

# GOLD PRICE FORECAST

Your Description Here



# Step 1: Importing Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_percentage_error
import tensorflow as tf
from keras import Model
from keras.layers import Input, Dense, Dropout
from keras.layers import LSTM
```

## Step 2: Reading Dataset

```
df = pd.read_csv('/kaggle/input/gold-price-10-years-20132023/Gold Price (2013-2023)
```

+ Code

+ Markdown

## Step 3: Dataset Overview

df								
	Date	Price	Open	High	Low	Vol.	Change %	
0	12/30/2022	1,826.20	1,821.80	1,832.40	1,819.80	107.50K	0.01%	
1	12/29/2022	1,826.00	1,812.30	1,827.30	1,811.20	105.99K	0.56%	
2	12/28/2022	1,815.80	1,822.40	1,822.80	1,804.20	118.08K	-0.40%	
3	12/27/2022	1,823.10	1,808.20	1,841.90	1,808.00	159.62K	0.74%	
4	12/26/2022	1,809.70	1,805.80	1,811.95	1,805.55	NaN	0.30%	
...								
2578	01/08/2013	1,663.20	1,651.50	1,662.60	1,648.80	0.13K	0.97%	
2579	01/07/2013	1,647.20	1,657.30	1,663.80	1,645.30	0.09K	-0.16%	
2580	01/04/2013	1,649.90	1,664.40	1,664.40	1,630.00	0.31K	-1.53%	
2581	01/03/2013	1,675.60	1,688.00	1,689.30	1,664.30	0.19K	-0.85%	
2582	01/02/2013	1,689.90	1,675.80	1,695.00	1,672.10	0.06K	0.78%	

2583 rows × 7 columns

## Dataset Basic Information:

[28]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2583 entries, 0 to 2582
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype  
---  --  
 0   Date        2583 non-null    object 
 1   Price       2583 non-null    object 
 2   Open         2583 non-null    object 
 3   High         2583 non-null    object 
 4   Low          2583 non-null    object 
 5   Vol.         2578 non-null    object 
 6   Change %    2583 non-null    object 
dtypes: object(7)
```

# Step 4: Data Preparation

+ Code

+ Markdown

## Step 4.1: Feature Subset Selection

Since we will not use Vol. and Change % features to predict Price, we will drop these two features:

```
df.drop(['Vol.', 'Change %'], axis=1, inplace=True)
```

## Step 4.2: Transforming Data

Date feature is stored as object in the data frame. To increase the speed of calculations, we convert it's data type to datetime and then sort this feature in ascending order:

```
df['Date'] = pd.to_datetime(df['Date'])
df.sort_values(by='Date', ascending=True, inplace=True)
df.reset_index(drop=True, inplace=True)
```

The ", " sign is redundant in the dataset. First, we remove it from the entire dataset and then change the data type of the numerical variables to float:

```
NumCols = df.columns.drop(['Date'])
df[NumCols] = df[NumCols].replace({',': ''}, regex=True)
df[NumCols] = df[NumCols].astype('float64')
```

Result:

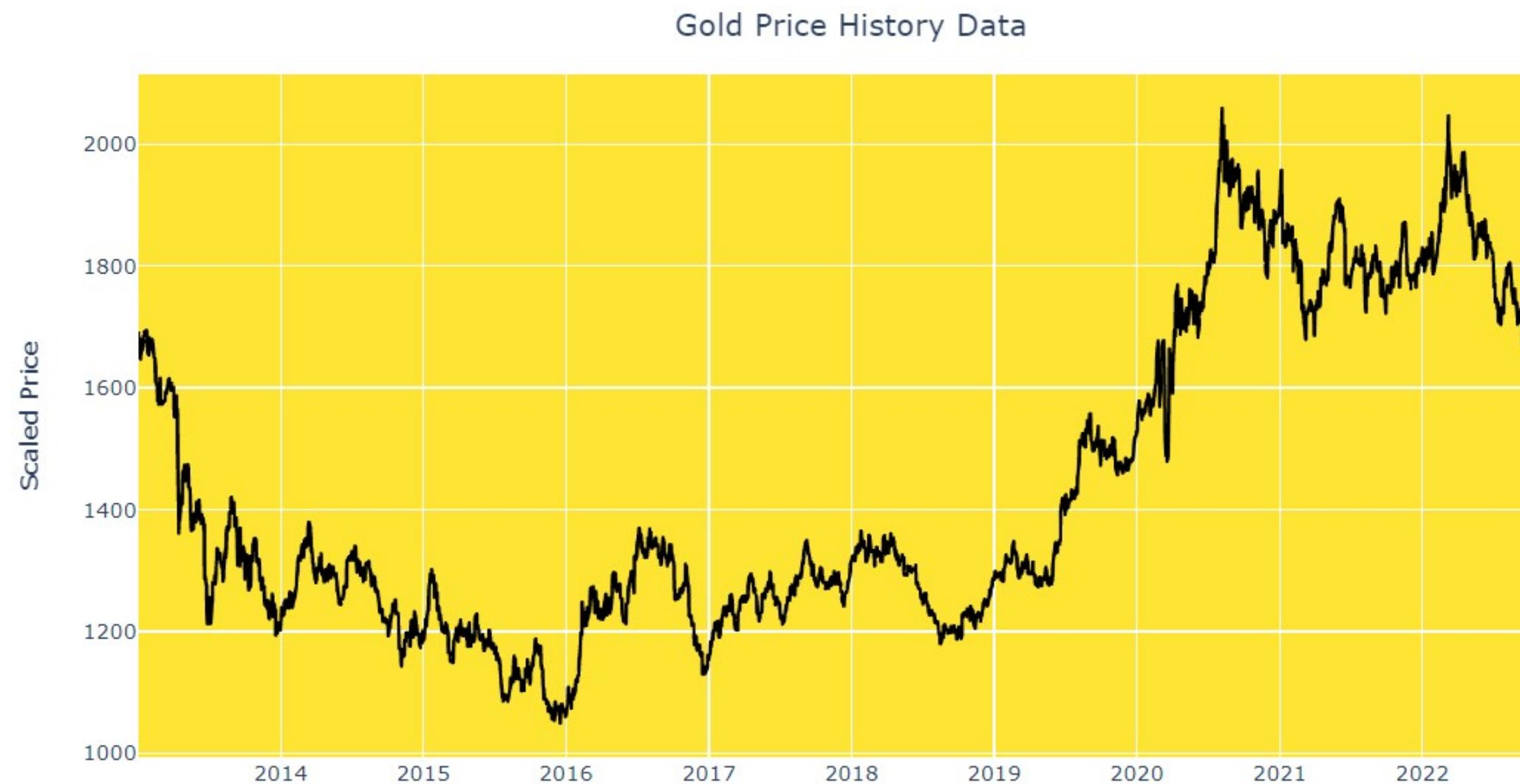
```
df.head()
```

	Date	Price	Open	High	Low
0	2013-01-02	1689.9	1675.8	1695.0	1672.1
1	2013-01-03	1675.6	1688.0	1689.3	1664.3
2	2013-01-04	1649.9	1664.4	1664.4	1630.0
3	2013-01-07	1647.2	1657.3	1663.8	1645.3
4	2013-01-08	1663.2	1651.5	1662.6	1648.8

# Step 5: Visualizing Gold Price History Data

Interactive Gold Price Chart:

```
fig = px.line(y=df.Price, x=df.Date)
fig.update_traces(line_color='black')
fig.update_layout(xaxis_title="Date",
                  yaxis_title="Scaled Price",
                  title={'text': "Gold Price History Data", 'y':0.95, 'x':0.5, 'xanchor': 'center'},
                  plot_bgcolor='rgba(255,223,0,0.8)')
```



# Step 6: Splitting Data to Training & Test Sets

Since we cannot train on future data in time series data, we should not divide the time series data randomly. In time series splitting, testing set is always later than training set. We consider the last year for testing and everything else for training:

```
test_size = df[df.Date.dt.year==2022].shape[0]
test_size
```

260

## Gold Price Training and Test Sets Plot:

+ Code

+ Markdown

```
:
plt.figure(figsize=(15, 6), dpi=150)
plt.rcParams['axes.facecolor'] = 'yellow'
plt.rc('axes', edgecolor='white')
plt.plot(df.Date[:-test_size], df.Price[:-test_size], color='black', lw=2)
plt.plot(df.Date[-test_size:], df.Price[-test_size:], color='blue', lw=2)
plt.title('Gold Price Training and Test Sets', fontsize=15)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Price', fontsize=12)
plt.legend(['Training set', 'Test set'], loc='upper left', prop={'size': 15})
plt.grid(color='white')
plt.show()
```

### Gold Price Training and Test Sets



## Step 7: Data Scaling

Since we aim to predict Price only based on its historical data, we scale Price using MinMaxScaler to avoid intensive computations:

+ Code

+ Markdown

```
scaler = MinMaxScaler()  
scaler.fit(df.Price.values.reshape(-1, 1))
```

MinMaxScaler()

## Step 8: Restructure Data & Create Sliding Window

The use of prior time steps to predict the next time step is called sliding window learning. We can do this by using previous time steps as input variables and use previous time steps is called the window width. Here we set window width to 60 of 60 time-stamp prices. `y_train` and `y_test` are also lists of gold prices containing `X_train` and `X_test` respectively:

+ Code

+ Markdown

```
window_size = 60
```

## Step 8: Restructure Data & Create Sliding Window

The use of prior time steps to predict the next time step is called sliding window. In this way, time series data can be expressed as supervised learning. We can do this by using previous time steps as input variables and use the next time step as the output variable. The number of previous time steps is called the window width. Here we set window width to 60. Therefore, X\_train and X\_test will be nested lists containing lists of 60 time-stamp prices. y\_train and y\_test are also lists of gold prices containing the next day's gold price corresponds to each list in X\_train and X\_test respectively:

```
window_size = 60
```

Training Set:

```
train_data = df.Price[:-test_size]
train_data = scaler.transform(train_data.values.reshape(-1,1))
```

## Training Set:

```
[40]:  
    train_data = df.Price[:-test_size]  
    train_data = scaler.transform(train_data.values.reshape(-1,1))
```

+ Code

+ Markdown

```
[41]:  
    X_train = []  
    y_train = []  
  
    for i in range(window_size, len(train_data)):  
        X_train.append(train_data[i-60:i, 0])  
        y_train.append(train_data[i, 0])
```

## Test Set:

```
[42]:  
    test_data = df.Price[-test_size-60:]  
    test_data = scaler.transform(test_data.values.reshape(-1,1))
```

```
[43]:  
    X_test = []  
    y_test = []  
  
    for i in range(window_size, len(test_data)):  
        X_test.append(test_data[i-60:i, 0])  
        y_test.append(test_data[i, 0])
```

## Step 9: Converting Data to Numpy Arrays

Now `X_train` and `X_test` are nested lists (two-dimensional lists) and `y_train` is a one-dimensional list. We need to convert them to numpy arrays with a higher dimension, which is the data format accepted by TensorFlow when training the neural network:

+ Code

+ Markdown

```
X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

```
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))
y_train = np.reshape(y_train, (-1,1))
y_test = np.reshape(y_test, (-1,1))
```

```
print('X_train Shape: ', X_train.shape)
print('y_train Shape: ', y_train.shape)
print('X_test Shape: ', X_test.shape)
print('y_test Shape: ', y_test.shape)
```

```
X_train Shape: (2263, 60, 1)
y_train Shape: (2263, 1)
X_test Shape: (260, 60, 1)
y_test Shape: (260, 1)
```

# Step 10: Creating an LSTM Network

We build an LSTM network, which is a type of Recurrent Neural Networks designed to solve vanishing gradient problem:

+ Code

+ Markdown

**Model Definition:**

```
def define_model():
    input1 = Input(shape=(window_size,1))
    x = LSTM(units = 64, return_sequences=True)(input1)
    x = Dropout(0.2)(x)
    x = LSTM(units = 64, return_sequences=True)(x)
    x = Dropout(0.2)(x)
    x = LSTM(units = 64)(x)
    x = Dropout(0.2)(x)
    x = Dense(32, activation='softmax')(x)
    dnn_output = Dense(1)(x)

    model = Model(inputs=input1, outputs=[dnn_output])
    model.compile(loss='mean_squared_error', optimizer='Nadam')
    model.summary()

    return model
```

```
model = define_model()
history = model.fit(X_train, y_train, epochs=150, batch_size=32, validation_split=0.1, verbose=1)
```

Model: "model\_1"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[None, 60, 1]	0
lstm_3 (LSTM)	(None, 60, 64)	16896
dropout_3 (Dropout)	(None, 60, 64)	0
lstm_4 (LSTM)	(None, 60, 64)	33024
dropout_4 (Dropout)	(None, 60, 64)	0
lstm_5 (LSTM)	(None, 64)	33024
dropout_5 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2080
dense_3 (Dense)	(None, 1)	33

Total params: 85,057  
Trainable params: 85,057  
Non-trainable params: 0

---

Epoch 1/150  
64/64 [=====] - 18s 173ms/step - loss: 0.0303 - val\_loss: 0.0671  
Epoch 2/150  
64/64 [=====] - 9s 145ms/step - loss: 0.0109 - val\_loss: 0.0353  
Epoch 3/150  
64/64 [=====] - 10s 162ms/step - loss: 0.0072 - val\_loss: 0.0171  
Epoch 4/150  
64/64 [=====] - 10s 149ms/step - loss: 0.0051 - val\_loss: 0.0077  
Epoch 5/150  
64/64 [=====] - 10s 149ms/step - loss: 0.0036 - val\_loss: 0.0032  
Epoch 6/150  
64/64 [=====] - 10s 161ms/step - loss: 0.0026 - val\_loss: 0.0021  
Epoch 7/150  
64/64 [=====] - 10s 150ms/step - loss: 0.0021 - val\_loss: 0.0026  
Epoch 8/150  
64/64 [=====] - 10s 149ms/step - loss: 0.0018 - val\_loss: 0.0021  
Epoch 9/150  
64/64 [=====] - 10s 163ms/step - loss: 0.0015 - val\_loss: 0.0058  
Epoch 10/150  
64/64 [=====] - 9s 146ms/step - loss: 0.0013 - val\_loss: 0.0033  
Epoch 11/150  
64/64 [=====] - 9s 148ms/step - loss: 0.0012 - val\_loss: 0.0022  
Epoch 12/150  
64/64 [=====] - 10s 161ms/step - loss: 0.0012 - val\_loss: 0.0053  
Epoch 13/150  
64/64 [=====] - 10s 149ms/step - loss: 0.0011 - val\_loss: 0.0018  
Epoch 14/150  
64/64 [=====] - 9s 148ms/step - loss: 0.0010 - val\_loss: 0.0071  
Epoch 15/150  
64/64 [=====] - 10s 149ms/step - loss: 0.0010 - val\_loss: 0.0079  
Epoch 16/150  
64/64 [=====] - 10s 161ms/step - loss: 9.9049e-04 - val\_loss: 0.0093  
Epoch 17/150  
64/64 [=====] - 10s 149ms/step - loss: 9.0258e-04 - val\_loss: 0.0051  
Epoch 18/150  
64/64 [=====] - 9s 147ms/step - loss: 8.6469e-04 - val\_loss: 0.0011  
Epoch 19/150  
64/64 [=====] - 10s 161ms/step - loss: 8.6761e-04 - val\_loss: 0.0014  
Epoch 20/150  
64/64 [=====] - 10s 150ms/step - loss: 7.9019e-04 - val\_loss: 0.0014  
Epoch 21/150  
64/64 [=====] - 10s 150ms/step - loss: 7.9578e-04 - val\_loss: 0.0051

# Step 11: Model Evaluation

Next, we evaluate our time series forecast using MAPE (Mean Absolute Percentage Error) metric:

+ Code + Markdown

```
result = model.evaluate(X_test, y_test)
y_pred = model.predict(X_test)
```

```
9/9 [=====] - 0s 42ms/step - loss: 5.8453e-04
```

```
MAPE = mean_absolute_percentage_error(y_test, y_pred)
Accuracy = 1 - MAPE
```

```
print("Test Loss:", result)
print("Test MAPE:", MAPE)
print("Test Accuracy:", Accuracy)
```

```
Test Loss: 0.0005845279665663838
Test MAPE: 0.025992325018147116
Test Accuracy: 0.9740076749818529
```

# Step 12: Visualizing Results

Returning the actual and predicted Price values to their primary scale:

+ Code + Markdown

```
y_test_true = scaler.inverse_transform(y_test)
y_test_pred = scaler.inverse_transform(y_pred)
```

Investigating the closeness of the prices predicted by the model to the actual prices:

```
plt.figure(figsize=(15, 6), dpi=150)
plt.rcParams['axes.facecolor'] = 'yellow'
plt.rc('axes', edgecolor='white')
plt.plot(df['Date'].iloc[: -test_size], scaler.inverse_transform(train_data), color='black', lw=2)
plt.plot(df['Date'].iloc[-test_size:], y_test_true, color='blue', lw=2)
plt.plot(df['Date'].iloc[-test_size:], y_test_pred, color='red', lw=2)
plt.title('Model Performance on Gold Price Prediction', fontsize=15)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Price', fontsize=12)
plt.legend(['Training Data', 'Actual Test Data', 'Predicted Test Data'], loc='upper left', prop={'size': 15})
plt.grid(color='white')
plt.show()
```

## Model Performance on Gold Price Prediction

