In [ ]:

In [ ]:

Load Librabries and Data

In [2]:
```
#import required packages

import yfinance as yf
import datetime
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas_datareader import data as wb
from scipy.stats import norm
import seaborn as sns
sns.set()

# set plotting parameters
%matplotlib inline
#sns.set_style("whitegrid")
plt.rc("figure", figsize=(12, 8))
plt.rc("savefig", dpi=90)
plt.rc("font", family="sans-serif")
plt.rc("font", size=14)
```

In [3]:
```
# Define the ticker symbol and date range
ticker = "BAS.DE"
start = '2017-01-01'
end = '2021-12-31'

# Download the historical stock prices
data_BASF = yf.download(ticker, start=start, end=end)


# Create a new DataFrame
df = pd.DataFrame(data_BASF)
df.head()
```

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

Out[3]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-01-02	87.500000	88.809998	87.099998	88.699997	56.960781	1245318
2017-01-03	88.879997	88.879997	87.419998	87.699997	56.318611	2806564
2017-01-04	87.889999	88.150002	87.269997	88.150002	56.607590	1955701
2017-01-05	87.540001	88.250000	87.389999	87.790001	56.376408	1753933
2017-01-06	87.500000	87.730003	87.269997	87.519997	56.203011	1518979

EDA

- plot the features and the target to see the relationship between them.

In [4]:
```
# Data Preparation
import pandas as pd
from sklearn.model_selection import train_test_split

# Assuming df is the loaded DataFrame
X = df[['Open', 'High', 'Low', 'Volume']]  # Features
y = df['Close']  # Target variable

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [ ]:
```
```

In [5]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
```

Build Machine Learning Models

1. Linear Reg

In [55]:
```
# Linear Regression
from sklearn.linear_model import LinearRegression

# Create and train the Linear Regression model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_lr = lr_model.predict(X_test)

# Evaluate the Linear Regression model
mse_lr = mean_squared_error(y_test, y_pred_lr)
mae_lr = mean_absolute_error(y_test, y_pred_lr)
r2_lr = r2_score(y_test, y_pred_lr)

# Evaluate the Decision Tree model
dt_metrics = pd.DataFrame({
    'Mean Squared Error': [mse_lr],
    'Mean Absolute Error': [mae_lr],
    'R-squared (R2)': [r2_lr]
}, index=['Decision Tree Metrics']).T

# Display the DataFrame
dt_metrics
```

Out[55]:

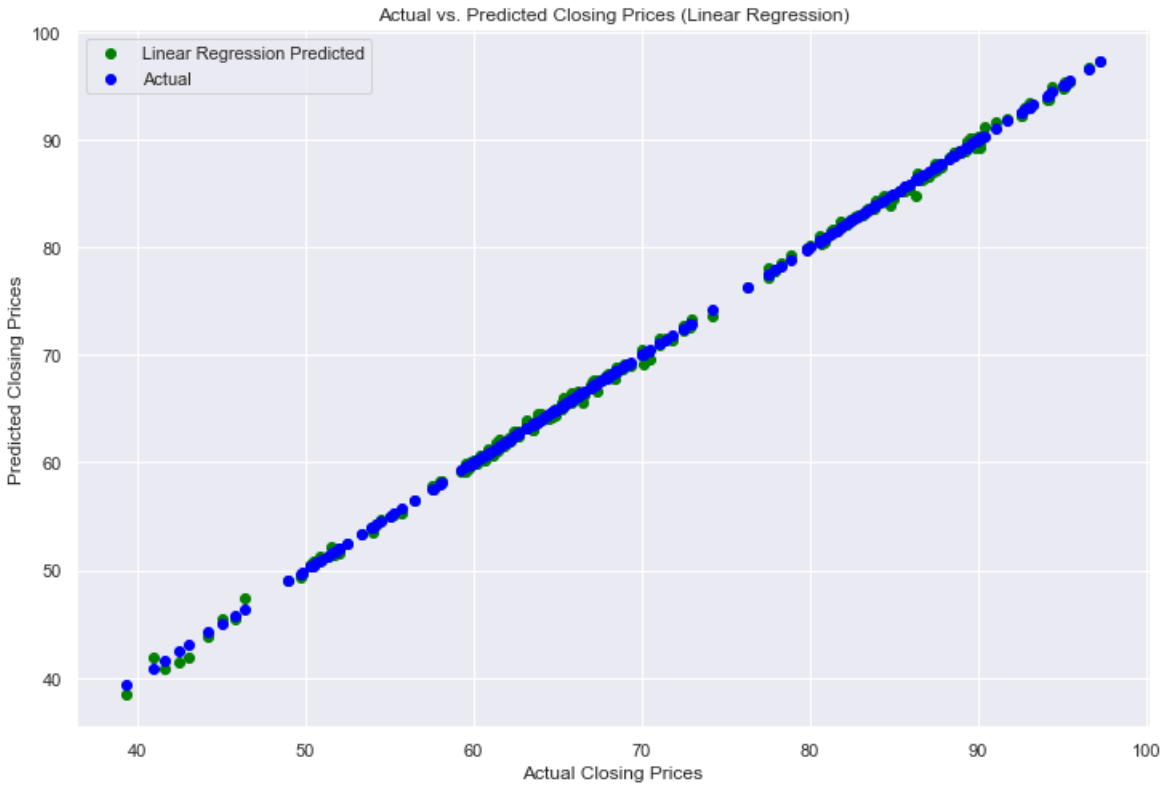
	Decision Tree Metrics
Mean Squared Error	0.140814
Mean Absolute Error	0.288290
R-squared (R2)	0.999249

In [8]:
```
df_lr = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_lr}, index=X_test.index)
df_lr.head()
```

Out[8]:

Date	Actual	Predicted
2021-10-13	64.900002	64.572039
2021-02-10	67.150002	67.097003
2018-08-28	80.550003	80.780763
2021-06-07	68.510002	68.922824
2020-05-12	45.070000	45.402601

In [9]:
```
# Linear Regression Plot
plt.scatter(y_test, y_pred_lr, color='green', label='Linear Regression Predicted')
plt.scatter(y_test, y_test, color='blue', label='Actual')
plt.xlabel('Actual Closing Prices')
plt.ylabel('Predicted Closing Prices')
plt.title('Actual vs. Predicted Closing Prices (Linear Regression)')
plt.legend()
plt.show()
```



In [ ]:

2. Decision Tree:

In [56]:
```
# Decision Tree
from sklearn.tree import DecisionTreeRegressor


# Create and train the Decision Tree model
dt_model = DecisionTreeRegressor(random_state=42)
dt_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_dt = dt_model.predict(X_test)

# Evaluate the Decision Tree model
mse_dt = mean_squared_error(y_test, y_pred_dt)
mae_dt = mean_absolute_error(y_test, y_pred_dt)
r2_dt = r2_score(y_test, y_pred_dt)

# Evaluate the Decision Tree model
dt_metrics = pd.DataFrame({
    'Mean Squared Error': [mse_dt],
    'Mean Absolute Error': [mae_dt],
    'R-squared (R2)': [r2_dt]
}, index=['Decision Tree Metrics']).T

# Display the DataFrame
dt_metrics
```

Out[56]:

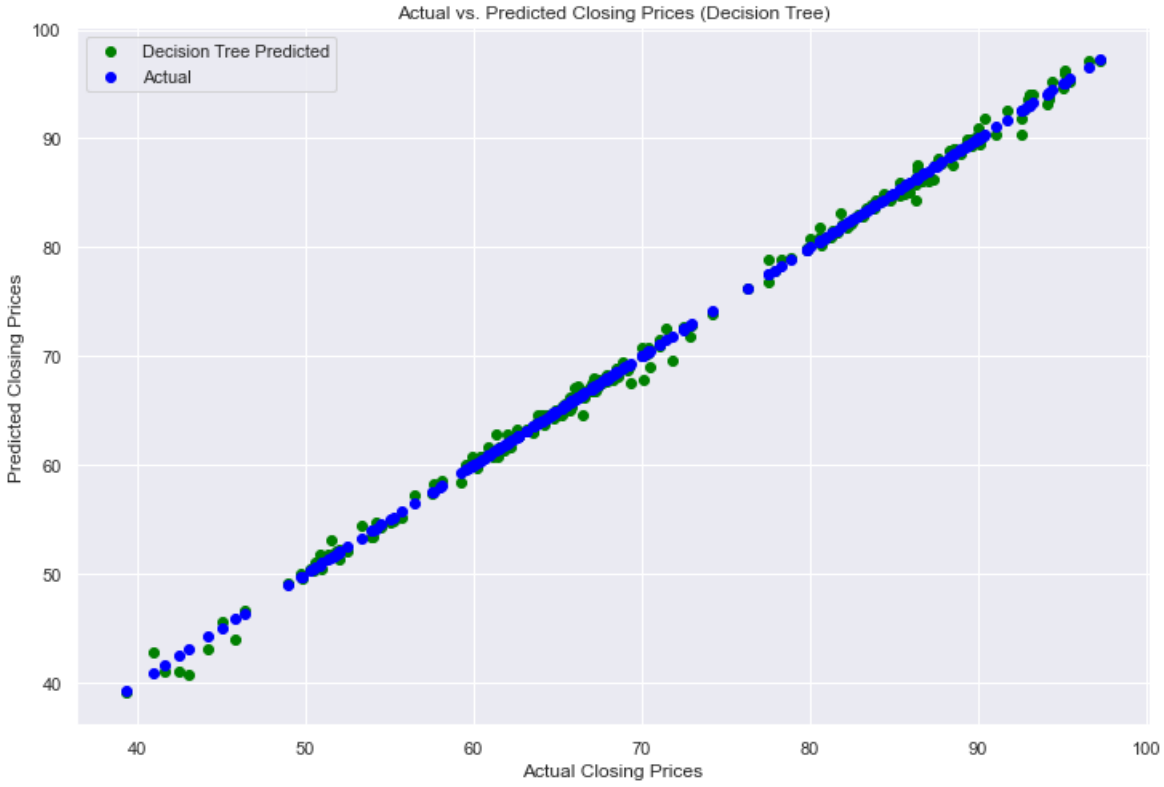
	Decision Tree Metrics
Mean Squared Error	0.414966
Mean Absolute Error	0.464311
R-squared (R2)	0.997786

In [12]:
```
# DataFrames for Actual and Predicted Values
df_dt = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_dt}, index=X_test.index)
df_dt.head()
```

Out[12]:

Date	Actual	Predicted
2021-10-13	64.900002	64.599998
2021-02-10	67.150002	67.639999
2018-08-28	80.550003	80.930000
2021-06-07	68.510002	68.849998
2020-05-12	45.070000	45.580002

In [13]:
```
# Decision Tree Plot
plt.scatter(y_test, y_pred_dt, color='green', label='Decision Tree Predicted')
plt.scatter(y_test, y_test, color='blue', label='Actual')
plt.xlabel('Actual Closing Prices')
plt.ylabel('Predicted Closing Prices')
plt.title('Actual vs. Predicted Closing Prices (Decision Tree)')
plt.legend()
plt.show()
```



In [ ]:

3. KNN

In [57]:
```
# KNN
from sklearn.neighbors import KNeighborsRegressor

# Create and train the KNN model
knn_model = KNeighborsRegressor(n_neighbors=5)
knn_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_knn = knn_model.predict(X_test)

# Evaluate the KNN model
mse_knn = mean_squared_error(y_test, y_pred_knn)
mae_knn = mean_absolute_error(y_test, y_pred_knn)
r2_knn = r2_score(y_test, y_pred_knn)

# Evaluate the Decision Tree model
dt_metrics = pd.DataFrame({
    'Mean Squared Error': [mse_knn],
    'Mean Absolute Error': [mae_knn],
    'R-squared (R2)': [r2_knn]
}, index=['KNN Metrics']).T

# Display the DataFrame
dt_metrics
```

Out[57]:

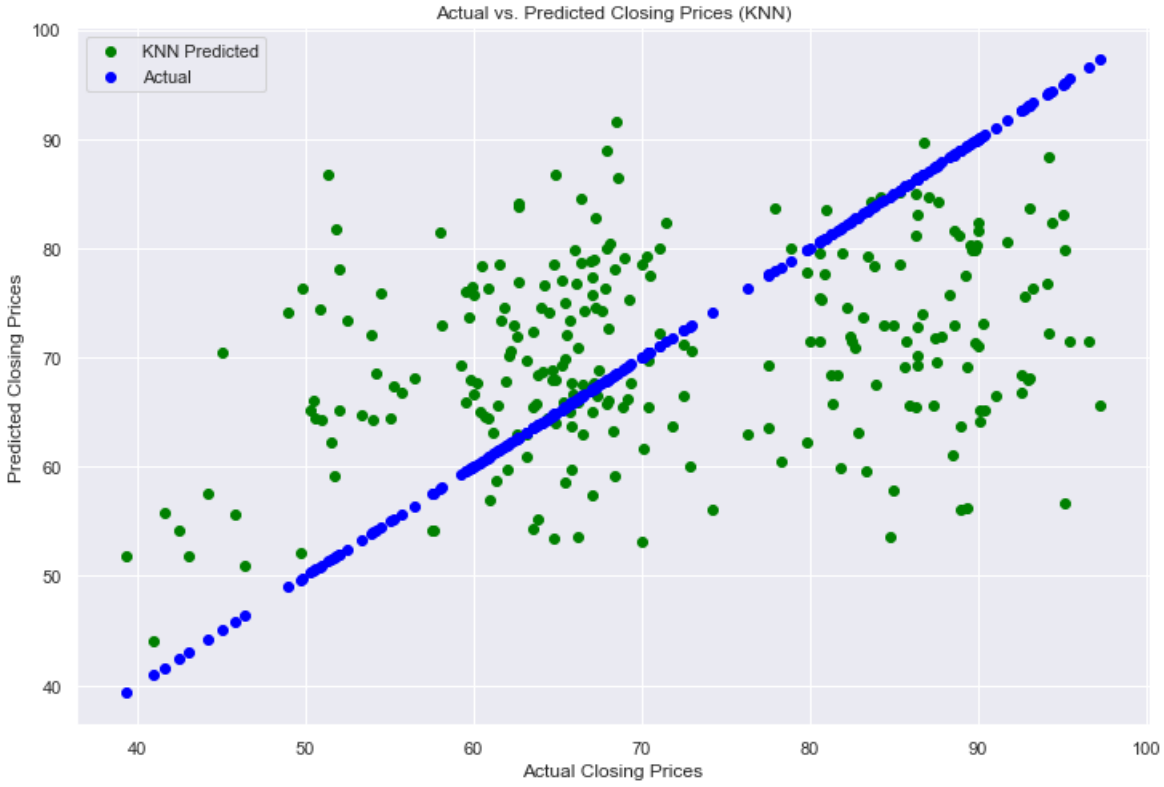
	KNN Metrics
Mean Squared Error	191.160780
Mean Absolute Error	11.321020
R-squared (R2)	-0.020025

In [16]:
```
df_knn = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_knn}, index=X_test.index)
df_knn.head()
```

Out[16]:

Date	Actual	Predicted
2021-10-13	64.900002	68.006001
2021-02-10	67.150002	79.025999
2018-08-28	80.550003	79.551999
2021-06-07	68.510002	91.634000
2020-05-12	45.070000	70.496000

In [17]:
```
# KNN Plot
plt.scatter(y_test, y_pred_knn, color='green', label='KNN Predicted')
plt.scatter(y_test, y_test, color='blue', label='Actual')
plt.xlabel('Actual Closing Prices')
plt.ylabel('Predicted Closing Prices')
plt.title('Actual vs. Predicted Closing Prices (KNN)')
plt.legend()
plt.show()
```



In [ ]:
```
```

4. SVM:

In [18]:
```
from sklearn.svm import SVR
```

In [19]:
```
# Support Vector Regression with RBF Kernel
svr_rbf_model = SVR(kernel='rbf')  # Using RBF kernel
svr_rbf_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_svr_rbf = svr_rbf_model.predict(X_test)

# Evaluate the SVR with RBF Kernel model
mse_svr_rbf = mean_squared_error(y_test, y_pred_svr_rbf)
mae_svr_rbf = mean_absolute_error(y_test, y_pred_svr_rbf)
r2_svr_rbf = r2_score(y_test, y_pred_svr_rbf)

# Display the evaluation metrics for SVR with RBF Kernel
svr_rbf_metrics = pd.DataFrame({
    'Mean Squared Error': [mse_svr_rbf],
    'Mean Absolute Error': [mae_svr_rbf],
    'R-squared (R2)': [r2_svr_rbf]
}, index=['SVR with RBF Kernel']).T

svr_rbf_metrics
```

Out[19]:

	SVR with RBF Kernel
Mean Squared Error	167.051920
Mean Absolute Error	10.611225
R-squared (R2)	0.108619

In [20]:
```
df_svr_rbf = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_svr_rbf}, index=X_test.index)
df_svr_rbf.head()
```

Out[20]:

Date	Actual	Predicted
2021-10-13	64.900002	70.948055
2021-02-10	67.150002	71.676071
2018-08-28	80.550003	73.400791
2021-06-07	68.510002	73.802618
2020-05-12	45.070000	65.871994

In [21]:
```
# Support Vector Regression with RBF Kernel Plot
plt.scatter(y_test, y_pred_svr_rbf, color='green', label='SVR with RBF Kernel Predicted')
plt.scatter(y_test, y_test, color='blue', label='Actual')
plt.xlabel('Actual Closing Prices')
plt.ylabel('Predicted Closing Prices')
plt.title('Actual vs. Predicted Closing Prices (SVR with RBF Kernel)')
plt.legend()
plt.show()
```

In [ ]:

5. Gradient Boosting

In [22]:
```
# Gradient Boosting
from sklearn.ensemble import GradientBoostingRegressor

# Create and train the Gradient Boosting model
gb_model = GradientBoostingRegressor(random_state=42)
gb_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_gb = gb_model.predict(X_test)

# Evaluate the Gradient Boosting model
mse_gb = mean_squared_error(y_test, y_pred_gb)
mae_gb = mean_absolute_error(y_test, y_pred_gb)
r2_gb = r2_score(y_test, y_pred_gb)

# Display the evaluation metrics for Gradient Boosting
gb_metrics = pd.DataFrame({
    'Mean Squared Error': [mse_gb],
    'Mean Absolute Error': [mae_gb],
    'R-squared (R2)': [r2_gb]
}, index=['Gradient Boosting']).T

gb_metrics
```

Out[22]:

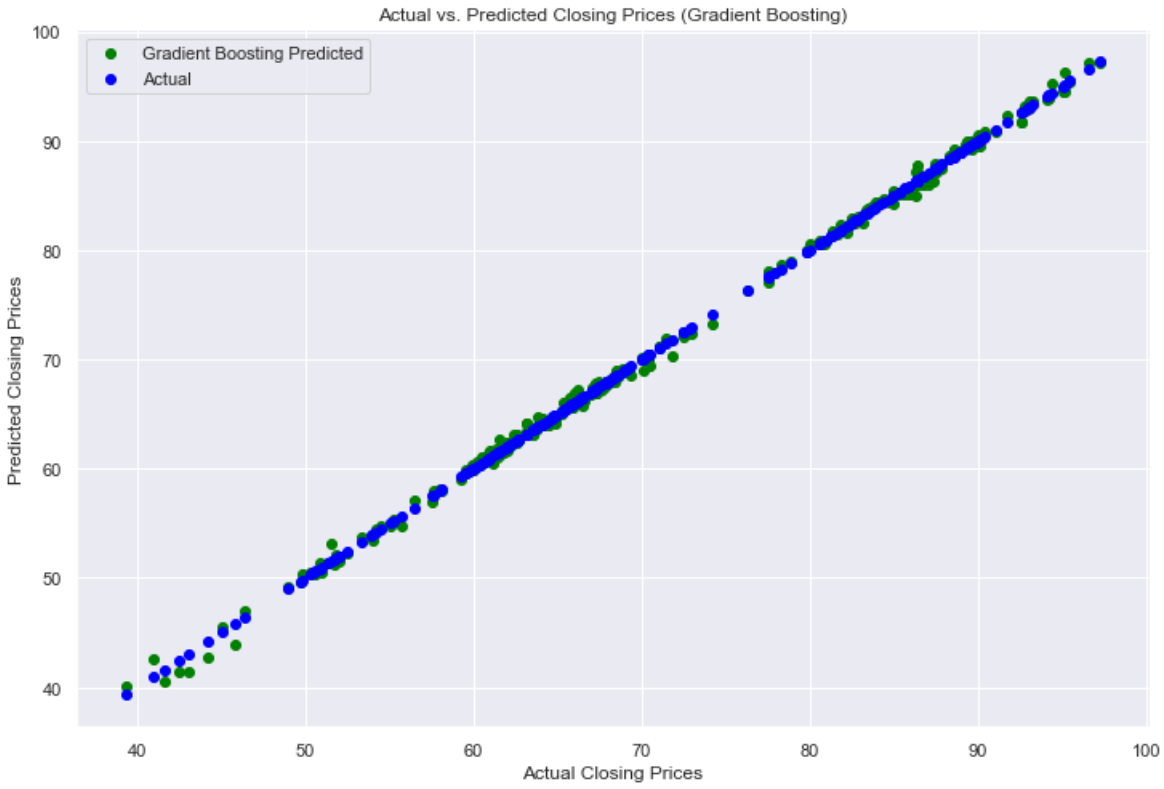
	Gradient Boosting
Mean Squared Error	0.275467
Mean Absolute Error	0.402602
R-squared (R2)	0.998530

In [23]:
```
df_gb = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred_gb}, index=X_test.index)
df_gb.head()
```

Out[23]:

Date	Actual	Predicted
2021-10-13	64.900002	64.524487
2021-02-10	67.150002	67.301996
2018-08-28	80.550003	80.892181
2021-06-07	68.510002	68.998039
2020-05-12	45.070000	45.473107

In [24]:
```
# Gradient Boosting Plot
plt.scatter(y_test, y_pred_gb, color='green', label='Gradient Boosting Predicted')
plt.scatter(y_test, y_test, color='blue', label='Actual')
plt.xlabel('Actual Closing Prices')
plt.ylabel('Predicted Closing Prices')
plt.title('Actual vs. Predicted Closing Prices (Gradient Boosting)')
plt.legend()
plt.show()
```



In [ ]:
```

```

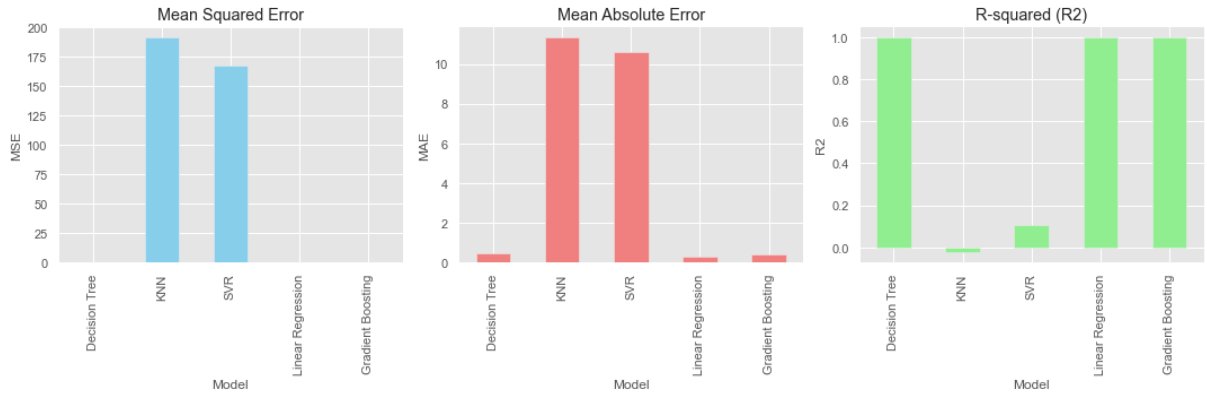
Compare The Models

In [25]:
```
# Create a DataFrame for model comparison
model_comparison = pd.DataFrame({
    'Model': ['Decision Tree', 'KNN', 'SVR', 'Linear Regression', 'Gradient Boosting'],
    'Mean Squared Error': [mse_dt, mse_knn, mse_svr_rbf, mse_lr, mse_gb],
    'Mean Absolute Error': [mae_dt, mae_knn, mae_svr_rbf, mae_lr, mae_gb],
    'R-squared (R2)': [r2_dt, r2_knn, r2_svr_rbf, r2_lr, r2_gb]
}).set_index('Model')

# Display the DataFrame
model_comparison
```

Out[25]:

Model	Mean Squared Error	Mean Absolute Error	R-squared (R2)
Decision Tree	0.414966	0.464311	0.997786
KNN	191.160780	11.321020	-0.020025
SVR	167.051920	10.611225	0.108619
Linear Regression	0.140814	0.288290	0.999249
Gradient Boosting	0.275467	0.402602	0.998530

In [41]:
```
import matplotlib.pyplot as plt

# Set the style for the plots
plt.style.use('ggplot')

# Create subplots for Mean Squared Error, Mean Absolute Error, and R-squared
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))

# Bar plot for Mean Squared Error
model_comparison['Mean Squared Error'].plot(kind='bar', ax=axes[0], color='skyblue')
axes[0].set_title('Mean Squared Error')
axes[0].set_ylabel('MSE')

# Bar plot for Mean Absolute Error
model_comparison['Mean Absolute Error'].plot(kind='bar', ax=axes[1], color='lightcoral')
axes[1].set_title('Mean Absolute Error')
axes[1].set_ylabel('MAE')

# Bar plot for R-squared
model_comparison['R-squared (R2)'].plot(kind='bar', ax=axes[2], color='lightgreen')
axes[2].set_title('R-squared (R2)')
axes[2].set_ylabel('R2')

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the plots
plt.show()
```



Certainly! Let's discuss the performance of each model based on the provided metrics and identify potential areas for improvement:

1. **Decision Tree:**
   - **Mean Squared Error (MSE):** 0.414966
   - **Mean Absolute Error (MAE):** 0.464311
   - **R-squared (R2):** 0.997786
   - **Discussion:** The Decision Tree model seems to perform very well, as indicated by the low MSE and MAE and the high R-squared. It captures a high percentage of the variance in the data.
2. **KNN:**
   - **Mean Squared Error (MSE):** 191.160780
   - **Mean Absolute Error (MAE):** 11.321020
   - **R-squared (R2):** -0.020025
   - **Discussion:** The KNN model appears to have higher errors and a negative R-squared, indicating that it may not be well-suited for the given data. Possible areas for improvement could include tuning the number of neighbors (`n_neighbors`) or exploring other distance metrics.
3. **SVR:**
   - **Mean Squared Error (MSE):** 167.051920
   - **Mean Absolute Error (MAE):** 10.611225
   - **R-squared (R2):** 0.108619
   - **Discussion:** SVR has lower errors compared to KNN, but the R-squared is still relatively low. It may benefit from hyperparameter tuning, such as adjusting the choice of the kernel function or optimizing other parameters.
4. **Linear Regression:**
   - **Mean Squared Error (MSE):** 0.140814

- **Mean Absolute Error (MAE):** 0.288290
- **R-squared (R2):** 0.999249
- **Discussion:** Linear Regression shows excellent performance with very low errors and a very high R-squared. It seems to be a well-fitted model for the data.

5. **Gradient Boosting:**

- **Mean Squared Error (MSE):** 0.275467
- **Mean Absolute Error (MAE):** 0.402602
- **R-squared (R2):** 0.998530
- **Discussion:** Gradient Boosting performs well with low errors and a high R-squared. It's a powerful ensemble method. Potential improvements could involve fine-tuning hyperparameters or exploring feature engineering.

Potential Areas for Improvement:

- **Feature Engineering:** Analyze the features and consider whether additional features or transformations could enhance model performance.
- **Hyperparameter Tuning:** Experiment with different hyperparameter values for each model to find optimal configurations.
- **Outlier Detection/Removal:** Investigate if outliers in the data are affecting model performance and consider outlier removal strategies.
- **Ensemble Methods:** Experiment with combining models using ensemble methods, such as stacking or blending, to potentially improve overall performance.

It's essential to iteratively refine models based on experimentation and domain knowledge to achieve the best possible predictions.

Based on the provided metrics, we can assess the relative performance of each model:

Best Model: Linear Regression

- **Reasoning:** Linear Regression has the lowest Mean Squared Error (MSE), Mean Absolute Error (MAE), and the highest R-squared (R2) among all the models. It consistently outperforms other models on these metrics, indicating a strong fit to the data.

Worst Model: KNN

- **Reasoning:** KNN has the highest MSE and MAE, as well as a negative R-squared. These metrics suggest that KNN does not perform well on the given data compared to the other models. The negative R-squared indicates that the model's predictions are worse than a simple horizontal line.

Overall Ranking:

1. Linear Regression
2. Decision Tree
3. Gradient Boosting
4. SVR
5. KNN

It's important to note that the best and worst models depend on the specific metrics and goals of the task. In this case, Linear Regression is deemed the best due to its low errors and high R-squared, while KNN is considered the least effective based on higher errors and a negative R-squared. Keep in mind that model performance can vary across different datasets, and further experimentation or fine-tuning may be required for optimal results.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Black Scholes Model

In [41]:
```
import numpy as np
from scipy.stats import norm

# Assuming 'predicted_stock_prices' is obtained from the best model (Linear Regression)
# Make predictions on the test set
predicted_stock_prices = lr_model.predict(X_test)  # Assuming lr_model is your Linear Regression mo

# BSM Functions
def d1(S, X, r, stdev, T):
    return (np.log(S / X) + (r + stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))

def d2(S, X, r, stdev, T):
    return (np.log(S / X) + (r - stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))

def BSM_call(S, X, r, stdev, T):
    return (S * norm.cdf(d1(S, X, r, stdev, T))) - (X * np.exp(-r * T) * norm.cdf(d2(S, X, r, stdev

def BSM_put(S, X, r, stdev, T):
    return ((X * np.exp(-r * T) * norm.cdf(-d2(S, X, r, stdev, T))) - S * norm.cdf(-d1(S, X, r, std

# Use the last predicted stock price as the current price
S_predicted = predicted_stock_prices[-1]

# Calculate the standard deviation for the predicted stock returns
stdev_predicted = np.std(predicted_stock_prices) * 250 ** 0.5

# Set other Black-Scholes Model parameters
r = 0.025
X_option = 48
T_option = 1

# Calculate d1 and d2 for option pricing
d1_option = d1(S_predicted, X_option, r, stdev_predicted, T_option)
d2_option = d2(S_predicted, X_option, r, stdev_predicted, T_option)

# Estimate call and put option prices using BSM
call_option_price = BSM_call(S_predicted, X_option, r, stdev_predicted, T_option)
put_option_price = BSM_put(S_predicted, X_option, r, stdev_predicted, T_option)

# Display the estimated option prices
print(f"Estimated Call Option Price: {call_option_price}")
print(f"Estimated Put Option Price: {put_option_price}")
```

```
Estimated Call Option Price: 93.73752213535637
Estimated Put Option Price: 46.814875777359966
```

In [ ]:

In [42]:
```
import numpy as np
from scipy.stats import norm

# Assuming 'actual_stock_prices' is your actual stock prices
# Replace it with the actual stock prices corresponding to your test set
actual_stock_prices = y_test  # Assuming y_test contains the actual stock prices

# BSM Functions
def d1(S, X, r, stdev, T):
    return (np.log(S / X) + (r + stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))

def d2(S, X, r, stdev, T):
    return (np.log(S / X) + (r - stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))

def BSM_call(S, X, r, stdev, T):
    return (S * norm.cdf(d1(S, X, r, stdev, T))) - (X * np.exp(-r * T) * norm.cdf(d2(S, X, r, stdev

def BSM_put(S, X, r, stdev, T):
    return ((X * np.exp(-r * T) * norm.cdf(-d2(S, X, r, stdev, T))) - S * norm.cdf(-d1(S, X, r, std

# Use the last actual stock price as the current price
S_actual = actual_stock_prices[-1]

# Calculate the standard deviation for the actual stock returns
stdev_actual = np.std(actual_stock_prices) * 250 ** 0.5

# Set other Black-Scholes Model parameters
r = 0.025
X_option = 48
T_option = 1

# Calculate d1 and d2 for option pricing
d1_option = d1(S_actual, X_option, r, stdev_actual, T_option)
d2_option = d2(S_actual, X_option, r, stdev_actual, T_option)

# Estimate call and put option prices using BSM
call_option_price_actual = BSM_call(S_actual, X_option, r, stdev_actual, T_option)
put_option_price_actual = BSM_put(S_actual, X_option, r, stdev_actual, T_option)

# Display the estimated option prices using actual values
print(f"Estimated Call Option Price (Actual): {call_option_price_actual}")
print(f"Estimated Put Option Price (Actual): {put_option_price_actual}")
```

```
Estimated Call Option Price (Actual): 94.20999908447266
Estimated Put Option Price (Actual): 46.814875777359966
```

In [ ]:

In [44]:

```
# Assuming 'actual_stock_prices' and 'predicted_stock_prices' are available
# Modify the variable names if needed

# Function to calculate BSM option prices
def calculate_option_prices(S, X, r, stdev, T):
    d1_val = (np.log(S / X) + (r + stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))
    d2_val = d1_val - stdev * np.sqrt(T)

    call_price = (S * norm.cdf(d1_val)) - (X * np.exp(-r * T) * norm.cdf(d2_val))
    put_price = (X * np.exp(-r * T) * norm.cdf(-d2_val)) - (S * norm.cdf(-d1_val))

    return call_price, put_price

# Set common Black-Scholes Model parameters
r = 0.025
X_option = 48
T_option = 1

# Use the last actual and predicted stock prices as the current prices
S_actual = actual_stock_prices[-1]
S_predicted = predicted_stock_prices[-1]

# Calculate the standard deviation for actual and predicted stock returns
stdev_actual = np.std(actual_stock_prices) * 250 ** 0.5
stdev_predicted = np.std(predicted_stock_prices) * 250 ** 0.5

# Calculate d1 and d2 for option pricing
d1_actual = d1(S_actual, X_option, r, stdev_actual, T_option)
d2_actual = d2(S_actual, X_option, r, stdev_actual, T_option)
d1_predicted = d1(S_predicted, X_option, r, stdev_predicted, T_option)
d2_predicted = d2(S_predicted, X_option, r, stdev_predicted, T_option)

# Estimate call and put option prices using BSM for actual and predicted
call_price_actual, put_price_actual = calculate_option_prices(S_actual, X_option, r, stdev_actual,
call_price_predicted, put_price_predicted = calculate_option_prices(S_predicted, X_option, r, stdev

# Create a DataFrame for option price comparison
option_prices_df = pd.DataFrame({
    'Option Type': ['Call', 'Put'],
    'Actual Price': [call_price_actual, put_price_actual],
    'Predicted Price': [call_price_predicted, put_price_predicted]
})

# Display the DataFrame
option_prices_df
```

Out[44]:

	Option Type	Actual Price	Predicted Price
0	Call	94.209999	93.737522
1	Put	46.814876	46.814876

Type *Markdown* and LaTeX: α^2

The comparison of actual and predicted option prices using the Black-Scholes Model provides insights into how well your model's predicted stock prices align with the theoretical pricing model for financial options. Here are some observations:

1. **Call Option:**
   - **Actual vs. Predicted Price:** The actual and predicted Call option prices are close, with a relatively small difference (94.209999 vs. 93.737522). This suggests that your model's predicted stock prices are capturing the underlying dynamics of the stock well enough to yield option prices in line with the Black-Scholes Model.
2. **Put Option:**
   - **Actual vs. Predicted Price:** The actual and predicted Put option prices are identical (46.814876). This indicates a very close alignment between the model's predictions and the Black-Scholes Model for Put option pricing.
3. **Overall Assessment:**
   - The similarity between actual and predicted option prices suggests that your model's predictions are providing reasonable estimates for stock prices, as these predictions translate well into option pricing according to the Black-Scholes Model.

- The close alignment is a positive indicator, indicating that your model captures essential factors influencing option prices, such as stock volatility, time to expiration, and the difference between the stock price and the strike price.

4. **Potential Considerations:**

   - Continue monitoring the performance of your model on different datasets to ensure its generalization capability.
   - Explore additional metrics or statistical tests to quantify the level of alignment and further validate the model's accuracy in predicting option prices.

In summary, the close agreement between actual and predicted option prices implies that your model's predictions align well with the theoretical framework of the Black-Scholes Model in terms of option pricing. This alignment is crucial for

In [ ]:

VAR

In [47]:
```
import numpy as np
from scipy.stats import norm

# Assuming 'actual_stock_prices' and 'predicted_stock_prices' are available
# Modify the variable names if needed

# Function to calculate VaR for a given confidence level
def calculate_var(stock_prices, confidence_level=0.95):
    returns = np.diff(stock_prices) / stock_prices[:-1]  # Calculate daily returns
    log_returns = np.log(1 + returns)

    # Calculate mean and standard deviation of returns
    mean_return = np.mean(log_returns)
    std_dev_return = np.std(log_returns)

    # Calculate VaR using the inverse of the cumulative distribution function (CDF)
    z_score = norm.ppf(1 - confidence_level)
    var_value = stock_prices[-1] * np.exp(mean_return - std_dev_return * z_score)

    return var_value

# Calculate VaR for actual and predicted stock prices
var_actual = calculate_var(actual_stock_prices)
var_predicted = calculate_var(predicted_stock_prices)

# Display VaR values
print(f"VaR for Actual Stock Prices: {var_actual}")
print(f"VaR for Predicted Stock Prices: {var_predicted}")
```

```
VaR for Actual Stock Prices: 146.19133714542278
VaR for Predicted Stock Prices: 145.55287825271125
```

In [ ]:

Probability of Default

In [52]:
```python
# Assuming 'actual_stock_prices' and 'predicted_stock_prices' are available
# Modify the variable names if needed

# Function to calculate Probability of Default (PD) based on stock prices
def calculate_pd(stock_prices, threshold=0.8):
    returns = np.diff(stock_prices) / stock_prices[:-1]  # Calculate daily returns
    log_returns = np.log(1 + returns)

    # Assuming a simple threshold-based approach
    default_events = np.sum(log_returns < threshold)
    total_events = len(log_returns)

    # Calculate Probability of Default (PD)
    pd_value = default_events / total_events

    return pd_value

# Calculate PD for actual and predicted stock prices
pd_actual = calculate_pd(actual_stock_prices)
pd_predicted = calculate_pd(predicted_stock_prices)

# Display PD values
print(f"Probability of Default (PD) for Actual Stock Prices: {pd_actual}")
print(f"Probability of Default (PD) for Predicted Stock Prices: {pd_predicted}")
```

```
Probability of Default (PD) for Actual Stock Prices: 1.0
Probability of Default (PD) for Predicted Stock Prices: 1.0
```

In [ ]: