ADS_505_Final_Team_Project_Team_6: Amy Ou, John Vincent Deniega, & Ghassan Seba

TSLA Price Prediction

TSLA closed below opening price = 0

Ref: https://github.com/vanguardfox/ADS-505-Group6-2023.git

Problem statement and justification for the proposed approach: According to Forbes (2023) and US News (2023), our competitors in JP Morgan to Vanguard Group leverage artificial intelligence to augment their investment strategies. In order to keep pace and potentially overmatch our competitors, we must also deploy artificial intelligence applications and data science mining practices to our investment activities. In this test case, Tesla, Inc. stock price and other market data are used as a proof-of-concept to both understand and predict price action ahead of our competitors. This includes data sources from the NASDAQ stock exchange, the Standard and Poor's 500 index, and US Federal Reserve interest rate decisions.

Import Packages

In [1]:
```
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from sklearn.linear_model import LogisticRegressionCV, LinearRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    mean_squared_error,
    mean_absolute_error,
    r2_score,
    precision_score,
    confusion_matrix,
    ConfusionMatrixDisplay,
)
```

```
from sklearn.neural_network import MLPClassifier
from sklearn.tree import plot_tree
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from mpl_toolkits.mplot3d import Axes3D


%matplotlib inline


#from datetime import date
```

In [2]:
```
# To ignore the warning messages
warnings.filterwarnings('ignore')
```

Reading Data and Checking Data Characteristics

In [3]:
```
# Read data into data frames
# df = pd.read_csv('/Users/johnvincent/Desktop/\
# usd_ADS/github/ADS-505-Group6-2023/TSLA_preprocessed.csv')
df = pd.read_csv('TSLA_preprocessed.csv')
```

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

Out[4]:

	Date	Open_TSLA	High_TSLA	Low_TSLA	Close_TSLA	Adj_Close_TSLA	Volume_TSLA	EPS_qtr
0	3/11/22	280.066681	281.266663	264.589996	265.116669	265.116669	67037100	0.68
1	3/14/22	260.203339	266.899994	252.013336	255.456665	255.456665	71152200	0.68
2	3/15/22	258.423340	268.523346	252.190002	267.296661	267.296661	66841200	0.68
3	3/16/22	269.666656	280.666656	267.420013	280.076660	280.076660	84028800	0.68
4	3/17/22	276.996674	291.666656	275.239990	290.533325	290.533325	66582900	0.68

5 rows × 28 columns

In [5]:
```
df['Date'] = pd.to_datetime(df['Date']) #convert to datetime64 data type
df['Date']
```

Out[5]:
```
0     2022-03-11
1     2022-03-14
2     2022-03-15
3     2022-03-16
4     2022-03-17
         ...
375   2023-09-08
376   2023-09-08
377   2023-09-11
378   2023-09-12
379   2023-09-13
Name: Date, Length: 380, dtype: datetime64[ns]
```

Add column for day of the week

'0' is 'Monday' and '4' is 'Friday'

There should be no '5' or '6' due to market close.

TSLA closed below opening price = 0

Ref: Click For reference

In [6]:
```
df.insert(1, 'day_week', df['Date'].dt.dayofweek)
df.head()
```

Out[6]:

	Date	day_week	Open_TSLA	High_TSLA	Low_TSLA	Close_TSLA	Adj_Close_TSLA	Volume_TSLA
0	2022-03-11	4	280.066681	281.266663	264.589996	265.116669	265.116669	67037100
1	2022-03-14	0	260.203339	266.899994	252.013336	255.456665	255.456665	71152200
2	2022-03-15	1	258.423340	268.523346	252.190002	267.296661	267.296661	66841200
3	2022-03-16	2	269.666656	280.666656	267.420013	280.076660	280.076660	84028800
4	2022-03-17	3	276.996674	291.666656	275.239990	290.533325	290.533325	66582900

5 rows × 29 columns

Add column on the difference between open and close TSLA price

In [7]:
```
df.insert(7, 'open_close_TSLA', df.iloc[df.index]['Close_TSLA'] - \
          df.iloc[df.index]['Open_TSLA'])
df.head()
```

Out[7]:

	Date	day_week	Open_TSLA	High_TSLA	Low_TSLA	Close_TSLA	Adj_Close_TSLA	open_close_TSL
0	2022-03-11	4	280.066681	281.266663	264.589996	265.116669	265.116669	-14.95001
1	2022-03-14	0	260.203339	266.899994	252.013336	255.456665	255.456665	-4.74667
2	2022-03-15	1	258.423340	268.523346	252.190002	267.296661	267.296661	8.87332
3	2022-03-16	2	269.666656	280.666656	267.420013	280.076660	280.076660	10.41000
4	2022-03-17	3	276.996674	291.666656	275.239990	290.533325	290.533325	13.53665

5 rows × 30 columns

Add binary column if:

TSLA closed at or above opening price = 1

TSLA closed below opening price = 0

For future use in logistic regression or classification purposes

In [8]:
```
df.insert(8, 'positive_TSLA', np.where(df['open_close_TSLA']>=0, 1, 0))
df.head()
```

Out[8]:

	Date	day_week	Open_TSLA	High_TSLA	Low_TSLA	Close_TSLA	Adj_Close_TSLA	open_close_TSL
0	2022-03-11	4	280.066681	281.266663	264.589996	265.116669	265.116669	-14.95001
1	2022-03-14	0	260.203339	266.899994	252.013336	255.456665	255.456665	-4.74667
2	2022-03-15	1	258.423340	268.523346	252.190002	267.296661	267.296661	8.87332
3	2022-03-16	2	269.666656	280.666656	267.420013	280.076660	280.076660	10.41000
4	2022-03-17	3	276.996674	291.666656	275.239990	290.533325	290.533325	13.53665

5 rows × 31 columns

In [9]:
```
df.shape # Check the data dimension
```

Out[9]:
```
(380, 31)
```

In [10]:
```
df.columns # Get attribute names
```

Out[10]:
```
Index(['Date', 'day_week', 'Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Close_TSLA',
       'Adj_Close_TSLA', 'open_close_TSLA', 'positive_TSLA', 'Volume_TSLA',
       'EPS_qtr', 'Revenue', 'oper_cash_fl', 'gross_mrgn', 'oper_mrgn',
       'net_mrgn', 'pe_ann', 'pe_qtr', 'fed_funds_rate', 'Open_NDQ',
       'High_NDQ', 'Low_NDQ', 'Close_NDQ', 'Adj Close_NDQ', 'Volume_NDQ',
       'Open_SPX', 'High_SPX', 'Low_SPX', 'Close_SPX', 'Adj_Close_SPX',
       'Volume_SPX'],
      dtype='object')
```

In [11]:
```
df.info() # Check data characteristics
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 380 entries, 0 to 379
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Date             380 non-null    datetime64[ns]
 1   day_week         380 non-null    int64
 2   Open_TSLA        380 non-null    float64
 3   High_TSLA        380 non-null    float64
 4   Low_TSLA         380 non-null    float64
 5   Close_TSLA       380 non-null    float64
 6   Adj_Close_TSLA   380 non-null    float64
 7   open_close_TSLA  380 non-null    float64
 8   positive_TSLA    380 non-null    int32
 9   Volume_TSLA      380 non-null    int64
 10  EPS_qtr          380 non-null    float64
 11  Revenue          380 non-null    int64
 12  oper_cash_fl     380 non-null    int64
 13  gross_mrgn       380 non-null    float64
 14  oper_mrgn        380 non-null    float64
 15  net_mrgn         380 non-null    float64
 16  pe_ann           380 non-null    float64
 17  pe_qtr           380 non-null    float64
 18  fed_funds_rate   380 non-null    float64
 19  Open_NDQ         380 non-null    float64
 20  High_NDQ         380 non-null    float64
 21  Low_NDQ          380 non-null    float64
 22  Close_NDQ        380 non-null    float64
 23  Adj Close_NDQ    380 non-null    float64
 24  Volume_NDQ       380 non-null    int64
 25  Open_SPX         380 non-null    float64
 26  High_SPX         380 non-null    float64
 27  Low_SPX          380 non-null    float64
 28  Close_SPX        380 non-null    float64
 29  Adj_Close_SPX    380 non-null    float64
 30  Volume_SPX       380 non-null    int64
dtypes: datetime64[ns](1), float64(23), int32(1), int64(6)
memory usage: 90.7 KB
```

Exploratory Data Analysis

Checking Missing Values

In [12]:
```
df.isnull().sum() # Checking Null Values
```

Out[12]:
```
Date                0
day_week            0
Open_TSLA           0
High_TSLA           0
Low_TSLA            0
Close_TSLA          0
Adj_Close_TSLA      0
open_close_TSLA     0
positive_TSLA       0
Volume_TSLA         0
EPS_qtr             0
Revenue             0
oper_cash_fl        0
gross_mrgn          0
oper_mrgn           0
net_mrgn            0
pe_ann              0
pe_qtr              0
fed_funds_rate      0
Open_NDQ            0
High_NDQ            0
Low_NDQ             0
Close_NDQ           0
Adj Close_NDQ       0
Volume_NDQ          0
Open_SPX            0
High_SPX            0
Low_SPX             0
Close_SPX           0
Adj_Close_SPX       0
Volume_SPX          0
dtype: int64
```

0 NaN values in the dataframe

Checking duplicated Values

In [13]:
```
df.duplicated().sum()
```

Out[13]:
```
0
```

0 duplicated values in the dataframe

Understanding of price range categories

In [14]:
```
df.nunique() #check cardinality to see if variable type fits
```

Out[14]:

```
Date                379
day_week              5
Open_TSLA           375
High_TSLA           373
Low_TSLA            375
Close_TSLA          377
Adj_Close_TSLA      377
open_close_TSLA     372
positive_TSLA         2
Volume_TSLA         379
EPS_qtr               6
Revenue               7
oper_cash_fl          7
gross_mrgn            6
oper_mrgn             7
net_mrgn              7
pe_ann              378
pe_qtr              378
fed_funds_rate       12
Open_NDQ            380
High_NDQ            380
Low_NDQ             380
Close_NDQ           380
Adj Close_NDQ       380
Volume_NDQ          380
Open_SPX            380
High_SPX            378
Low_SPX             379
Close_SPX           379
Adj_Close_SPX       379
Volume_SPX          380
dtype: int64
```

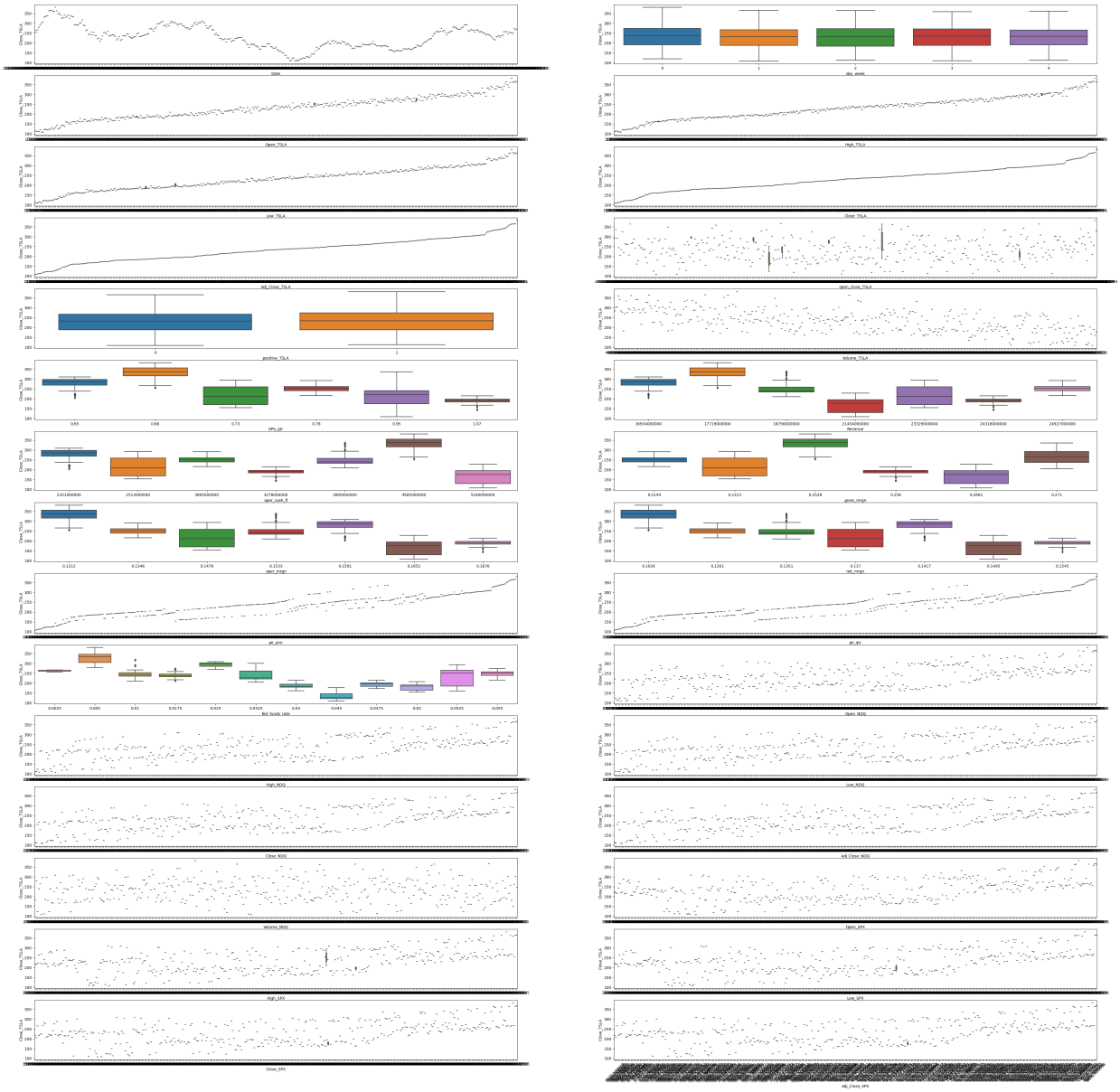
In [15]:

```
# Ref. https://towardsdatascience.com/how-to-perform-exploratory-data-analysis-with-se
# Divide features into arrays by type
columns = ['Date', 'day_week', 'Open_TSLA', 'High_TSLA', 'Low_TSLA',
           'Close_TSLA', 'Adj_Close_TSLA', 'open_close_TSLA',
           'positive_TSLA', 'Volume_TSLA', 'EPS_qtr', 'Revenue',
           'oper_cash_fl', 'gross_mrgn', 'oper_mrgn', 'net_mrgn',
           'pe_ann', 'pe_qtr','fed_funds_rate', 'Open_NDQ',
           'High_NDQ', 'Low_NDQ', 'Close_NDQ','Adj Close_NDQ',
           'Volume_NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX',
           'Close_SPX', 'Adj_Close_SPX', 'Volume_SPX'] #32 columns

tesla = ['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Close_TSLA',
       'Adj_Close_TSLA','open_close_TSLA', 'positive_TSLA',
         'Volume_TSLA', 'EPS_qtr', 'Revenue', 'oper_cash_fl',
       'gross_mrgn', 'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr',] #16 columns

nasdaq = ['Open_NDQ', 'High_NDQ', 'Low_NDQ', 'Close_NDQ',
       'Adj Close_NDQ', 'Volume_NDQ'] #6 columns

spx = ['Open_SPX', 'High_SPX', 'Low_SPX',
       'Close_SPX', 'Adj_Close_SPX', 'Volume_SPX'] #6 columns
```

Data Visualization: Analyzing the Relationship Between

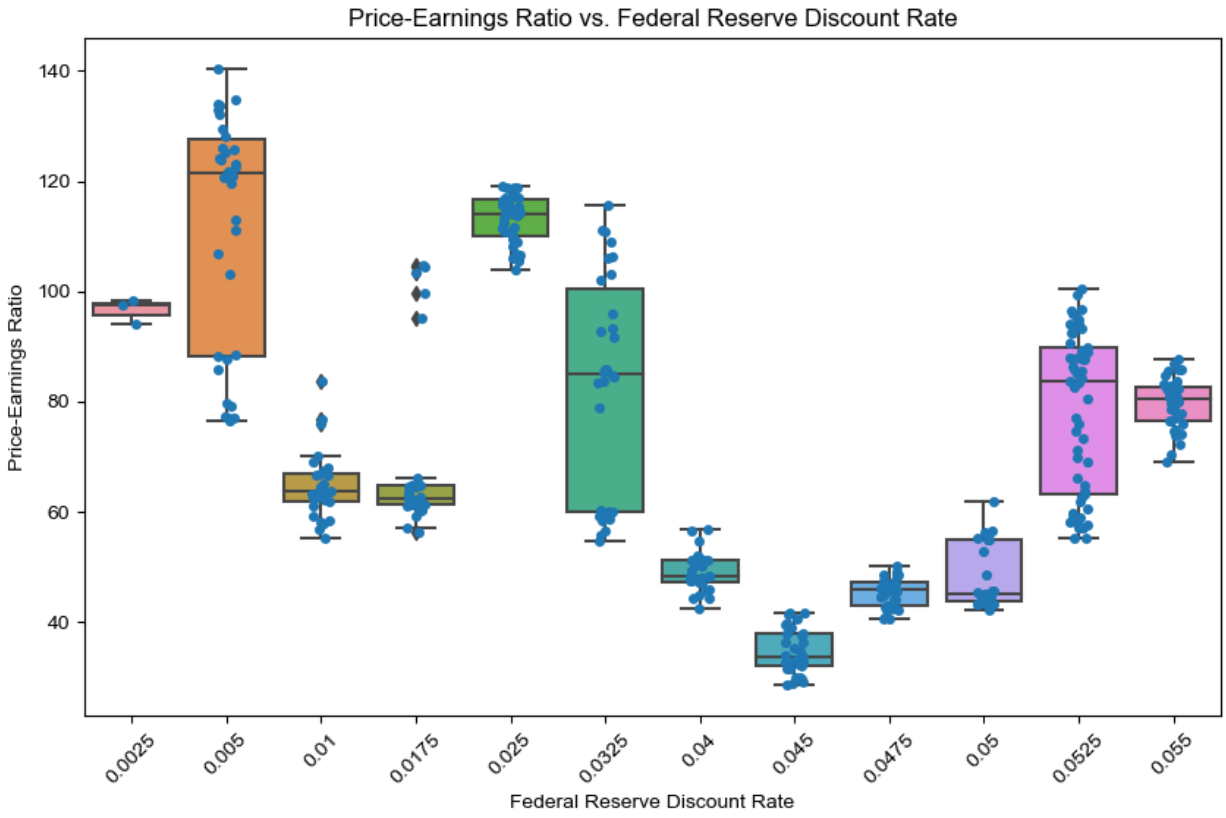
Variables

In [16]:
```
# Ref. https://towardsdatascience.com/how-to-perform-exploratory
# -data-analysis-with-seaborn-97e3413e841d
# Generate a 15-by-2 grid of subplots to display

fig, ax = plt.subplots(15,2, figsize=(50,50))

target = 'Close_TSLA'

# FOR loop that iterates through each feature in the "columns" set and zips it into a
# Each subplot uses each feature and plots the respective feature as a boxplot measuri
for var, subplot in zip(columns, ax.flatten()):
    sns.boxplot(x=var, y=target, data=df, ax=subplot)
    plt.xticks(rotation=45)
plt.show()
```



Drill Down: Explore P/E ratio vs. US Federal Reserve Discount Rate

In [17]:
```
target_y = 'pe_ann'
column_x = 'fed_funds_rate'

plt.figure(figsize=(10,6))
plt.setp(sns.boxplot(x=column_x, y=target_y, data=df).\
         get_xticklabels(), rotation=45)
sns.stripplot(x=column_x, y=target_y, data=df)
sns.set_style("darkgrid")
plt.title("Price-Earnings Ratio vs. Federal Reserve Discount Rate")
plt.xlabel("Federal Reserve Discount Rate")
plt.ylabel("Price-Earnings Ratio")
plt.show()
```



In [18]:
```
fed_stats = df.groupby('fed_funds_rate').describe().transpose()
fed_stats.loc['pe_ann']
```

Out[18]:

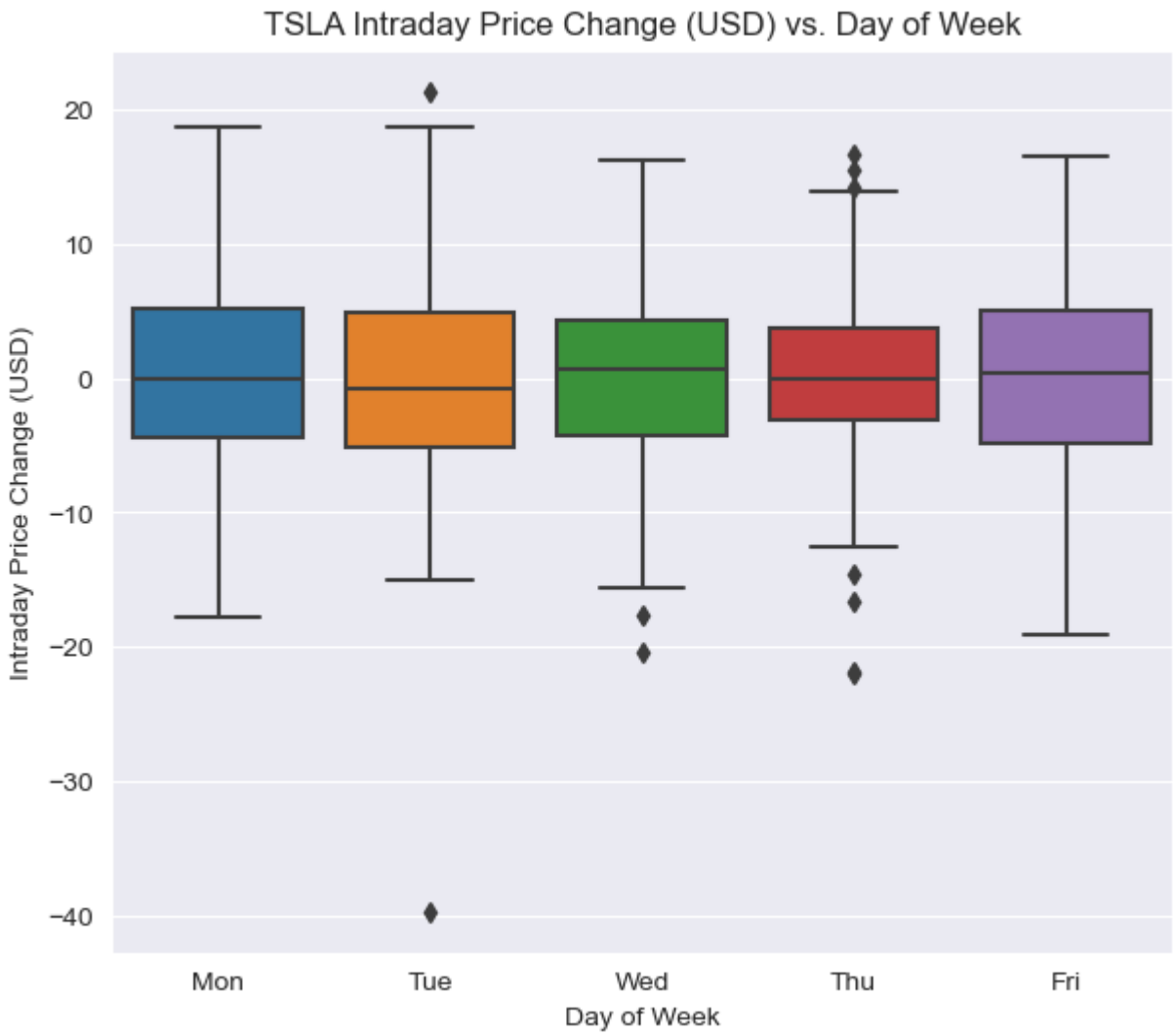
fed_funds_rate	0.0025	0.0050	0.0100	0.0175	0.0250	0.0325	0.0400	0.(
count	3.000000	34.000000	29.000000	28.000000	39.000000	30.000000	29.000000	32.00
mean	96.552695	111.857331	64.635783	68.900195	112.952335	83.007139	48.960889	34.51
std	2.316729	21.408135	6.194704	15.677199	4.317682	20.162340	3.470707	4.03
min	93.917892	76.382454	55.101756	56.078949	103.926920	54.547368	42.355262	28.44
25%	95.693628	88.244079	61.714911	61.293640	109.894233	59.897368	47.118422	31.92
50%	97.469364	121.380514	63.541228	62.378508	114.020515	84.982693	48.202631	33.60
75%	97.870097	127.617342	66.807893	64.624780	116.580769	100.499039	51.236841	37.83
max	98.270831	140.373780	83.563160	104.708980	118.969234	115.692303	56.660526	41.56

Interpretation: TSLA's Price-Earnings (PE) ratio exhibits large differences in both central tendencies and variation across different US Federal Reserve interest discount rates, suggesting that TSLA stock is highly sensitive to monetary policy changes.

Supporting Evidence: Throughout the range of interest discount rates, the median PE ratio went as low as 33.61 at 4.5 percent to as high as 121.38 at 0.5%. The PE standard deviation during the period of increase beginning March 2022 through September 2023 ranged from a low of 2.59 at 4.75 percent to 21.41 at 0.5%, demonstrating significant changes in PE variation and supporting the thesis of TSLA's tendency for changes in volatility. Because the movement between the high and low is not monotonic, it cannot yet be concluded that the two are covariant, but the data support that an interest change occurring in itself with all else being equal causes the market to reset and reprice TSLA shares.

Explore relationship between PE ratio and the trading day of the week

In [19]:
```
target_y = 'open_close_TSLA'
column_x = 'day_week'

plt.figure(figsize=(7, 6))
sns.boxplot(x=column_x, y=target_y, data=df)
sns.set_style("whitegrid")
plt.title("TSLA Intraday Price Change (USD) vs. Day of Week")
plt.xlabel("Day of Week")
plt.xticks([0,1,2,3,4],['Mon','Tue','Wed','Thu','Fri'])
plt.ylabel("Intraday Price Change (USD)")
plt.show()
```



Statistics on the open-close TSLA column by day of the week

In [20]:
```
day_week_stats = df.groupby('day_week').describe().transpose()
display(day_week_stats.loc['open_close_TSLA'])
upper_IQR = day_week_stats.loc['open_close_TSLA'].loc['75%']
lower_IQR = day_week_stats.loc['open_close_TSLA'].loc['25%']
IQR = upper_IQR-lower_IQR
print("Interquartile Range of Intraday TSLA Price Changes\n")
IQR
```

day_week	0	1	2	3	4
count	68.000000	78.000000	79.000000	77.000000	78.000000
mean	-0.018677	-0.473930	0.029958	-0.209610	-0.269958
std	7.253562	8.599311	7.299539	7.823672	7.671168
min	-17.759995	-39.669983	-20.333328	-21.983337	-19.019990
25%	-4.479168	-5.168335	-4.269997	-3.179993	-4.812504
50%	-0.025009	-0.834992	0.716660	-0.036667	0.415001
75%	5.107502	4.852505	4.339996	3.760010	5.092506
max	18.690003	21.326660	16.226685	16.599991	16.449982

```
Interquartile Range of Intraday TSLA Price Changes
```

Out[20]:
```
day_week
0     9.586670
1    10.020840
2     8.609993
3     6.940003
4     9.905011
dtype: float64
```

Key Findings:

1. Wednesdays exhibit a tendency for positive price action (positive median).

Business Recommendation: When possible, buy shares or CALL options on Wednesdays.

2. Thursdays exhibit the lowest volatility (minimum interquartile range).

Business Recommendation: Deploy short-dated "Iron Condor" strategy to take advantage of lower volatility tendency.

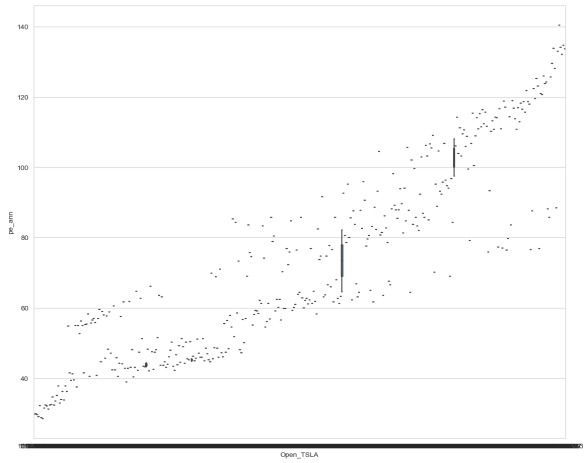
3. Tuesdays exhibit the greatest volatility (maximum interquartile range).

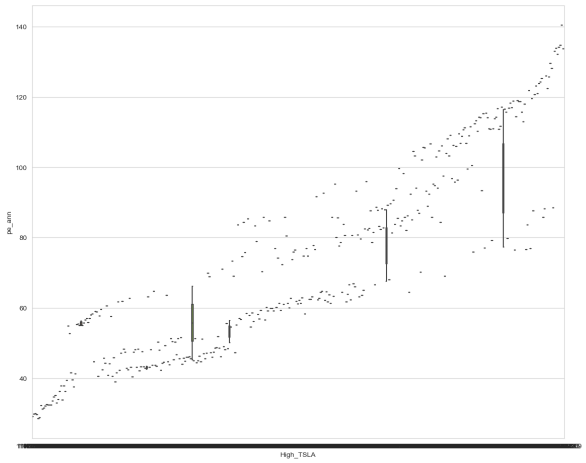
Business Recommendation: Exercise increased hedging on Tuesdays in order to account for higher volatility tendencies.

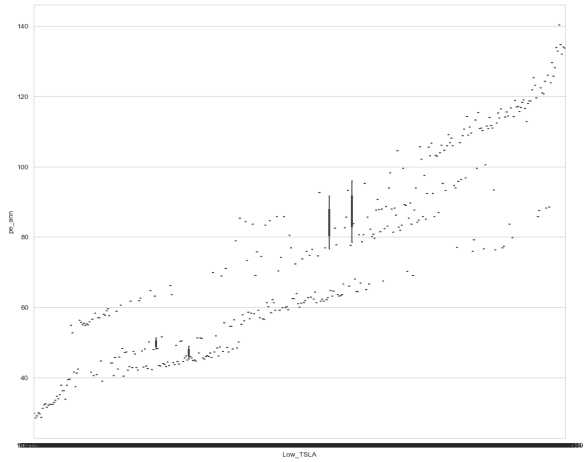
Use a FOR loop to run analyze P/E Ratio vs. the 'tesla' variable set (defined above)

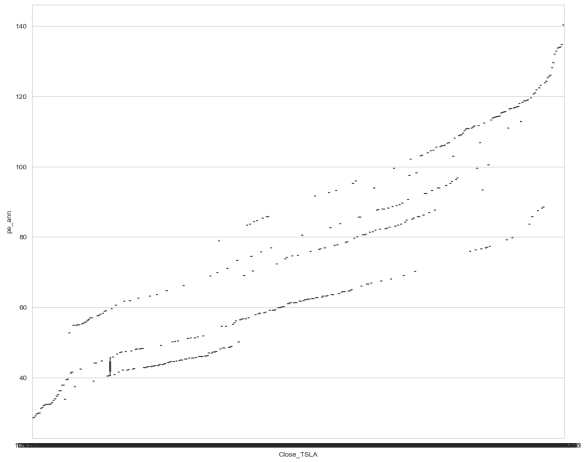
In [21]:

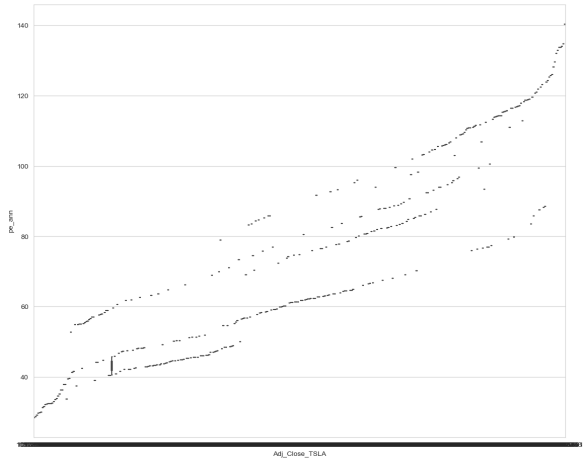
```
# Ref. https://towardsdatascience.com/how-to-perform-
# exploratory-data-analysis-with-seaborn-97e3413e841d
# Generate a 8-by-2 grid of subplots to display

fig, ax = plt.subplots(8,2, figsize=(30,105))

target = 'pe_ann'

plt.xticks(rotation=45)

# FOR loop that iterates through each feature in the "categorical"
# set and zips it into a tuple with a subplot
# Each subplot uses each feature and plots the respective
# feature as a boxplot measuring price range.
for var, subplot in zip(tesla, ax.flatten()):
    sns.boxplot(x=var, y=target, data=df, ax=subplot)
plt.show()
```

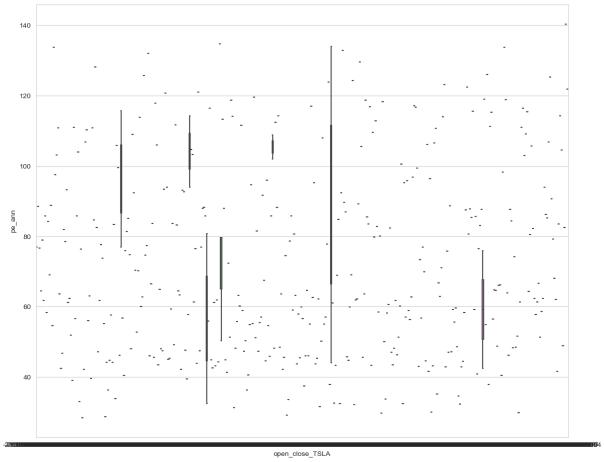

pe_ann
Open_TSLA

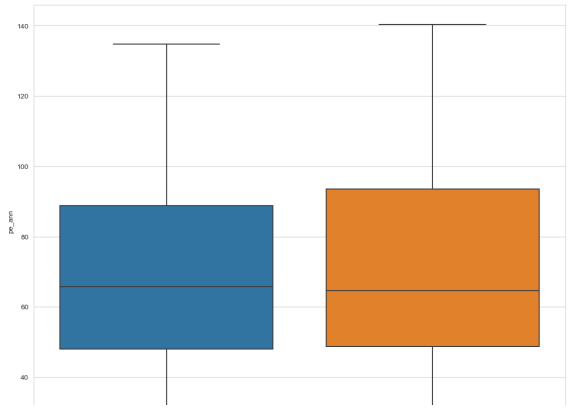

pe_ann
High_TSLA

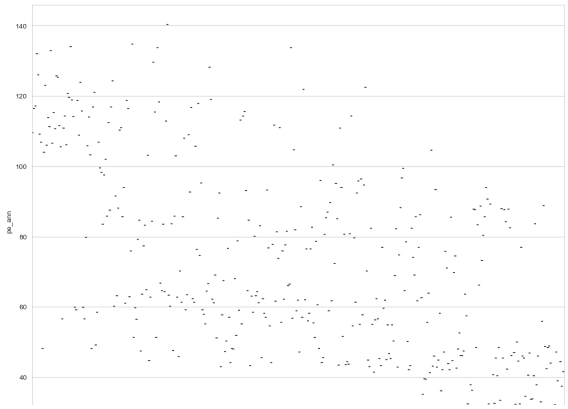

pe_ann
Low_TSLA


pe_ann
Close_TSLA


pe_ann
Adj_Close_TSLA

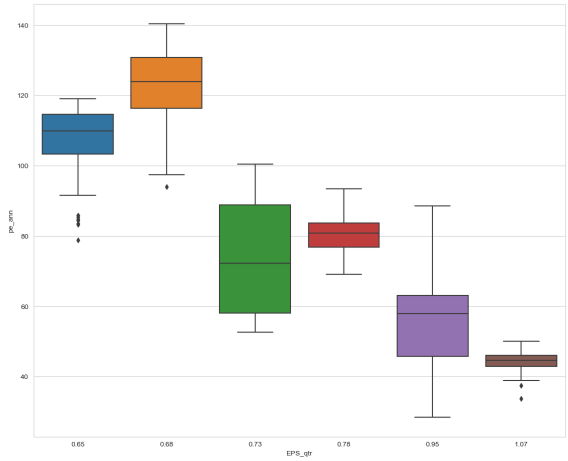

pe_ann
open_close_TSLA

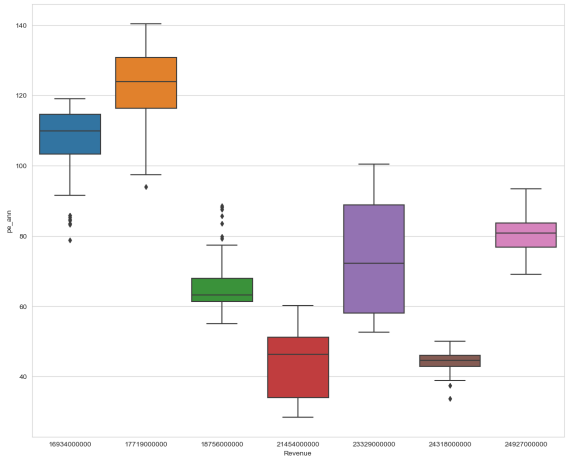

pe_ann

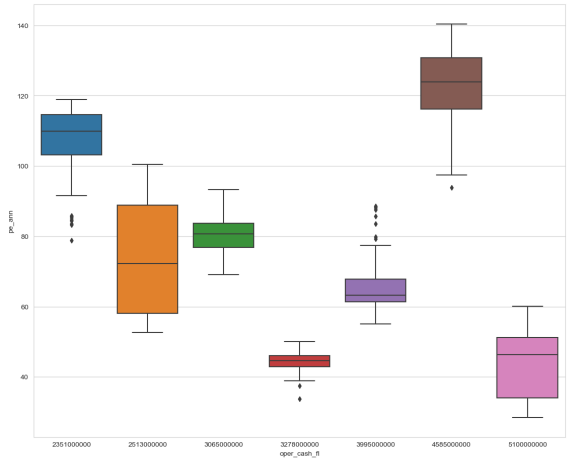

pe_ann

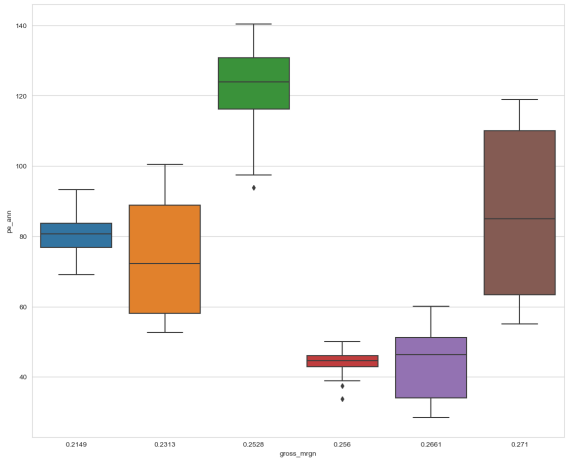

0
1
positive_TSLA

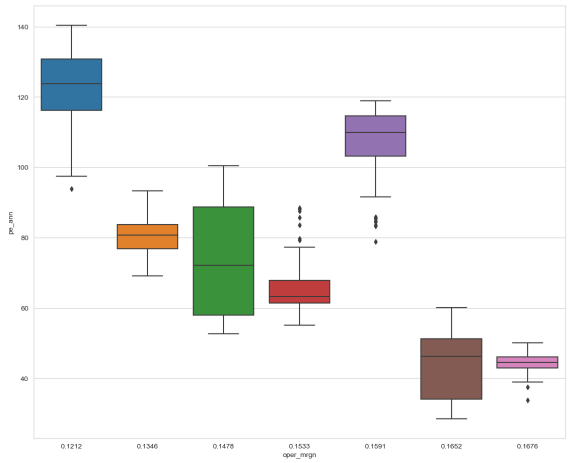

Volume_TSLA

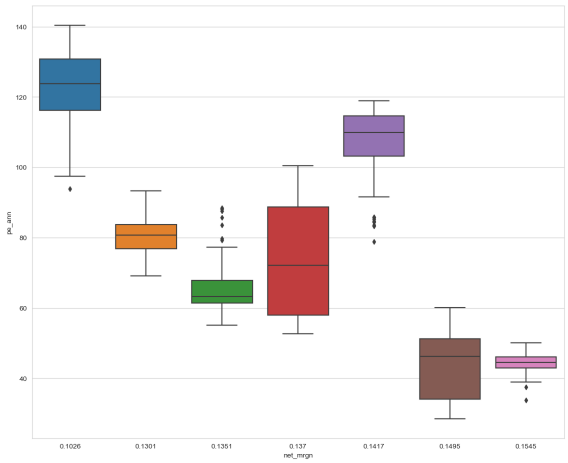

pe_ann
140
120
100
80
60
40
0.65
0.68
0.73
0.78
0.95
1.07
EPS_qtr

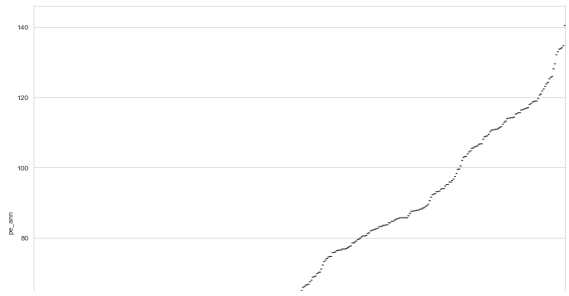

pe_ann
140
120
100
80
60
40
16934000000
17719000000
18756000000
21454000000
23329000000
24318000000
24927000000
Revenue

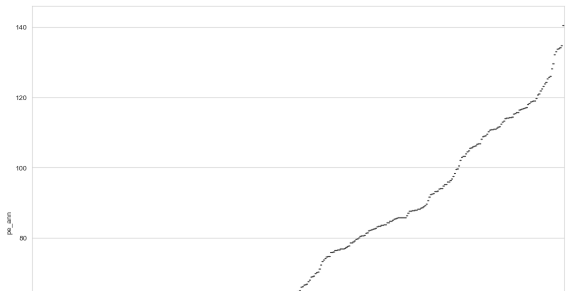

pe_ann
140
120
100
80
60
40
2351000000
2513000000
3065000000
3278000000
3995000000
4585000000
5100000000
oper_cash_fl


pe_ann
140
120
100
80
60
40
0.2149
0.2313
0.2528
0.256
0.2661
0.271
gross_mrgn


pe_ann
140
120
100
80
60
40
0.1212
0.1346
0.1478
0.1533
0.1591
0.1652
0.1676
oper_mrgn

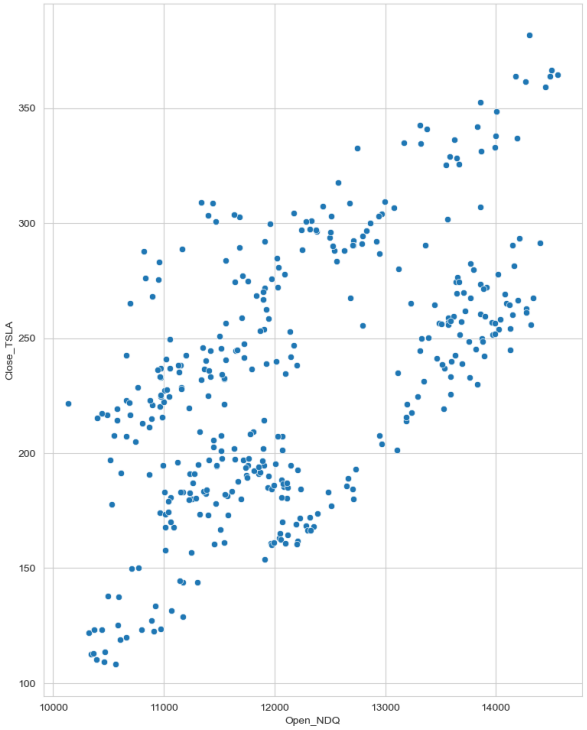

pe_ann
140
120
100
80
60
40
0.1026
0.1301
0.1351
0.137
0.1417
0.1495
0.1545
net_mrgn

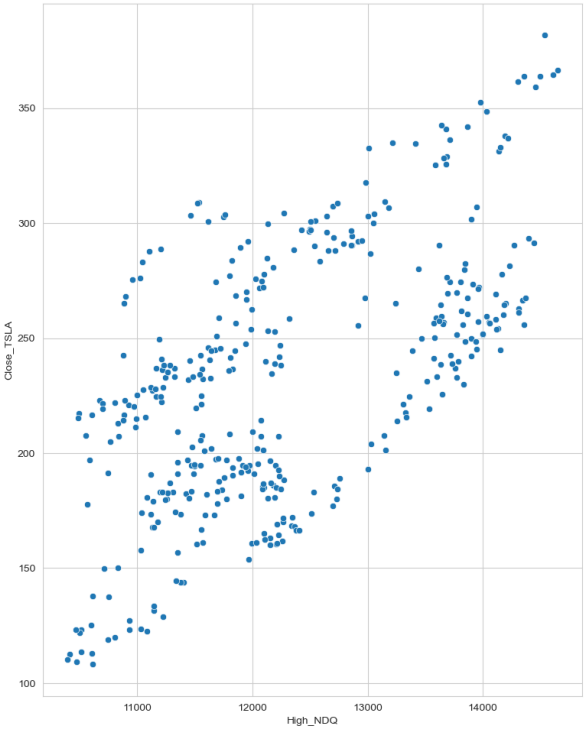

pe_ann
140
120
100
80

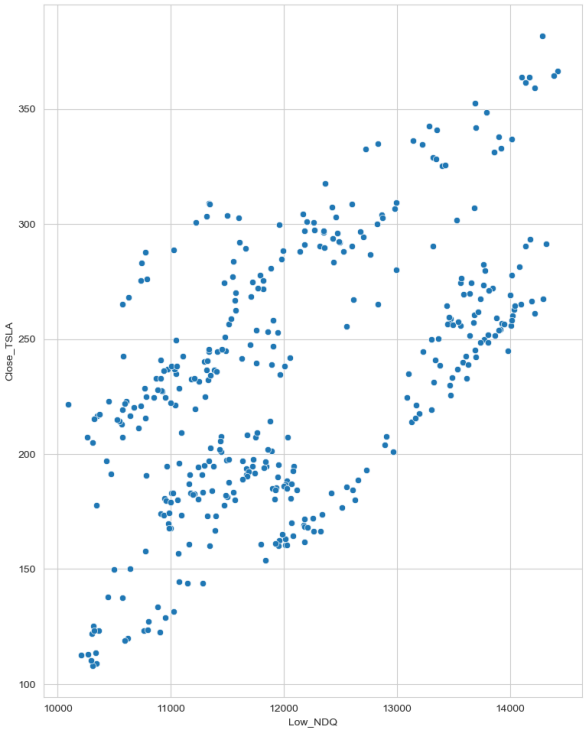

pe_ann
140
120
100
80

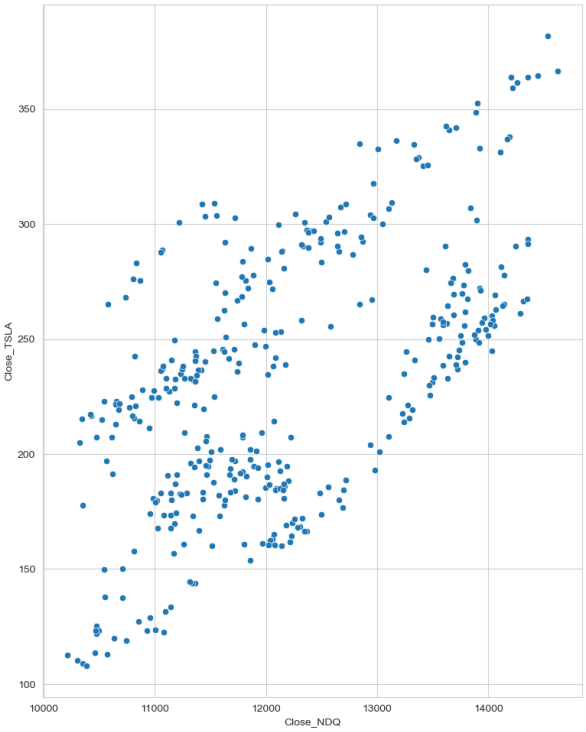
Investigate TSLA's closing price against the 'nasdaq' variable set (above)

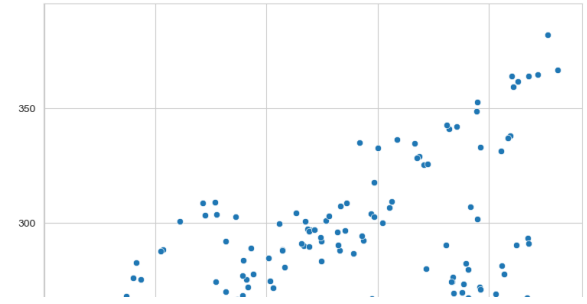
In [22]:
```
# Ref. https://towardsdatascience.com/how-to-perform-
# exploratory-data-analysis-with-seaborn-97e3413e841d
# Generate a 3-by-2 grid of subplots to display
fig, ax = plt.subplots(3,2, figsize=(20,40))

target = 'Close_TSLA'

plt.xticks(rotation=45)

# FOR loop that iterates through each feature in the "nominal" set and zips it into a
# Each subplot uses each feature and plots the respective feature as a boxplot measuri
for var, subplot in zip(nasdaq, ax.flatten()):
    sns.scatterplot(x=var, y=target, data=df, ax=subplot)
plt.show()
```

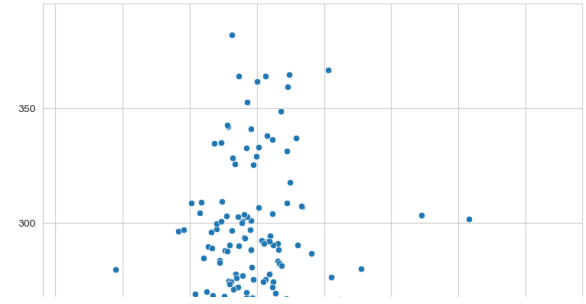

Close_TSLA
Open_NDQ
100
150
200
250
300
350
10000
11000
12000
13000
14000

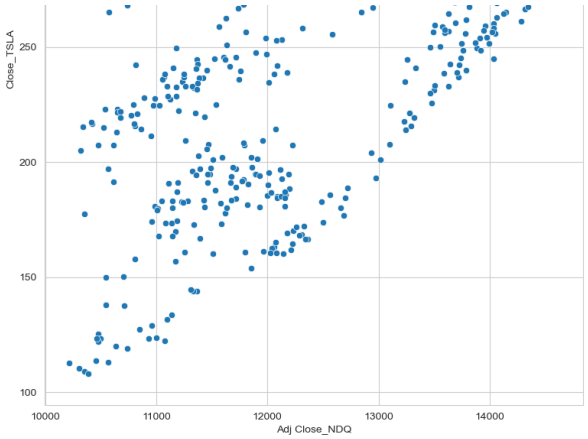

Close_TSLA
High_NDQ
100
150
200
250
300
350
11000
12000
13000
14000

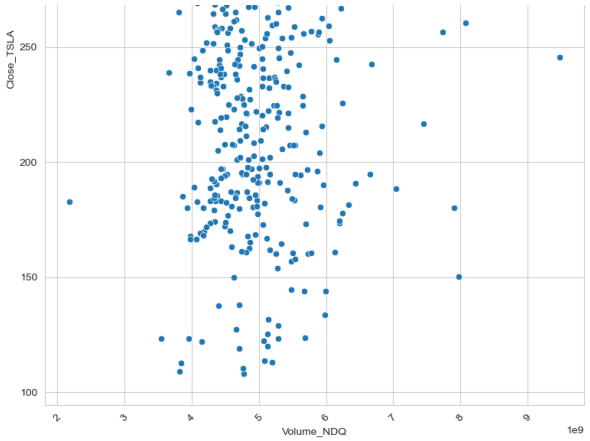

Close_TSLA
Low_NDQ
100
150
200
250
300
350
10000
11000
12000
13000
14000


Close_TSLA
Close_NDQ
100
150
200
250
300
350
10000
11000
12000
13000
14000


300
350

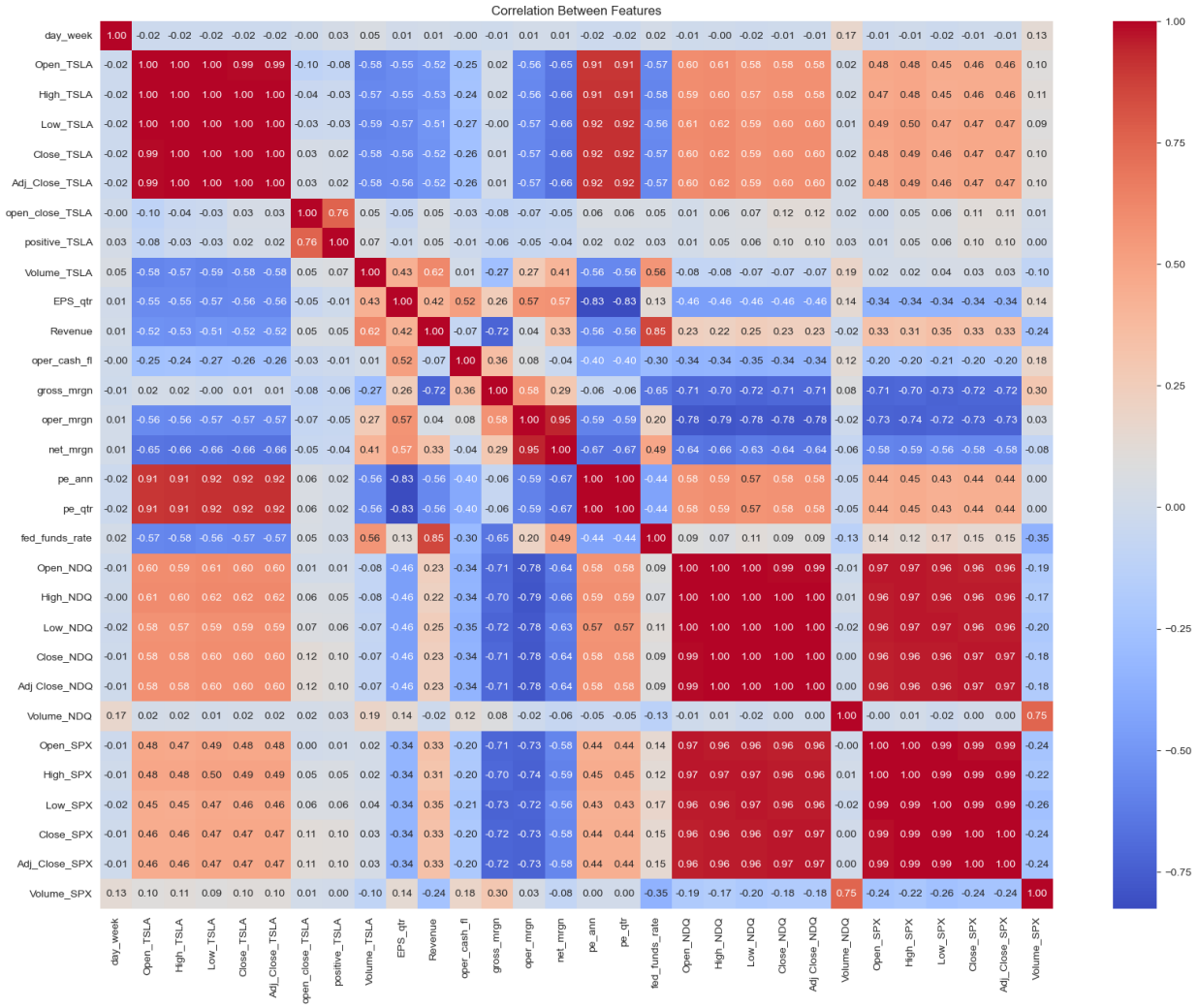

300
350





Heatmap of correlation values by feature

In [23]:
```
plt.figure(figsize=(20, 15))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", \
            fmt='.2f', annot_kws=None)
plt.title('Correlation Between Features')
plt.show()
```



Display highest correlations between target and the other features in our dataset

In [24]:
```
df.corr().abs()['open_close_TSLA'].sort_values(ascending=False)
```

Out[24]:
```
open_close_TSLA    1.000000
positive_TSLA      0.764211
Adj Close_NDQ      0.115798
Close_NDQ          0.115798
Close_SPX          0.111286
Adj_Close_SPX      0.111286
Open_TSLA          0.102563
gross_mrgn         0.080839
Low_NDQ            0.068830
oper_mrgn          0.067029
Low_SPX            0.061890
High_NDQ           0.058220
pe_ann             0.055624
pe_qtr             0.055624
net_mrgn           0.054955
Volume_TSLA        0.053045
Revenue            0.051464
fed_funds_rate     0.050581
High_SPX           0.050329
EPS_qtr            0.048268
High_TSLA          0.042456
oper_cash_fl       0.034188
Adj_Close_TSLA     0.033831
Close_TSLA         0.033831
Low_TSLA           0.028370
Volume_NDQ         0.015990
Volume_SPX         0.010348
Open_NDQ           0.010313
day_week           0.003698
Open_SPX           0.002230
Name: open_close_TSLA, dtype: float64
```

Modeling - Data Preparation

Handle Outliers

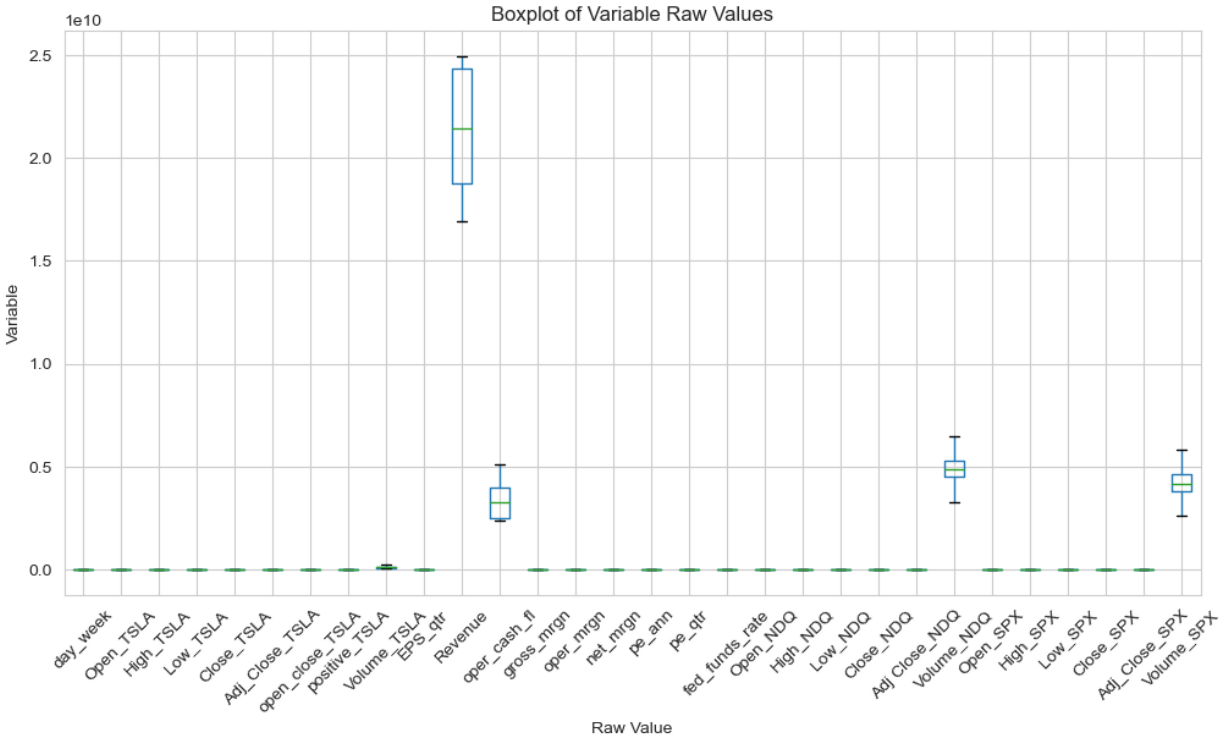
Set outliers to 1.5*IQR max or min to prepare for value normalization using z score

In [25]:
```
df_prep = df.copy().drop(columns='Date')
IQR = df_prep.quantile(.75)-df_prep.quantile(.25)
clipped_min = df_prep.quantile(.25) - (1.5*IQR)
clipped_max = df_prep.quantile(.75) + (1.5*IQR)
outliers = df_prep.columns

# Replace outlier variables with clipped_min or clipped_max, as appropriate
df_prep[outliers] = np.where(df_prep[outliers]<clipped_min[outliers],\
```

```
                                clipped_min[outliers], df_prep[outliers])
df_prep[outliers] = np.where(df_prep[outliers]>clipped_max[outliers],\
                                clipped_max[outliers], df_prep[outliers])
ax = df_prep.boxplot(return_type='axes', rot=45, figsize=(12, 6))
ax.set_xlabel("Raw Value")
ax.set_ylabel("Variable")
ax.set_title("Boxplot of Variable Raw Values")
```

Out[25]:

```
Text(0.5, 1.0, 'Boxplot of Variable Raw Values')
```



*****Outlier values have been handled.**

Standardization

Standardization, or z-score normalization, rescales data to match the standard normal distribution, facilitating data interchange across systems. It simplifies data processing, analysis, and storage, aiding businesses in making informed decisions by enabling easy data comparison and evaluation. Standardization is particularly useful for normally distributed data, and it lacks a bounding range, unlike normalization, which does not impact outliers in the data.

Ref: https://www.simplilearn.com/normalization-vs-standardization-article

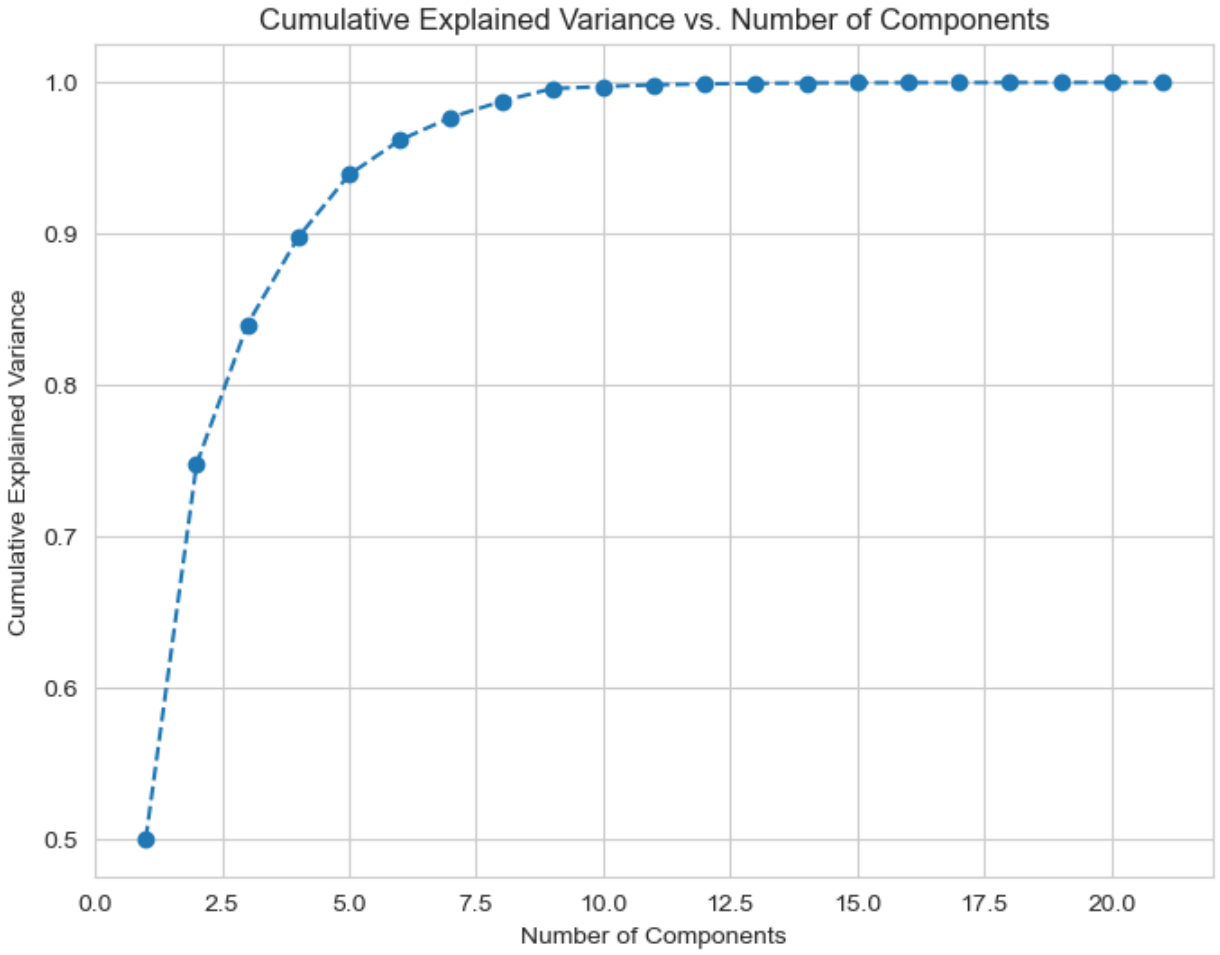
Standardize using stats.zscore

In [26]:
```
# Standardize values to z score now that outliers are handled with clipping
df_z = df_prep.copy()
df_z = stats.zscore(df_z) # Standardize all columns by z score
df_z['positive_TSLA'] = df['positive_TSLA'] # Reset this column back to binary
```

Feature Selection - Principle Component Analysis (PCA)

Data Selection: Identifying features to keep based on variance threshold

In [27]:
```
# Create new df excluding target variables close_TSLA and positive_TSLA
PCA_df =  df_z[['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Volume_TSLA',
                'EPS_qtr', 'Revenue', 'oper_cash_fl', 'gross_mrgn',
                'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr',
                'fed_funds_rate', 'Open_NDQ', 'High_NDQ', 'Low_NDQ',
                'Volume_NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX',
                'Volume_SPX']]

# Initialize PCA with a large number of components
pca_tsla = PCA(n_components=len(PCA_df.columns))

# Fit PCA to the PCA_df data
pca_tsla.fit(PCA_df)

# Calculate the cumulative explained variance
cumulative_variance = np.cumsum(pca_tsla.explained_variance_ratio_)

# Plot the explained variance
plt.figure(figsize=(8, 6))
plt.plot(range(1, len(cumulative_variance) + 1), \
         cumulative_variance, marker='o', linestyle='--')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Cumulative Explained Variance vs. Number of Components')
plt.grid(True)
plt.show()
```



*****Selected threshold = .95**

In [28]:
```
# Create a PCA object with n components
pca_tsla = PCA(n_components=5)

# Fit the PCA model to the PCA data and transform it to the principal components
principalComponents_tsla = pca_tsla.fit_transform(PCA_df)
```

In [29]:
```
# Create a DataFrame 'principal_tsla_Df' with n principal components columns
principal_tsla_Df = pd.DataFrame(data = principalComponents_tsla
             , columns = ['principal component 1', 'principal component 2',
                          'principal component 3', 'principal component 4',
                          'principal component 5'])

# Display the last 5 rows
principal_tsla_Df.head()
```

Out[29]:

	principal component 1	principal component 2	principal component 3	principal component 4	principal component 5
0	3.850764	2.017726	1.940497	-1.362708	-1.901865
1	3.094240	2.269720	2.841905	-0.581519	-2.546669
2	3.244731	2.253335	2.276140	-1.014104	-2.405786
3	3.991752	2.076801	3.663642	0.177781	-2.299092
4	4.673806	1.780332	2.328535	-1.020899	-1.670959

In [30]:
```
# Extract explained variance ratio
explained_variance_ratio = pca_tsla.explained_variance_ratio_

# Round the explained variance ratio to 4 decimal places
rounded_explained_variance_ratio = np.round(explained_variance_ratio, 4)

# Print explained variation per principal component
print('Explained variation per principal component: {}'\
      .format(rounded_explained_variance_ratio))
```

```
Explained variation per principal component: [0.4995 0.248  0.0925 0.0576 0.0414]
```

Principal Component Analysis (PCA) is a technique that enables us to simplify complex data. It achieves this by condensing large datasets into a more compact form while striving to retain essential information. In PCA, we transform data into a set of "principal components," each capturing a different aspect of the original data's variability. The first component explains the most variation in the data, followed by subsequent components. For example, if we see "Explained variation per principal component: [0.4995 0.248 0.0925 0.0576 0.0414]," it means that the first component (PC1) captures 49.95% of the data's complexity, while the second (PC2) covers 24.80%, and so on, highlighting their importance.

We also measure information loss, which indicates how much simplification occurs during the process. We calculate it as follows:

Information Loss = 1 - Total Variance Retained

With five retained components, they collectively preserve 93.90% of the original data's richness:

0.4995 + 0.248 + 0.0925 + 0.0576 + 0.0414 = 0.9390

Information Loss = 1 − 0.9390 = 0.0610

Therefore, the information loss is minimal, just about 6.10%.

In summary, PCA offers a way to simplify complex data while retaining critical information. This simplification means gaining insights from large datasets in a more manageable and

comprehensible manner. The choice of how many components to retain depends on the balance between simplification and preserving essential details tailored to specific business goals.

In [31]:
```
# Set a threshold for cumulative explained variance
threshold = 0.95

# Find the number of components that exceed the threshold
num_components_to_keep = np.argmax(cumulative_variance >= threshold)+1

print(f"Number of components to retain: {num_components_to_keep}")
```

```
Number of components to retain: 6
```

In [32]:
```
# Select the corresponding columns from the original dataset
selected_columns = PCA_df.columns[:num_components_to_keep]

# Create a list of features (columns) to keep
features_to_keep = df[selected_columns].columns.tolist()

# Create a list of features (columns) to delete
features_to_delete = \
[col for col in df.columns if col not in features_to_keep]

print("Features to Keep:")
print(features_to_keep)

print("\nFeatures to Delete:")
print(features_to_delete)
```

```
Features to Keep:
['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Volume_TSLA', 'EPS_qtr', 'Revenue']

Features to Delete:
['Date', 'day_week', 'Close_TSLA', 'Adj_Close_TSLA', 'open_close_TSLA', 'positive_TSL
A', 'oper_cash_fl', 'gross_mrgn', 'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr', 'fed_f
unds_rate', 'Open_NDQ', 'High_NDQ', 'Low_NDQ', 'Close_NDQ', 'Adj Close_NDQ', 'Volume_
NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX', 'Close_SPX', 'Adj_Close_SPX', 'Volume_SPX']
```

Identify and Display the data features that have the strongest influence on each of the retained principal components obtained through PCA analysis, based on component loadings.

In [33]:
```
PCA_cols =  ['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Volume_TSLA',
                'EPS_qtr', 'Revenue', 'oper_cash_fl', 'gross_mrgn',
                'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr',
                'fed_funds_rate', 'Open_NDQ', 'High_NDQ', 'Low_NDQ',
                'Volume_NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX',
                'Volume_SPX']


# Get the loadings of the retained components
component_loadings = pca_tsla.components_[:num_components_to_keep]

# Format a DataFrame to represent the loadings
```

```
loadings_df = pd.DataFrame(component_loadings, columns=PCA_cols)

# Absolute values of loadings for each feature
absolute_loadings = np.abs(loadings_df)

# Identify the most influential features for each component
most_influential_features = []

for i in range(num_components_to_keep-1):
    component_name = f"Principal Component {i + 1}"
    most_influential_feature = absolute_loadings.iloc[i-1].idxmax()
    most_influential_features.append\
    ((component_name, most_influential_feature))

# Display the most influential features for each component
print("Most influential features for each component:")
for component, feature in most_influential_features:
    print(f"{component}: {feature}")
```

```
Most influential features for each component:
Principal Component 1: EPS_qtr
Principal Component 2: High_NDQ
Principal Component 3: Revenue
Principal Component 4: Volume_NDQ
Principal Component 5: oper_cash_fl
```

In [34]:

```
# List the top 5 most influential features
top_5_influential_features = most_influential_features[:5]

# Initialize a dictionary to store the principal components for each feature
feature_to_components = {}

# Identify the most influential features for each principal component
for i in range(num_components_to_keep):
    component_name = f"Principal Component {i + 1}"

    # Find the feature with the highest loading for the current component
    most_influential_feature = absolute_loadings.iloc[i-1].idxmax()

    # Append the component and its most influential feature to a list
    most_influential_features\
    .append((component_name, most_influential_feature))

    # Store the mapping of features to components
    if most_influential_feature in feature_to_components:
        feature_to_components[most_influential_feature].append(component_name)
    else:
        feature_to_components[most_influential_feature] = [component_name]

# Sort the top 5 most influential features by loading value in descending order
sorted_top_5_influential_features = sorted(top_5_influential_features, \
          key=lambda x: -absolute_loadings\
                                    .loc[int(x[0].split()[-1]) - 1, x[1]])

print("\nTop 5 Most Influential Features\
(Sorted by Loading Value in Descending Order):")
for component, feature in sorted_top_5_influential_features:
    # Get the loading value for the feature in the current component
    loading_value = absolute_loadings\
    .loc[int(component.split()[-1]) - 1, feature]
```

```
    # Print the component, feature, and its loading value
    print(f"{component}: {feature} (Loading: {loading_value:.4f})")

# List the principal components each of the top 5 features has influence on
print("\nPrincipal Components Influenced by Top 5 Features:")
for component, feature in sorted_top_5_influential_features:
    # Identify the components where the feature is influential
    influenced_components = ', '.join(feature_to_components[feature])

    # Print the feature and the components it influences
    print(f"{feature} is influential in {influenced_components}")
```

```
Top 5 Most Influential Features(Sorted by Loading Value in Descending Order):
Principal Component 4: Volume_NDQ (Loading: 0.4848)
Principal Component 1: EPS_qtr (Loading: 0.2094)
Principal Component 2: High_NDQ (Loading: 0.1567)
Principal Component 3: Revenue (Loading: 0.0538)
Principal Component 5: oper_cash_fl (Loading: 0.0053)

Principal Components Influenced by Top 5 Features:
Volume_NDQ is influential in Principal Component 4
EPS_qtr is influential in Principal Component 1, Principal Component 6
High_NDQ is influential in Principal Component 2
Revenue is influential in Principal Component 3
oper_cash_fl is influential in Principal Component 5
```

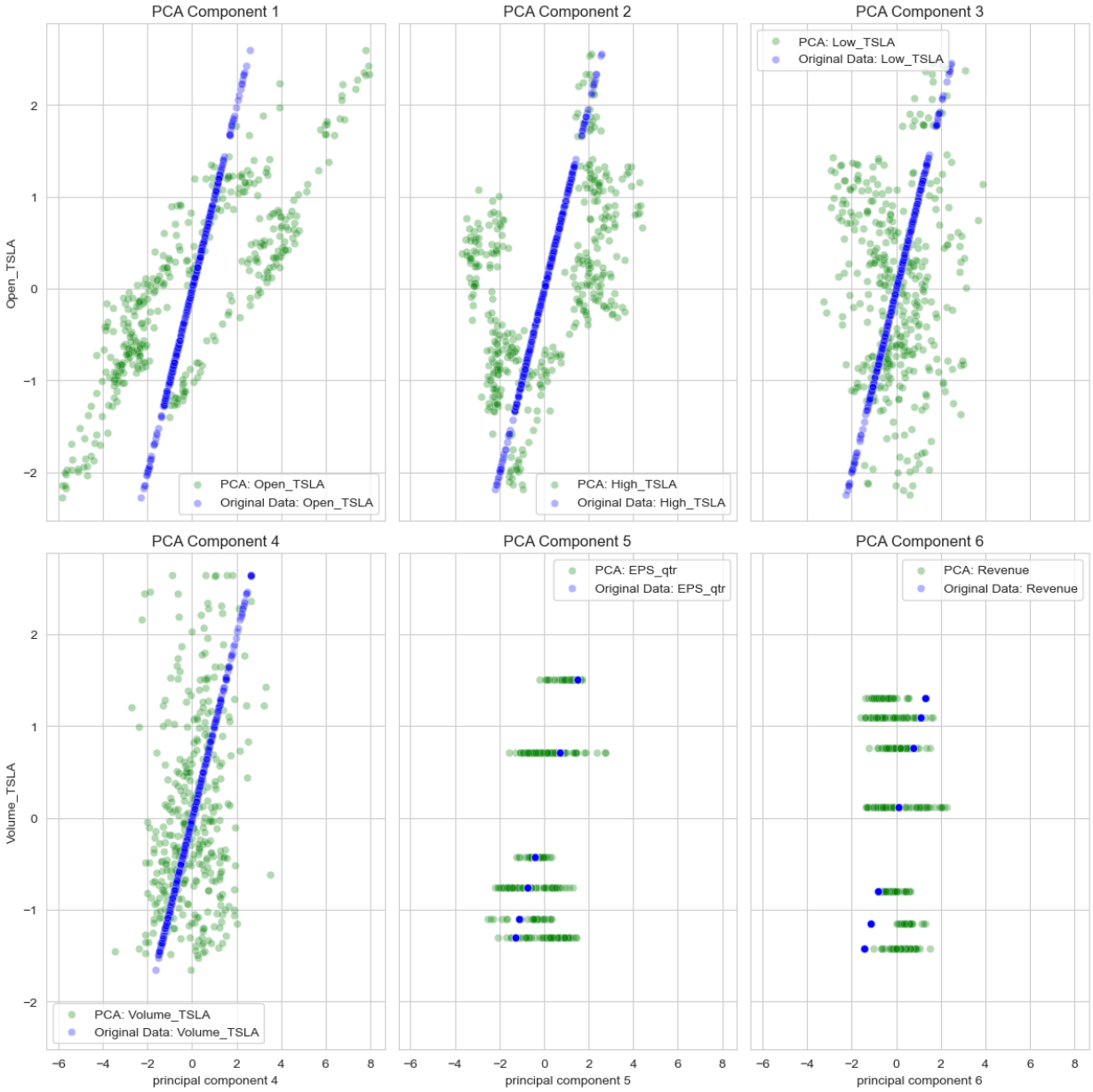
Visualize PCA findings

In [35]:

```
scaled_data = df_z[['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Volume_TSLA',
                'EPS_qtr', 'Revenue', 'oper_cash_fl', 'gross_mrgn',
                'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr',
                'fed_funds_rate', 'Open_NDQ', 'High_NDQ', 'Low_NDQ',
                'Volume_NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX',
                'Volume_SPX']]


# Adjust the layout to add more space between the subplots
plt.subplots_adjust(wspace=0.8, hspace=0.8)

# Create a PCA object with n components
pca_plots = PCA(n_components=6)

# Fit PCA model to PCA data and transform it to the principal components
principalPlots = pca_plots.fit_transform(PCA_df)

# Create a DataFrame 'principal_tsla_plots_Df' with n principal components
principal_tsla_plots_Df = pd.DataFrame(data = principalPlots
             , columns = ['principal component 1', 'principal component 2',
                          'principal component 3', 'principal component 4',
                          'principal component 5', 'principal component 6'])

# Set the number of rows and columns for subplots
rows = 2
cols = 3

# Initialize the component counter
comps = 1
```

```
# Create a figure and a grid of subplots
fig, axes = plt.subplots\
(rows, cols, figsize=(12, 12), sharex=True, sharey=True)

# Check if there are enough components for the specified rows and columns
n_components = min(rows * cols, principal_tsla_plots_Df.shape[1])

for row in range(rows):
    for col in range(cols):
        if comps <= n_components:
            # Select the corresponding component from principal_tsla_plots_Df
            component = principal_tsla_plots_Df.iloc[:, comps-1]

            # Choose the specific column from scaled_data to plot against
            selected_column = scaled_data.iloc[:, comps-1]

            # Create a scatterplot for Original Data with a custom label
            ax = sns.scatterplot(x=component, y=selected_column,
             ax=axes[row, col], color='green', alpha=0.3, \
                                label=f'PCA: {selected_column.name}')

            # Create a scatterplot for the PCA Data with a custom label
            sns.scatterplot(x=selected_column, y=selected_column,
            ax=ax, color='blue', alpha=0.3, \
                           label=f'Original Data: {selected_column.name}')

            # Set the title of the subplot based on the component number
            ax.set_title(f'PCA Component {comps}')

            # Increment the component counter
            comps += 1
        else:
            # Remove any empty subplots
            fig.delaxes(axes[row, col])

# Add the legend
for ax in axes.flat:
    handles, labels = ax.get_legend_handles_labels()
    ax.legend(handles, labels)

# Adjust the layout and display the plots
plt.tight_layout()
plt.show()
```

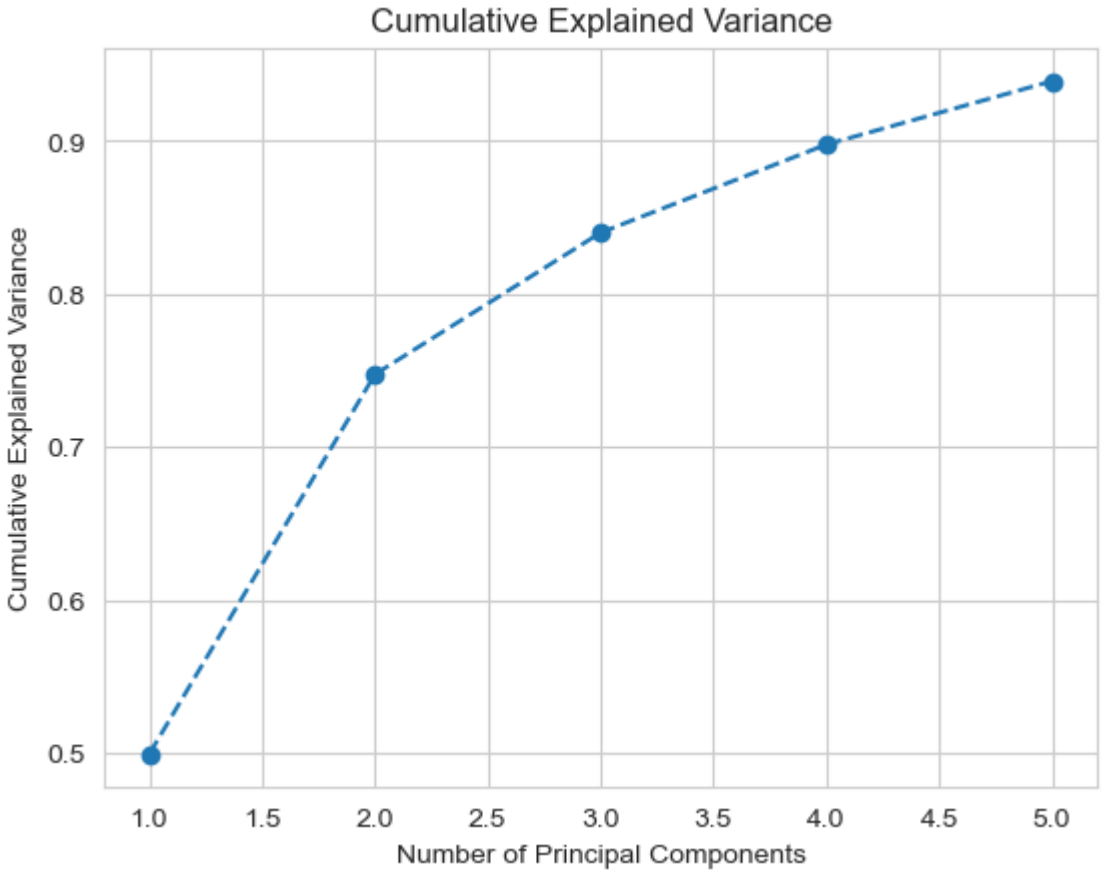
```
<Figure size 640x480 with 0 Axes>
```



Original Data: The wide scattering of data points along the X-axis suggests that the data varies significantly in that direction. This nebulous pattern means considerable variability in the data. **PCA Data:** The upward-sloping line of the PCA data on the Y-axis indicates that the PCA (Principal Component Analysis) focuses on the primary source of data variability. In simpler terms, the plots provide us data on: **PCA Efficiency:** The upward-sloping line in the PCA data shows that PCA has successfully found the most important direction in the data. **Feature Highlighting:** This allows us to spotlight what's truly important in our data. The PCA has identified the main factors that drive the variations. **Simplification:** By using this PCA component, we can simplify our analysis, presentations, or decisions. We get the main picture without all the complexities. **Straightforward Relationships:** The straight line indicates a clear relationship in the data, which is suitable for understanding our data more plainly and making informed decisions. **Data Compression:** We've streamlined our dataset just like condensing a large file into a smaller one while preserving all the crucial details. This process ensures we supply our data mining models with the most pertinent

information, leading to more accurate and dependable results. In short, seeing the original data scattered nebulously while the PCA data forms a line means that the PCA has effectively pinpointed the key factors in the data.

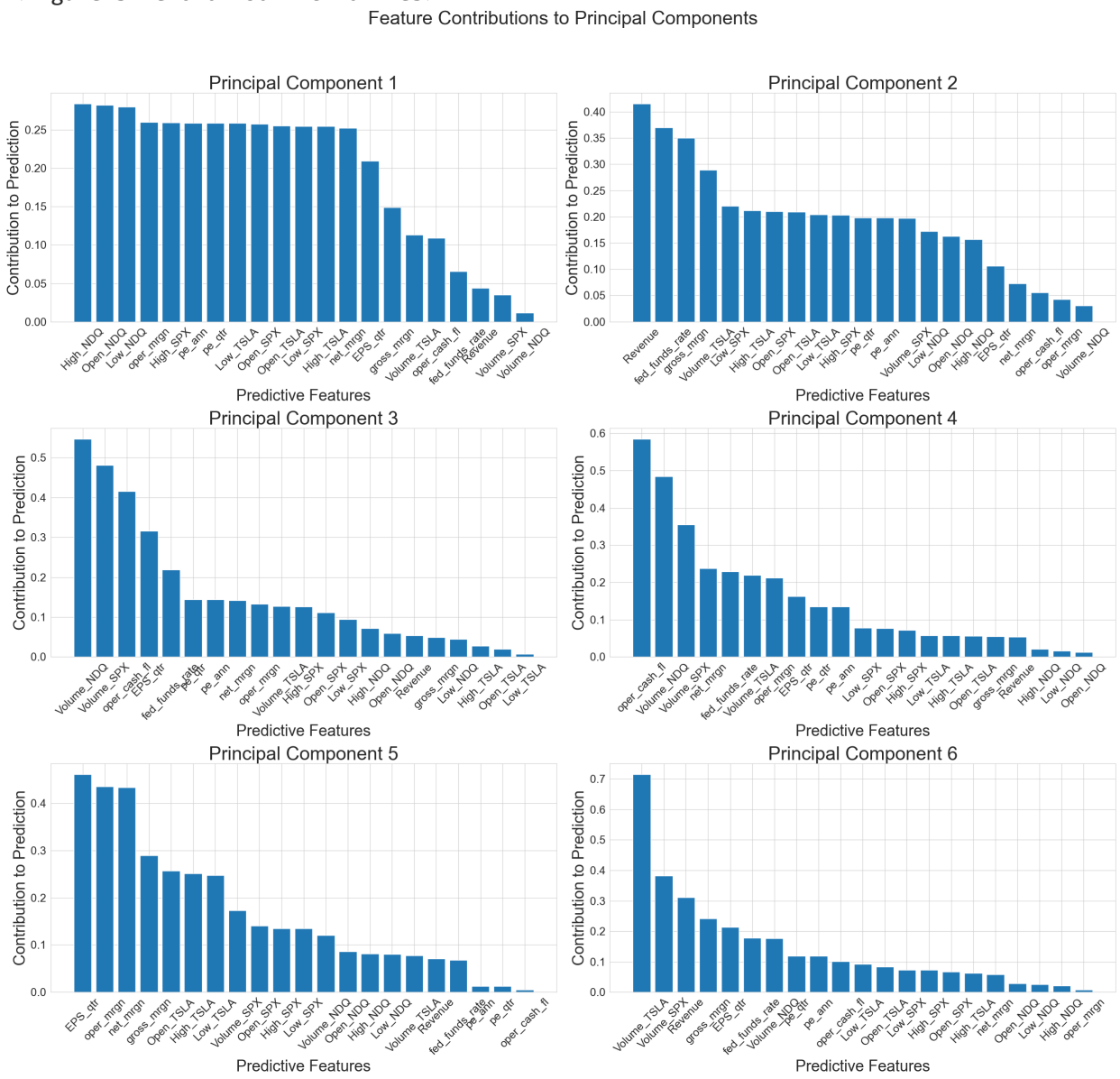
In [36]:
```
cumulative_variance = np.cumsum(rounded_explained_variance_ratio)
plt.plot(range(1, len(cumulative_variance) + 1), \
         cumulative_variance, marker='o', linestyle='--')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Cumulative Explained Variance')
plt.grid(True)
plt.show()
```



In [37]:
```
features_to_compare = ['Open_TSLA', 'High_TSLA', 'Low_TSLA', 'Volume_TSLA',
                'EPS_qtr', 'Revenue', 'oper_cash_fl', 'gross_mrgn',
                'oper_mrgn', 'net_mrgn', 'pe_ann', 'pe_qtr',
                'fed_funds_rate', 'Open_NDQ', 'High_NDQ', 'Low_NDQ',
                'Volume_NDQ', 'Open_SPX', 'High_SPX', 'Low_SPX',
                'Volume_SPX']

# Adjust the layout to add more space between the subplots
plt.subplots_adjust(wspace=0.8, hspace=0.8)

# Create a PCA object with n components
pca_groupings = PCA(n_components=21)

# Fit the PCA model to PCA data and transform to principal components
principalGroupings = pca_groupings.fit_transform(PCA_df)
```

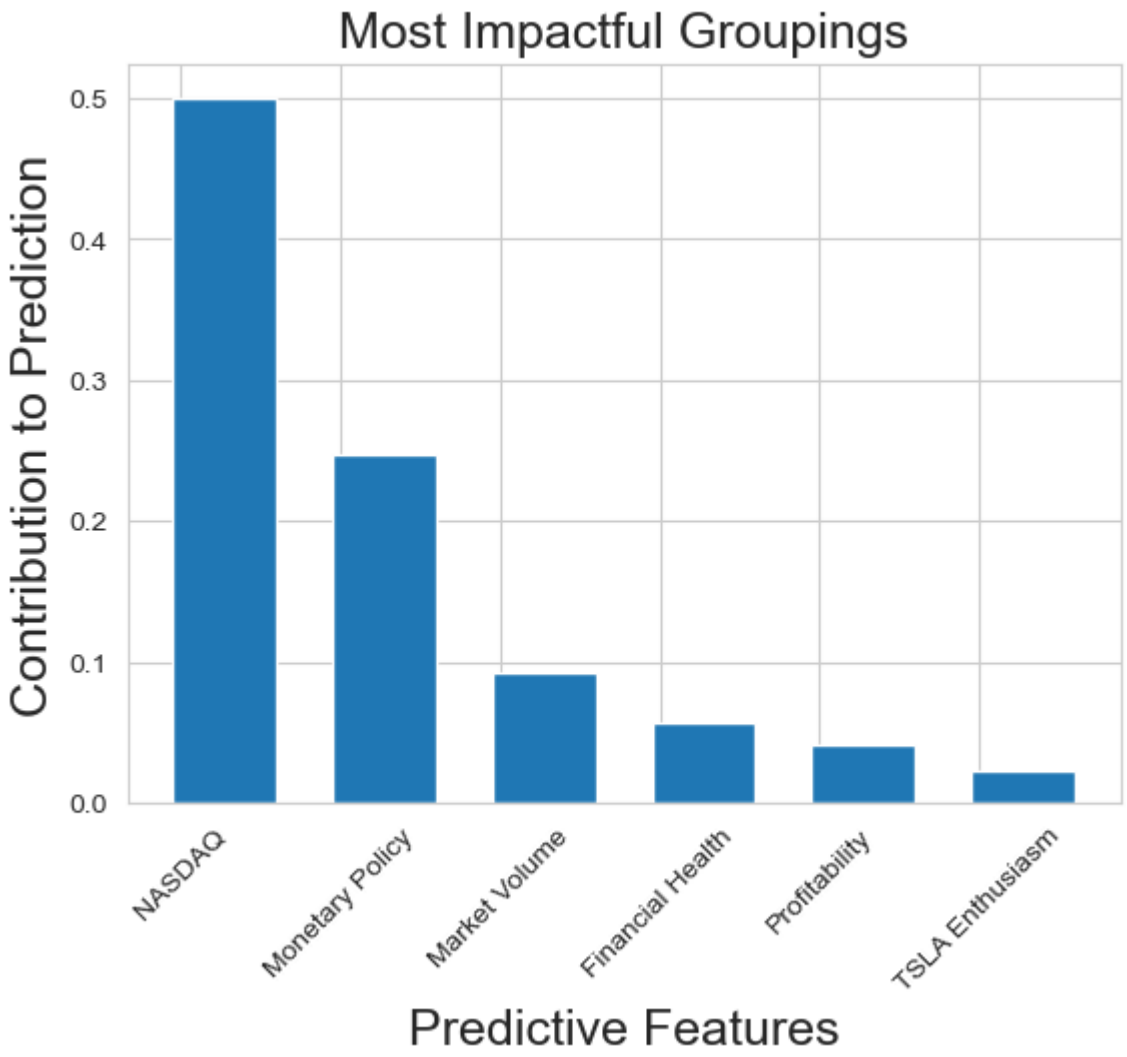
```
# Get the loadings of the retained components
group_component_loadings = \
pca_groupings.components_[:num_components_to_keep]

# Create a figure with a grid of subplots
fig, axes = plt.subplots(3, 2, figsize=(25, 25))
fig.suptitle('Feature Contributions to Principal Components', \
             fontsize=30)

# Format a DataFrame to represent the loadings
group_loadings_df = pd.DataFrame\
(group_component_loadings, columns=features_to_compare)

# Absolute values of loadings for each feature
group_absolute_loadings = np.abs(group_loadings_df)


# Extract the absolute loadings for the principal components
loadings_components = group_absolute_loadings.columns.tolist()

# Initialize a list to keep track of empty subplots
empty_subplots = []

# Iterate through all principal components
for i in range(len(group_absolute_loadings)):
    # Extract the absolute loadings for the current principal component
    loadings_component = group_absolute_loadings.iloc[i]

    # Sort the features by their absolute loadings in descending order
    sorted_features = loadings_component[features_to_compare]\
    .sort_values(ascending=False)

    # Calculate the subplot coordinates in the grid
    row, col = divmod(i, 2)

    # Create bar chart for the feature contributions to principal component
    ax = axes[row, col]
    ax.bar(sorted_features.index, sorted_features.values)

    ax.set_xlabel('Predictive Features', fontsize=25)
    ax.set_ylabel('Contribution to Prediction', fontsize=25)
    ax.tick_params(axis='x', rotation=45, labelrotation=45, labelsize=18)
    ax.tick_params(axis='y', labelsize=18)


    # Add a custom title to the subplot (customize as needed)
    ax.set_title(f'Principal Component {i + 1}', fontsize=30)

# Remove any empty subplots
for i in range(len(group_absolute_loadings), 6):
    row, col = divmod(i, 2)
    empty_subplots.append(axes[row, col])

for empty_subplot in empty_subplots:
    fig.delaxes(empty_subplot)

# Adjust the layout to prevent overlapping titles
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



In [38]:
```
# Create a PCA object with n components
pca_tsla_chart = PCA(n_components=6)

# Fit the PCA model to the PCA data and transform it to the principal components
principalComponents_chart = pca_tsla_chart.fit_transform(PCA_df)

# Create a DataFrame 'principal_tsla_Df' with n principal components columns
principal_tsla_chart_Df = pd.DataFrame(data = principalComponents_chart
        , columns = ['principal component 1', 'principal component 2',
                     'principal component 3', 'principal component 4',
                     'principal component 5', 'principal component 6'])

# Extract explained variance ratio
explainedVariance_ratio_chart = pca_tsla_chart.explained_variance_ratio_

# Round the explained variance ratio to 4 decimal places
rounded_explainedVariance_ratio_chart = \
np.round(explainedVariance_ratio_chart, 4)

# Data labels
custom_labels = [
```

```
    "NASDAQ",
    "Monetary Policy",
    "Market Volume",
    "Financial Health",
    "Profitability",
    "TSLA Enthusiasm"
]

# Create a bar plot
bar_width = .65  # Adjust the bar width as needed
x_positions = np.arange(1, len(custom_labels) + 1)
plt.bar\
(x_positions, rounded_explainedVariance_ratio_chart, width=bar_width)

# Calculate the tick positions in the center of the bars
ticks = x_positions + (bar_width / 2) - 0.6  # Adjusting the offset

# Set the x-tick positions and labels
plt.xticks(ticks, custom_labels, rotation=45, fontsize=10)

plt.xlabel('Predictive Features', fontsize=18)
plt.ylabel('Contribution to Prediction', fontsize=18)
plt.title('Most Impactful Groupings', fontsize=18)
plt.show()
```



Modeling

Perform Train_Test_Split on Normalized/Standardized Data sets

Partition standardized dataframe for regression on target TSLA closing price

In [39]:
```
# Set X and y variables after feature selection
X = df_z[features_to_keep]
y_close = df_z['Close_TSLA']
y_positive = df_z['positive_TSLA']

# Regression on Close: Split the data into training and testing sets
X_close_train, X_close_test, y_close_train, y_close_test = \
    train_test_split(X, y_close, test_size=0.4, random_state=14)

# Classification on Positive: Split the data into training and testing sets
X_positive_train, X_positive_test, y_positive_train, y_positive_test = \
    train_test_split(X, y_positive, test_size=0.4, random_state=14)

# Output the shapes of the training and testing datasets for verification.
print("CLOSE Training data shapes:", \
      X_close_train.shape, y_close_train.shape)
print("CLOSE Testing data shapes:", \
      X_close_test.shape, y_close_test.shape)
print("POSITIVE Training data shapes:", \
      X_positive_train.shape, y_positive_train.shape)
print("POSITIVE Testing data shapes:", \
      X_positive_test.shape, y_positive_test.shape)
```

```
CLOSE Training data shapes: (228, 6) (228,)
CLOSE Testing data shapes: (152, 6) (152,)
POSITIVE Training data shapes: (228, 6) (228,)
POSITIVE Testing data shapes: (152, 6) (152,)
```

Linear Regression

In [40]:
```
# Print coefficients of the regression model in descending order
lr_full_z = LinearRegression().fit(X_close_train, y_close_train)
lr_full_coef_z = pd.DataFrame(zip(X_close_train.columns, lr_full_z.coef_))
lr_full_coef_z.sort_values(by=1, ascending=False)
```

Out[40]:

	0	1
1	High_TSLA	0.906834
2	Low_TSLA	0.836512
3	Volume_TSLA	0.000288
4	EPS_qtr	-0.003222
5	Revenue	-0.003675
0	Open_TSLA	-0.747768

In [41]:
```
# Print coefficients of the regression model in descending order
lr_full_z = LinearRegression().fit(X_close_test, y_close_test)
lr_full_coef_z = pd.DataFrame(zip(X_close_test.columns, lr_full_z.coef_))
lr_full_coef_z.sort_values(by=1, ascending=False)
```

Out[41]:

	0	1
2	Low_TSLA	0.925065
1	High_TSLA	0.629531
4	EPS_qtr	0.004226
5	Revenue	0.003861
3	Volume_TSLA	-0.003442
0	Open_TSLA	-0.559089

Logistic Regression on Z-scores Standardized df (Cross Validated x5 w/ L2 penalty)

In [42]:
```
logit_train = LogisticRegressionCV\
(solver='lbfgs', cv=5, max_iter=500) #l2 penalty by default
logit_train.fit(X_positive_train, y_positive_train)

# Derived from Shmueli (2019)
print('intercept ', logit_train.intercept_[0])
print(pd.DataFrame({'coeff': logit_train.coef_[0]}, \
                   index=X_positive_train.columns)\
      .sort_values(by='coeff', ascending=False))
```

```
intercept  0.11296305681842383
                coeff
Low_TSLA     11.896122
High_TSLA     8.450929
Volume_TSLA   0.220437
EPS_qtr       0.040376
Revenue      -0.285781
Open_TSLA   -20.441825
```

Predictive Models

Build model(s) based on previous records' ability to predict future price movement.

Can previous data be used to predict future data?

Predictive Model #1: Random Forest Classifier

In [43]:
```
rf_train = RandomForestClassifier\
(n_estimators=100, min_samples_split=128, criterion='gini', random_state=14)
rf_train.fit(X_positive_train, y_positive_train)

rf_train_pred = rf_train.predict(X_positive_test[X_positive_train.columns])
rf_train_pred = pd.Series(rf_train_pred, index=X_positive_test.index)
precision = precision_score(y_positive_test, rf_train_pred)
print("Precision =", round(precision,4))
```

```
Precision = 0.4831
```

In [44]:
```
# Referencing directly from Schmueli's (2019) use
# of DecisionTreeRegressor and using param_grid as default values
param_grid = {
    'n_estimators': [100, 200],
    'criterion': ["gini", "entropy", "log_loss"],
    'max_depth': [1, 3, 5, 10],
    'min_samples_split': [2, 4, 8, 16, 32, 64, 128],
    'min_impurity_decrease': [0, 0.001, 0.005, 0.01],
}

grid_search = GridSearchCV(RandomForestClassifier(), \
       param_grid, cv=5, n_jobs=-1) # CV with 5 folds and all CPUs
grid_search.fit(X_positive_train, y_positive_train)

rf = grid_search.best_estimator_
```

In [45]:
```
rf
```

Out[45]:
```
RandomForestClassifier(criterion='entropy', max_depth=10,
                       min_impurity_decrease=0.005, min_samples_split=8,
                       n_estimators=200)
```

In [46]:
```
rf_cv = RandomForestClassifier(criterion='entropy', max_depth=10,
           min_impurity_decrease=0.01, n_estimators=200, random_state=14)
rf_cv.fit(X_positive_train, y_positive_train)

rf_cv_pred = rf_cv.predict(X_positive_test[X_positive_train.columns])
rf_cv_pred = pd.Series(rf_cv_pred, index=X_positive_test.index)
```

```
precision = precision_score(y_positive_test, rf_cv_pred)
print("Precision =", round(precision,4))
```

```
Precision = 0.5408
```

In [47]:
```
pd.DataFrame(zip(rf_cv.feature_names_in_,\
        rf_cv.feature_importances_)).sort_values(by=1, ascending=False)
```

Out[47]:

	0	1
0	Open_TSLA	0.255384
3	Volume_TSLA	0.243466
1	High_TSLA	0.210342
2	Low_TSLA	0.208979
5	Revenue	0.046203
4	EPS_qtr	0.035627

Predictive Model #2: Train MLP on known positive_TSLA = 1 for intraday

In [48]:
```
# Referencing directly from Schmueli's (2019) use of
# DecisionTreeRegressor and using param_grid as default values
param_grid = {
    'hidden_layer_sizes': [1, 2, 4, 8, 16],
    'activation': ["identity", "tanh", "logistic", "relu"],
    'solver': ['lbfgs'],
    'max_iter': [500, 1000, 2000],
}

grid_search = GridSearchCV(MLPClassifier(), \
                param_grid, cv=5, n_jobs=-1) # CV with 5 folds and all CPUs
grid_search.fit(X_positive_train, y_positive_train)

rf = grid_search.best_estimator_
```

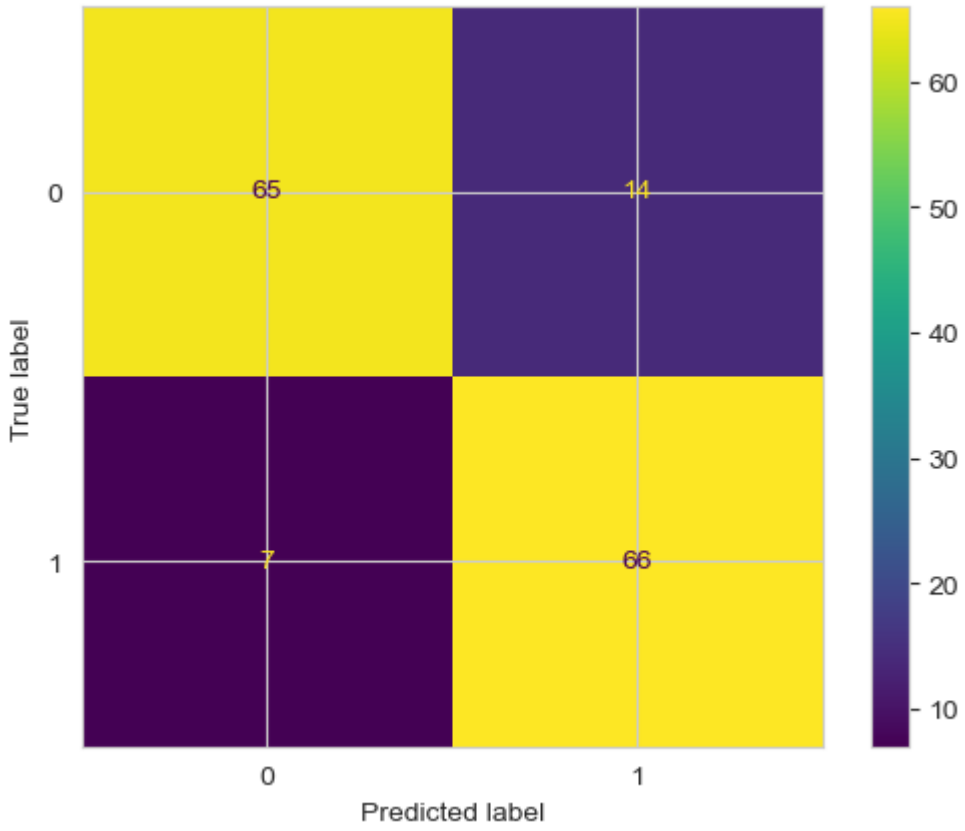
In [49]:
```
rf
```

Out[49]:
```
MLPClassifier(activation='identity', hidden_layer_sizes=1, max_iter=500,
              solver='lbfgs')
```

In [50]:
```
# Create and fit the MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(1), activation='identity', \
                    max_iter=500, solver='lbfgs', random_state=14)
mlp.fit(X_positive_train, y_positive_train)

# Predict on the test data
nn_train_pred = mlp.predict(X_positive_test)

# Calculate precision using the true test labels and predicted values
precision = precision_score(y_positive_test, nn_train_pred)
print("Precision =", round(precision, 4))
```

```
# Reference: https://scikit-learn.org/stable/modules/generated/sklearn
# .metrics.ConfusionMatrixDisplay.html
#            #sklearn.metrics.ConfusionMatrixDisplay
cm = confusion_matrix(y_positive_test, nn_train_pred, labels=mlp.classes_)
cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mlp.classes_)
cmd.plot()
print(classification_report(y_positive_test, nn_train_pred))
```

```
Precision = 0.825
              precision    recall  f1-score   support

           0       0.90      0.82      0.86        79
           1       0.82      0.90      0.86        73

    accuracy                           0.86       152
   macro avg       0.86      0.86      0.86       152
weighted avg       0.87      0.86      0.86       152
```



Predictive Model #3: Regression Tree model

In [51]:

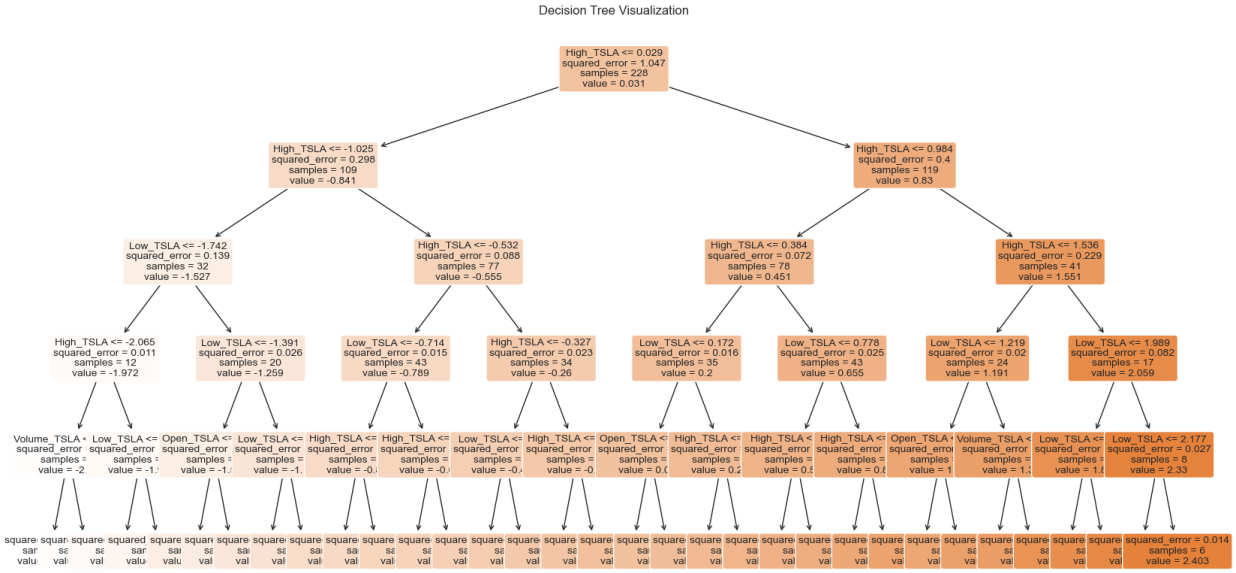
```
# Create a base Decision Tree Regressor with a range of max_depth values to search
tree_reg = DecisionTreeRegressor(random_state=1)

# Specify the values of max_depth to search
param_grid = {'max_depth': [2, 3, 4, 5, 10, 15, 20]}

# find optimal max_depth parameter
grid_search = GridSearchCV\
(tree_reg, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_close_train, y_close_train)
```

```
# Set variable to hold optimal max_depth
best_max_depth = grid_search.best_params_['max_depth']
print("The best max_depth is: ", best_max_depth)

# Fit a Decision Tree Regression model
rt_model = DecisionTreeRegressor\
(max_depth=best_max_depth,random_state=1)
rt_model.fit(X_close_train, y_close_train)

# Make predictions on the validation set
y_pred_rt = rt_model.predict(X_close_test)

# Calculate evaluation measures
mse_rt = mean_squared_error(y_close_test, y_pred_rt)
rmse_rt = mean_squared_error\
(y_close_test, y_pred_rt, squared=False)
mae_rt = mean_absolute_error(y_close_test, y_pred_rt)
r2_rt = r2_score(y_close_test, y_pred_rt)

# Apply formatting to the outcomes
mse_rt = '{:,.4f}'.format(mse_rt)
rmse_rt = '{:.4f}'.format(rmse_rt)
mae_rt = '{:.4f}'.format(mae_rt)
r2_rt = '{:.4f}'.format(r2_rt)


print('Mean Squared Error (MSE) - Tree:', mse_rt)
print('Root Mean Squared Error (RMSE) - Tree:', rmse_rt)
print('Mean Absolute Error (MAE) - Tree:', mae_rt)
print('R-squared (R2) - Tree:', r2_rt)
```

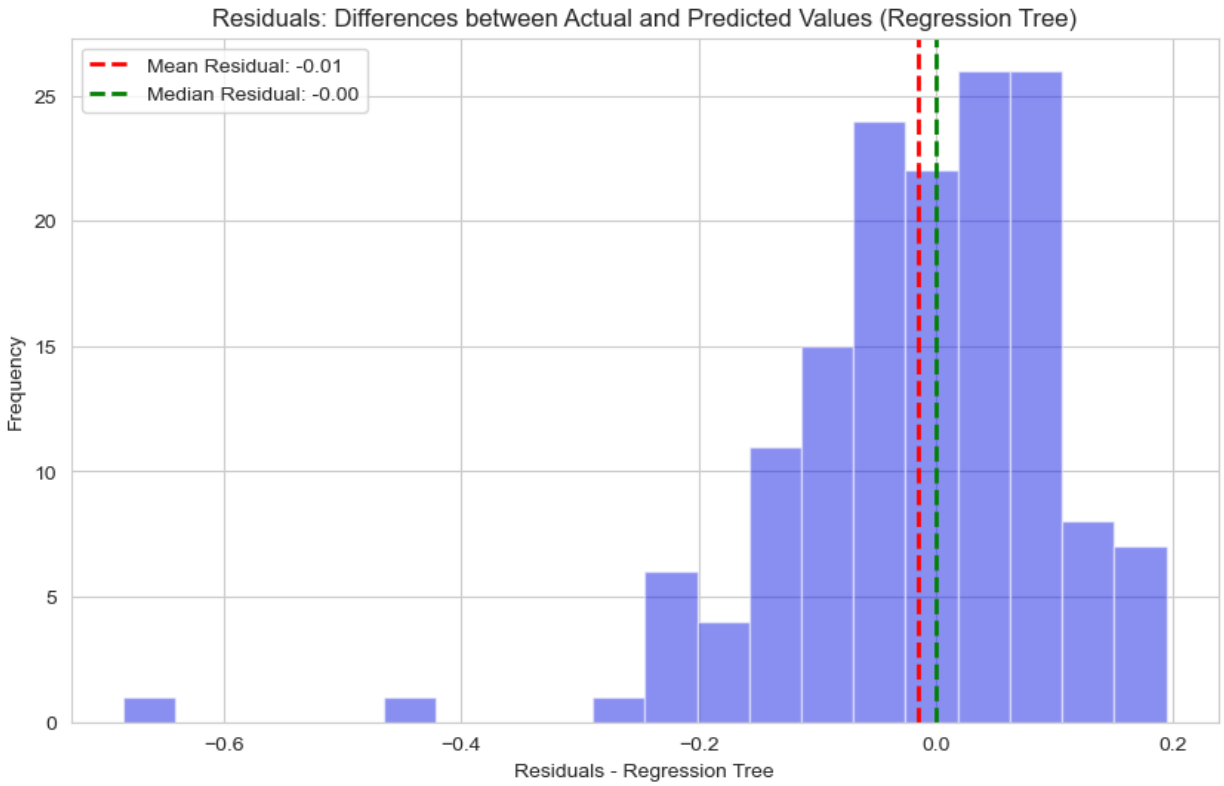
```
The best max_depth is:  5
Mean Squared Error (MSE) - Tree: 0.0139
Root Mean Squared Error (RMSE) - Tree: 0.1181
Mean Absolute Error (MAE) - Tree: 0.0856
R-squared (R2) - Tree: 0.9849
```

In [52]:

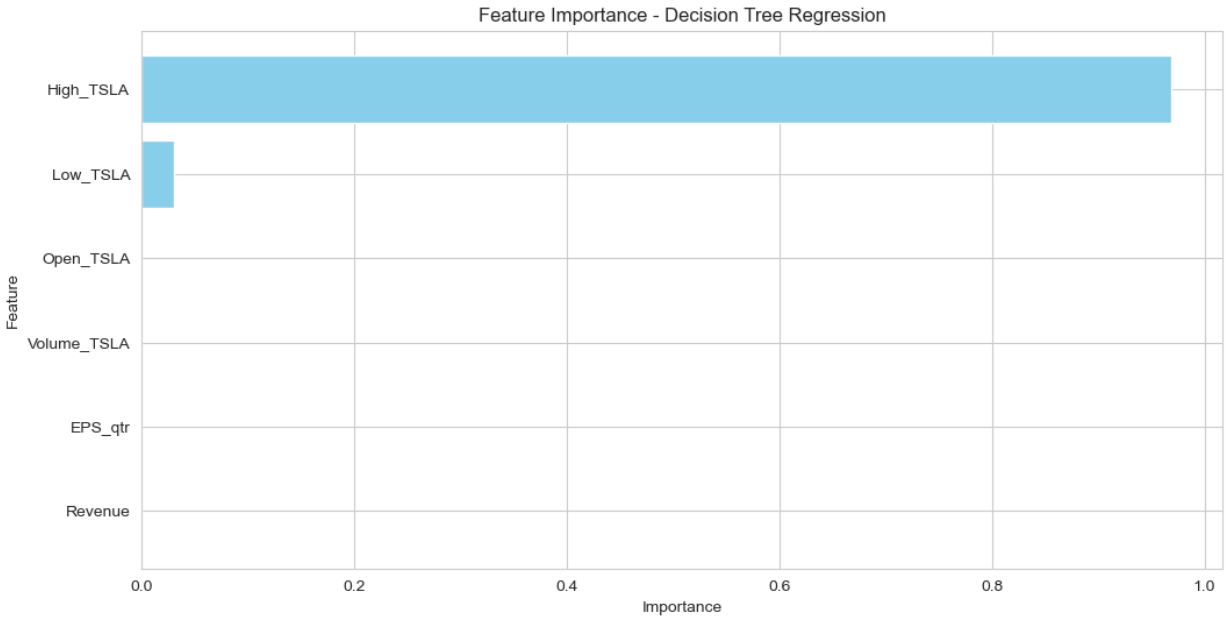
```
# Visualize the Decision Tree
plt.figure(figsize=(20, 10))
plot_tree(rt_model, feature_names=X_close_test\
      .columns.to_list(), filled=True, rounded=True, fontsize=10)
plt.title("Decision Tree Visualization")
plt.show()
```



In [53]:
```
# Calculate residuals
residuals_rt = y_close_test - y_pred_rt

# Create a histogram of residuals for the decision tree model
plt.figure(figsize=(10, 6))
plt.hist(residuals_rt, bins=20, alpha=0.5, color='#141ee5')

# Calculate mean and median of residuals for the decision tree model
mean_residual_rt = residuals_rt.mean()
median_residual_rt = np.median(residuals_rt)

# Add mean and median lines with value labels for the decision tree model
plt.axvline(mean_residual_rt, color='red', linestyle='dashed',\
            linewidth=2, label=f'Mean Residual: {mean_residual_rt:.2f}')

plt.axvline(median_residual_rt, color='green', linestyle='dashed',\
            linewidth=2, label=f'Median Residual: {median_residual_rt:.2f}')

plt.xlabel('Residuals - Regression Tree')
plt.ylabel('Frequency')
plt.title('Residuals: Differences between Actual and \
Predicted Values (Regression Tree)')
plt.legend()
plt.show()
```



In [54]:
```
# Get feature importance from the trained Decision Tree model
feature_importances = rt_model.feature_importances_

# Get the names of the features
feature_names = X_close_train.columns

# Create a DataFrame to store feature names and their importance scores
feature_importance_df = pd.DataFrame({'Feature': \
 feature_names, 'Importance': feature_importances})

# Sort the features by importance in descending order
feature_importance_df = feature_importance_df\
.sort_values(by='Importance', ascending=False)

# Create a bar chart to visualize feature importance
plt.figure(figsize=(12, 6))
plt.barh(feature_importance_df['Feature'], \
         feature_importance_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance - Decision Tree Regression')
plt.gca().invert_yaxis()
plt.show()
```



Predictive Model #4: Linear Regression Model

In [55]:

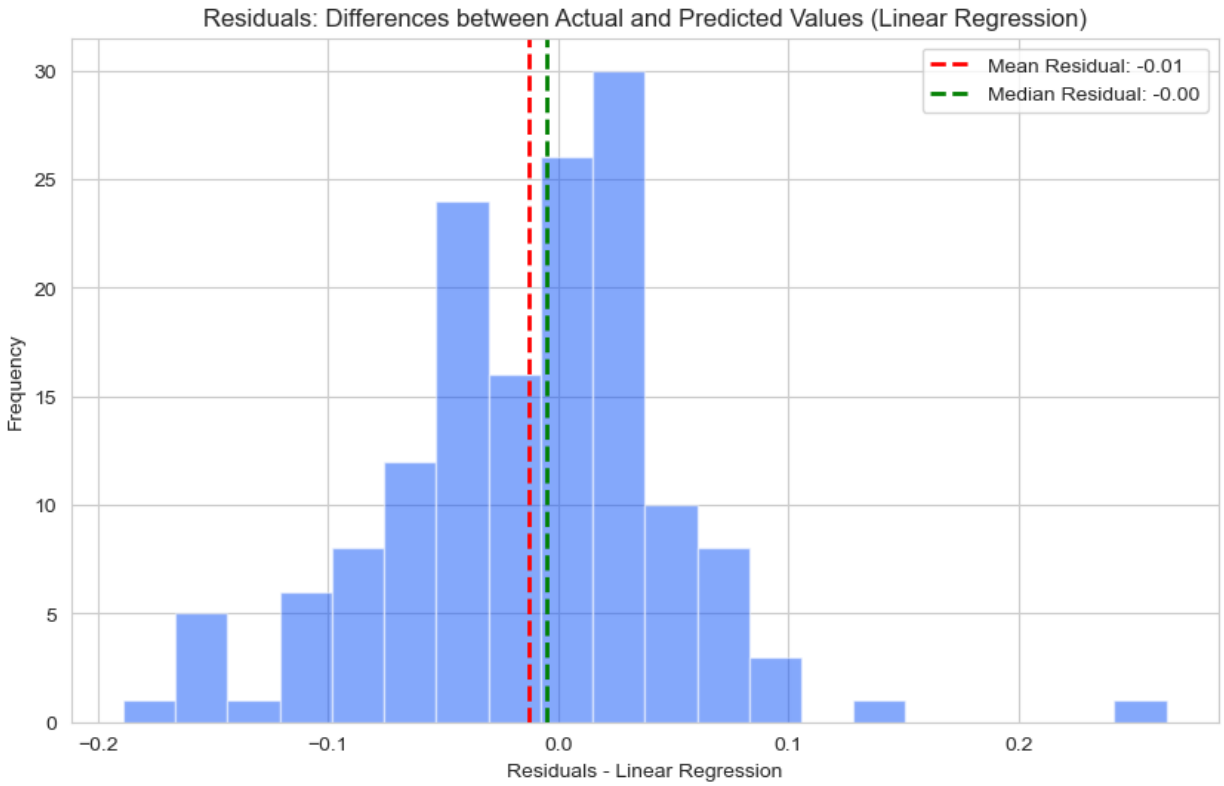
```
# Fit a multiple linear regression model
linear_reg_model = LinearRegression()
linear_reg_model.fit(X_close_train, y_close_train)

# Make predictions on the validation set (X_valid_linear)
y_pred_linear = linear_reg_model.predict(X_close_test)

# Calculate evaluation measures for the linear regression model
mse_linear = mean_squared_error(y_close_test, y_pred_linear)
rmse_linear = mean_squared_error(y_close_test, y_pred_linear, squared=False)
mae_linear = mean_absolute_error(y_close_test, y_pred_linear)
r2_linear = r2_score(y_close_test, y_pred_linear)

# Apply formatting to the outcomes
mse_linear = '{:,.4f}'.format(mse_linear)
rmse_linear = '{:.4f}'.format(rmse_linear)
mae_linear = '{:.4f}'.format(mae_linear)
r2_linear = '{:.4f}'.format(r2_linear)

# Print coefficients
coefficients = pd.DataFrame({'Predictor': X_close_test.columns, \
                             'Coefficient': linear_reg_model.coef_})
display(coefficients)

# Print evaluation measures for the linear regression model
print('Mean Squared Error (MSE) - Linear Regression:', mse_linear)
print('Root Mean Squared Error (RMSE) - Linear Regression:', rmse_linear)
print('Mean Absolute Error (MAE) - Linear Regression:', mae_linear)
print('R-squared (R2) - Linear Regression:', r2_linear)
```

	Predictor	Coefficient
0	Open_TSLA	-0.747768
1	High_TSLA	0.906834
2	Low_TSLA	0.836512
3	Volume_TSLA	0.000288
4	EPS_qtr	-0.003222
5	Revenue	-0.003675

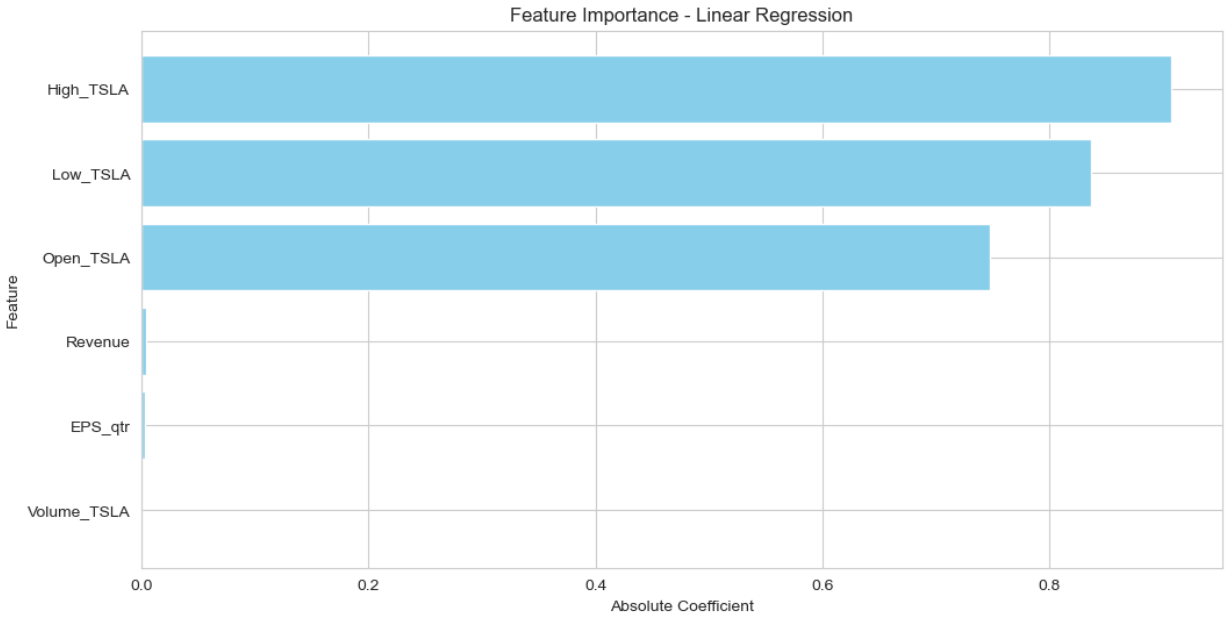
```
Mean Squared Error (MSE) - Linear Regression: 0.0040
Root Mean Squared Error (RMSE) - Linear Regression: 0.0634
Mean Absolute Error (MAE) - Linear Regression: 0.0475
R-squared (R2) - Linear Regression: 0.9957
```

In [56]:

```
# Calculate residuals
residuals = y_close_test - y_pred_linear

# Create a histogram of residuals
plt.figure(figsize=(10, 6))
plt.hist(residuals, bins=20, alpha=0.5, color='#0b52f9')

# Calculate mean and median of residuals
mean_residual = residuals.mean()
median_residual = np.median(residuals)

# Add mean and median lines with value labels
plt.axvline(mean_residual, color='red', linestyle='dashed',\
            linewidth=2, label=f'Mean Residual: {mean_residual:.2f}')

plt.axvline(median_residual, color='green', linestyle='dashed',\
            linewidth=2, label=f'Median Residual: {median_residual:.2f}')

plt.xlabel('Residuals - Linear Regression')
plt.ylabel('Frequency')
plt.title('Residuals: Differences between Actual \
and Predicted Values (Linear Regression)')
plt.legend()
plt.show()
```



In [57]:
```
# Get the coefficients (importance) of the features from the trained Linear Regression
coefficients = linear_reg_model.coef_

# Get the names of the features
feature_names = X_close_train.columns

# Create a DataFrame to store feature names and their coefficients
feature_importance_df = pd.DataFrame({'Feature': feature_names, \
                                      'Coefficient': coefficients})

# Sort the features by absolute coefficient values in descending order
feature_importance_df['Absolute Coefficient'] = \
abs(feature_importance_df['Coefficient'])
feature_importance_df = feature_importance_df\
.sort_values(by='Absolute Coefficient', ascending=False)

# Create a bar chart to visualize feature importance
plt.figure(figsize=(12, 6))
plt.barh(feature_importance_df['Feature'], \
         feature_importance_df['Absolute Coefficient'], color='skyblue')
plt.xlabel('Absolute Coefficient')
plt.ylabel('Feature')
plt.title('Feature Importance - Linear Regression')
plt.gca().invert_yaxis()  # Invert y-axis to display important features on top
plt.show()
```



Predictive Model #5: Random Forest Closing Price Prediction Model

In [58]:
```
rf = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model on the training data
rf.fit(X_close_train, y_close_train)
#predict
y_pred = rf.predict(X_close_test)
```

In [59]:
```
#Model evaluation
mse = mean_squared_error(y_close_test, y_pred)
r2 = r2_score(y_close_test, y_pred)
```

In [60]:
```
print(f"Mean Squared Error (MSE): {mse}")
print(f"R-squared (R2) Score: {r2}")
```

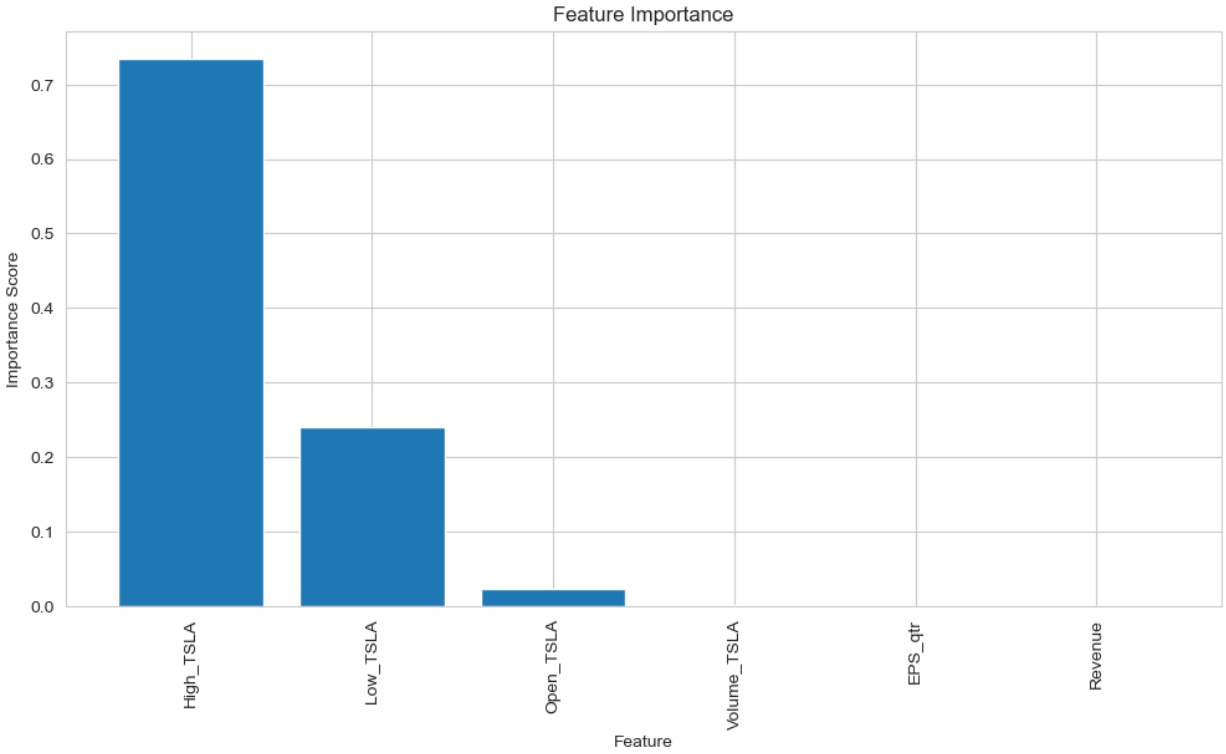
```
Mean Squared Error (MSE): 0.008505917555843784
R-squared (R2) Score: 0.9908071678375914
```

In [61]:
```
# absolute error
error = np.abs(y_close_test - y_pred)

# Set a threshold to define prediction accuracy
threshold = 5
accurate_predictions = (error <= threshold).astype(int)

# percentage of accuracy
accuracy_percentage = \
(np.sum(accurate_predictions) / len(accurate_predictions)) * 100

# scatterplot of predicted vs. actual
plt.figure(figsize=(10, 6))
plt.scatter(y_close_test, y_pred,\
            c=accurate_predictions, cmap='coolwarm', alpha=0.7)
plt.colorbar(label='Accuracy')
plt.xlabel('Actual "Close_TSLA" Prices')
plt.ylabel('Predicted "Close_TSLA" Prices')
plt.title(f'Scatterplot of Actual vs. Predicted\
```

```
"Close_TSLA" Prices\nAccuracy: {accuracy_percentage:.2f}%')
plt.grid(True)

plt.show()
```

In [62]:

```
# Feature importance plot
FI = rf.feature_importances_
feature_names = rf.feature_names_in_
plt.figure(figsize=(12, 6))
sorted_idx = np.argsort(FI)[::-1]
plt.bar(range(X_close_test.shape[1]), FI[sorted_idx], align="center")
plt.xticks(range(X_close_test.shape[1]), \
           [feature_names[i] for i in sorted_idx], rotation=90)
plt.xlabel("Feature")
plt.ylabel("Importance Score")
plt.title("Feature Importance")
plt.show()
```


Feature Importance
Importance Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
High_TSLA
Low_TSLA
Open_TSLA
Volume_TSLA
EPS_qtr
Revenue
Feature