STOCK PRICE PREDICTION Using Regression and Deep Learning Algorithms

Discussed Algorithms:

1. Support Vector Regressor
2. Random Forest Regressor
3. KNN Regressor
4. LSTM (Long Short-Term Memory)
5. GRU (Gated Recurrent Unit)

In [1]:
```
!pip install yfinance
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import yfinance as yf
import datetime
import warnings
warnings.filterwarnings('ignore')
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: yfinance in /usr/local/lib/python3.8/dist-packages (0.2.12)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.8/dist-packages (from yfinance) (1.21.6)
Requirement already satisfied: requests>=2.26 in /usr/local/lib/python3.8/dist-packages (from yfinance) (2.28.2)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.8/dist-packages (from yfinance) (1.3.5)
Requirement already satisfied: appdirs>=1.4.4 in /usr/local/lib/python3.8/dist-packages (from yfinance) (1.4.4)
Requirement already satisfied: html5lib>=1.1 in /usr/local/lib/python3.8/dist-packages (from yfinance) (1.1)
Requirement already satisfied: multitasking>=0.0.7 in /usr/local/lib/python3.8/dist-packages (from yfinance) (0.0.11)
Requirement already satisfied: cryptography>=3.3.2 in /usr/local/lib/python3.8/dist-packages (from yfinance) (39.0.1)
Requirement already satisfied: frozendict>=2.3.4 in /usr/local/lib/python3.8/dist-packages (from yfinance) (2.3.5)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.8/dist-packages (from yfinance) (2022.7.1)
Requirement already satisfied: lxml>=4.9.1 in /usr/local/lib/python3.8/dist-packages (from yfinance) (4.9.2)
Requirement already satisfied: beautifulsoup4>=4.11.1 in /usr/local/lib/python3.8/dist-packages (from yfinance) (4.11.2)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.8/dist-packages (from beautifulsoup4>=4.11.1->yfinance) (2.4)
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.8/dist-packages (from cryptography>=3.3.2->yfinance) (1.15.1)
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.8/dist-packages (from html5lib>=1.1->yfinance) (1.15.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.8/dist-packages (from html5lib>=1.1->yfinance) (0.5.1)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.8/dist-packages (from requests>=2.26->yfinance) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests>=2.26->yfinance) (2022.12.7)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.8/dist-packages (from requests>=2.26->yfinance) (3.0.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests>=2.26->yfinance) (2.10)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/dist-packages (from cffi>=1.12->cryptography>=3.3.2->yfinance) (2.21)
```

Data Ingestion

In [2]:
```
df = yf.download("ADANIENT.NS", start="2020-01-01", end="2023-02-12")
df=df.round(2)
```

```
[*********************100%**********************]  1 of 1 completed
```

Top 5 rows of dataframe

In [3]:
```
df.head()
```

Out[3]:

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-01	209.00	210.45	206.65	207.85	206.12	1553127
2020-01-02	208.00	213.20	207.50	211.20	209.44	2991937
2020-01-03	210.25	212.35	205.80	208.30	206.56	2512421
2020-01-06	207.75	207.75	197.75	199.55	197.89	4353179
2020-01-07	200.55	205.70	200.55	204.05	202.35	2966120

Bottom 5 rows of dataframe

In [4]:
```
df.tail()
```

Out[4]:

Date	Open	High	Low	Close	Adj Close	Volume
2023-02-06	1575.00	1615.00	1435.20	1572.70	1572.70	19308603
2023-02-07	1571.00	1962.70	1525.60	1802.95	1802.95	19188072
2023-02-08	1869.85	2222.15	1840.85	2164.25	2164.25	19173006
2023-02-09	2168.00	2168.00	1731.40	1925.70	1925.70	18279862
2023-02-10	1769.00	1990.00	1733.15	1846.95	1846.95	11334878

Shape of the dataframe

In [5]:
```
df.shape
```

Out[5]: (776, 6)

List of columns in the dataframe

In [6]:
```
df.columns
```

Out[6]: Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume'], dtype='object')

Checking duplicate

In [7]:
```
df.duplicated().sum()
```

Out[7]: 0

Checking Null

In [8]:
```
df.isnull().sum()
```

Out[8]:
```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

Basic information about the dataframe

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 776 entries, 2020-01-01 to 2023-02-10
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Open       776 non-null    float64
 1   High       776 non-null    float64
 2   Low        776 non-null    float64
 3   Close      776 non-null    float64
 4   Adj Close  776 non-null    float64
 5   Volume     776 non-null    int64  
dtypes: float64(5), int64(1)
memory usage: 42.4 KB
```

Basic statistics of the dataframe

In [10]:
```
df.describe()
```

Out[10]:

	Open	High	Low	Close	Adj Close	Volume
count	776.000000	776.000000	776.000000	776.000000	776.000000	7.760000e+02
mean	1457.520941	1484.426740	1427.072358	1456.515851	1455.890941	4.753229e+06
std	1157.218274	1172.336313	1135.864261	1154.038099	1154.213288	5.389994e+06
min	121.000000	129.800000	116.400000	120.900000	120.770000	2.482490e+05
25%	312.587500	319.112500	307.012500	312.750000	312.410000	1.714098e+06
50%	1443.925000	1474.375000	1415.175000	1442.225000	1441.615000	3.077650e+06
75%	2136.037500	2176.250000	2076.287500	2111.450000	2110.550000	5.361654e+06
max	4175.000000	4190.000000	4066.400000	4165.300000	4165.300000	4.926454e+07

Checking unique and making Date as index to perform our analysis

In [11]:
```
df.nunique()
df['Date'] = df.index
```

In [12]:
```
df
```

Out[12]:

Date	Open	High	Low	Close	Adj Close	Volume	Date
2020-01-01	209.00	210.45	206.65	207.85	206.12	1553127	2020-01-01
2020-01-02	208.00	213.20	207.50	211.20	209.44	2991937	2020-01-02
2020-01-03	210.25	212.35	205.80	208.30	206.56	2512421	2020-01-03
2020-01-06	207.75	207.75	197.75	199.55	197.89	4353179	2020-01-06
2020-01-07	200.55	205.70	200.55	204.05	202.35	2966120	2020-01-07
...	...	...	...	...	...	...	...
2023-02-06	1575.00	1615.00	1435.20	1572.70	1572.70	19308603	2023-02-06
2023-02-07	1571.00	1962.70	1525.60	1802.95	1802.95	19188072	2023-02-07
2023-02-08	1869.85	2222.15	1840.85	2164.25	2164.25	19173006	2023-02-08
2023-02-09	2168.00	2168.00	1731.40	1925.70	1925.70	18279862	2023-02-09
2023-02-10	1769.00	1990.00	1733.15	1846.95	1846.95	11334878	2023-02-10

776 rows × 7 columns

Cheking range of date

In [13]:
```
print("Starting date: ",df.iloc[0][-1])
print("Ending date: ", df.iloc[-1][-1])
print("Duration: ", df.iloc[-1][-1]-df.iloc[0][-1])
```

```
Starting date:  2020-01-01 00:00:00
Ending date:  2023-02-10 00:00:00
Duration:  1136 days 00:00:00
```

Creating monthvise data

In [14]:
```
monthvise= df.groupby(df['Date'].dt.strftime('%B'))[['Open','Close']].mean().sort_values(by='Close')
```

In [15]:
```
monthvise
```

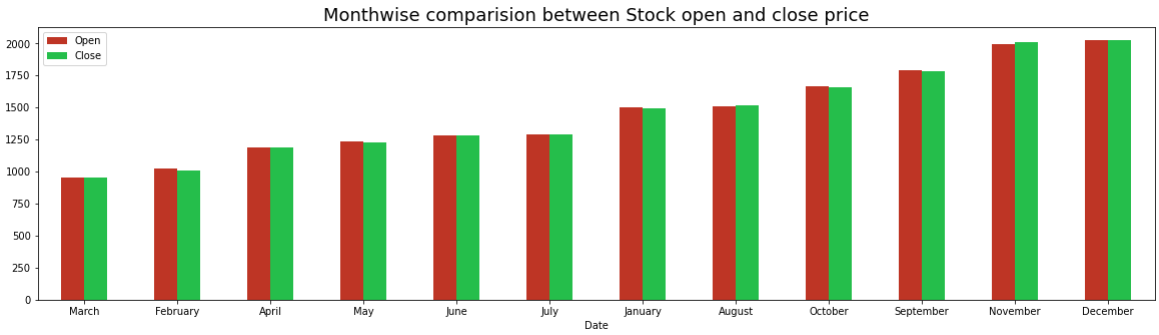
Out[15]:

Date	Open	Close
March	951.176190	955.553175
February	1023.341791	1009.479104
April	1184.605357	1189.132143
May	1235.105000	1230.002500
June	1284.721212	1285.371212
July	1289.846154	1291.502308
January	1501.656548	1490.852976
August	1508.315323	1519.850000
October	1667.537500	1660.111667
September	1787.856923	1786.126154
November	1996.344262	2007.223770
December	2026.534328	2024.473881

Potting monthvise data

In [16]:
```
monthvise.plot.bar(rot=0, color={"Open": "#BE3525", "Close":"#25BE4B"},figsize=(20,5))
plt.title("Monthwise comparision between Stock open and close price", fontsize = 18)
```

Out[16]:
```
Text(0.5, 1.0, 'Monthwise comparision between Stock open and close price')
```



In [17]:
```
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
```

Minimum low price in each month

In [18]:
```
df.groupby(df['Date'].dt.strftime('%B'))['Low'].min()
```

Out[18]:
```
Date
April        128.00
August       171.20
December     395.10
February     215.60
January      194.55
July         145.20
June         141.00
March        116.40
May          127.30
November     333.05
October      296.00
September    257.50
Name: Low, dtype: float64
```

Maximum low price in each month

In [19]:
```
df.groupby(df['Date'].dt.strftime('%B'))['Low'].max()
```

Out[19]:
```
Date
April        2336.30
August       3161.00
December     4066.40
February     1941.20
January      3822.55
July         2560.00
June         2207.05
March        1991.00
May          2311.00
November     4022.00
October      3326.10
September    3812.00
Name: Low, dtype: float64
```

Minimum high price in each month

In [20]:
```
df.groupby(df['Date'].dt.strftime('%B'))['High'].min()
```

Out[20]:
```
Date
April        134.60
August       178.70
December     423.95
February     224.65
January      203.55
July         150.00
June         148.45
March        129.80
May          135.40
November     344.30
October      311.95
September    281.25
Name: High, dtype: float64
```

Maximum high price in each month

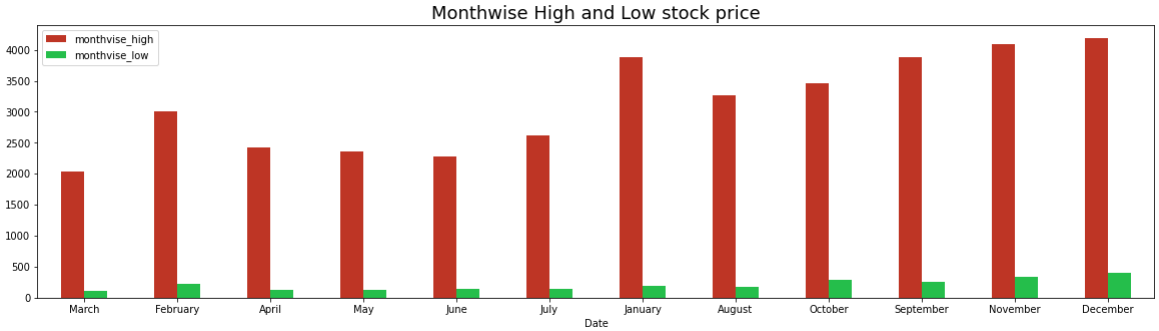
In [21]:
```
df.groupby(df['Date'].dt.strftime('%B'))['High'].max()
```

Out[21]:
```
Date
April        2420.95
August       3263.10
December     4190.00
February     3010.75
January      3880.00
July         2622.00
June         2274.00
March        2042.00
May          2362.90
November     4096.00
October      3460.05
September    3885.00
Name: High, dtype: float64
```

In [22]:
```
monthvise['monthvise_high']= df.groupby(df['Date'].dt.strftime('%B'))['High'].max()
monthvise['monthvise_low']= df.groupby(df['Date'].dt.strftime('%B'))['Low'].min()
```

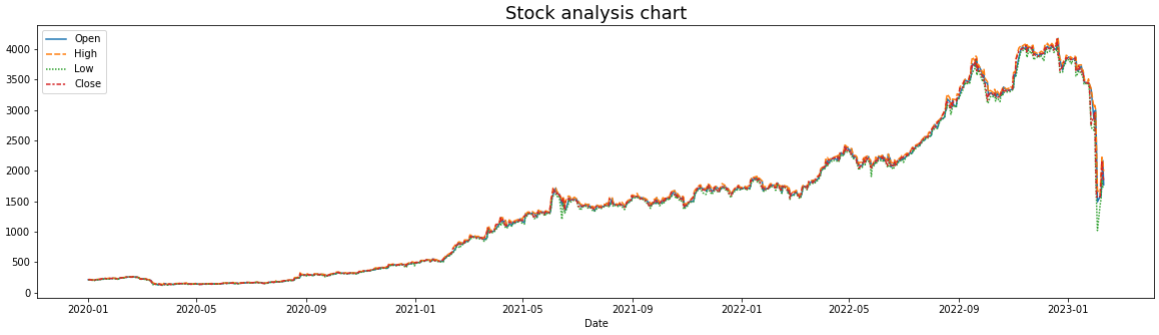
In [23]:
```
monthvise[['monthvise_high', 'monthvise_low']].plot.bar(rot=0, color={"monthvise_high": "#BE3525", "monthvise_low":"#25BE4B"},figsize=(20,5))
plt.title("Monthwise High and Low stock price", fontsize = 18)
```

Out[23]: Text(0.5, 1.0, 'Monthwise High and Low stock price')



In [24]:
```
plt.figure(figsize=(20,5))
sns.lineplot(data=df[['Open','High','Low','Close']])
plt.title("Stock analysis chart",fontsize=18)
```

Out[24]: Text(0.5, 1.0, 'Stock analysis chart')



In [25]:
```
closedf = df[['Date','Close']]
print("Shape of close dataframe:", closedf.shape)
```

```
Shape of close dataframe: (776, 2)
```

In [26]:
```
plt.figure(figsize=(20,5))
sns.lineplot(data=closedf)
plt.title("Stock close price chart",fontsize=18)
```

Out[26]: Text(0.5, 1.0, 'Stock close price chart')

Model Creation

In [27]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM, GRU
```

Evaluation Metrics

mean_squared_error:

- This is a metric that measures the average squared difference between the predicted and actual values. It is calculated as the mean of the squared differences between the predicted and actual values. This metric is useful for penalizing large errors in the predictions, and is commonly used as a loss function during training. However, it can be sensitive to outliers and does not provide an easily interpretable measure of error in the original units of the response variable.

mean_absolute_error:

- This is a metric that measures the average absolute difference between the predicted and actual values. It is calculated as the mean of the absolute differences between the predicted and actual values. This metric provides an easily interpretable measure of error in the original units of the response variable, and is less sensitive to outliers than the mean squared error.

explained_variance_score:

- This is a metric that measures the proportion of variance in the target variable that is explained by the model. It is calculated as 1 - (variance of residuals / variance of target variable). This metric provides an indication of how well the model fits the data and can be interpreted as the amount of information captured by the model.

r2_score:

- This is a metric that measures the proportion of variance in the target variable that is explained by the model, normalized by the total variance in the target variable. It is calculated as 1 - (sum of squared residuals / total sum of squares). This metric provides a normalized indication of how well the model fits the data, and can be interpreted as the percentage of variance in the target variable that is explained by the model. It ranges from 0 to 1, with 1 indicating a perfect fit and values close to 0 indicating poor performance.

mean_poisson_deviance:

- It is a metric that measures the goodness of fit of a Poisson regression model. It compares the predicted values of the model with the observed values and calculates the mean deviation of the predicted values from the observed values. A lower value of this metric indicates a better fit of the model to the data.

mean_gamma_deviance:

- It is a metric that measures the goodness of fit of a Gamma regression model. It compares the predicted values of the model with the observed values and calculates the mean deviation of the predicted values from the observed values. A lower value of this metric indicates a better fit of the model to the data.

accuracy_score:

- It is a metric used to evaluate the performance of a classification model. It calculates the proportion of correctly predicted labels to the total number of samples. A higher value of this metric indicates a better classification performance of the model.

In [28]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, explained_variance_score, r2_score
from sklearn.metrics import mean_poisson_deviance, mean_gamma_deviance, accuracy_score
from sklearn.preprocessing import MinMaxScaler
```

Scaling

MinMaxScaler:

- MinMaxScaler is a technique used in data preprocessing to scale and transform features or variables of a dataset to a particular range, usually between 0 and 1. This technique helps to normalize the data and make it more consistent, making it easier to compare and analyze the data.
- MinMaxScaler is a simple linear scaling technique that works by subtracting the minimum value of the feature and then dividing by the range (i.e., the difference between the maximum and minimum values) of the feature. The resulting values are then rescaled to fit within the desired range.
- MinMaxScaler is commonly used in machine learning algorithms, especially in algorithms that involve distance calculations or optimization, such as clustering, gradient descent, and neural networks. It helps to improve the performance of the algorithms by reducing the influence of the scale of the variables on the model.

In [29]:
```
close_stock = closedf.copy()
del closedf['Date']
scaler=MinMaxScaler(feature_range=(0,1))
closedf=scaler.fit_transform(np.array(closedf).reshape(-1,1))
print(closedf.shape)
```

```
(776, 1)
```

In [30]:
```
training_size=int(len(closedf)*0.75)
test_size=len(closedf)-training_size
train_data,test_data=closedf[0:training_size,:],closedf[training_size:len(closedf),:1]
print("train_data: ", train_data.shape)
print("test_data: ", test_data.shape)
# print(train_data)
# print(test_data)
```

```
train_data:  (582, 1)
test_data:  (194, 1)
```

In [31]:
```
# convert an array of values into a dataset matrix
"""
This function creates a sliding window of size time_step over the input dataset and constructs a set of input-output pairs for training a time series forecasting model.
The input matrix dataX contains n rows, where n is the number of time steps in the input sequence, and time_step columns, representing the past time_step values of the
input sequence. The output matrix dataY contains n rows and 1 column,representing the next value in the sequence to be predicted.
"""
def create_dataset(dataset, time_step=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-time_step-1):
        a = dataset[i:(i+time_step), 0]   ###i=0, 0,1,2,3-----99   100
        dataX.append(a)
        dataY.append(dataset[i + time_step, 0])
    return np.array(dataX), np.array(dataY)
```

In [32]:
```
# reshape into X=t,t+1,t+2,t+3 and Y=t+4
time_step = 15
X_train, y_train = create_dataset(train_data, time_step)
X_test, y_test = create_dataset(test_data, time_step)

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

```
X_train:  (566, 15)
y_train:  (566,)
X_test:  (178, 15)
y_test (178,)
```

Support Vector Regressor

- Support Vector Regressor (SVR) is a machine learning algorithm used for regression analysis. It is an extension of the popular Support Vector Machine (SVM) algorithm used for classification problems.
- SVR tries to fit a line or a curve through the data points in such a way that the margin between the predicted values and the actual values is minimized. The algorithm does this by identifying the support vectors, which are the data points closest to the line or curve being fitted, and using them to optimize the margin.
- SVR is particularly useful when dealing with nonlinear data, as it is able to transform the data into a higher dimensional space, where it can be more easily separated. It is also highly effective in dealing with noisy data, as it is able to effectively filter out the noise and identify the underlying trends in the data.
- In summary, SVR is a powerful regression algorithm that is highly effective in dealing with nonlinear and noisy data, making it a valuable tool for data scientists and machine learning practitioners.

In [33]:
```
from sklearn.svm import SVR
svr_rbf = SVR(kernel= 'rbf', C= 1e2, gamma= 0.1)
svr_rbf.fit(X_train, y_train)
```

Out[33]:
```
SVR(C=100.0, gamma=0.1)
```

In [34]:
```
# Lets Do the prediction

train_predict =svr_rbf.predict(X_train)
test_predict=svr_rbf.predict(X_test)

print(train_predict.shape)
print(test_predict.shape)

train_predict = train_predict.reshape(-1,1)
test_predict = test_predict.reshape(-1,1)

print("Train data prediction:", train_predict.shape)
print("Test data prediction:", test_predict.shape)
```

```
(566,)
(178,)
Train data prediction: (566, 1)
Test data prediction: (178, 1)
```

In [35]:
```
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
print(train_predict.shape)
print(test_predict.shape)
# print(train_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))
print(original_ytrain.shape)
print(original_ytest.shape)
```

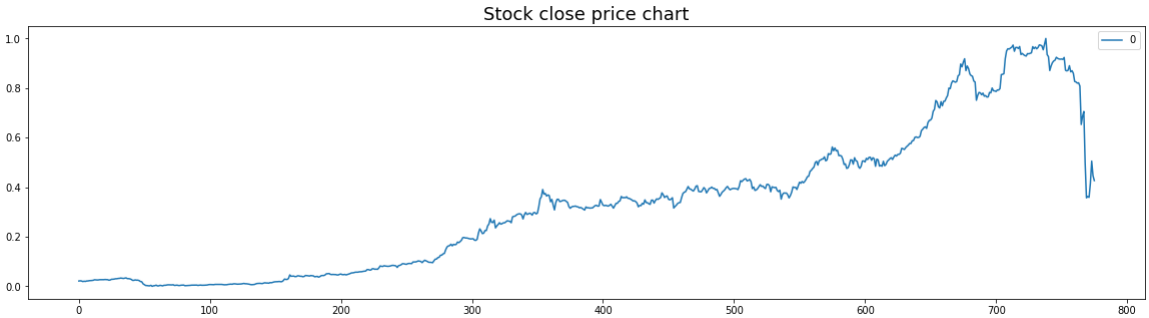
```
(566, 1)
(178, 1)
(566, 1)
(178, 1)
```

In [36]:
```
# Evaluation metrices RMSE and MAE
print("-------------------------------------------------------------------------------------------------------")
print("Train data RMSE: ", np.sqrt(mean_squared_error(original_ytrain,train_predict)))
print("Train data MSE: ", mean_squared_error(original_ytrain,train_predict))
print("Test data MAE: ", mean_absolute_error(original_ytrain,train_predict))
print("-------------------------------------------------------------------------------------------------------")
print("Test data RMSE: ", np.sqrt(mean_squared_error(original_ytest,test_predict)))
print("Test data MSE: ", mean_squared_error(original_ytest,test_predict))
print("Test data MAE: ", mean_absolute_error(original_ytest,test_predict))
print("-------------------------------------------------------------------------------------------------------")
print("Train data explained variance regression score:", explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:", explained_variance_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------")
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------")
print("Train data MGD: ", mean_gamma_deviance(original_ytrain, train_predict))
print("Test data MGD: ", mean_gamma_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------")
print("Train data MPD: ", mean_poisson_deviance(original_ytrain, train_predict))
print("Test data MPD: ", mean_poisson_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------")
```

```
-------------------------------------------------------------------------------------------------------
Train data RMSE:  223.56371663422217
Train data MSE:  49980.73539530679
Test data MAE:  182.52147205638119
-------------------------------------------------------------------------------------------------------
Test data RMSE:  1018.6151462017375
Test data MSE:  1037576.816071587
Test data MAE:  891.8595375886738
-------------------------------------------------------------------------------------------------------
Train data explained variance regression score: 0.9203253647034753
Test data explained variance regression score: 0.3674685081234186
-------------------------------------------------------------------------------------------------------
Train data R2 score: 0.8899002291580416
Test data R2 score: -1.134778818453622
-------------------------------------------------------------------------------------------------------
Train data MGD:  0.27937958841738786
Test data MGD:  0.13571715142418472
-------------------------------------------------------------------------------------------------------
Train data MPD:  108.34200696977018
Test data MPD:  372.3713804549531
-------------------------------------------------------------------------------------------------------
```

In [37]:
```
plt.figure(figsize=(20,5))
sns.lineplot(data=closedf)
plt.title("Stock close price chart",fontsize=18)
```

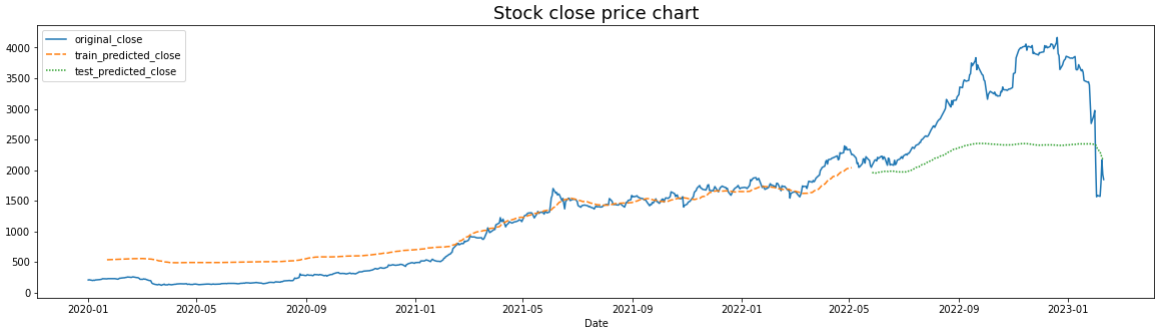
Out[37]:
```
Text(0.5, 1.0, 'Stock close price chart')
```



In [38]:
```
# shift train predictions for plotting
look_back=time_step
trainPredictPlot = np.empty_like(closedf)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# shift test predictions for plotting
testPredictPlot = np.empty_like(closedf)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(closedf)-1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

plotdf = pd.DataFrame({'Date': close_stock['Date'],'original_close': close_stock['Close'],'train_predicted_close': trainPredictPlot.reshape(1,-1)[0].tolist(),
                       'test_predicted_close': testPredictPlot.reshape(1,-1)[0].tolist()})

plt.figure(figsize=(20,5))
sns.lineplot(data=plotdf)
plt.title("Stock close price chart",fontsize=18)
```

```
Train predicted data:  (776, 1)
Test predicted data:  (776, 1)
```

Out[38]:
```
Text(0.5, 1.0, 'Stock close price chart')
```



In [39]:
```
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
temp_input
```

Out[39]:
```
[0.8246587874592028,
 0.8197631292651566,
 0.8211601226387102,
 0.8080432202551675,
 0.6528904163782019,
 0.685379784393235,
 0.705419839778459,
 0.4980837701513202 7,
 0.3571234299278014,
 0.36245178518445254,
 0.3589654831371773,
 0.41589605380278905,
 0.5052294530709129,
 0.4462466620512313,
 0.42677529423400257]
```

In [40]:
```
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
temp_input
```

Out[40]:
```
[0.8246587874592028,
 0.8197631292651566,
 0.8211601226387102,
 0.8080432202551675,
 0.6528904163782019,
 0.685379784393235,
 0.705419839778459,
 0.4980837701513202 7,
 0.3571234299278014,
 0.36245178518445254,
 0.3589654831371773,
 0.41589605380278905,
 0.5052294530709129,
 0.4462466620512313,
 0.42677529423400257]
```

In [41]:
```
# CReating array of last 15 days
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
from numpy import array

lst_output=[]
n_steps=time_step
i=0
pred_days = 10
while(i<pred_days):
    if(len(temp_input)>time_step):
        x_input=np.array(temp_input[1:])
        #print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)

        yhat = svr_rbf.predict(x_input)
        #print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat.tolist())
        temp_input=temp_input[1:]

        lst_output.extend(yhat.tolist())
        i=i+1

    else:
        yhat = svr_rbf.predict(x_input)

        temp_input.extend(yhat.tolist())
        lst_output.extend(yhat.tolist())

        i=i+1

print("Output of predicted next days: ", len(lst_output))
print("Output of predicted next days: ",lst_output)
```

```
Output of predicted next days:  10
Output of predicted next days:  [0.4862338398203704, 0.4761172790939312, 0.4654626142603577, 0.4547085305133641
3, 0.4427945754093995, 0.4342864229584369, 0.424721958276958, 0.41397205548867544, 0.41031426313954716, 0.41272147235010775]
```

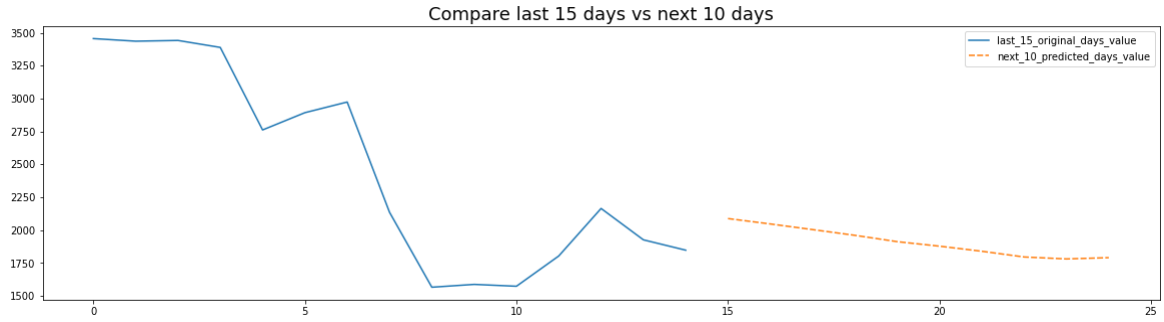
In [42]:
```
# Creating list of days to use for comparision
last_days=np.arange(1,time_step+1)
day_pred=np.arange(time_step+1,time_step+pred_days+1)
print(last_days)
print(day_pred)

temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat
next_predicted_days_value = temp_mat
len(last_original_days_value)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25]
```

Out[42]:
```
25
```

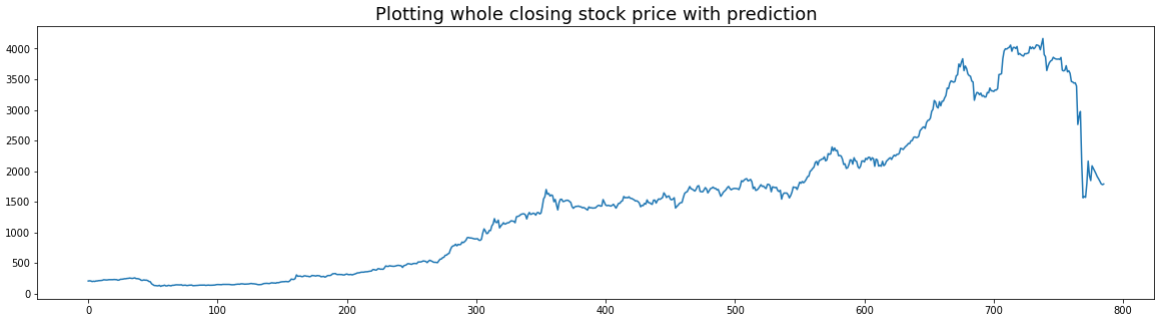
In [43]:
```
#  Creating empty dataframe for ploting graph
temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat.copy()
next_predicted_days_value = temp_mat.copy()

# Storing last 15 days and next 10 days values for comparision
last_original_days_value[0:time_step] = scaler.inverse_transform(closedf[len(closedf)-time_step:]).reshape(1,-1).tolist()[0]
next_predicted_days_value[time_step:] = scaler.inverse_transform(np.array(lst_output).reshape(-1,1)).reshape(1,-1).tolist()[0]
new_pred_plot = pd.DataFrame({
    'last_15_original_days_value':last_original_days_value,
    'next_10_predicted_days_value':next_predicted_days_value
})


plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Compare last 15 days vs next 10 days",fontsize=18)
```

Out[43]:
```
Text(0.5, 1.0, 'Compare last 15 days vs next 10 days')
```



In [44]:
```
# Creating List of close price
svrdf=closedf.tolist()
# Adding next 10 days prediction
svrdf.extend((np.array(lst_output).reshape(-1,1)).tolist())
# Tranforming to iriginal values
svrdf=scaler.inverse_transform(svrdf).reshape(1,-1).tolist()[0]

plt.figure(figsize=(20,5))
sns.lineplot(data=svrdf)
plt.title("Plotting whole closing stock price with prediction",fontsize=18)
```

Out[44]:
```
Text(0.5, 1.0, 'Plotting whole closing stock price with prediction')
```



Random Forest Regressor

- Random Forest Regressor is a popular machine learning algorithm that is widely used for solving regression problems. This algorithm is based on the concept of decision trees and ensemble learning. The Random Forest Regressor algorithm involves creating a large number of decision trees, and then combining the results of each tree to produce a final prediction.
- In this algorithm, each decision tree is trained on a randomly selected subset of the data, which helps to prevent overfitting. The final prediction is then made by aggregating the predictions of all the decision trees. This technique helps to improve the accuracy and stability of the model by reducing the impact of individual decision trees.
- Random Forest Regressor is highly flexible and can be used to solve a wide range of regression problems. It is also easy to use and implement, making it a popular choice among data scientists and machine learning practitioners. Overall, Random Forest Regressor is a powerful tool for solving complex regression problems, and it can provide highly accurate predictions in a variety of contexts.

In [45]:
```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 100, random_state = 0)
regressor.fit(X_train, y_train)
```

Out[45]:
```
RandomForestRegressor(random_state=0)
```

In [46]:
```
# Lets Do the prediction

train_predict=regressor.predict(X_train)
test_predict=regressor.predict(X_test)

train_predict = train_predict.reshape(-1,1)
test_predict = test_predict.reshape(-1,1)

print("Train data prediction:", train_predict.shape)
print("Test data prediction:", test_predict.shape)
```

```
Train data prediction: (566, 1)
Test data prediction: (178, 1)
```

In [47]:
```
# Transform back to original form
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))
```

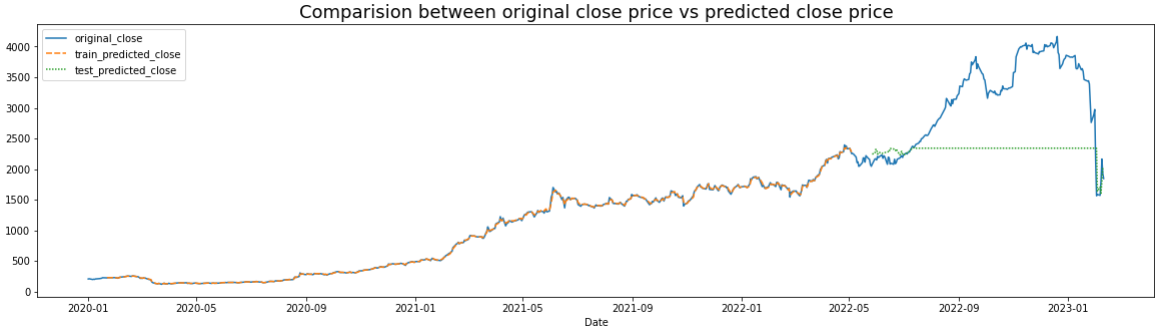
In [48]:
```
# Evaluation metrices RMSE and MAE
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data RMSE: ", np.sqrt(mean_squared_error(original_ytrain,train_predict)))
print("Train data MSE: ", mean_squared_error(original_ytrain,train_predict))
print("Test data MAE: ", mean_absolute_error(original_ytrain,train_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Test data RMSE: ", np.sqrt(mean_squared_error(original_ytest,test_predict)))
print("Test data MSE: ", mean_squared_error(original_ytest,test_predict))
print("Test data MAE: ", mean_absolute_error(original_ytest,test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data explained variance regression score:", explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:", explained_variance_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MGD: ", mean_gamma_deviance(original_ytrain, train_predict))
print("Test data MGD: ", mean_gamma_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MPD: ", mean_poisson_deviance(original_ytrain, train_predict))
print("Test data MPD: ", mean_poisson_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
```

```
-------------------------------------------------------------------------------------------------------------------
Train data RMSE:  13.573556802112037
Train data MSE:  184.24144426016196
Test data MAE:  8.062864840989498
-------------------------------------------------------------------------------------------------------------------
Test data RMSE:  1056.6664325851684
Test data MSE:  1116543.9497522665
Test data MAE:  876.7603623595508
-------------------------------------------------------------------------------------------------------------------
Train data explained variance regression score: 0.9995945892932148
Test data explained variance regression score: 0.11218360278131945
-------------------------------------------------------------------------------------------------------------------
Train data R2 score: 0.9995941448113519
Test data R2 score: -1.2972509956691538
-------------------------------------------------------------------------------------------------------------------
Train data MGD:  0.00017954496805427188
Test data MGD:  0.14833336232114536
-------------------------------------------------------------------------------------------------------------------
Train data MPD:  0.1412093803688879
Test data MPD:  404.07448750953745
-------------------------------------------------------------------------------------------------------------------
```

In [49]:
```
# shift train predictions for plotting
look_back=time_step
trainPredictPlot = np.empty_like(closedf)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# shift test predictions for plotting
testPredictPlot = np.empty_like(closedf)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(closedf)-1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

plotdf = pd.DataFrame({'Date': close_stock['Date'],'original_close': close_stock['Close'],'train_predicted_close': trainPredictPlot.reshape(1,-1)[0].tolist(),
                       'test_predicted_close': testPredictPlot.reshape(1,-1)[0].tolist()})

plt.figure(figsize=(20,5))
sns.lineplot(data=plotdf)
plt.title("Comparision between original close price vs predicted close price",fontsize=18)
```

```
Train predicted data:  (776, 1)
Test predicted data:  (776, 1)
```

Out[49]:
```
Text(0.5, 1.0, 'Comparision between original close price vs predicted close price')
```



In [50]:
```
# CReating array of last 15 days
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
from numpy import array

lst_output=[]
n_steps=time_step
i=0
pred_days = 10
while(i<pred_days):
    if(len(temp_input)>time_step):
        x_input=np.array(temp_input[1:])
        #print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)

        yhat = regressor.predict(x_input)
        #print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat.tolist())
        temp_input=temp_input[1:]

        lst_output.extend(yhat.tolist())
        i=i+1

    else:
        yhat = regressor.predict(x_input)

        temp_input.extend(yhat.tolist())
        lst_output.extend(yhat.tolist())

        i=i+1

print("Output of predicted next days: ", len(lst_output))
print("Output of predicted next days: ",lst_output)
```

```
Output of predicted next days:  10
Output of predicted next days:  [0.4791820789239438, 0.48538361685293197, 0.4844365047967557, 0.5128993175749179, 0.512615097418653, 0.5043477400850555, 0.5007559835822
368, 0.499420062308377, 0.5043311739689442, 0.5146825239837796]
```

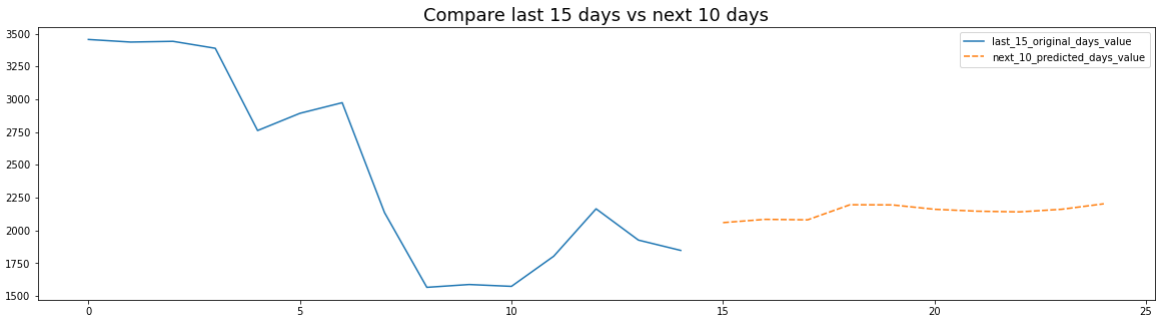
In [51]:
```
# Creating list of days to use for comparision
last_days=np.arange(1,time_step+1)
day_pred=np.arange(time_step+1,time_step+pred_days+1)
print(last_days)
print(day_pred)

temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat
next_predicted_days_value = temp_mat
len(last_original_days_value)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25]
```

Out[51]:
```
25
```

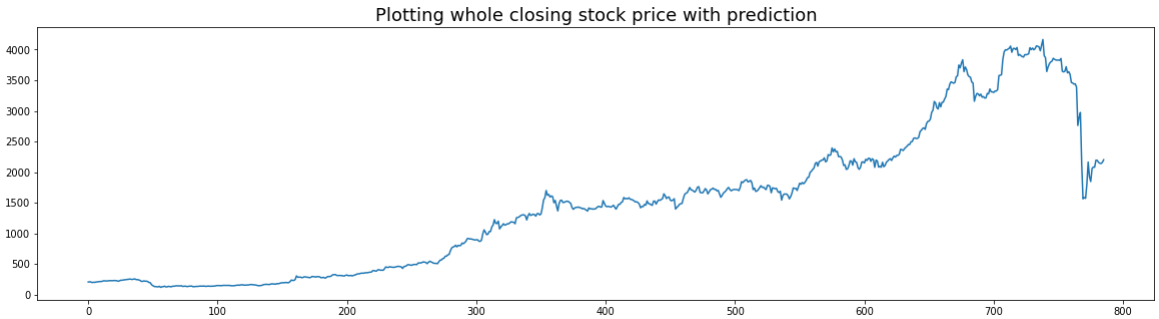
In [52]:
```
#  Creating empty dataframe for ploting graph
temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat.copy()
next_predicted_days_value = temp_mat.copy()

# Storing last 15 days and next 10 days values for comparision
last_original_days_value[0:time_step] = scaler.inverse_transform(closedf[len(closedf)-time_step:]).reshape(1,-1).tolist()[0]
next_predicted_days_value[time_step:] = scaler.inverse_transform(np.array(lst_output).reshape(-1,1)).reshape(1,-1).tolist()[0]
new_pred_plot = pd.DataFrame({
    'last_15_original_days_value':last_original_days_value,
    'next_10_predicted_days_value':next_predicted_days_value
})

plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Compare last 15 days vs next 10 days",fontsize=18)
```

Out[52]:
```
Text(0.5, 1.0, 'Compare last 15 days vs next 10 days')
```



In [53]:
```
# Creating list of close price
rfdf=closedf.tolist()
# Adding next 10 days prediction
rfdf.extend((np.array(lst_output).reshape(-1,1)).tolist())
# Tranforming to iriginal values
rfdf=scaler.inverse_transform(rfdf).reshape(1,-1).tolist()[0]

plt.figure(figsize=(20,5))
sns.lineplot(data=rfdf)
plt.title("Plotting whole closing stock price with prediction",fontsize=18)
```

Out[53]:
```
Text(0.5, 1.0, 'Plotting whole closing stock price with prediction')
```



K Nearest Neighbors (KNN)

- K Nearest Neighbors (KNN) is a type of non-parametric algorithm used for regression and classification problems. The KNN regression algorithm is called the K Neighbors Regressor.
- In KNN regression, the prediction for a new data point is based on the K-nearest neighbors of that data point in the training dataset. The K neighbors are the K training data points that are closest to the new data point based on some distance metric, such as Euclidean distance. The prediction for the new data point is then computed as the average of the output values of the K nearest neighbors.
- KNN regression can be useful when there is no clear functional relationship between the input and output variables, and the data has complex patterns that are difficult to model with linear regression or other parametric models. However, KNN regression can be sensitive to outliers, and the choice of the value of K can have a significant impact on the accuracy of the predictions.

In [54]:
```
from sklearn import neighbors
K = time_step
neighbor = neighbors.KNeighborsRegressor(n_neighbors = K)
neighbor.fit(X_train, y_train)
```

Out[54]:
```
KNeighborsRegressor(n_neighbors=15)
```

In [55]:
```
# Lets Do the prediction

train_predict=neighbor.predict(X_train)
test_predict=neighbor.predict(X_test)

train_predict = train_predict.reshape(-1,1)
test_predict = test_predict.reshape(-1,1)

print("Train data prediction:", train_predict.shape)
print("Test data prediction:", test_predict.shape)
```

```
Train data prediction: (566, 1)
Test data prediction: (178, 1)
```

In [56]:
```
# Transform back to original form

train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))
```

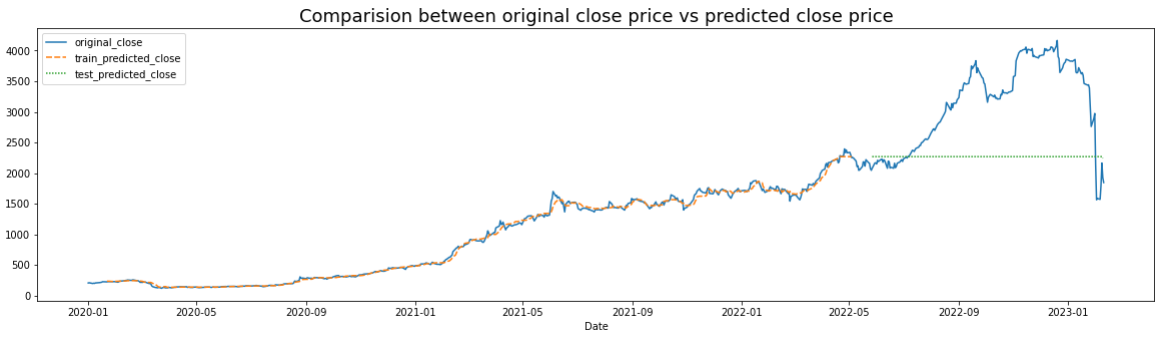
In [57]:
```
# Evaluation metrices RMSE and MAE
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data RMSE: ", np.sqrt(mean_squared_error(original_ytrain,train_predict)))
print("Train data MSE: ", mean_squared_error(original_ytrain,train_predict))
print("Test data MAE: ", mean_absolute_error(original_ytrain,train_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Test data RMSE: ", np.sqrt(mean_squared_error(original_ytest,test_predict)))
print("Test data MSE: ", mean_squared_error(original_ytest,test_predict))
print("Test data MAE: ", mean_absolute_error(original_ytest,test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data explained variance regression score:", explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:", explained_variance_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MGD: ", mean_gamma_deviance(original_ytrain, train_predict))
print("Test data MGD: ", mean_gamma_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MPD: ", mean_poisson_deviance(original_ytrain, train_predict))
print("Test data MPD: ", mean_poisson_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
```

```
-------------------------------------------------------------------------------------------------------------------
Train data RMSE:  42.282687137254165
Train data MSE:  1787.8256315469189
Test data MAE:  26.9667667844523
-------------------------------------------------------------------------------------------------------------------
Test data RMSE:  1117.5718099973767
Test data MSE:  1248966.7505008124
Test data MAE:  939.4057116104873
-------------------------------------------------------------------------------------------------------------------
Train data explained variance regression score: 0.9961438140059866
Test data explained variance regression score: 0.0007437133798527951
-------------------------------------------------------------------------------------------------------------------
Train data R2 score: 0.9960616987568939
Test data R2 score: -1.5697063799255386
-------------------------------------------------------------------------------------------------------------------
Train data MGD:  0.0024885103449328147
Test data MGD:  0.17350006730154086
-------------------------------------------------------------------------------------------------------------------
Train data MPD:  1.514015931295652
Test data MPD:  461.7832964635234
-------------------------------------------------------------------------------------------------------------------
```

In [58]:
```
# shift train predictions for plotting
look_back=time_step
trainPredictPlot = np.empty_like(closedf)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# shift test predictions for plotting
testPredictPlot = np.empty_like(closedf)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(closedf)-1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

plotdf = pd.DataFrame({'Date': close_stock['Date'],'original_close': close_stock['Close'],'train_predicted_close': trainPredictPlot.reshape(1,-1)[0].tolist(),
                       'test_predicted_close': testPredictPlot.reshape(1,-1)[0].tolist()})

plt.figure(figsize=(20,5))
sns.lineplot(data=plotdf)
plt.title("Comparision between original close price vs predicted close price",fontsize=18)
```

```
Train predicted data:  (776, 1)
Test predicted data:  (776, 1)
```

Out[58]:
```
Text(0.5, 1.0, 'Comparision between original close price vs predicted close price')
```



In [59]:
```
# CReating array of last 15 days
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
from numpy import array

lst_output=[]
n_steps=time_step
i=0
pred_days = 10
while(i<pred_days):
    if(len(temp_input)>time_step):
        x_input=np.array(temp_input[1:])
        #print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)

        yhat = neighbor.predict(x_input)
        #print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat.tolist())
        temp_input=temp_input[1:]

        lst_output.extend(yhat.tolist())
        i=i+1

    else:
        yhat = neighbor.predict(x_input)

        temp_input.extend(yhat.tolist())
        lst_output.extend(yhat.tolist())

        i=i+1

print("Output of predicted next days: ", len(lst_output))
print("Output of predicted next days: ",lst_output)
```

```
Output of predicted next days:  10
Output of predicted next days:  [0.5169007351729139, 0.5091921339794941, 0.5165702370355717, 0.5310561105067088, 0.5291250453301684, 0.5285167968878779, 0.5258794052681898, 0.5088913064978736, 0.5146877987670194, 0.5191095506544028]
```

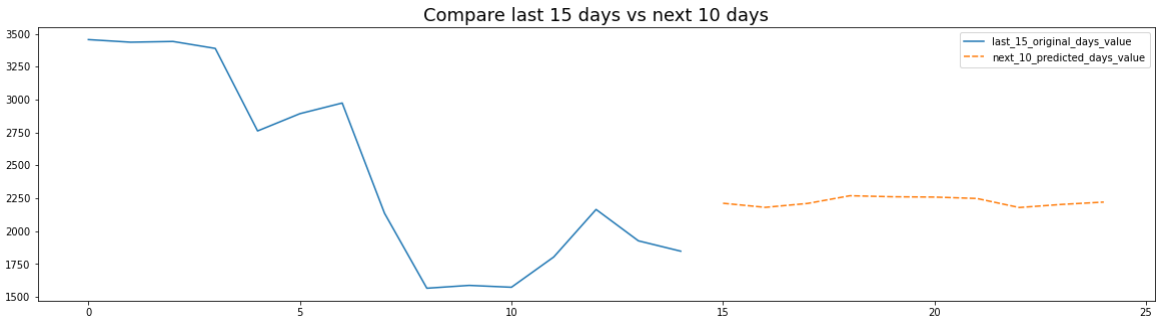
In [60]:
```
# Creating list of days to use for comparision
last_days=np.arange(1,time_step+1)
day_pred=np.arange(time_step+1,time_step+pred_days+1)
print(last_days)
print(day_pred)

temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat
next_predicted_days_value = temp_mat
len(last_original_days_value)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25]
```

Out[60]:
```
25
```

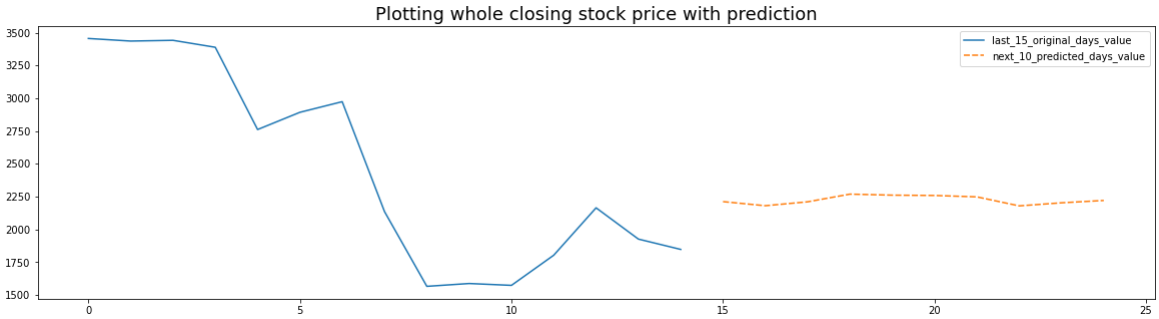
In [61]:
```
#  Creating empty dataframe for ploting graph
temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat.copy()
next_predicted_days_value = temp_mat.copy()

# Storing last 15 days and next 10 days values for comparision
last_original_days_value[0:time_step] = scaler.inverse_transform(closedf[len(closedf)-time_step:]).reshape(1,-1).tolist()[0]
next_predicted_days_value[time_step:] = scaler.inverse_transform(np.array(lst_output).reshape(-1,1)).reshape(1,-1).tolist()[0]
new_pred_plot = pd.DataFrame({
    'last_15_original_days_value':last_original_days_value,
    'next_10_predicted_days_value':next_predicted_days_value
})

plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Compare last 15 days vs next 10 days",fontsize=18)
```

Out[61]:
```
Text(0.5, 1.0, 'Compare last 15 days vs next 10 days')
```



In [62]:
```
# Creating list of close price
knndf=closedf.tolist()
# Adding next 10 days prediction
knndf.extend((np.array(lst_output).reshape(-1,1)).tolist())
# Tranforming to iriginal values
knndf=scaler.inverse_transform(knndf).reshape(1,-1).tolist()[0]

plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Plotting whole closing stock price with prediction",fontsize=18)
```

Out[62]:
```
Text(0.5, 1.0, 'Plotting whole closing stock price with prediction')
```



Long short-term memory (LSTM)

- LSTM (Long Short-Term Memory) is a deep learning model that is commonly used in the field of natural language processing,time series forecasting, speech recognition, and image classification. It is a type of recurrent neural network (RNN) that is designed to address the vanishing gradient problem, which occurs when RNNs are unable to effectively propagate error gradients over time.
- LSTM has the ability to selectively remember or forget previous inputs based on their relevance to the current output, allowing it to maintain long-term dependencies in sequential data. It consists of a memory cell, an input gate, an output gate, and a forget gate, each of which controls the flow of information within the network.
- The LSTM model has proven to be effective in a variety of applications, including language translation, sentiment analysis, and speech recognition. It has also been used in predictive maintenance, financial forecasting, and other areas that involve time-series data.
- Overall, LSTM is a powerful deep learning model that is well-suited for tasks that involve sequential data and long-term dependencies.

In [63]:
```
# reshape input to be [samples, time steps, features] which is required for LSTM
# LSTM requies 3-dimensional input
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)

print("X_train: ", X_train.shape)
print("X_test: ", X_test.shape)
```

```
X_train:  (566, 15, 1)
X_test:  (178, 15, 1)
```

Sequential Model

- In deep learning, a sequential model is a type of **neural network architecture** in which the layers are arranged sequentially, or one after the other, **with no branching**. This means that the output of each layer is fed as input to the next layer, in a chain-like sequence.
- The sequential model is a popular choice for many deep learning applications, as it is simple and easy to use. However, it may not be appropriate for all types of problems, such as those that require more complex network architectures with multiple inputs or outputs. In those cases, other types of neural network architectures, such as recurrent neural networks or convolutional neural networks, may be more appropriate.

1.Input shape: This parameter defines the shape of the input data that is fed to the LSTM network. In Keras, the input shape is specified as a tuple that contains the number of time steps in the sequence and the number of input features in each time step.

2.Number of hidden units: This parameter determines the number of hidden units in the LSTM layer. More hidden units generally allow the network to capture more complex patterns in the input data, but also require more computational resources.

3.Activation function: This parameter determines the activation function used in the LSTM layer. The most commonly used activation functions in LSTM are the hyperbolic tangent (tanh) function and the sigmoid function. The activation function is used to control the output of the LSTM cell and can help to prevent vanishing or exploding gradients during training.

4.Dropout: This parameter is used to prevent overfitting in the LSTM network by randomly dropping out some of the LSTM units during training.

5.Recurrent dropout: This parameter is similar to the dropout parameter, but is used to randomly drop out some of the connections between the LSTM cells during training.

6.Number of time steps: This parameter determines the number of time steps in the input sequence. In many applications, the number of time steps is fixed, but it can also be variable.

7.Batch size: This parameter determines the number of samples in each training batch. A larger batch size can help to speed up training, but can also require more memory.

8.Learning rate: This parameter determines the step size used in the optimization algorithm during training. A smaller learning rate can help to prevent overshooting the optimal solution, but can also make the training process slower.

In [64]:
```
tf.keras.backend.clear_session()
model=Sequential()
model.add(LSTM(32,return_sequences=True,input_shape=(time_step,1)))
model.add(LSTM(32,return_sequences=True))
model.add(LSTM(32))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

In [65]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 15, 32)            4352

 lstm_1 (LSTM)               (None, 15, 32)            8320

 lstm_2 (LSTM)               (None, 32)                8320

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 21,025
Trainable params: 21,025
Non-trainable params: 0
_________________________________________________________________
```

Epochs

An epoch is a complete pass through the training data during the training of a model. During each epoch, the model processes every sample in the training dataset, updates its parameters (weights and biases), and tries to minimize the loss function.

In [66]:
```
model.fit(X_train,y_train,validation_data=(X_test,y_test),epochs=10,batch_size=5,verbose=1)
```

```
Epoch 1/10
114/114 [==============================] - 11s 39ms/step - loss: 0.0039 - val_loss: 0.0108
Epoch 2/10
114/114 [==============================] - 3s 28ms/step - loss: 5.2446e-04 - val_loss: 0.0103
Epoch 3/10
114/114 [==============================] - 3s 24ms/step - loss: 4.4817e-04 - val_loss: 0.0101
Epoch 4/10
114/114 [==============================] - 3s 24ms/step - loss: 4.8426e-04 - val_loss: 0.0086
Epoch 5/10
114/114 [==============================] - 3s 25ms/step - loss: 3.9042e-04 - val_loss: 0.0106
Epoch 6/10
114/114 [==============================] - 3s 30ms/step - loss: 4.2133e-04 - val_loss: 0.0107
Epoch 7/10
114/114 [==============================] - 3s 24ms/step - loss: 4.7043e-04 - val_loss: 0.0130
Epoch 8/10
114/114 [==============================] - 3s 24ms/step - loss: 4.2426e-04 - val_loss: 0.0180
Epoch 9/10
114/114 [==============================] - 3s 25ms/step - loss: 4.1399e-04 - val_loss: 0.0103
Epoch 10/10
114/114 [==============================] - 4s 31ms/step - loss: 4.0220e-04 - val_loss: 0.0068
```

Out[66]:
```
<keras.callbacks.History at 0x7f84c0110af0>
```

In [67]:
```
### Lets Do the prediction and check performance metrics
train_predict=model.predict(X_train)
test_predict=model.predict(X_test)
train_predict.shape, test_predict.shape
```

```
18/18 [==============================] - 1s 8ms/step
6/6 [==============================] - 0s 7ms/step
```

Out[67]:
```
((566, 1), (178, 1))
```

In [68]:
```
# Transform back to original form
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))
```

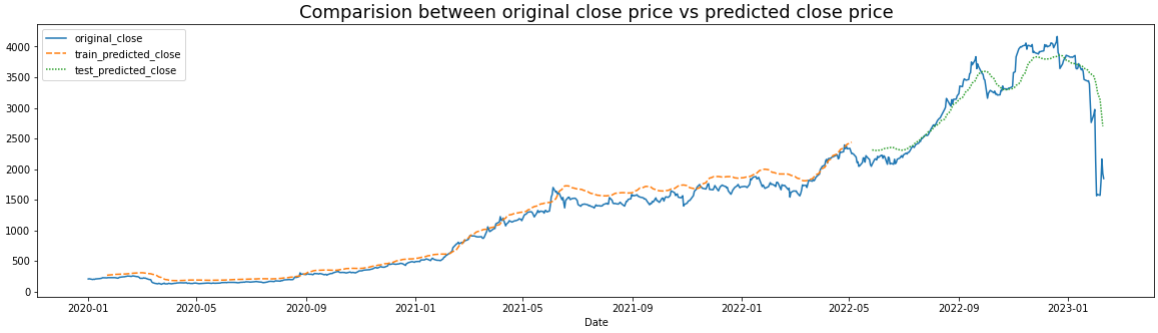
In [69]:
```
# Evaluation metrices RMSE and MAE
print("-------------------------------------------------------------------------------------")
print("Train data RMSE: ", np.sqrt(mean_squared_error(original_ytrain,train_predict)))
print("Train data MSE: ", mean_squared_error(original_ytrain,train_predict))
print("Test data MAE: ", mean_absolute_error(original_ytrain,train_predict))
print("-------------------------------------------------------------------------------------")
print("Test data RMSE: ", np.sqrt(mean_squared_error(original_ytest,test_predict)))
print("Test data MSE: ", mean_squared_error(original_ytest,test_predict))
print("Test data MAE: ", mean_absolute_error(original_ytest,test_predict))
print("-------------------------------------------------------------------------------------")
print("Train data explained variance regression score:", explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:", explained_variance_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------")
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------")
print("Train data MGD: ", mean_gamma_deviance(original_ytrain, train_predict))
print("Test data MGD: ", mean_gamma_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------")
print("Train data MPD: ", mean_poisson_deviance(original_ytrain, train_predict))
print("Test data MPD: ", mean_poisson_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------")
```

```
-------------------------------------------------------------------------------------
Train data RMSE:  118.02044291952107
Train data MSE:  13928.824946919933
Test data MAE:  93.6089766323777
-------------------------------------------------------------------------------------
Test data RMSE:  333.706895137686
Test data MSE:  111360.29186243455
Test data MAE:  201.18580445707516
-------------------------------------------------------------------------------------
Train data explained variance regression score: 0.9858399671258268
Test data explained variance regression score: 0.7722996888233522
-------------------------------------------------------------------------------------
Train data R2 score: 0.9693169693758119
Test data R2 score: 0.7708800075306073
-------------------------------------------------------------------------------------
Train data MGD:  0.028975330624765604
Test data MGD:  0.014142457048322529
-------------------------------------------------------------------------------------
Train data MPD:  13.180303195705264
Test data MPD:  38.962020359061746
-------------------------------------------------------------------------------------
```

In [70]:
```
# shift train predictions for plotting
look_back=time_step
trainPredictPlot = np.empty_like(closedf)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# shift test predictions for plotting
testPredictPlot = np.empty_like(closedf)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(closedf)-1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

plotdf = pd.DataFrame({'Date': close_stock['Date'],'original_close': close_stock['Close'],'train_predicted_close': trainPredictPlot.reshape(1,-1)[0].tolist(),
                       'test_predicted_close': testPredictPlot.reshape(1,-1)[0].tolist()})

plt.figure(figsize=(20,5))
sns.lineplot(data=plotdf)
plt.title("Comparision between original close price vs predicted close price",fontsize=18)
```

```
Train predicted data:  (776, 1)
Test predicted data:  (776, 1)
```

Out[70]:
```
Text(0.5, 1.0, 'Comparision between original close price vs predicted close price')
```



In [71]:
```
# CReating array of last 15 days
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
from numpy import array
lst_output=[]
n_steps=time_step
i=0
pred_days = 10
while(i<pred_days):
    if(len(temp_input)>time_step):
        x_input=np.array(temp_input[1:])
        #print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)
        yhat = model.predict(np.expand_dims(x_input, 2))
        #print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat[0])
        temp_input=temp_input[1:]

        lst_output.extend(yhat.tolist())
        i=i+1

    else:
        yhat = model.predict(np.expand_dims(x_input, 2))
        temp_input.extend(yhat[0])
        lst_output.extend(yhat.tolist())

        i=i+1

print("Output of predicted next days: ", len(lst_output))
print("Output of predicted next days: ",lst_output)
```

```
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 32ms/step
Output of predicted next days:  10
Output of predicted next days:  [[0.5785232782363892], [0.5568419694900513], [0.5415095090866089], [0.5315434336662292], [0.5260884165763855], [0.5271589159965515], [0.
528874397277832], [0.5310568809509277], [0.538675844669342], [0.5489208698272705]]
```

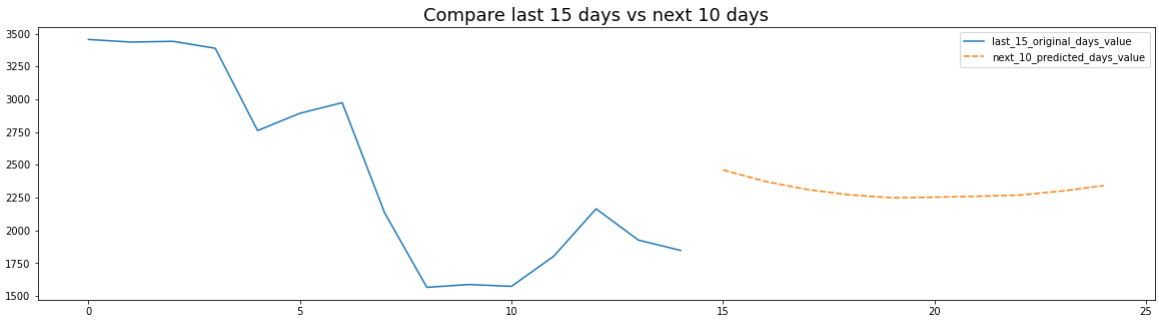
In [72]:
```
# Creating list of days to use for comparision
last_days=np.arange(1,time_step+1)
day_pred=np.arange(time_step+1,time_step+pred_days+1)
print(last_days)
print(day_pred)

temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat
next_predicted_days_value = temp_mat
len(last_original_days_value)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25]
```

Out[72]:
```
25
```

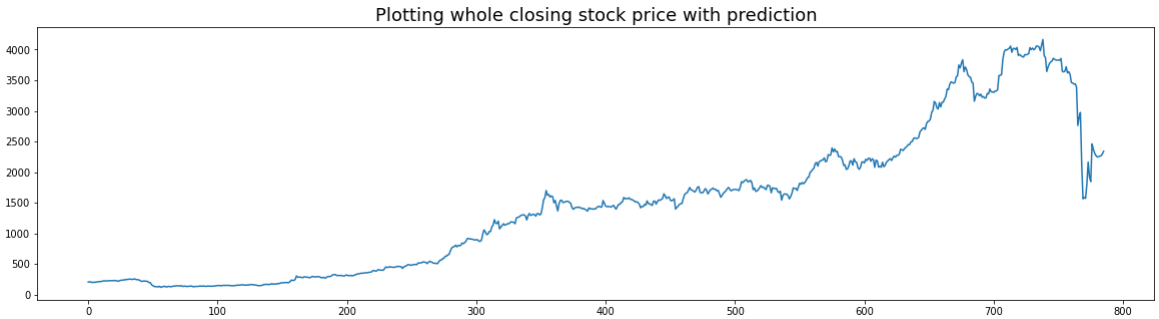
In [73]:
```
#  Creating empty dataframe for ploting graph
temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat.copy()
next_predicted_days_value = temp_mat.copy()

# Storing last 15 days and next 10 days values for comparision
last_original_days_value[0:time_step] = scaler.inverse_transform(closedf[len(closedf)-time_step:]).reshape(1,-1).tolist()[0]
next_predicted_days_value[time_step:] = scaler.inverse_transform(np.array(lst_output).reshape(-1,1)).reshape(1,-1).tolist()[0]
new_pred_plot = pd.DataFrame({
    'last_15_original_days_value':last_original_days_value,
    'next_10_predicted_days_value':next_predicted_days_value
})

plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Compare last 15 days vs next 10 days",fontsize=18)
```

Out[73]: Text(0.5, 1.0, 'Compare last 15 days vs next 10 days')



In [74]:
```
# Creating list of close price
lstmdf=closedf.tolist()
# Adding next 10 days prediction
lstmdf.extend((np.array(lst_output).reshape(-1,1)).tolist())
# Tranforming to iriginal values
lstmdf=scaler.inverse_transform(lstmdf).reshape(1,-1).tolist()[0]

plt.figure(figsize=(20,5))
sns.lineplot(data=lstmdf)
plt.title("Plotting whole closing stock price with prediction",fontsize=18)
```

Out[74]: Text(0.5, 1.0, 'Plotting whole closing stock price with prediction')



Save and load model

Hierarchical Data Format version 5

The HDF5 (Hierarchical Data Format version 5), format is designed to store large and complex datasets, and it provides a way to store and organize data in a hierarchical structure. In the context of deep learning, the .h5 file contains the weights and architecture of the trained model, along with any other relevant information such as the optimizer state and the training configuration.

In [75]:
```
model.save('lstm_model.h5')
lstm_model = tf.keras.models.load_model('lstm_model.h5')
```

In [76]:
```
lstm_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 15, 32)            4352

 lstm_1 (LSTM)               (None, 15, 32)            8320

 lstm_2 (LSTM)               (None, 32)                8320

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 21,025
Trainable params: 21,025
Non-trainable params: 0
_________________________________________________________________
```

GRU

- The GRU (Gated Recurrent Unit) Deep Learning model is a type of artificial neural network that is used for sequence data processing.
- This model is similar to the LSTM (Long Short-Term Memory) model and is designed to overcome some of the issues associated with LSTM.
- GRU networks have fewer parameters than LSTM networks, which makes them faster to train and more memory-efficient.
- The main difference between the two models is that GRU has fewer gates to control the flow of information and therefore, it is less prone to overfitting.
- The GRU model is widely used in various applications, such as natural language processing, speech recognition, image recognition, and music analysis.
- It is capable of learning long-term dependencies in the data and can model sequences with arbitrary lengths. In conclusion, the GRU Deep Learning model is a powerful tool for processing sequence data and has proven to be effective in various real-world applications.

In [77]:
```
# reshape input to be [samples, time steps, features] which is required for LSTM
#  GRU requires 3-Dimensional input
X_train =X_train.reshape(X_train.shape[0],X_train.shape[1] , 1)
X_test = X_test.reshape(X_test.shape[0],X_test.shape[1] , 1)

print("X_train: ", X_train.shape)
print("X_test: ", X_test.shape)
```

```
X_train:  (566, 15, 1)
X_test:  (178, 15, 1)
```

In [78]:
```
tf.keras.backend.clear_session()
model=Sequential()
model.add(GRU(32,return_sequences=True,input_shape=(time_step,1)))
model.add(GRU(32,return_sequences=True))
model.add(GRU(32,return_sequences=True))
model.add(GRU(32))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

In [79]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 15, 32)            3360

 gru_1 (GRU)                 (None, 15, 32)            6336

 gru_2 (GRU)                 (None, 15, 32)            6336

 gru_3 (GRU)                 (None, 32)                6336

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 22,401
Trainable params: 22,401
Non-trainable params: 0
_________________________________________________________________
```

In [80]:
```
model.fit(X_train,y_train,validation_data=(X_test,y_test),epochs=10,batch_size=5,verbose=1)
```

```
Epoch 1/10
114/114 [==============================] - 15s 48ms/step - loss: 0.0028 - val_loss: 0.0051
Epoch 2/10
114/114 [==============================] - 5s 41ms/step - loss: 3.1427e-04 - val_loss: 0.0075
Epoch 3/10
114/114 [==============================] - 4s 34ms/step - loss: 3.0530e-04 - val_loss: 0.0044
Epoch 4/10
114/114 [==============================] - 4s 34ms/step - loss: 2.9781e-04 - val_loss: 0.0036
Epoch 5/10
114/114 [==============================] - 5s 41ms/step - loss: 2.6310e-04 - val_loss: 0.0056
Epoch 6/10
114/114 [==============================] - 4s 34ms/step - loss: 3.2631e-04 - val_loss: 0.0040
Epoch 7/10
114/114 [==============================] - 4s 35ms/step - loss: 2.5648e-04 - val_loss: 0.0032
Epoch 8/10
114/114 [==============================] - 5s 39ms/step - loss: 2.3246e-04 - val_loss: 0.0063
Epoch 9/10
114/114 [==============================] - 4s 34ms/step - loss: 3.3626e-04 - val_loss: 0.0062
Epoch 10/10
114/114 [==============================] - 4s 37ms/step - loss: 2.1153e-04 - val_loss: 0.0036
```

Out[80]:
```
<keras.callbacks.History at 0x7f84bbca4100>
```

In [81]:
```
### Lets Do the prediction and check performance metrics
train_predict=model.predict(X_train)
test_predict=model.predict(X_test)
train_predict.shape, test_predict.shape
```

```
18/18 [==============================] - 2s 9ms/step
6/6 [==============================] - 0s 9ms/step
```

Out[81]:
```
((566, 1), (178, 1))
```

In [82]:
```
# Transform back to original form
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
original_ytrain = scaler.inverse_transform(y_train.reshape(-1,1))
original_ytest = scaler.inverse_transform(y_test.reshape(-1,1))
```

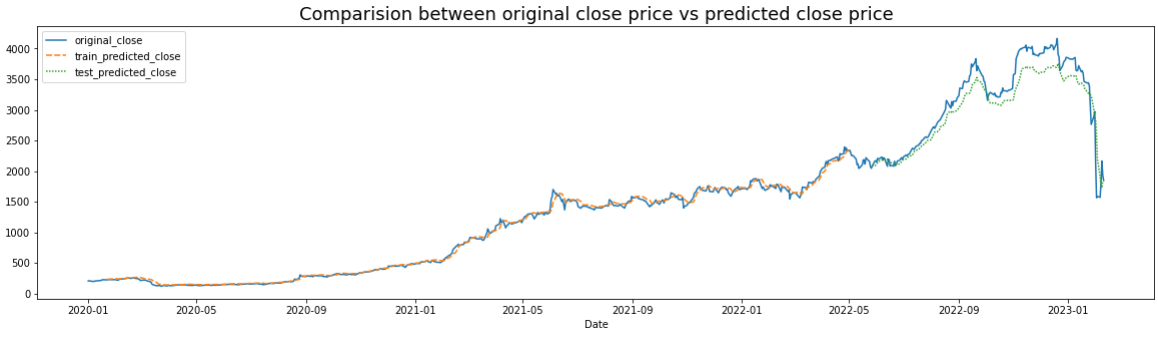
In [83]:
```
# Evaluation metrices RMSE and MAE
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data RMSE: ", np.sqrt(mean_squared_error(original_ytrain,train_predict)))
print("Train data MSE: ", mean_squared_error(original_ytrain,train_predict))
print("Test data MAE: ", mean_absolute_error(original_ytrain,train_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Test data RMSE: ", np.sqrt(mean_squared_error(original_ytest,test_predict)))
print("Test data MSE: ", mean_squared_error(original_ytest,test_predict))
print("Test data MAE: ", mean_absolute_error(original_ytest,test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data explained variance regression score:", explained_variance_score(original_ytrain, train_predict))
print("Test data explained variance regression score:", explained_variance_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data R2 score:", r2_score(original_ytrain, train_predict))
print("Test data R2 score:", r2_score(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MGD: ", mean_gamma_deviance(original_ytrain, train_predict))
print("Test data MGD: ", mean_gamma_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
print("Train data MPD: ", mean_poisson_deviance(original_ytrain, train_predict))
print("Test data MPD: ", mean_poisson_deviance(original_ytest, test_predict))
print("-------------------------------------------------------------------------------------------------------------------")
```

```
-------------------------------------------------------------------------------------------------------------------
Train data RMSE:  42.76864755146723
Train data MSE:  1829.1572133816244
Test data MAE:  28.871343875521077
-------------------------------------------------------------------------------------------------------------------
Test data RMSE:  241.50743266914483
Test data MSE:  58325.84003444153
Test data MAE:  194.81931275785658
-------------------------------------------------------------------------------------------------------------------
Train data explained variance regression score: 0.9960557677725231
Test data explained variance regression score: 0.928065168821531
-------------------------------------------------------------------------------------------------------------------
Train data R2 score: 0.9959706517234209
Test data R2 score: 0.8799965786191493
-------------------------------------------------------------------------------------------------------------------
Train data MGD:  0.0038684865334660257
Test data MGD:  0.0065364554675948405
-------------------------------------------------------------------------------------------------------------------
Train data MPD:  1.7596534333657043
Test data MPD:  18.928285633949322
-------------------------------------------------------------------------------------------------------------------
```

In [84]:
```
# shift train predictions for plotting
look_back=time_step
trainPredictPlot = np.empty_like(closedf)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(train_predict)+look_back, :] = train_predict
print("Train predicted data: ", trainPredictPlot.shape)

# shift test predictions for plotting
testPredictPlot = np.empty_like(closedf)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(closedf)-1, :] = test_predict
print("Test predicted data: ", testPredictPlot.shape)

plotdf = pd.DataFrame({'Date': close_stock['Date'],'original_close': close_stock['Close'],'train_predicted_close': trainPredictPlot.reshape(1,-1)[0].tolist(),
                       'test_predicted_close': testPredictPlot.reshape(1,-1)[0].tolist()})

plt.figure(figsize=(20,5))
sns.lineplot(data=plotdf)
plt.title("Comparision between original close price vs predicted close price",fontsize=18)
```

```
Train predicted data:  (776, 1)
Test predicted data:  (776, 1)
```

Out[84]:
```
Text(0.5, 1.0, 'Comparision between original close price vs predicted close price')
```



In [85]:
```
# CReating array of last 15 days
x_input=test_data[len(test_data)-time_step:].reshape(1,-1)
temp_input=list(x_input)
temp_input=temp_input[0].tolist()
from numpy import array

lst_output=[]
n_steps=time_step
i=0
pred_days = 10
while(i<pred_days):
    if(len(temp_input)>time_step):
        x_input=np.array(temp_input[1:])
        #print("{} day input {}".format(i,x_input))
        x_input=x_input.reshape(1,-1)

        yhat = model.predict(np.expand_dims(x_input, 2))
        #print("{} day output {}".format(i,yhat))
        temp_input.extend(yhat[0])
        temp_input=temp_input[1:]

        lst_output.extend(yhat.tolist())
        i=i+1

    else:
        yhat = model.predict(np.expand_dims(x_input, 2))

        temp_input.extend(yhat[0])
        lst_output.extend(yhat.tolist())

        i=i+1

print("Output of predicted next days: ", len(lst_output))
print("Output of predicted next days: ",lst_output)
```

```
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 28ms/step
Output of predicted next days:  10
Output of predicted next days:  [[0.44352987408638], [0.4430696666240692], [0.44347307085990906], [0.4441938102245331], [0.4449681341648102], [0.44572314620018005], [0.4464779496192932], [0.4472326934337616], [0.44795235991477966], [0.448639452457428]]
```

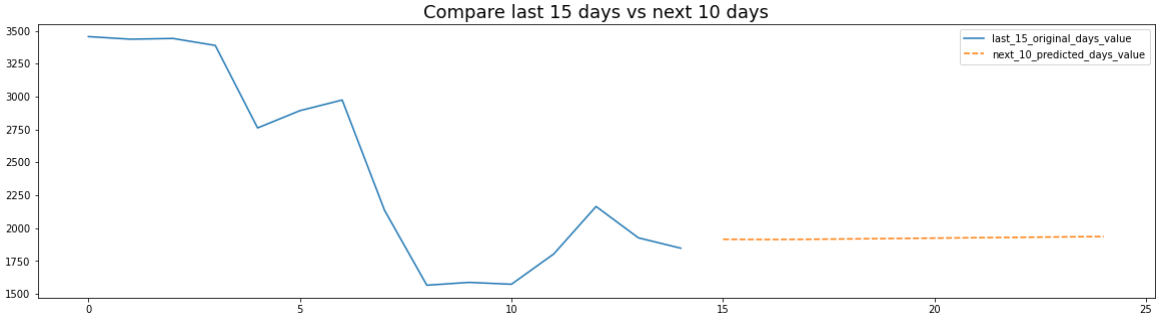
In [86]:
```
# Creating list of days to use for comparision
last_days=np.arange(1,time_step+1)
day_pred=np.arange(time_step+1,time_step+pred_days+1)
print(last_days)
print(day_pred)

temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat
next_predicted_days_value = temp_mat
len(last_original_days_value)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25]
```

Out[86]:
```
25
```

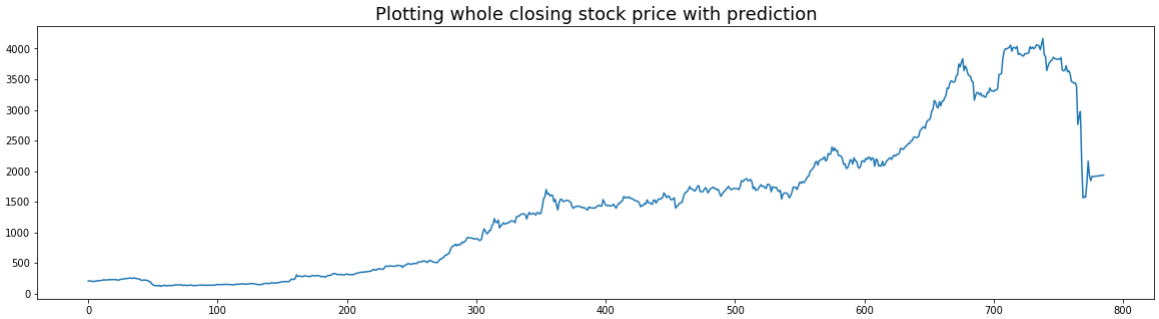
In [87]:
```
#  Creating empty dataframe for ploting graph
temp_mat = np.empty((len(last_days)+pred_days,1))
temp_mat[:] = np.nan
temp_mat = temp_mat.reshape(1,-1).tolist()[0]

last_original_days_value = temp_mat.copy()
next_predicted_days_value = temp_mat.copy()

# Storing last 15 days and next 10 days values for comparision
last_original_days_value[0:time_step] = scaler.inverse_transform(closedf[len(closedf)-time_step:]).reshape(1,-1).tolist()[0]
next_predicted_days_value[time_step:] = scaler.inverse_transform(np.array(lst_output).reshape(-1,1)).reshape(1,-1).tolist()[0]
new_pred_plot = pd.DataFrame({
    'last_15_original_days_value':last_original_days_value,
    'next_10_predicted_days_value':next_predicted_days_value
})

plt.figure(figsize=(20,5))
sns.lineplot(data=new_pred_plot)
plt.title("Compare last 15 days vs next 10 days",fontsize=18)
```

Out[87]:
```
Text(0.5, 1.0, 'Compare last 15 days vs next 10 days')
```



In [88]:
```
# Creating list of close price
grudf=closedf.tolist()
# Adding next 10 days prediction
grudf.extend((np.array(lst_output).reshape(-1,1)).tolist())
# Tranforming to iriginal values
grudf=scaler.inverse_transform(grudf).reshape(1,-1).tolist()[0]

plt.figure(figsize=(20,5))
sns.lineplot(data=grudf)
plt.title("Plotting whole closing stock price with prediction",fontsize=18)
```

Out[88]:
```
Text(0.5, 1.0, 'Plotting whole closing stock price with prediction')
```



In [89]:
```
model.save('gru_model.h5')
gru_model = tf.keras.models.load_model('gru_model.h5')
```

In [90]:
```
gru_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 15, 32)            3360

 gru_1 (GRU)                 (None, 15, 32)            6336

 gru_2 (GRU)                 (None, 15, 32)            6336

 gru_3 (GRU)                 (None, 32)                6336

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 22,401
Trainable params: 22,401
Non-trainable params: 0
_________________________________________________________________
```

Yahoo finance github link and syntax:

- https://github.com/ranaroussi/yfinance (https://github.com/ranaroussi/yfinance)

```
import yfinance as yf

adanient = yf.Ticker("ADANIENT.NS")

# get all stock info (slow)
adanient.info
# fast access to subset of stock info (opportunistic)
# msft.fast_info

# get historical market data
# hist = msft.history(period="1mo")

# show meta information about the history (requires history() to be called first)
# msft.history_metadata

# show actions (dividends, splits, capital gains)
# msft.actions
# msft.dividends
# msft.splits
# msft.capital_gains  # only for mutual funds & etfs

# show share count
# - yearly summary:
# msft.shares
# - accurate time-series count:
# msft.get_shares_full(start="2022-01-01", end=None)

# show financials:
# - income statement
# msft.income_stmt
# msft.quarterly_income_stmt
# - balance sheet
# msft.balance_sheet
# msft.quarterly_balance_sheet
```

THE END