import pandas as pd

import numpy as np

import pymongo

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

plt.style.use('seaborn-darkgrid')

from sklearn.linear\_model import LinearRegression

myclient = pymongo.MongoClient("mongodb://localhost:27017")

mydb = myclient["golddata"]

mycol= mydb["golddata"]

value = mycol.find()

list\_deger = list(value)

df = pd.DataFrame(list\_deger)

Explanation: In this code snippet, you are first importing the pandas, numpy, pymongo, seaborn and matplotlib libraries. Then you specify that the graphics can be drawn in the Jupyter Notebook environment with the '%matplotlib inline' command and adjust the appearance of the graphics using the 'seaborn-darkgrid' template. Next, you import the Linear Regression class from the sklearn library.

Next, you create a MongoClient object using the pymongo library to connect to the MongoDB database. You are connecting to a database that has the database name of this object "golddata" and has a collection named "golddata" in the database. Then, you assign a 'value' to the variable using the 'find()' method to retrieve all the data in the collection and convert this variable to a list shape. Finally, you convert this list to a pandas DataFrame object and assign this object to the variable 'df'.

df.set\_index("Date", input=True)

In this line of code, you are setting the 'Date' column of the pandas DataFrame object as an index. with the parameter 'inplace=True', you are also applying this operation to the DataFrame object 'df' itself, that is, the change is made to the object 'df' itself. As a result of this operation, the 'Date' column will now be the index of the DATAFRAME and will not be the first column of the DATAFRAME. For example, if the headings of your DATAFRAME are as follows: "Date, Open, High, Low, Close, Volume", then as a result of this operation, the headings of the DATAFRAME will be as follows: "Open, High, Low, Close, Volume" and the "Date" column will be used as an index.

#We remove the columns that are not related and dropna()

We are lowering the NaN values using the # function.

#Then we draw the closing price of the Gold ETF.

df = df[['Close']]

# Drop rows with missing values

df = df.dropna()

# Plot the closing price of GLD

df.Close.plot(figsize=(10, 7),color='r')

plt.ylabel("The Price of Gold")

plt.title("Gold Price Series")

plt.show()

In this block of code, first of all, you select only the 'Close' column from inside the 'df' DataFrame object. Then, you lower the NaN values in the DataFrame object using the 'dropna()' method. This method extracts lines with NaN values from the DataFrame object.

Then, the values of the 'Close' column are 'df.Close.you are making a graphic with the plot() method. you determine the graphic size with the 'figsize' parameter and you determine the graphic line color with the 'color' parameter. 'plt.ylabel()' and 'plt.with the title()' functions, you specify the labels of the graphic axes and the graphic title. Finally, 'plt.you are showing the graph with the show() function.

This block of code shows the values in the 'Close' column in a chart under the name 'Gold Price Series' and labels the y-axis of the chart as 'Gold Price'.

#Define explanatory variables

#Explanatory variable, to determine the value of the Gold ETF price the next day

#is a manipulated variable. Simply, to estimate the Gold ETF price

#these are the features we want to use.

#The explanatory variables in this strategy are the activity of the last 3 days and 9 days

#are the averages.Using the dropna() function, the NaN values are

#we lower it and store the property variables in X.

#But what do you think is useful for predicting Gold ETF prices to X

#you can add more variables. These variables are technical indicators,

#Gold miners ETF (GDX) or Oil ETF (USO)

it could be the price of another ETF, such as #, or US economic data.

#Define the dependent variable

#Similarly, the dependent variable depends on the values of the explanatory variables.

#Simply put, it is the Gold ETF price that we are trying to predict.

#We store the gold ETF price in y.

df['S\_3'] = df['Close'].rolling(window=3).mean()

df['S\_9'] = df['Close'].rolling(window=9).mean()

df['next day'] = df['Close'].shift(-1)

df = df.dropna()

X = df[['S\_3', 'S\_9']]

# Define dependent variable

y = df['next day\_f']

In this code block, you create explanatory variables to predict the Gold ETF price. First of all, you create the 3-day and 9-day moving averages of the 'Close' column. You are adding these moving averages to the 'S\_3' and 'S\_9' columns. Then, you add the values of the values in the 'Close' column for the next day to the 'next day' column. You are using the 'shift(-1)' method to create the values of this column. This method returns the value of each value for the next day.

Then, you lower the NaN values using the 'dropna()' method and select the columns 'S\_3' and 'S\_9' for the variable 'X'. These columns will be your explanatory variables. Then, as your dependent variable, you assign the column 'next day\_f' to the variable 'y'.

With this code block, you will have created your explanatory variables and your dependent variable. Using these variables, you can proceed to the process of estimating the values of the Gold ETF price for the next day.

#Separate the data into training and test dataset

#In this step, we separate the predictors and output data into training and test data. The training data is used to create a linear regression model by matching the input with the expected output.

#Test data is used to estimate how well the model is trained.

#The first 80% of the data is used for training, and the rest of the data is used for testing.

#X\_train and y\_train are training datasets

#X\_test and y\_test are the test dataset

# Split the data into train and test dataset

t = .8

t=int(t\*len(df))

# Train dataset

X\_train = X[:t]

y\_train = y[:t]

# Test dataset

X\_test = X[t:]

y\_test = y[t:]

In this code block, you are separating your explanatory variables and your dependent variable into training and test datasets. This separation process assigns the first 80% of your dataset to the training dataset and the remaining 20% to the test dataset. You are doing this separation according to the value you have assigned to the variable 't'.

Then, you assign the training dataset to the 'X\_train' and 'y\_train' variables, and the test dataset to the 'X\_test' and 'y\_test' variables. The variables 'x\_train' and 'y\_train' will hold your training data. Using this data, you will be able to create a linear regression model. The variables 'x\_test' and 'y\_test' will hold the test data that you will use to predict how well your model is trained.

#Creating a linear regression model

#Now we will create a linear regression model.

#However, what is linear regression?

#between the variables 'x' and 'y', Decode the observed values of 'y'

#a mathematical relationship that describes the "best" in terms of the observed values of 'x',

#if we try to catch it by placing a line from a distribution graph,

#then such an equation is called a linear regression analysis between x and Y. Dec.

#To break it down further, the regression on a dependent variable

#explains the change in terms of independent variables.

#Dependent variable - 'y' is the variable you want to predict.

#Independent variables - 'x' to estimate the dependent variable

#are the explanatory variables that you use. The following regression equation

#this explains the relationship:

#Then, to create a coefficient and constant for regression

#to fit independent and dependent variables (x's and y's)

we use the #fit method.

# Creating a linear regression model

linear = LinearRegression().fit(X\_train,y\_train)

print("Linear Regression model")

print("Gold Price (y) =%.2f \* 3-Day Moving Average (x1) \

+ %.2f \* 9-Day Moving Average (x2) \

+ %.2f (constant)" % (linear.coef\_[0], linear.coef\_[1], linear.intercept\_))

In this code block, you are creating a linear regression model. Linear regression describes the change of a dependent variable in terms of independent variables. For example, when you want to predict the value of the Gold ETF price for the next day, your dependent variable is the Gold ETF price, and your explanatory variables are the characteristics that you use to predict the Gold ETF price.

To create a linear regression model, you are using the 'LinearRegression' class. Using the 'fit()' method of this class, you are fitting your training data into the 'X\_train' and 'y\_train' variables. the 'fit()' method automatically determines the coefficient and constant values to fit the data. These coefficients and constant values form the equation of the linear regression model.

Finally, you print the equation of the model on the screen. This equation is the formula that you will use to estimate the value of the Gold ETF price for the next day. This formula includes the 3-day and 9-day moving averages as the 'x1' and 'x2' variables, and the Gold ETF price as the 'y' variable. Thanks to this formula, 3 days and

#Predict Gold ETF prices

#Now it's time to check whether the model works in the test data set.

#Using the linear model created using the training data set

We are predicting the #Gold ETF prices. Estimation method,

#finds the Gold ETF price (y) for the given X explanatory variable.

# Predicting the Gold ETF prices

predicted\_price = linear.predict(X\_test)

predicted\_price = pd.DataFrame(

predicted\_price, index=y\_test.index, columns=['price'])

predicted\_price.plot(figsize=(10, 7))

y\_test.plot()

plt.legend(['Predicted Event', 'Actual Event'])

plt.ylabel("The Price of Gold")

plt.show()

#The graph shows the estimated and actual price of the Gold ETF.

In this code block, you are running the linear regression model on the test data set. You are doing this with the 'predict()' method. the 'predict()' method predicts the dependent variable (the value of the Gold ETF price for the next day) using the given explanatory variables.

Then, you assign the predicted prices to the 'predicted\_price' variable and convert this variable to a Pandas DATAFRAME. This DataFrame shows the estimated prices sorted by dates.

Finally, you show the estimated prices and the actual prices on a graph. In this graph, see the estimated prices as the blue line and the actual prices as the red line

Let's calculate the goodness of fit using the #score() function.

#R square

r2\_score = linear.score(X[t:], y[t:])\*100

float("{0:.2f}".format(r2\_score))

#As can be seen, the R-square value of the model is 99.35%.

#R-square is always Decently between 0% and 100%.

# a score close to 100% indicates that the model explains Gold ETF prices well.

In this code block, you are measuring the goodness of fit of the linear regression model. You are making this measurement with the 'score()' method. the 'score()' method measures the accuracy of the model by using the given explanatory variables and the dependent variable.

The metric you use for this measurement is the R-square value. The R-square measures how well a model explains the change of the dependent variable in terms of explanatory variables. The R-squared value is always between 0 and 100, and a score close to 100 indicates that the model describes the dependent variable Decently.

In this measurement, you see that your R-square value is 99.35%. This value is the model's Gold ETF f

#Drawing cumulative returns

#Cumulative returns of this strategy to analyze its performance

#let's calculate.

#1-The steps for calculating cumulative returns are as follows:

#2-Create the daily percentage change of the gold price

#3- The estimated price of the next day is from the estimated price of the current day

#create a trading signal represented by a "1" when it is in excess.

#Otherwise the position will not be taken

#4-Multiplying the daily percentage change by the trading signal, the strategy returns

#calculate.

#5-Finally, we will draw the cumulative return graph

gold = pd.DataFrame()

gold['price'] = df[t:]['Close']

gold['predicted\_price\_next\_day'] = predicted\_price

gold['actual\_price\_next\_day'] = y\_test

gold['gold\_returns'] = gold['price'].pct\_change().shift(-1)

gold['signal'] = np.where(gold.predicted\_price\_next\_day.shift(1) < gold.predicted\_price\_next\_day,1,0)

gold['strategy\_returns'] = gold.signal \*gold['gold\_returns']

((gold['strategy\_returns']+1).cumprod()).plot(figsize=(10,7),color='g')

plt.ylabel('Cumulative Returns')

plt.show()

In this code block, you are measuring the performance of a strategy that tries to predict the future prices of a Gold ETF using a linear regression model. You do this by drawing a graph showing the cumulative returns generated by the strategy.

First, you are creating a Pandas DATAFRAME named 'gold'. You are adding the 'price' and 'predicted\_price\_next\_day' columns to this DATAFRAME, which contain the daily prices and predicted prices of the Gold ETF. Then, you add the actual prices to the 'actual\_price\_next\_day' column as well.

Then, in the 'gold\_returns' column, you add data containing the daily percentage changes of the Gold ETF. These changes are calculated using the 'pct\_change()' method and shifted back one day using the 'shift(-1)' method. This transaction shows the ratio of the next day's price to today's price.

Finally

#Sharpe ratio, Nobel laureate William F. Developed by Sharpe

#and for investors to understand the return on an investment compared to the risk

#it is used to help. Per ratio, unit volatility or total risk

#it is the average return that exceeds the risk-free rate. Volatility, the volatility of an asset or portfolio

#it is a measure of price fluctuations

#The formula for calculating the Sharpe ratio is {R(p) – R(f)}/s(p).

#Here

#R(p): Portfolio return

#R(f): The risk-free rate of return

#s(p): Portfolio standard deviation

sharpe = gold['strategy\_returns'].mean()/gold['strategy\_returns'].std()\*(252\*\*0.5)

'The Sharpe Rate is %.2f'% (sharpe)

In this code block, you are calculating the Sharpe ratio of a strategy over the 'gold' DATAFRAME. The Sharpe ratio is a metric that measures the return on an investment compared to its risk. This ratio is the average return per unit volatility or total risk that exceeds the risk-free ratio.

To calculate the Sharpe ratio, you divide the average of the strategy returns by the standard deviation of the strategy returns. This way, you will have measured the relationship between the return on an investment and volatility. Dec. Then, taking this ratio as a 252-day year, you take the square root. This number shows how many days an investment needs to be made in order to get a return above bond interest rates during the year.

Finally, hesa

#The below output is based on the data points in the train dataset.

#Sometimes, it is difficult to understand why there is more weight to 3 days moving average compared to 9 days moving average.

#But this equation is created by what the machine learning model learned from the data.

#Gold ETF Price = 1.2 \* 3 Days Moving Average – 0.2 \* 9 Days Moving Average + 0.39

#Using this equation, you can predict the next day price of the Gold ETF and buy/sell if the next day price is significantly

#higher or lower than the previous day's price.

#To learn more about R2 score you can refer to the following link

#How to use this model to predict daily movements?

#Is it necessary to predict gold prices and buy GLD or

#to give a trading signal that we should not take a position

#you can use the code below.

#import datetime and get today's date

# Get the data rolling is used to get a moving average

import datetime as dt

current\_date = dt.datetime.now()

# Get the data

df['S\_3'] = df['Close'].rolling(window=3).mean()

df['S\_9'] = df['Close'].rolling(window=9).mean()

df = df.dropna()

# Forecast the price

df['predicted\_gold\_price'] = linear.predict(df[['S\_3', 'S\_9']])

df['signal'] = np.where(df.predicted\_gold\_price.shift(1) < df.predicted\_gold\_price,"Buy","No Position")

# Print the forecast

df.tail(1)[['signal','predicted\_gold\_price']].T

Explanatory variables are other variables that we try to explain the value of a variable. For example, these are the characteristics that we want to use to predict the Gold ETF price. The explanatory variables in this strategy are the moving averages of the last 3 days and 9 days. The dependent variable, on the other hand, is the variable that we are trying to predict depending on the values of the explanatory variables. For example, it is the Gold ETF price that we are trying to predict.

It is used to separate the data into training and test dataset, to separate the predictors and output data into training and test data. The training data is used to create a linear regression model by matching the input with the expected output. The test data, on the other hand, is used to estimate how well the model is trained. The first eighty percent of the data is for training, and the rest of the data is for testing

from statsmodels.tsa.api import Decryption

model = ARIMA(df['Close'],order=(0,0,0)) Decode = ARIMA(df['Close'],order =(0,0,0))

result = modelo.fit()

print(result.aic)

AIC (Akaike Information Criterion) is used as a model conformity measurement. It is a criterion used to measure the suitability of the model. A model with a low AIC value is considered a more suitable model. For this reason, the model with the lowest AIC value is selected. In this example, the AIC value is calculated for the ARIMA model.Dec.

model2= ARIMA(df['Close'], order=(0,1,0)) Decryption model2 = ARIMA(df['Close'],order=(0,1,0))

result2 = model2.fit()

print(result2.aic)

According to the Akaike information criterion (AIC), it measures the relative quality of statistical models for a given dataset. It is often used to select the best model by comparing the A Dec values between models. In this case, you are trying to fit two different ARIMA models into the "Close" column of the Dec data frame. The ARIMA order of the first model is (0,0,0), while the order of the second model is (0,1,0). Dec.

The AIC value of the first model is printed as a result of the first model fitting. If the AIC value of the second model is, the second model is printed as a result of fitting.

In general, a lower AIC value means a better model fitting. Therefore, if the second model has a lower AIC value than the first model, it can be considered a better fitting for the data.

model3 = ARIMA(df['Close'], order=(1,0,0)) model3 = ARIMA(df['Close'],Dec= (1,0,0))

result3 = model3.fit()

print(result3.aic)

The Akaike information criterion (AIC) measures the relative quality of statistical models for a dataset. By comparing the A Dec values, it aims to select the best model from among the models. In this case, you are fitting an ARIMA model to the Dec data frame containing the "Close" column (1,0,0) respectively. The AIC value of this model is printed as a result of model fitting.

In general, a low AIC value indicates a better model fitting. Therefore, if this model has a lower AIC value than the previous two models, it will be a better fit for the data.

model4 = ARIMA(df['Close'], order=(1,1,0)) Decryption model4 = ARIMA(df['Close'], order =(1,1,0))

result4 = model4.fit()

print(result4.aic)

This code snippet uses the ARIMA model to Decode the dataset 'Close' variable. The ARIMA (AutoRegressive Integrated Moving Average) model is a regulatory prediction model on a time series Deciency. This model relies on historical data to make predictions for a time series of data and is therefore called "auto-regressive". The ranking of this model is indicated as (1,1,0). This shows that the auto-regressive components of the model are of the first order and the moving average components are of the zeroth order.

the fit() method of the ARIMA model assigned to the model4 variable is called and the result is assigned to the result4 variable.Dec. Later, result4.the aic (Akaike Improvement Index) item is printed. The Akaike Improvement Index is a measurement used to measure the suitability of a model to a dataset. This measurement indicates that the lower the values, the better the fit of the model.

model5 = ARIMA(Dec['Close'], order=(1,1,1))

result5 = model5.fit()

print(result5.aic)

This code snippet uses the ARIMA model to Decode the dataset 'Close' variable. The ARIMA (AutoRegressive Integrated Moving Average) model is a regulatory prediction model on a time series Deciency. This model relies on historical data to make predictions for a time series of data and is therefore called "auto-regressive". The ranking of this model is indicated as (1,1,1). This shows that the auto-regressive components of the model include the first order, the moving average components include the first order, and the first order derivative of the observed series.

the fit() method of the ARIMA model assigned to the model5 variable is called and the result is assigned to the result5 variable Dec. Later, result5.the aic (Akaike Improvement Index) item is printed. The Akaike Improvement Index is a measurement used to measure the suitability of a model to a dataset. This measurement indicates that the lower the values, the better the fit of the model.

The difference Dec Dec between these two code snippets is the ordering of the ARIMA model used. Sorting (1,1,0) was used for model4, while sorting (1,1,1) was used for model5. This ranking indicates which order of auto-regressive components, which order of moving average components and which order derivatives the model contains. The change of this ranking can affect the predictive power of the model, and therefore, the Akaike Improvement Index is used to determine which one fits better.

residuals = pd.DataFrame(result5.residence)

residuals = residuals.iloc[1:]

residuals.description()

This code snippet creates a dataset using the residuals element of the result5 model (residuals). Residuals are the difference obtained by subtracting the model's estimates from the actual values. If the model has predicted correctly, the residuals values will normally look like a random variable with a distribution.

The Residuals dataset skips the first row from zero by selecting data according to index numbers using the iloc[] method. Then, the describe() method of the residuals dataset is called and the summary statistics of the dataset (mean, median, standard deviation, etc.) are shown. These statistics provide information about the distribution of residuals values and can be used to measure how accurate the model's predictions are.

plt.style.use('fivethirtyeight')

plt.figure(figsize=(18,10))

plt.plot(df['Open'])

plt.plot(df['Close'])

plt.legend(['Open','Close'])

plt.title('Open and Close Prices')

This code snippet creates two graphs in the dataset containing the values of the variables 'Open' and 'Close'. Firstly, plt.style.the 'fivethirtyeight' style is set using the use() function. Later, plt.the dimensions of the graphics are determined using the figure() function. The dimensions are indicated as (18,10).

Then, plt.using the plot() function, the values of the 'Open' and 'Close' variables are shown in the graphs. plt.using the legend() function, the legends of the graphics ('Open' and 'Close') are added and plt.using the title() function, the titles of the charts ('Open and Close Prices') are determined. As a result of this code snippet, two graphical outputs are obtained, showing the values of the variables 'Open' and 'Close'.

plt.figure(figsize=(18,10))

plt.plot(df['High'])

plt.plot(df['Low'])

plt.legend(['High','Low'])

plt.title('High and Low Prices of Gold', fontsize =20)

This code snippet creates two graphs in the dataset containing the values of the variables 'High' and 'Low'. Firstly, plt.the dimensions of the graphics are determined using the figure() function. The dimensions are indicated as (18,10).

Then, plt.using the plot() function, the values of the 'High' and 'Low' variables are shown in the graphs. plt.using the legend() function, the legends ('High' and 'Low') of the graphics are added and plt.using the title() function, the titles of the charts ('High and Low Prices of Gold') are determined. As a result of this code snippet, two graphical outputs are obtained that show the values of the variables 'High' and 'Low'. The title font size is specified as 20.

plt.plot(df['Close'])

plt.plot(residuals)

plt.legend(['Close','Residuals'])

This code snippet creates two graphs in the dataset that contain the values of the 'Close' variable and the values of the residuals dataset. Firstly, plt.using the plot() function, the values of the 'Close' variable and the values of the residuals dataset are shown in the graphs. Later, plt.using the legend() function, the legends of the graphics ('Close' and 'Residuals') are added.

This code snippet shows the estimated values of the 'Close' variable and the residuals values obtained as the difference from the actual values. This graph shows how accurate the model's predictions are. If the model has predicted correctly, the residuals values will normally look like a random variable with a distribution.

df['PriceDiff'] = df['Close'].shift(-1) - df['Close']

df = df.dropna()

print(df['PriceDiff'])

This code snippet adds a new variable named 'Pric October' to the dataset. This variable shows the difference between the next value of the variable 'Close' and the current value of the variable 'Close'. For example, the difference between the value df['Close'][0] and the value df['Close'][1] is assigned to the value df['PriceDiff'][0]. Dec.

df['Close'].the shift(-1) function creates a new data set by sliding the values of the 'Close' variable down one row. This dataset shows the next values of the values of the variable 'Close'. Decrypts the difference between this dataset and the current values of the 'Close' variable and assigns it to the 'PriceDiff' variable.

Then, df.using the dropna() function, missing (NaN) values are discarded in the dataset. After this operation, the values of the variable df['PriceDiff'] are printed.

This code snippet shows how different the next values of the 'Close' variable are. The average of these values can be used to predict how different the next values of the 'Close' variable will be.

plt.figure(figsize=(18,10))

plt.plot(df['PriceDiff'])

plt.plot(df['Close'])

plt.legend(['PriceDiff','Close'])

plt.title('Price Difference and Close Price of Gold',fontsize=30)

This code snippet creates two graphs in the dataset containing the values of the variables 'PriceDiff' and 'Close'. Firstly, plt.the dimensions of the graphics are determined using the figure() function. The dimensions are indicated as (18,10).

Then, plt.using the plot() function, the values of the 'PriceDiff' and 'Close' variables are shown in the graphs. Later, plt.using the legend() function, the legends of the graphics ('PriceDiff' and 'Close') are added and plt.using the title() function, the titles of the charts ('Price Difference and Close Price of Gold') are determined. The title font size is specified as 30.

This code snippet shows the effect of the 'PriceDiff' variable on the 'Close' variable. It shows how the values of the 'PriceDiff' variable adapt to the values of the 'Close' variable. This graph can be used to predict how different the next values of the 'Close' variable will be.

df['MA10'] = df['Close'].rolling(10).mean()

df['MA50'] = df['Close'].rolling(50).mean()

This code snippet adds two new variables named 'MA10' and 'MA50 October' to the dataset. The 'MA10' variable shows the 10-day moving average of the 'Close' variable. The 'MA50' variable, on the other hand, shows the 50-day moving average of the 'Close' variable.

df['Close'].the rolling() function calculates the moving average of the dataset for the specified period. For example, df['Close'].rolling(10).the mean() function calculates the 10-day moving average of the 'Close' variable. The result of this function is assigned to the variable 'MA10'. Similarly, df['Close'].rolling(50).the mean() function calculates the 50-day moving average of the 'Close' variable and assigns it to the 'MA50' variable.

This code snippet shows how the changes of the 'Close' variable follow a trend. Moving averages show a regular average of the changes of variables in the data set and can therefore be used to determine trends.

from statsmodels.tsa.stattools import ads

def adf\_test(timeseries):

#Perform Dickey-Fuller test:

print ('Results of Dickey-Fuller Test:')

dftest = adfuller(timeseries, autolag='AIC')

dfoutput = pd.Series(dftest[0:4], index=['Test Statistical','p-value','#Lags Used','Number of Observations Used'])

for key,value in dftest[4].items():

dfoutput['Critical Value (%s)'%key] = value

print (dfoutput)

This code snippet defines the adf\_test() function, which is used to perform the Dickey-Fuller test. This function takes a time series dataset and applies the Dickey-Fuller test.

The Dickey-Fuller test is used to test whether a time series is stationary. A time series is stationary if its mean and variance do not change over time. If a time series is not stationary, it has this time series in train, and therefore modeling becomes difficult.

The Dickey-Fuller test gives an output that contains some important statistics, such as the 'Test Statistical' value, the 'p-value' value, and the 'Critical Value' values. The 'Test Statistical' value indicates whether the null hypothesis should be rejected. If the 'Test Statistical' value is lower than the 'Critical Value' values, the null hypothesis is rejected and the time series is stationary. If the 'Test Statistical' value is higher than the 'Critical Value' values, the null hypothesis is accepted and the time series is not stationary. the 'p-value' value indicates the probability that the null hypothesis will be rejected. If the 'p-value' is lower, the null hypothesis is rejected more strongly.

print(adf\_test(df['High']))

This code snippet performs the Dickey-Fuller test for the variable 'High' using the adf\_test() function. This function tests whether the variable 'High' is stationary.

The Dickey-Fuller test output contains 'Test Statistical' value, 'p-value' value, '#Lags Used' value, 'Number of Observations Used' value and 'Critical Value' values. These values give information about whether the variable 'High' is stationary or not. If the 'Test Statistical' value is lower than the 'Critical Value' values, the 'High' variable is stationary. the 'p-value' value indicates the probability that the null hypothesis will be rejected. If the 'p-value' is low, the null hypothesis is rejected more strongly and the variable 'High' is stationary.

from statsmodels.tsa.stattools import cpsu

def kpss\_test(timeseries):

print("Results of CPSU Test:")

kpsstest = kpss(timeseries, regression="c", nlags="auto")

kpss\_output = pd.Series(

kpsstest[0:3], index=["Test Statistical", "p-value", "Lags Used"]

)

for key, value in kpsstest[3].items():

kpss\_output["Critical Value (%s)" %key] = value

print(kpss\_output)

This code snippet defines the kpss\_test() function, which is used to perform the KPSS (Kwiatkowski-Phillips-Schmidt-Shin) test. This function takes a time series dataset and applies the KPSS test.

The CPSS test is used to test whether a time series is stationary. A time series is stationary if its mean and variance do not change over time. If a time series is not stationary, it has this time series in train, and therefore modeling becomes difficult.

The CPSU test gives an output that contains some important statistics, such as the 'Test Statistical' value, the 'p-value' value, and the 'Critical Value' values. The 'Test Statistical' value indicates whether the null hypothesis should be rejected. If the 'Test Statistical' value is higher than the 'Critical Value' values, null

kpss\_test(df['High'])

This code snippet performs the KPSS test for the variable 'High' using the kpss\_test() function. This function tests whether the variable 'High' is stationary.

The CPSU test output includes the 'Test Statistical' value, the 'p-value' value and the 'Critical Value' values. These values give information about whether the variable 'High' is stationary or not. If the 'Test Statistical' value is higher than the 'Critical Value' values, the 'High' variable is stationary. the 'p-value' value, on the other hand, indicates the probability that the null hypothesis will be rejected. If the 'p-value' value is lower, the null hypothesis is rejected more strongly and the variable 'High' is stationary.

df['high\_diff'] = df['High'] - df['High'].shift(1)

df['High\_diff'].dropna().plot(figsize= (20,10))

This code snippet creates a new variable called 'high\_diff', which calculates the difference of the variable 'High' compared to its previous values. Firstly, the variable 'High\_diff' is assigned the difference of the variable 'High' compared to the previous values. df['High'].the shift(1) function returns the previous values of the variable 'High' in the dataset. The result of this function is subtracted from the variable 'High' and assigned to the variable 'High\_diff'.

Then, df['High\_diff'].with the dropna() function, the NaN values of the data in the 'high\_diff' variable are extracted. Finally, plt.the values of the variable 'high\_diff' are shown in the graph using the plot() function. The graphic dimensions are determined as (20,10).

This code snippet shows how the values of the variable 'High' have changed compared to their previous values. This graph shows the trends of the changes of the variable 'High' and can therefore be used to determine trends.

kpss\_test(df['high\_diff'].dropna())

This code snippet performs the KPSS test for the variable 'High\_diff' using the kpss\_test() function. This function tests whether the variable 'high\_diff' is stationary.

The CPSU test output includes the 'Test Statistical' value, the 'p-value' value and the 'Critical Value' values. These values give information about whether the variable 'high\_diff' is stationary. If the 'Test Statistical' value is higher than the 'Critical Value' values, the 'high\_diff' variable is stationary. the 'p-value' value indicates the probability that the null hypothesis will be rejected. If the value of 'p-value' is lower, the null hypothesis is rejected more strongly and the variable 'High\_diff' was stationary

adf\_test(df['High\_diff']).dropna())

Results of the Dickey-Fuller Test:

Test Statistical -1.627129e+01

p-value 3.485839e-29

#Lags Used 2.500000e+01

Number of Observations Used 5.675000e+03

Critical Value (1%) -3.431503e+00

Critical Value (5%) -2.862049e+00

Critical Value (10%) -2.567041e+00

dtype: float64

This output shows the results of the Dickey-Fuller test. This test tested whether the variable 'high\_diff' is stationary or not.

The 'Test Statistical' value indicates whether the null hypothesis should be rejected. If the 'Test Statistical' value is lower than the 'Critical Value' values, the null hypothesis is rejected and the 'High\_diff' variable is stationary. As can be seen in this output, the 'Test Statistical' value is lower than the 'Critical Value' values, and therefore the 'High\_diff' variable is stationary.

the 'p-value' value indicates the probability that the null hypothesis will be rejected. If the 'p-value' is lower, the null hypothesis is rejected more strongly. As seen in this output,

from sklearn.preprocessing import MinMaxScaler

from keras.models import Sequential

import math

from keras.layers import Denses, LSTM

import matplotlib.pyplot as plt

plt.style.use("fivethirtyeight")

#Create a new dataframe with only the "Close column"

data = df.filter(['Close'])

#Convert the dataframe to a numpy array

dataset = data.values

#Get the number of rows to train the model on

training\_data\_len = math.ceil(len(dataset)\* .8)

training\_data

This code snippet contains the necessary steps to train an LSTM (Long Short-Term Memory) artificial neural network model on a time series dataset.

First, a new dataset is created from the dataset using the 'Close' variable. This data set is assigned to a variable named data. Then, the data variable is converted to the numpy array and the dataset is assigned to the variable.

Finally, the training\_data\_len variable is calculated, which indicates that up to 80% of the dataset dataset will be used as model training data. This variable is calculated by rounding the number of rows of the dataset. This step specifies how many rows the data to be used in the model training will consist of.

scaler=MinMaxScaler(feature\_range=(0,1))

scaled\_data = scaler.fit\_transform(dataset)

scaled\_data

array([[0.01494925],

[0.01406664],

[0.01423213],

...,

[0.8162511 ],

[0.81067961],

[0.80135702]])

This code snippet performs the scaling of the dataset dataset using the MinMaxScaler class. Scaling ensures that the values in the dataset are in the same December, which makes the model work better.

The MinMaxScaler class converts the values of the dataset into a value December between 0 and 1. Decryption of the data set is performed by using the MinMaxScaler class. This class specifies the December of values to be converted with the parameter 'feature\_range'. The scaling operation is performed by assigning the scaled\_data variable.

the scaled\_data variable contains the scaled version of the dataset dataset. After Decaling, the values in the dataset will be in a value December between 0 and 1

#create the training data set

#Create the scaled training data set

train\_data = scaled\_data[0:training\_data\_len, :]

#Split the data into x\_train and y\_train data sets

x\_train = []

y\_train = []

for i in range(60, len(train\_data)):

x\_train.append(train\_data[i-60:i, 0])

y\_train.append(train\_data[i, 0])

if i<= 60:

print(x\_train)

print(y\_train)

print()

This code snippet creates training data from the scaled dataset. Up to 80% of the scaled dataset is assigned to the train\_data variable and this variable is used as training data.

Then, two datasets named x\_train and y\_train are created. The creation of these datasets is performed using the for loop. the values of the train\_ Deca dataset between i-60 and i are added to the x\_train dataset. if y\_train is in the dataset, i is in the train\_data dataset. the value in the line is added. This process is repeated for all rows of the train\_data dataset.

This code snippet performs the creation of the x\_train and y\_train datasets. The creation of these data sets is necessary for the creation of data sets that will be used in the training of the LSTM artificial neural network model.

#Convert the x\_train and y\_train to numpy arrays

x\_train, y\_train = np.array(x\_train), np.array(y\_train)

This code snippet converts the x\_train and y\_train datasets into numpy arrays. The data sets to be used in the training of the LSTM artificial neural network model should be as a numpy array. Therefore, the x\_train and y\_train datasets are np.it is converted to a numpy array using the array() function.

After this process, the x\_train and y\_train datasets become available as a numpy array. The fact that these datasets are numpy sequences is necessary for the training of the LSTM artificial neural network model.

#Reshape the data

x\_train = np.reshape(x\_train, (x\_train.shape[0], x\_train.shape[1], 1))

x\_train.shape

This code snippet changes the shape of the x\_train dataset. In the training of the LSTM artificial neural network model, the dataset should be 3-dimensional. Therefore, the x\_train dataset is np.it is made 3-dimensional using the reshape() function.

np.the reshape() function converts the dataset into the desired shape by changing the shape of the dataset. This function takes two parameters that specify the new shape of the dataset. The first parameter specifies the number of new rows of the dataset, and the second parameter specifies the number of new columns of the dataset.

In this code snippet, the x\_train dataset (x\_train.shape[0], x\_train.shape[1] is converted to the form 1). This figure is the x\_train of the number of rows of the dataset.shape[0] is up, the number of columns is also x\_train.shape[1] indicates that it is up to. After this process, the x\_train dataset becomes 3-dimensional and becomes ready for the training of the LSTM artificial neural network model.

#Build the LSTM model

model= Sequential()

model.add(LSTM(50, return\_sequences=True, input\_shape= (x\_train.shape[1], 1)))

model.add(LSTM(50, return\_sequences=False))

model.add(Dense(25))

model.add(Even(1))

In this code snippet, the LSTM artificial neural network model is created. This model is created using the Sequential() class. The Sequential() class is used in cases where the artificial neural network model does not have a complex structure and has a unidirectional structure.

Then, LSTM layers are added to the model object. The first LSTM layer is created to produce 50 outputs, and the return\_sequences parameter is specified as True. This parameter specifies whether the output of the LSTM layer will be transmitted to other layers. The second LSTM layer is created to produce 50 outputs and the return\_sequences parameter is set to False. This parameter indicates that the output of the second LSTM layer will not be transmitted to other layers.

Finally, the Dense layers are added to the model object. The first Dense layer is created in such a way that it produces 25 outputs. The second Density layer, on the other hand, is created in such a way as to produce 1 output. This layer is the output layer of the artificial neural network model.

In this code snippet, the creation of the LSTM artificial neural network model is performed. This model is created in such a way that it makes predictions by processing the dataset.

#Compile the model

model.compile(optimizer = 'man', loss='mean\_squared\_error')

In this code snippet, the created LSTM artificial neural network model is compiled. model.the compile() function is used to compile an artificial neural network model. This function determines the optimization algorithm and loss function of the model to be used during training.

In this code snippet, the optimizer parameter is given the value 'adam' and the loss parameter is given the value 'mean\_squared\_error'. These parameters indicate that the artificial neural network model will be trained using the Adam optimization algorithm, and the loss function has a square average error.

After this process, the LSTM artificial neural network model is compiled and becomes ready for the training process.

#Train the model

model.fit(x\_train,y\_train,batch\_size=1,epochs=1)

In this code snippet, the LSTM artificial neural network model is trained. model.the fit() function performs the training of the artificial neural network model. This function takes the training data and performs the training of the model on these data.

In this code snippet, model the x\_train and y\_train datasets.it is given to the fit() function. These data sets are the data to be used in the training of the LSTM artificial neural network model. model.the fit() function also retrieves the batch\_size and epochs parameters. the batch\_size parameter specifies how many parts the dataset will be divided into, and the epochs parameter specifies how many times the training process will be repeated.

In this code snippet, the batch\_size parameter is given a value of 1 and the epochs parameter is given a value of 1. These parameters indicate that the dataset will be divided into one part and the training process will be repeated for one time. After this process, the LSTM artificial neural network model is trained and becomes ready to make predictions.

#Create the testing data set

#Create a new array containing scaled values from index 4502 to 2003

test\_data= scaled\_data[training\_data\_len - 60:, :]

#Create the data sets x test and y\_test

x\_test = []

y\_test = dataset[training\_data\_len:,:]

for i in range(60, len(test\_data)):

x\_test.append(test\_data[i-60:i,0])

In this code snippet, datasets are created to test the prediction performance of the model. The process of creating datasets is performed in a similar way to educational data.

First, the test\_data variable is assigned the data that comes after the training data. Then, the x\_test and y\_test datasets are created. the x\_test dataset is created by taking from the test\_data dataset, and the y\_test dataset is taken from the original dataset containing the target variables of the test data.

In this code snippet, the for loop is used when creating the x\_test dataset. This loop is added to the x\_test dataset by taking data samples from the test\_data dataset. After this process, the x\_test dataset will be created and will be used to test the prediction performance of the model.

#convert the data to a numpy array

x\_test = np.array(x\_test)

In this code snippet, the x\_test dataset is converted to a numpy array. This process is performed in order to make the dataset available to artificial neural network libraries such as tensorflow or Keras.

In this code snippet, the x\_test dataset is np.it is converted to a numpy array with the array() function. After this process, the x\_test dataset becomes available as a numpy array.

#reshape the data

x\_test = np.reshape(x\_test, (x\_test.shape[0], x\_test.shape[1], 1))

In this code snippet, the x\_test dataset is reshaped. This process is performed in order to bring the dataset to the shape expected by the LSTM artificial neural network model.

The LSTM artificial neural network model waits for the dataset (batch\_size, timesteps, features) in the form of. Therefore, the dataset is reshaped and brought to this shape.

In this code snippet, the x\_test dataset is np.it is reshaped with the reshape() function. This function takes parameters that specify the new shape of the dataset. In this code snippet, the necessary parameters are given to convert the x\_test dataset (batch\_size, timesteps, features) to its shape. After this process, the x\_test dataset is converted to the shape expected by the LSTM artificial neural network model.

#get the models predicted price values

predictions = modelo.predict(x\_test)

predictions = scaler.inverse\_transform(predictions)

In this code snippet, the predictions of the LSTM artificial neural network model are taken. This is the process, the model.it is performed with the predict() function. model.the predict() function returns the predictions of the artificial neural network model according to the given data set.

In this code snippet, the model.the predict() function is given the x\_test dataset. After this process, the predictions made by the model according to the x\_test dataset are assigned to the predictions variable.

Next, the predictions variable scalar.it is returned from the scaling operation with the inverse\_transform() function. This function returns the given dataset to its previous state before the scaling operation. After this process, the predictions variable contains the predictions made by the LSTM artificial neural network model based on real data.

#get the root mean squared error (RMSE)

rmse = np.sqrt(np.mean(predictions - y\_test)\*\*2)

rmse

In this code snippet, the accuracy value of the predictions of the LSTM artificial neural network model is calculated. This value is called root mean squared error (RMSE) and is calculated as the average squared error value of the model's estimates.

To calculate the RMSE value, first of all, the squares of the differences between the estimates and the actual values are taken.Dec. Then, the average of these squares is taken, and finally, the square root of this average is taken. As a result of these operations, the RMSE value is calculated.

In this code snippet, np.the square root operation is performed with the sqrt() function and np.the mean() function is used to average the squares. After this process, the RMSE value is obtained and printed on the screen.

#Plot the data

train =data[:training\_data\_len]

valid = data[training\_data\_len: ]

valid['Predictions'] = predictions

#Visualize the data

plt.figure(figsize=(16,8))

plt.title('Model')

plt.xlabel('Date', fontsize=18)

plt.ylabel('Close Price USD($)', fontsize=18)

plt.plot(train[ 'Close' ])

plt.plot(valid[[ 'Close', 'Predictions']])

plt.legend(['Train', 'Val', 'Predictions'], loc='lower right')

plt.show()

In this code snippet, the predictions of the LSTM artificial neural network model are shown graphically. This graph shows how similar the model's predictions are to the actual data and shows the accuracy value of the model.

In this code snippet, the train and valid variables are created first. The train variable contains the dataset to be used as the model's training data, and the valid variable contains the dataset to be used as the model's test data. Then, the predictions variable is added to the valid variable and this variable contains the predictions of the model.

Finally, plt.with the plot() function, the Close columns of the train and valid variables are plotted, and the predictions variable is also shown in the graph. After this process, it is visually shown how similar the model's predictions are to the actual data.

#Show the and predicted prices

Valid

In this code snippet, the actual values are shown with the predictions of the model. In this code snippet, the valid variable is printed, and this variable contains the model's estimates and the actual values. In this variable, the Close column contains the actual values, and the Predictions column contains the model's predictions. In this way, a comparison of the model's estimates with the actual values can be made.