecast Plot with Prediction Window={}".format(prediction_len))
    plt.show()
```

Train-test split

```
x_train,y_train,x_test,y_test,scaler = get_reg_train_test(dji_series,
sequence_length=WINDOW+1, roll_mean_window =None, normalize=True,
scale=False)
print("Data Split Complete")
print("x_train shape={}".format(x_train.shape))
print("y_train shape={}".format(y_train.shape))
print("x_test shape={}".format(x_test.shape))
print("y_test shape={}".format(y_test.shape))
```

> *We are creating seven-day window that is comprised of six days of historical data (x_train) and one day forecast (y_train)*

```
window = 6
pred_length = int(window/2)
stock_index = '^DJI'

# prepare LSTM model
lstm_model=None
try:
lstm_model = get_reg_model(layer_units=[50,100],window_size=window)
except:
print("Model Build Failed. Trying Again")
lstm_model = get_reg_model(layer_units=[50,100],window_size=window)
```

```
> Compilation Time :  0.011904001235961914
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 6, 50)             10400
```

```
_________________________________________________________________
dropout_3 (Dropout)          (None, 6, 50)             0
_________________________________________________________________
lstm_4 (LSTM)                (None, 100)               60400
_________________________________________________________________
dropout_4 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 70,901
Trainable params: 70,901
Non-trainable params: 0
_________________________________________________________________
None
```

```
# eatrly stopping to avoid overfitting
callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss',patience=2,verbose=0)]
lstm_model.fit(x_train, y_train, epochs=20, batch_size=16,verbose=1,validation_split=0.05,
               callbacks=callbacks)
print("Model Fit Complete")
```

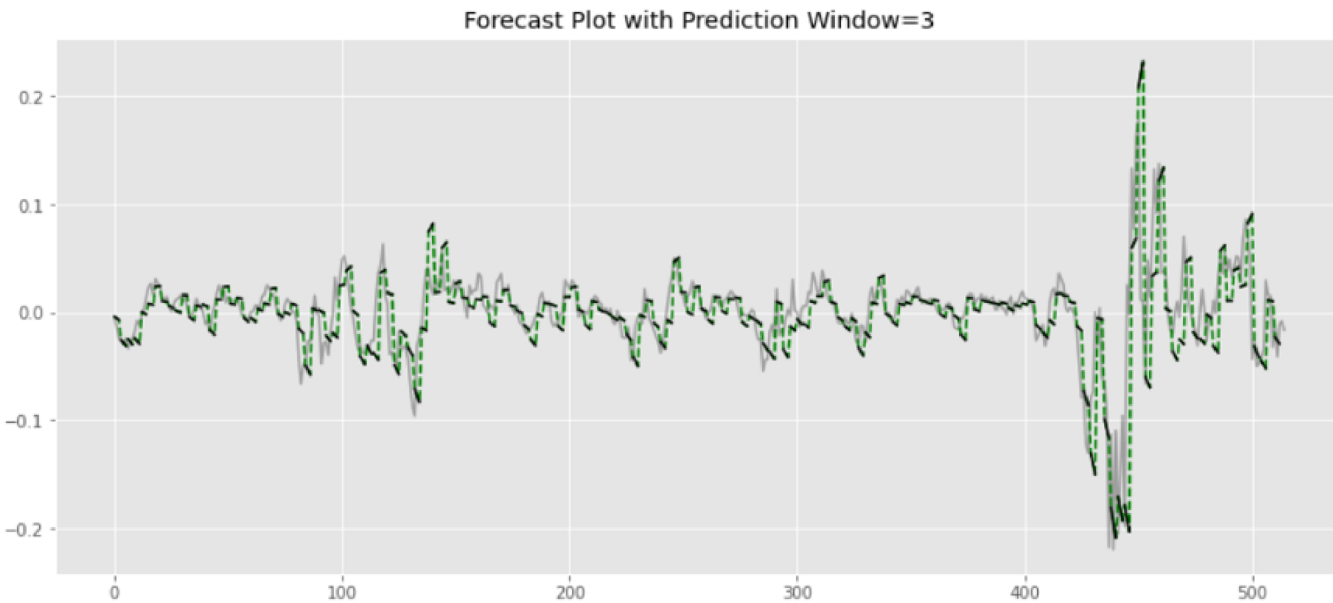
```
Train on 4404 samples, validate on 232 samples
Epoch 1/20
4404/4404 [==============================] - 9s 2ms/step - loss: 3.0409e-04 - val_loss: 1.1142e-04
Epoch 2/20
4404/4404 [==============================] - 7s 1ms/step - loss: 1.7071e-04 - val_loss: 1.0512e-04
Epoch 3/20
4404/4404 [==============================] - 7s 1ms/step - loss: 1.5794e-04 - val_loss: 1.1398e-04
Epoch 4/20
4404/4404 [==============================] - 7s 2ms/step - loss: 1.5487e-04 - val_loss: 1.0658e-04
Model Fit Complete
```

```
# Performance on train data
train_pred = predict_reg_multiple(lstm_model,x_train,
window_size=window,prediction_len=pred_length)
train_offset = y_train.shape[0] -
np.array(train_pred).flatten().shape[0]
train_rmse = math.sqrt(mean_squared_error(y_train[train_offset:],
np.array(train_pred).flatten()))
print('Train Score: %.2f RMSE' % (train_rmse))

# Performance on test data
test_pred = predict_reg_multiple(lstm_model,x_test,
window_size=window,prediction_len=pred_length)
test_offset = y_test.shape[0] -
np.array(test_pred).flatten().shape[0]
test_rmse = math.sqrt(mean_squared_error(y_test[test_offset:],
```

```
np.array(test_pred).flatten()))
print('Test Score: %.2f RMSE' % (test_rmse))
```

```
Train Score: 0.03 RMSE
Test Score: 0.05 RMSE
```

```
# Plot Test Predictions
plot_reg_results(test_pred,y_test,prediction_len=pred_length)
plt.show()
```



The gray line is the original/true test data (normalized) and the black lines denote the predicted/forecast values in three-day periods. The dotted line is used to explain the overall flow of the predicted series. As is evident, the forecasts are having some similarity to the actual data.

Here also, we can bring the data to original shape to see the actual plot and generate report.

Window size along with other hyper-parameters of the network (like epochs, batch size, LSTM units, etc.) have an impact on the final output and therefore, these hyper-parameters can be chosen carefully for robust output.

For simplified version of developing RNN network architecture can be found [here](#).

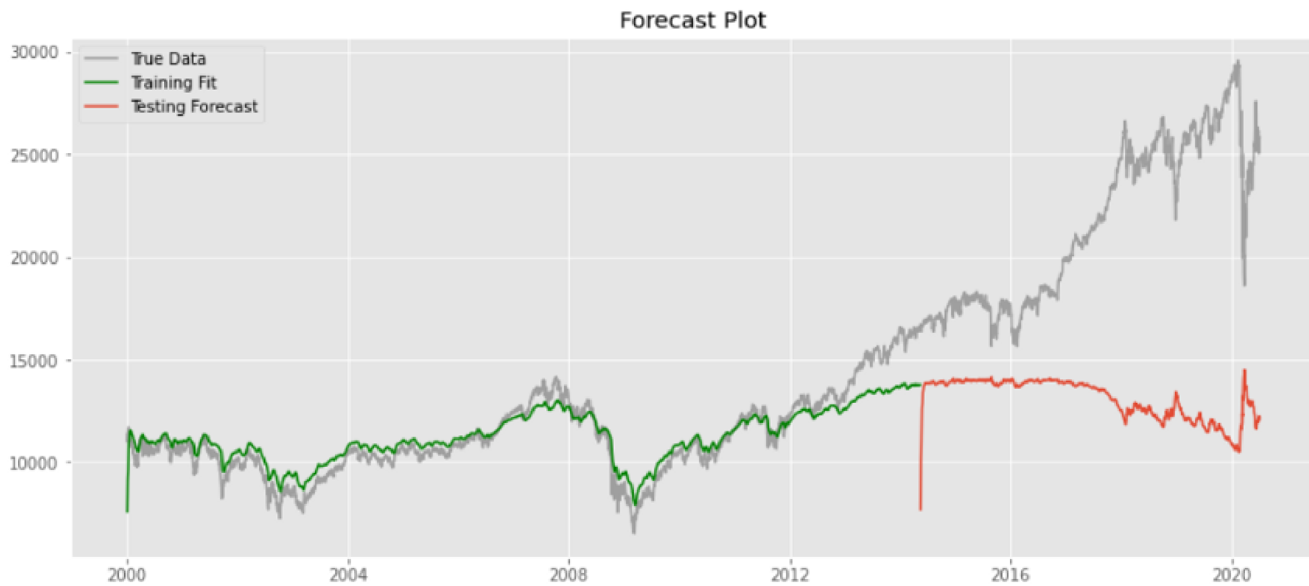
LSTM Sequence Modeling

Let us model it now as sequence where value at each time step is a function of previous values. Here we do not divide the time series into windows of fixed sizes, rather we would utilize the LSTMs to learn from the data and determine which past values to utilize for forecasting.

```
train_percent = 0.7
verbose=True

# split train and test datasets
train,test,scaler = \
get_seq_train_test(dji_series,scaling=True,train_size=train_percent)
train = np.reshape(train,(1,train.shape[0],1))
test = np.reshape(test,(1,test.shape[0],1))
train_x = train[:,:-1,:]
train_y = train[:,1:,:]
test_x = test[:,:-1,:]
test_y = test[:,1:,:]
print("Data Split Complete")
print("train_x shape={}".format(train_x.shape))
print("train_y shape={}".format(train_y.shape))
print("test_x shape={}".format(test_x.shape))
print("test_y shape={}".format(test_y.shape))

# build RNN model
seq_lstm_model=None
try:
seq_lstm_model = get_seq_model(input_shape=(train_x.shape[1],1),
verbose=verbose)
except:
print("Model Build Failed. Trying Again")
seq_lstm_model = get_seq_model(input_shape=
(train_x.shape[1],1),verbose=verbose)

# train the model
seq_lstm_model.fit(train_x, train_y, epochs=10, batch_size=1,
verbose=2)
print("Model Fit Complete")

# train fit performance
train_predict = seq_lstm_model.predict(train_x)
# inverse train transformation
train_predict = scaler.inverse_transform(train_predict.reshape(-1,1))

# pad_sequences is used to ensure that all sequences in a list have
the same length.
test_predict = pad_sequences(test_x,maxlen=train_x.shape[1],
padding='post', value=0,dtype='float')
# inverse test transformation
test_predict = scaler.inverse_transform(test_predict.reshape(-1,1))
```

```
# plot the true and forecasted values
train_size = len(train_predict)+1
plt.plot(dji_series.index, dji_series.values,c='black',
alpha=0.3,label='True Data')
plt.plot(dji_series.index[1:train_size], train_predict, label=
'Training Fit',c='g')
plt.plot(dji_series.index[train_size+1:],test_predict[:test_x.shape[1
]],label='Testing Forecast')
plt.title('Forecast Plot')
plt.legend()
```

Though I have used padding utility of keras.preprocessing.sequence module; however, this can also be done taking the last few values from train window as per window size.



Here, we could observe poor performance model on test data; rather generalization issue where model under-fit the data during in validation stage. We can go back to training stage and try adjusting the hyper-parameters to validate again on test set.

1. increase the number of training set,
2. tune the number of epochs,
3. tune batch size,
4. tuning the number of neuron and validate again on test set.

Also, I would recommend to use time series split using the below code snippet to get better result when dealing with time series observation with temporal order.

> *This split provides train/test indices to split time series data samples that are observed at fixed time intervals, in train/test sets. In each split, test indices must be higher than before, and thus shuffling in cross validator is inappropriate. This CV object is a variation of Kfold. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set.*

```
# taken from sklearn user guide
tscv = TimeSeriesSplit()
print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=5)
for train_index, test_index in tscv.split(X):
  print("TRAIN:", train_index, "TEST:", test_index)
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]
```

However, this was an experimentation and illustration of how supervised algorithms can effectively use to predict stock prices.

Conclusion

We have experimented with the prediction modeling of DOW Jones Industrial Average and found that Prophet is quite powerful and effective in time series forecasting. The forecast error using the historical data. can also be measured by comparing the predicted values with the actual values and using cross validation. The performance of neural networks too can be improved significantly optimizing parameters as suggested and selecting the window size.

I can be reached here.

Machine Learning Predictive Modeling Analytics Time Series Forecasting