SPY ETF Analysis

Project Subject:

The objective of this project is to acquire a deeper understanding of the impact that different macroeconomic factors have on the overall performance of the stock market. The SPY ETF will be utilized as an indicator of the general market conditions. Additionally, the project aims to enhance our comprehension of how various policies and macroeconomic indicators influence one another.

Data sources:

Flat File:

The flat file, obtained from Kaggle, consists of 11 features and 7639 rows. It encompasses the historical prices of the SPY ETF, spanning from 01/28/1993 to 05/30/2023. The columns include the date, opening price, daily low and high prices, closing price, and volume traded for each day. Additionally, the dataset provides the date information categorized by day, week, month, and year.

- https://www.kaggle.com/datasets/gkitchen/s-and-p-500-spy

In [1]:
```
import pandas as pd
spy_df=pd.read_csv("C:/Users/predi/Documents/GitHub/DSC540/Data Sets/spy.csv")
spy_df.head(5)
```

Out[1]:

	Date	Open	High	Low	Close	Volume	Day	Weekday	Week	Month	Year
0	1993-01-29	25.140215	25.140215	25.015139	25.122347	1003200	29	4	4	1	1993
1	1993-02-01	25.140216	25.301027	25.140216	25.301027	480500	1	0	5	2	1993
2	1993-02-02	25.283171	25.372511	25.229567	25.354643	201300	2	1	5	2	1993
3	1993-02-03	25.390365	25.640516	25.372497	25.622648	529400	3	2	5	2	1993
4	1993-02-04	25.711979	25.783451	25.426092	25.729847	531500	4	3	5	2	1993

Website:

The website dataset, sourced from bankrate.com, encompasses the federal interest rates starting from 01/09/1991 until the present day. After extracting the relevant information from the website, the dataset will consist of 5 features. These include the date of the interest rate change, the type of meeting associated with the change, the amount of the rate change, as well as the lower and upper federal target rates.

- https://www.bankrate.com/banking/federal-reserve/history-of-federal-funds-rate/

API:

The API dataset will be created by aggregating multiple API requests from econdb.com, each pulling in an economic indicator dataset. These datasets will be combined to form a comprehensive dataset. The final dataset will include the following features: Consumer Price Index, Producer Price Index, Unemployment Rate, Job Vacancy Rate, Real GDP (Gross Domestic Product), Real GDP per capita, Industrial Production, Retail Trade, 3-month Yield, and Long-term Yield. All of these features will be merged into a single dataset using their respective date information.

- https://www.econdb.com/api/series/?page=1

Relationship:

To merge all the datasets together, the Spy dataset will serve as the left join to the other two datasets. The NaN values in the resulting merged dataset will then be filled in using the previously stated value for each column. This approach is reasonable since many of the indicators are typically updated on a monthly basis. By implementing this method, a final dataset will be created with no missing values, ensuring that each row/date contains accurate data.

Summary

The main goal of this project is to enhance our understanding of how macroeconomic data impacts the overall market and its interrelationships. This will be achieved by merging all the datasets based on the observation date. The analysis will evaluate how the market is influenced by the indicators through the daily, monthly, and yearly percent price movements of the SPY. This approach will provide insights into the short-term and long-term reactions of the market to various metrics, while also mitigating the influence of unaccounted smaller variables.

To explore the relationships between indicators, the project will calculate correlations, Variance Inflation Factors (VIFs), and plot variables against each other to identify patterns. The information gained from this analysis can be valuable in improving trading strategies, understanding future economic periods, and gaining insights into government economic policies. It is worth noting that the research at this level does not present any negative ethical implications, as broadening knowledge and potentially improving trading strategies have minimal impact on society or individuals. However, if the research were to be drastically scaled up, it could lead to ethical concerns related to algo-trading, including market manipulation, unfair trading practices, and market instability.

The project faces several challenges. The primary challenge is the limited availability of data, as companies are often reluctant to publicly provide information that can be used for profit. Another significant hurdle is the multitude of variables that impact the market and each other. Quantifying all these variables is a near-impossible task, and financial companies are continuously attempting to address this challenge without a definitive answer. Consequently, this project aims to gain a general understanding of the effects rather than providing a comprehensive explanation for every economic indicator.

CSV Data Preparation

In [2]:
```
import pandas as pd
import numpy as np
```

In [3]:
```
#Shows a random sample of the rows of the dataframe and its shape.
spy_df=pd.read_csv("C:/Users/predi/Documents/GitHub/DSC540/Data Sets/spy.csv")
print(spy_df.shape)
spy_df.sample(5)
```

```
(7639, 11)
```

Out[3]:

	Date	Open	High	Low	Close	Volume	Day	Weekday	Week	Month	Year
23	1993-03-04	25.837066	25.837066	25.658386	25.658386	89500	4	3	9	3	1993
6178	2017-08-10	223.014952	223.150785	220.669717	220.724045	120479500	10	3	32	8	2017
5098	2013-04-29	131.864033	132.678467	131.656268	132.387604	88572800	29	0	18	4	2013
1416	1998-09-08	64.709311	66.072456	64.027738	66.072456	14746500	8	1	37	9	1998
7441	2022-08-17	420.623412	424.168848	419.270422	421.354218	63563400	17	2	33	8	2022

In [4]:
```
#checks for any NaN values in the dataframe.None found
spy_df.isnull().sum()
```

Out[4]:
```
Date       0
Open       0
High       0
Low        0
Close      0
Volume     0
Day        0
Weekday    0
Week       0
Month      0
Year       0
dtype: int64
```

Step 1 - clean header information

In [5]:
```
#Strips and lowers all header names
spy_df.columns=[x.lower() for x in spy_df.columns]
spy_df.columns=[x.strip() for x in spy_df.columns]
spy_df.head(5)
```

Out[5]:

	date	open	high	low	close	volume	day	weekday	week	month	year
0	1993-01-29	25.140215	25.140215	25.015139	25.122347	1003200	29	4	4	1	1993
1	1993-02-01	25.140216	25.301027	25.140216	25.301027	480500	1	0	5	2	1993
2	1993-02-02	25.283171	25.372511	25.229567	25.354643	201300	2	1	5	2	1993
3	1993-02-03	25.390365	25.640516	25.372497	25.622648	529400	3	2	5	2	1993
4	1993-02-04	25.711979	25.783451	25.426092	25.729847	531500	4	3	5	2	1993

Step 2 - change data types as needed

In [6]:
```
#Changes the "date" data type to date from object with a standard format.
spy_df['date'] =pd.to_datetime(spy_df['date'])
spy_df['date'].dtypes
```

Out[6]:
```
dtype('<M8[ns]')
```

Step 3 - remove unneeded columns

In [7]:
```
#Drops the repetive date columns.
#Keeping year to factor in for inflation and stadard growth in future models.
#Also month to factor in seasonal fluctuations
spy_df.drop(columns=['day','weekday','week'], inplace=True)
spy_df.head(5)
```

Out[7]:

	date	open	high	low	close	volume	month	year
0	1993-01-29	25.140215	25.140215	25.015139	25.122347	1003200	1	1993
1	1993-02-01	25.140216	25.301027	25.140216	25.301027	480500	2	1993
2	1993-02-02	25.283171	25.372511	25.229567	25.354643	201300	2	1993
3	1993-02-03	25.390365	25.640516	25.372497	25.622648	529400	2	1993
4	1993-02-04	25.711979	25.783451	25.426092	25.729847	531500	2	1993

Step 4 - add additional coulmns needed for EDA and future models.

In [8]:
```
#Creates a new column to show the percent changed during each trading day.
spy_df['day_change']=(spy_df['close']/spy_df['open'] - 1)*100
```

In [9]:
```
#Creates a new column to show the percnet change during after hours each day.
```

```
spy_df['aftermarket']=(spy_df['open']/spy_df['close'].shift(1) - 1)*100
```

In [10]:
```
#Creates a new columns to show the rolling 7, 30, and 90 day price averages.
#Goal is to have multiple time frames to test models against.
spy_df['7_day_avg']= spy_df['close'].rolling(window=7).mean()
spy_df['30_day_avg']= spy_df['close'].rolling(window=30).mean()
spy_df['90_day_avg']= spy_df['close'].rolling(window=90).mean()
```

In [11]:
```
#Creates a new column to show the rolling 90 day volume average
spy_df['90_day_volume']= spy_df['volume'].rolling(window=90).mean()
spy_df.tail(5)
```

Out[11]:

	date	open	high	low	close	volume	month	year	day_change	aftermarket	7_day_avg	30_
7634	2023-05-24	412.420013	412.820007	409.880005	411.089996	89213700	5	2023	-0.322491	-0.403290	415.328574	41
7635	2023-05-25	414.739990	416.160004	412.410004	414.649994	90961600	5	2023	-0.021699	0.887882	415.957145	41
7636	2023-05-26	415.329987	420.769989	415.250000	420.019989	93830000	5	2023	1.129223	0.163992	416.641427	41
7637	2023-05-30	422.029999	422.579987	418.739990	420.179993	72216000	5	2023	-0.438359	0.478551	416.777139	41
7638	2023-05-31	418.279999	419.220001	416.220001	417.850006	110811800	5	2023	-0.102800	-0.452186	416.667140	41

Step 5 - round columns to correct significant figures

In [12]:
```
#Rounds all floats to the second decimal place in the dataframe.
spy_df=spy_df.round(decimals=2)
#changes 90_day_average column to a integer and rounds to 0 decimal places
spy_df['90_day_volume']=spy_df['90_day_volume'].round(decimals=0)
spy_df['90_day_volume']=spy_df['90_day_volume'].astype('Int64')
```

In [13]:
```
#Data frame after all changes
spy_df.tail(5)
```

Out[13]:

	date	open	high	low	close	volume	month	year	day_change	aftermarket	7_day_avg	30_day_avg	90_day_a
7634	2023-05-24	412.42	412.82	409.88	411.09	89213700	5	2023	-0.32	-0.40	415.33	412.55	404
7635	2023-05-25	414.74	416.16	412.41	414.65	90961600	5	2023	-0.02	0.89	415.96	412.59	404
7636	2023-05-26	415.33	420.77	415.25	420.02	93830000	5	2023	1.13	0.16	416.64	412.84	405
7637	2023-05-30	422.03	422.58	418.74	420.18	72216000	5	2023	-0.44	0.48	416.78	413.05	405
7638	2023-05-31	418.28	419.22	416.22	417.85	110811800	5	2023	-0.10	-0.45	416.67	413.17	405

Web Scraping Data Preparation

Step 1 - scrape tables

In [14]:
```
import requests
from bs4 import BeautifulSoup
```

In [15]:
```
#Gets the HTML for the page that the tables needed are on
url='https://www.bankrate.com/banking/federal-reserve/history-of-federal-funds-rate/'
page=requests.get(url).text
soup=BeautifulSoup(page,'html.parser')
```

In [16]:
```
#Prints all of the tables on the page and their class.
# Way to check for table nots needed.
for table in soup.find_all('table'):
    print(table.get('class'))
```

```
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
['Table', 'Table--text', 'Table--stripedOdd', 'Table--fixed', 'Table--spacing1']
```

In [17]:
```
#Creates a list of all of the tables
table_soup=soup.find_all('table')
```

In [18]:
```
#Loops through all of the tables on the page and turns them in dataframes.
#Then concats them together by their row axis.
fed_df=[]
for i in range(7):
    temp_df = pd.read_html(str(table_soup))[i]
    if i==0:
        fed_df=temp_df
    else:
        fed_df = pd.concat([fed_df, temp_df],axis=0, ignore_index=True)
fed_df.sample(10)
```

Out[18]:

	Meeting date	Rate change	Target	Target & target range	Target range
7	Nov. 5, 1991	-25 basis points	4.75 percent	NaN	NaN
49	Nov. 10, 2004	+25 basis points	2 percent	NaN	NaN
81	Dec. 18-19, 2018	+25 basis points	NaN	NaN	2.25-2.5 percent
8	Dec. 6, 1991 (After a Dec. 2, 1991, conference...	-25 basis points	4.5 percent	NaN	NaN
3	April 30, 1991: Conference call	-25 basis points	5.75 percent	NaN	NaN
83	Sept. 17-18, 2019	-25 basis points	NaN	NaN	1.75-2 percent
70	Oct 8, 2008: Emergency meeting	-50 basis points	NaN	1.50 percent	NaN
88	May 3-4, 2022	+50 basis points	NaN	NaN	0.75-1 percent
74	Dec. 13-14, 2016	+25 basis points	NaN	NaN	0.5-0.75 percent
17	Aug. 16, 1994	+50 basis points	4.75 percent	NaN	NaN

step 2 - clean header information

In [19]:
```
#Strips and lowers all header names
fed_df.columns=[x.lower() for x in fed_df.columns]
fed_df.columns=[x.strip() for x in fed_df.columns]
fed_df.columns=fed_df.columns.str.replace(' ','_')


fed_df.sample(2)
```

Out[19]:

	meeting_date	rate_change	target	target_&_target_range	target_range
64	Oct. 30-31, 2007	-25 basis points	NaN	4.5 percent	NaN
48	Sept. 21, 2004	+25 basis points	1.75 percent	NaN	NaN

step 3 - format date and meeting columns

In [20]:
```
#spilts meeting date information into meeting date and meeting type.
fed_df[['meeting_date', 'meeting_type']] = fed_df[
    "meeting_date"].apply(lambda x: pd.Series(str(x).split(": ")))
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    r'\s\(After.*', '', regex=True)
```

In [21]:
```
#fills meeting type 'NaN' values with 'normal meeting'
fed_df['meeting_type'].fillna(value='normal meeting', inplace=True)
```

In [22]:
```
#removes multi date meetings and leaves last day of meeting
#Manually changes spercific ones as needed.
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    '\d{2}-', '', regex=True)
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    '\d{1}-', '', regex=True)
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    r'^(\w+\s){2,}', '', regex=True)
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    '1, 1992', 'July 1, 1992', regex=True)
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    'Jan. Feb. 1, 1995', 'Feb. 1, 1995', regex=True)
fed_df['meeting_date']=fed_df['meeting_date'].str.replace(
    'Jan. Feb. 1, 2023', 'Feb. 1, 2023', regex=True)
```

In [23]:
```
#formats date to match other dataframes(yyyy-mm-dd).
fed_df['meeting_date']=pd.to_datetime(
    fed_df['meeting_date'],format="%Y/%m/%d",infer_datetime_format=True)
```

step 4 - reformat rate change column

In [24]:
```
#removes none integar text from rate_chnage column
#changes data type to int and turns rate changes into percentages
fed_df['rate_change']=fed_df['rate_change'].str.replace(' basis points','')
fed_df['rate_change']=fed_df['rate_change'].str.replace('+','')
fed_df['rate_change']=fed_df['rate_change'].str.replace('-100 to 75','-87.5')
```

```
C:\Users\predi\AppData\Local\Temp\ipykernel_6168\1515315568.py:4: FutureWarning: The default value of regex w
ill change from True to False in a future version. In addition, single character regular expressions will *no
t* be treated as literal strings when regex=True.
  fed_df['rate_change']=fed_df['rate_change'].str.replace('+','')
```

In [25]:
```
#changes data type to int and turns rate changes into percentages
fed_df['rate_change']=pd.to_numeric(fed_df['rate_change'])
fed_df['rate_change']=fed_df['rate_change']/100
```

step 5 - combine and split target columns

In [26]:
```
#combines all of the target columns into one.
cols = ['target', 'target_&_target_range', 'target_range']
fed_df['lower_target']=fed_df[cols].apply(
    lambda row: ''.join(row.values.astype(str)), axis=1)
```

In [27]:
```
#drops the columns that were combined into one
fed_df.drop(columns=cols,inplace=True)
```

In [28]:
```
#removes unneeded text from columns
fed_df['lower_target']=fed_df['lower_target'].str.replace(
    ' percent','', regex=True)
fed_df['lower_target']=fed_df['lower_target'].str.replace(
    'nan','', regex=True)
```

In [29]:
```
#splits the target ranges into lower and upper target columns
fed_df[['lower_target', 'upper_target']] = fed_df[
```

```
    "lower_target"].apply(lambda x: pd.Series(str(x).split("-")))
```

In [30]:
```
#fills 'NaN' values in upper target column
#with values from lower target column.
fed_df['upper_target'].fillna(value=fed_df['lower_target'],
                              inplace=True)
```

In [31]:
```
#Changes both columns to numeric data types.
fed_df['lower_target']=pd.to_numeric(fed_df['lower_target'])
fed_df['upper_target']=pd.to_numeric(fed_df['upper_target'])
```

In [32]:
```
#final data frame
fed_df.sample(10)
```

Out[32]:

	meeting_date	rate_change	meeting_type	lower_target	upper_target
3	1991-04-30	-0.25	Conference call	5.75	5.75
46	2004-06-30	0.25	normal meeting	1.25	1.25
40	2001-09-17	-0.50	Emergency meeting	3.00	3.00
50	2004-12-14	0.25	normal meeting	2.25	2.25
2	1991-03-08	-0.25	Unscheduled move	6.00	6.00
83	2019-09-18	-0.25	normal meeting	1.75	2.00
29	1999-11-16	0.25	normal meeting	5.50	5.50
9	1991-12-20	-0.50	normal meeting	4.00	4.00
16	1994-05-17	0.50	normal meeting	4.25	4.25
17	1994-08-16	0.50	normal meeting	4.75	4.75

ethical implications

The data being scraped for this project is publicly available information provided by the government, so there are no ethical issues with obtaining and utilizing it within the bounds of the law. The project's objectives are entirely lawful. The only ethical consideration pertains to the website's policy on data scraping. However, according to the terms of service, there shouldn't be any problems since it primarily focuses on the usage of visitor data and does not explicitly mention the use of the data found on the website. While there is always a possibility that something was missed in the terms of service, there don't appear to be any significant ethical concerns associated with this data.

API Data Preparation

In [33]:
```
#import needed packages
import json
import datetime
```

Step 1 - pull data sets using API

In [34]:
```
#get api key from json file
with open("apikey.json") as f:
    key = json.load(f)
    API_TOKEN = key['econdb']
```

In [35]:
```
#consumer price index dataset
cpi_df = pd.read_csv(
    'https://www.econdb.com/api/series/CPIUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])

#producer price index dataset
ppi_df = pd.read_csv(
```

```
    'https://www.econdb.com/api/series/PPIUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

In [36]:
```
#consumer confidence index dataset
cci_df=pd.read_csv(
    'https://www.econdb.com/api/series/CONFUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

In [37]:
```
#industrial production data set
ip_df=pd.read_csv(
    'https://www.econdb.com/api/series/IPUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

In [38]:
```
#GDP data set
gdp_df=pd.read_csv(
    'https://www.econdb.com/api/series/RGDPUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])

#government debt data set
goverment_debt_df=pd.read_csv(
    'https://www.econdb.com/api/series/GDEBTUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

In [39]:
```
#unemployment percentage data set
unemployment_df = pd.read_csv(
    'https://www.econdb.com/api/series/URATEUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])

#total job vacancies data set
job_vacancy_df=pd.read_csv(
    'https://www.econdb.com/api/series/JVRUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

In [40]:
```
#10 year bond interest rate data set
bond_10year_df= pd.read_csv(
    'https://www.econdb.com/api/series/Y10YDUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])

#3 month bond interest rate data set
bond_3month_df=pd.read_csv(
    'https://www.econdb.com/api/series/M3YDUS/?token=%s&format=csv'
    % API_TOKEN,parse_dates=['Date'])
```

Step 2 - merge data sets together

In [41]:
```
#merges all of the data sets together by the date column
merged_df = pd.merge(cpi_df, ppi_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, cci_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, gdp_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, goverment_debt_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, unemployment_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, job_vacancy_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, bond_10year_df, on='Date', how='outer')
merged_df = pd.merge(merged_df, bond_3month_df, on='Date', how='outer')
#'ip_df' column cuasing issues with date order when merging using outer join
#used left to resolve issue
merged_df = pd.merge(merged_df, ip_df, on='Date', how='left')
```

In [42]:
```
#shows final merged dat set
merged_df.sample(5)
```

Out[42]:

	Date	CPIUS	PPIUS	CONFUS	RGDPUS	GDEBTUS	URATEUS	JVRUS	Y10YDUS	M3YDUS	IPUS
307	1972-08-01	41.9	NaN	100.80	NaN	NaN	5.6	NaN	6.21	NaN	41.82
456	1985-01-01	105.7	NaN	103.30	7829260.0	NaN	7.3	NaN	11.38	8.02	54.53
846	2017-07-01	244.2	112.7	100.50	18127994.0	19844910.0	4.3	4.3	2.32	1.09	100.10
791	2012-12-01	231.2	NaN	78.44	NaN	16432730.0	7.9	2.4	1.72	0.07	98.20
352	1976-05-01	56.4	NaN	91.92	NaN	NaN	7.4	NaN	7.90	NaN	43.84

Step 3 - clean header information

In [43]:
```
#Strips and lowers all header names
merged_df.columns=[x.lower() for x in merged_df.columns]
merged_df.columns=[x.strip() for x in merged_df.columns]
merged_df.columns=merged_df.columns.str.replace(' ','_')
merged_df.sample(2)
```

Out[43]:

	date	cpius	ppius	confus	rgdpus	gdebtus	urateus	jvrus	y10ydus	m3ydus	ipus
159	1960-04-01	29.54	NaN	100.4	3260177.0	NaN	5.2	NaN	4.28	NaN	23.55
67	1952-08-01	26.69	NaN	NaN	NaN	NaN	3.4	NaN	NaN	NaN	17.80

Step 4 - drop oldest observations

In [44]:
```
#droping everything before 1960 due to incomplete data
#before then and not needed for model.
filtered_df=merged_df[merged_df['date'] >= '2001-06-01']
```

Step 5 - Fill in/drop 'NaN' values

In [45]:
```
#check for NaN values in the data frame
filtered_df.isna().sum()
```

Out[45]:
```
date         0
cpius        0
ppius      158
confus      13
rgdpus     178
gdebtus      0
urateus      4
jvrus        5
y10ydus      0
m3ydus       0
ipus         1
dtype: int64
```

In [46]:
```
#have to drop ppius column due data not starting until 2014
droped_df = filtered_df.drop(columns=['ppius'])
```

In [47]:
```
#backfill real gdp NaN values, then forward fill last two months
droped_df['rgdpus'] = droped_df['rgdpus'].fillna(method='backfill')
droped_df['rgdpus'] = droped_df['rgdpus'].fillna(method='ffill')
droped_df.isna().sum()
```

Out[47]:
```
date        0
cpius       0
confus     13
rgdpus      0
gdebtus     0
urateus     4
jvrus       5
y10ydus     0
m3ydus      0
ipus        1
dtype: int64
```

In [48]:
```
droped_df.tail(5)
```

Out[48]:

	date	cpius	confus	rgdpus	gdebtus	urateus	jvrus	y10ydus	m3ydus	ipus
914	2023-03-01	301.8	NaN	20404088.0	31458438.0	3.5	NaN	3.66	4.86	102.7
915	2023-04-01	302.9	NaN	20404088.0	31457816.0	NaN	NaN	3.46	5.07	103.3
916	2023-05-01	303.3	NaN	20404088.0	31464458.0	NaN	NaN	3.57	5.31	102.8
917	2023-06-01	303.8	NaN	20404088.0	32332274.0	NaN	NaN	3.75	5.42	102.2
918	2023-07-01	304.3	NaN	20404088.0	32608586.0	NaN	NaN	3.90	5.49	NaN

In [49]:
```
#manually updating last year of consumer confidence not included in data set
index_positions = list(range(906, 918))
new_values = [51.5,58.2,58.6,59.9,56.8,59.7,64.9,67.0,62.0,63.5,59.2,64.4]
droped_df.loc[index_positions, 'confus'] = new_values
```

In [50]:
```
#manually updating 3 months of unemployment rate
index_positions = list(range(915, 918))
new_values = [3.4,3.7,3.6]
droped_df.loc[index_positions, 'urateus'] = new_values
```

In [51]:
```
#manually updating 4 months of job vacancies
index_positions = list(range(914, 918))
new_values = [5.9,6.2,5.9,5.9]
droped_df.loc[index_positions, 'jvrus'] = new_values
```

In [52]:
```
droped_df.isna().sum()
```

Out[52]:
```
date       0
cpius      0
confus     1
rgdpus     0
gdebtus    0
urateus    1
jvrus      1
y10ydus    0
m3ydus     0
ipus       1
dtype: int64
```

Step 5 - create additional feature

In [53]:
```
#Creates a new column to calulate difference bewteen short and long yields
econ_df=droped_df
econ_df['yield_diff'] = econ_df['y10ydus'] - econ_df['m3ydus']
```

In [54]:
```
#shows final/cleaned data frame
econ_df
```

Out[54]:

	date	cpius	confus	rgdpus	gdebtus	urateus	jvrus	y10ydus	m3ydus	ipus	yield_diff
653	2001-06-01	177.7	99.63	13248142.0	5726815.0	4.5	3.1	5.28	3.57	89.77	1.71
654	2001-07-01	177.4	99.42	13248142.0	5718302.0	4.6	3.6	5.24	3.59	89.23	1.65
655	2001-08-01	177.4	98.45	13284881.0	5769876.0	4.9	3.0	4.97	3.44	89.13	1.53
656	2001-09-01	178.1	88.01	13284881.0	5807463.0	5.0	2.9	4.73	2.69	88.67	2.04
657	2001-10-01	177.6	88.98	13284881.0	5815983.0	5.3	2.9	4.57	2.20	88.39	2.37
...	...	...	...	...	...	...	...	...	...	...	...
914	2023-03-01	301.8	62.00	20404088.0	31458438.0	3.5	5.9	3.66	4.86	102.70	-1.20
915	2023-04-01	302.9	63.50	20404088.0	31457816.0	3.4	6.2	3.46	5.07	103.30	-1.61
916	2023-05-01	303.3	59.20	20404088.0	31464458.0	3.7	5.9	3.57	5.31	102.80	-1.74
917	2023-06-01	303.8	64.40	20404088.0	32332274.0	3.6	5.9	3.75	5.42	102.20	-1.67
918	2023-07-01	304.3	NaN	20404088.0	32608586.0	NaN	NaN	3.90	5.49	NaN	-1.59

266 rows × 11 columns

ethical implications

The API data used for this project shares similar ethical considerations as the previously mentioned scraped data. It is important to note that the data obtained through the API is also publicly available and provided by the government, which ensures its legality. As long as the project's objectives remain lawful and within the scope of authorized usage, there should be no inherent ethical issues with utilizing this data. Overall, while precautions should be taken to respect the API provider's policies, there are no major ethical concerns associated with the usage of this data.

EDA

Step 1 - Load data into database

In [55]:
```
#import needed packages
import sqlite3
```

In [56]:
```
#Creates a database called 'final_project_db'
db_file = 'final_project_db'
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
```

In [57]:
```
#Creates a table in the database for the spy data
table_name = 'spy_data'
spy_df.to_sql(table_name, conn, index=False, if_exists='replace')
```

Out[57]: 7639

In [58]:
```
#Creates a table in the database for the federal interest rate data
table_name = 'fed_interest_data'
fed_df.to_sql(table_name, conn, index=False, if_exists='replace')
```

Out[58]: 98

In [59]:
```
#Creates a table in the database for the economical indicator data
table_name = 'econ_indicators_data'
econ_df.to_sql(table_name, conn, index=False, if_exists='replace')
```

Out[59]: 266

In [60]:
```
#drops merge table if it already exists
#Then merges the three tables into one table. Using a left join on the date column.
cursor.execute('DROP TABLE IF EXISTS merged_data')

query = f'''
    CREATE TABLE merged_data AS
    SELECT *
    FROM spy_data
    LEFT JOIN fed_interest_data
    ON spy_data.date = fed_interest_data.meeting_date
    LEFT JOIN econ_indicators_data
    ON spy_data.date= econ_indicators_data.date
    '''

cursor.execute(query)
conn.commit()
```

In [61]:
```
cursor.execute(f"PRAGMA table_info(merged_data)")
columns = cursor.fetchall()
for column in columns:
    print(column[1:3])
```

```
('date', 'NUM')
('open', 'REAL')
('high', 'REAL')
('low', 'REAL')
('close', 'REAL')
('volume', 'INT')
('month', 'INT')
('year', 'INT')
('day_change', 'REAL')
('aftermarket', 'REAL')
('7_day_avg', 'REAL')
('30_day_avg', 'REAL')
('90_day_avg', 'REAL')
('90_day_volume', 'INT')
('meeting_date', 'NUM')
('rate_change', 'REAL')
('meeting_type', 'TEXT')
('lower_target', 'REAL')
('upper_target', 'REAL')
('date:1', 'NUM')
('cpius', 'REAL')
('confus', 'REAL')
('rgdpus', 'REAL')
('gdebtus', 'REAL')
('urateus', 'REAL')
('jvrus', 'REAL')
('y10ydus', 'REAL')
('m3ydus', 'REAL')
('ipus', 'REAL')
('yield_diff', 'REAL')
```

Step 2 - Fill in NaN values from merging data

In [62]:
```
#gets merged data table and converts to data frame
merged_data_df = pd.read_sql_query("SELECT * FROM merged_data", conn)
#shows NaN value in dataframe
merged_data_df.isnull().sum()
```

Out[62]:
```
date                0
open                0
high                0
low                 0
close               0
volume              0
month               0
year                0
day_change          0
aftermarket         1
7_day_avg           6
30_day_avg         29
90_day_avg         89
90_day_volume      89
meeting_date     7556
rate_change      7556
meeting_type     7556
lower_target     7556
upper_target     7556
date:1           7469
cpius            7469
confus           7469
rgdpus           7469
gdebtus          7469
urateus          7469
jvrus            7469
y10ydus          7469
m3ydus           7469
ipus             7469
yield_diff       7469
dtype: int64
```

In [63]:
```
#fills in NaN values for rate change with 0
merged_data_df['rate_change'].fillna(value=0, inplace=True)
#fills in NaN values between meetings with 'no meeting'
merged_data_df['meeting_type'].fillna(value='no meeting', inplace=True)
#forward fills all other NaN values in the dataframe
merged_data_df.fillna(method='ffill', inplace=True)
```

In [64]:
```
#drops redundant date colulms
final_df = merged_data_df.drop(columns=['meeting_date','date:1'])
#drops all data before 2001-06-01 to get rid of NaN values
final_df = final_df[final_df['date']>= '2001-06-1']
```

In [65]:
```
#shows new total of NaN values in data frame
final_df.isnull().sum()
```

Out[65]:
```
date             0
open             0
high             0
low              0
close            0
volume           0
month            0
year             0
day_change       0
aftermarket      0
7_day_avg        0
30_day_avg       0
90_day_avg       0
90_day_volume    0
rate_change      0
meeting_type     0
lower_target     0
upper_target     0
cpius            0
confus           0
rgdpus           0
gdebtus          0
urateus          0
jvrus            0
y10ydus          0
m3ydus           0
ipus             0
yield_diff       0
dtype: int64
```

In [66]:
```
#Creates a table in the database for the final dataframe
table_name = 'all_data'
final_df.to_sql(table_name, conn, index=False, if_exists='replace')
```

Out[66]:
```
5528
```

Step 3 - Visualizations

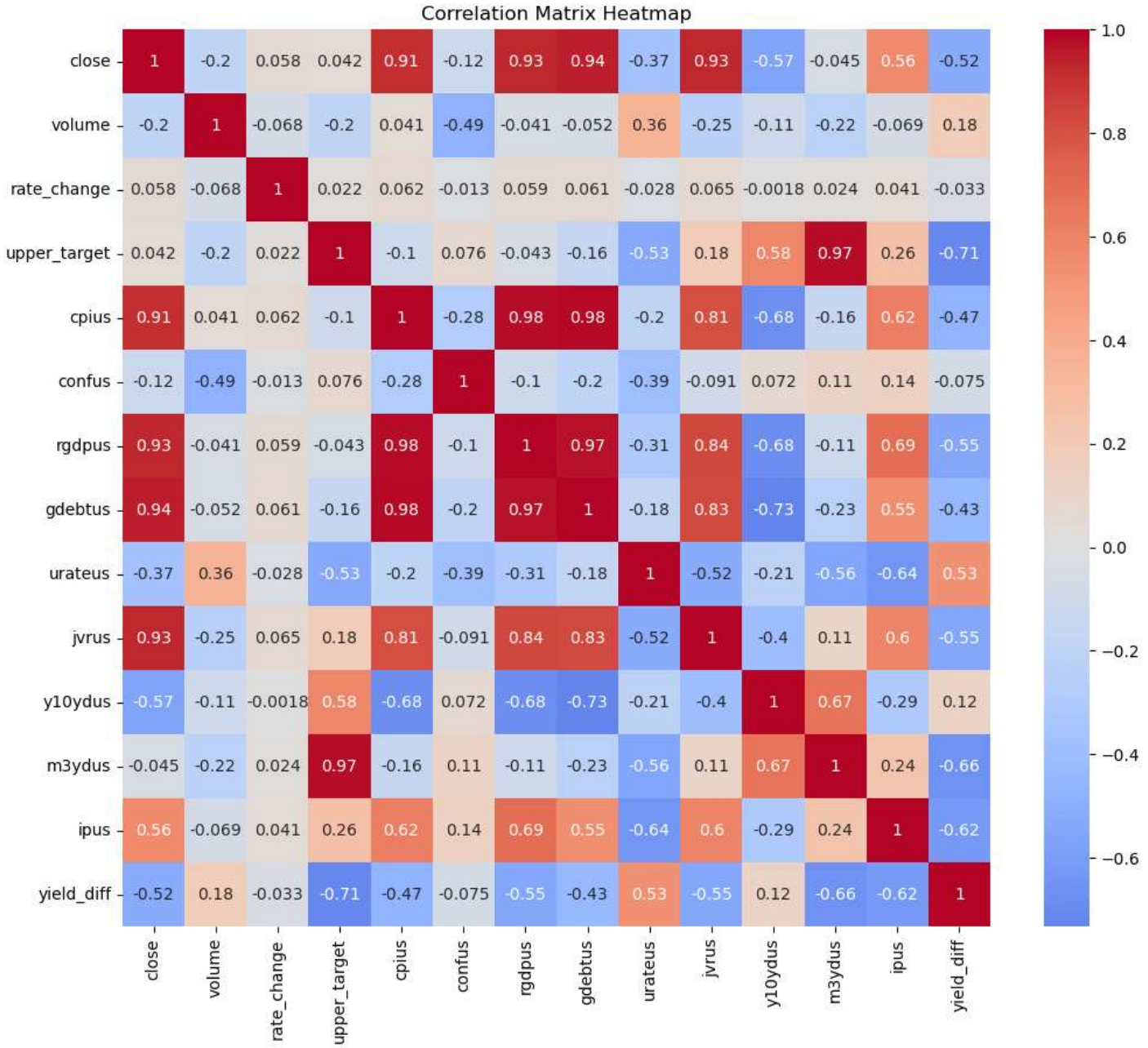
In [67]:
```
#imports needed packages
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.dates import YearLocator, DateFormatter
import calendar
```

In [68]:
```
#turns the sql table into a dataframe to make visualizations easier
all_data_df = pd.read_sql_query("SELECT * FROM all_data", conn)
```

Visualization 1 - Correlation Matrix Heat Map

In [69]:
```
#Creates a list of columns to be used
corr_columns= ['close', 'volume','rate_change', 'meeting_type', 'upper_target',
               'cpius','confus', 'rgdpus', 'gdebtus', 'urateus', 'jvrus', 'y10ydus',
               'm3ydus','ipus', 'yield_diff']
corr_data = all_data_df[corr_columns]

#create correlation heat map
plt.figure(figsize=(12, 10))
sns.heatmap(corr_data.corr(), annot=True, cmap='coolwarm', center=0)
plt.title("Correlation Matrix Heatmap")
plt.show()
```

```
C:\Users\predi\AppData\Local\Temp\ipykernel_6168\3869103266.py:9: FutureWarning: The default value of numeric
_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid colum
ns or specify the value of numeric_only to silence this warning.
  sns.heatmap(corr_data.corr(), annot=True, cmap='coolwarm', center=0)
```

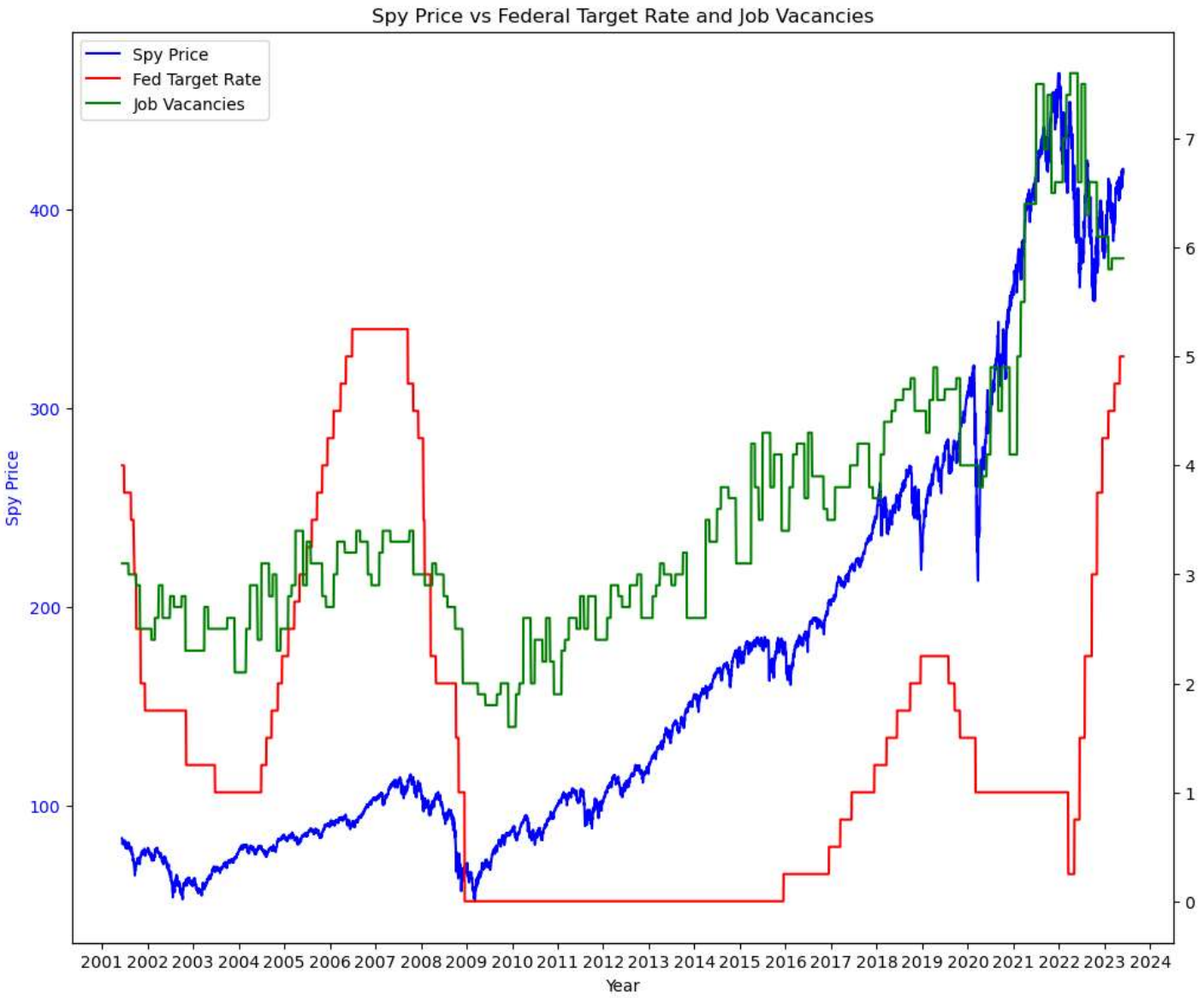


Visualization 2 - Line Plot: Spy Price vs Federal Target Rate and Job Vacancies

In [70]:

```
#converts date column to datetime format
all_data_df['date'] = pd.to_datetime(all_data_df['date'])

#Plots SPY price
fig, ax1 = plt.subplots(figsize=(12, 10))
ax1.plot(all_data_df['date'], all_data_df['close'],
         color='blue', label='Spy Price')
ax1.set_xlabel('Year')
ax1.set_ylabel('Spy Price', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

#creates secodary axis
ax2 = ax1.twinx()
ax2.plot(all_data_df['date'], all_data_df['lower_target'],
         color='red', label='Fed Target Rate')
ax2.plot(all_data_df['date'], all_data_df['jvrus'],
         color='green', label='Job Vacancies')

#Creates legend
lines = ax1.get_lines() + ax2.get_lines()
labels = [line.get_label() for line in lines]
ax1.legend(lines, labels, loc='upper left')
```

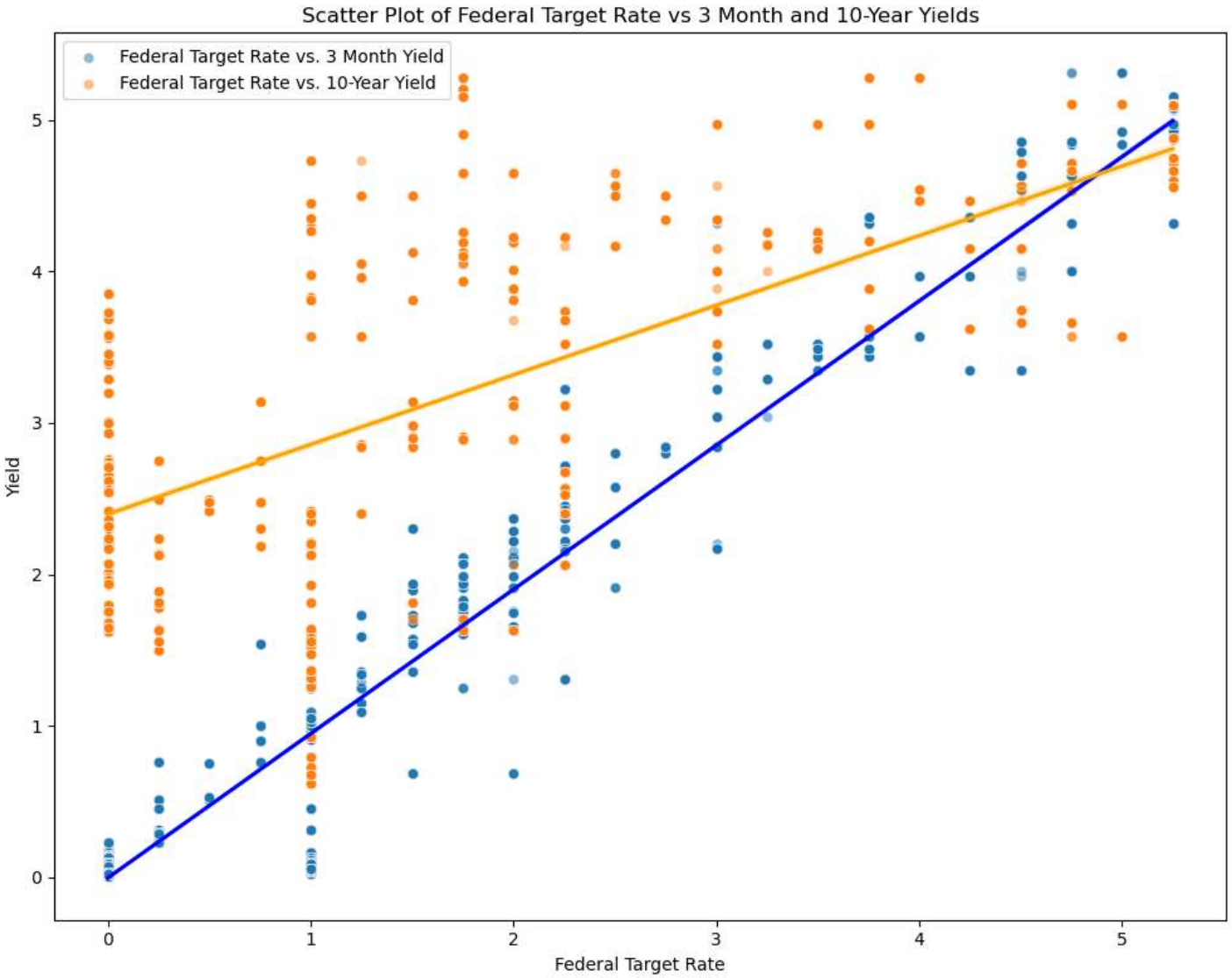
```
#Sets the x-axis date format to show years
ax1.xaxis.set_major_locator(YearLocator())
ax1.xaxis.set_major_formatter(DateFormatter("%Y"))
plt.xticks(rotation=45)
plt.title('Spy Price vs Federal Target Rate and Job Vacancies')
plt.show()
```



Visualization 3 - Scatterplot: Federal Target Rate vs 3 Month and 10-Year Yields

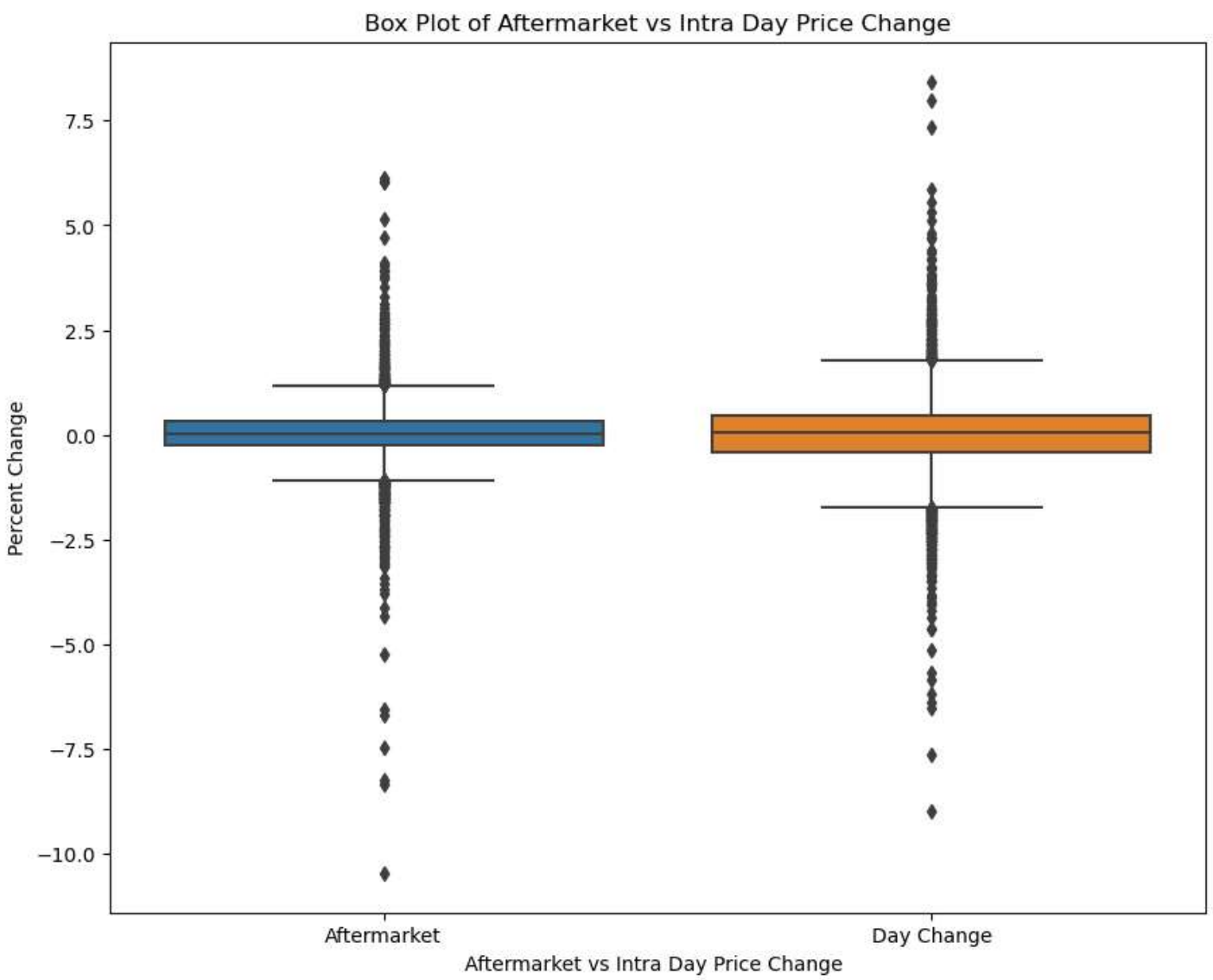
In [72]:

```
#Creates scatter plot for 'lower_target' versus 'm3ydus' and 'y10ydus'
plt.figure(figsize=(10,8))
sns.scatterplot(data=all_data_df, x='lower_target', y='m3ydus',
                alpha=0.5, label='Federal Target Rate vs. 3 Month Yield')
sns.scatterplot(data=all_data_df, x='lower_target', y='y10ydus',
                alpha=0.5, label='Federal Target Rate vs. 10-Year Yield')

#Adds trendlines for both scatter plots
sns.regplot(data=all_data_df, x='lower_target', y='m3ydus',
            scatter=False, color='blue')
sns.regplot(data=all_data_df, x='lower_target', y='y10ydus',
            scatter=False, color='orange')
plt.xlabel('Federal Target Rate')
plt.ylabel('Yield')
plt.title('Federal Target Rate vs 3 Month and 10-Year Yields')

plt.legend()
plt.tight_layout()
plt.show()
```



Visualization 4 - Box Plot: Aftermarket vs Intra Day Price Change

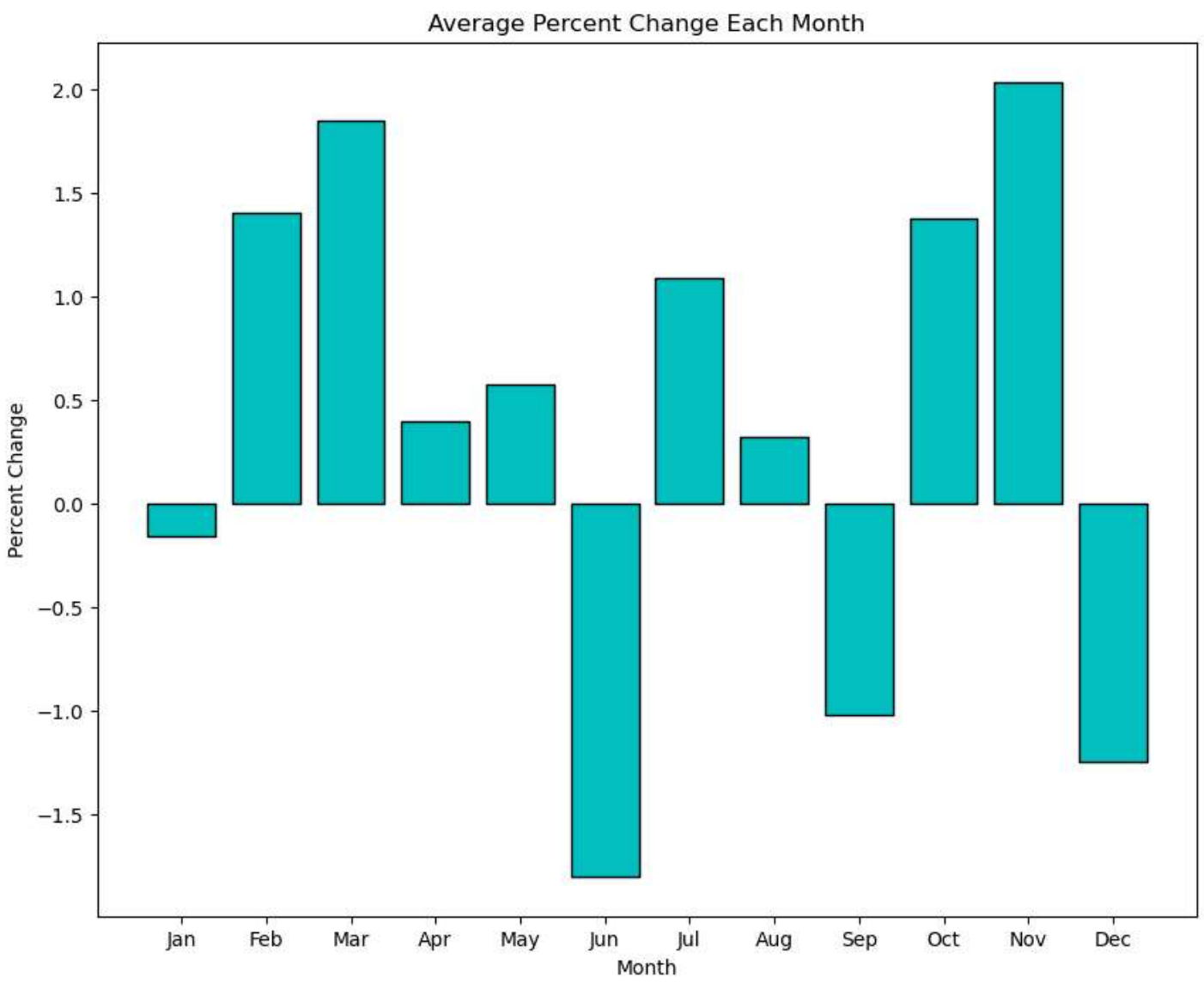
In [71]:
```
# Create a box plot using Seaborn
plt.figure(figsize=(10,8))
sns.boxplot(data=all_data_df[['aftermarket', 'day_change']])
plt.xlabel('Aftermarket vs Intra Day Price Change ')
plt.ylabel('Percent Change')
plt.title('Aftermarket vs Intra Day Price Change')
plt.xticks(ticks=[0, 1], labels=['Aftermarket', 'Day Change'])
plt.show()
```



Visualization 5 - Bar Graph: Average Percent Change Each Month

In [73]:

```
#Gets day of month
all_data_df['day'] = all_data_df['date'].dt.day

#Calculates the average day change for each month
average_day_change = all_data_df.groupby('month')['day_change'].mean()
#Calculates the average percentchange for each month
days_in_month = all_data_df.groupby('month')['day'].max()
change_per_month = average_day_change * days_in_month

# Creates thge bar graph
plt.figure(figsize=(10, 8))
#uses calender libary to look up month names.
plt.bar([calendar.month_abbr[i] for i in change_per_month.index],
        change_per_month, color='c',edgecolor='black')
plt.xlabel('Month')
plt.ylabel('Percent Change')
plt.title('Average Percent Change Each Month')
plt.show()
```



In [75]:
```
#final dataset
all_data_df.head(10)
```

Out[75]:

	date	open	high	low	close	volume	month	year	day_change	aftermarket	...	confus	rgdpus	gdebtus	urateu
0	2001-06-11	83.82	84.00	82.96	83.41	7012200	6	2001	-0.48	-0.23	...	99.63	13248142.0	5726815.0	4.
1	2001-06-12	82.59	83.84	82.05	83.27	9364400	6	2001	0.82	-0.98	...	99.63	13248142.0	5726815.0	4.
2	2001-06-13	83.46	83.73	82.45	82.55	7629400	6	2001	-1.09	0.23	...	99.63	13248142.0	5726815.0	4.
3	2001-06-14	82.14	82.22	80.53	80.70	12603000	6	2001	-1.76	-0.50	...	99.63	13248142.0	5726815.0	4.
4	2001-06-15	80.21	81.19	79.87	80.83	16821100	6	2001	0.78	-0.61	...	99.63	13248142.0	5726815.0	4.
5	2001-06-18	80.70	81.22	80.21	80.44	11368300	6	2001	-0.32	-0.16	...	99.63	13248142.0	5726815.0	4.
6	2001-06-19	81.18	81.52	80.19	80.79	7732300	6	2001	-0.48	0.92	...	99.63	13248142.0	5726815.0	4.
7	2001-06-20	80.39	81.50	80.33	81.21	8787200	6	2001	1.02	-0.49	...	99.63	13248142.0	5726815.0	4.
8	2001-06-21	81.08	82.46	81.03	82.14	12259100	6	2001	1.31	-0.17	...	99.63	13248142.0	5726815.0	4.
9	2001-06-22	81.92	81.98	81.04	81.49	12212000	6	2001	-0.52	-0.27	...	99.63	13248142.0	5726815.0	4.

10 rows × 29 columns

Summary

The project aimed to analyze the relationship between SPY ETF price movements and various influencing variables. Its goal was to comprehend how changes in economic indicators and time variables affected fluctuations in the stock market, as represented by the SPY ETF. The data was collected from three different sources: SPY price data was sourced from a CSV file on Kaggle.com, federal interest rate data was scraped from bankrate.com, and the remaining indicators were obtained through the econdb.com API.

The data utilized in this project raises ethical considerations analogous to those of the previously mentioned scraped and API data. It is crucial to emphasize that the data accessed is publicly accessible and originates from government sources, ensuring its legality and ethical use. As long as the project's goals align with authorized usage and legal parameters, there should be minimal ethical concerns associated with the incorporation of this data. In general, while it's important to be attentive to adhering to the API provider's guidelines, the utilization of this data does not inherently raise significant ethical issues.

Upon obtaining the data, each dataset underwent a process of cleaning and feature engineering. This involved rectifying date formats, transforming text strings into usable formats, generating new features to capture different types of price movements, and standardizing variables to ensure consistency. Additionally, irrelevant or redundant features were identified and removed to streamline the dataset. The datasets were uploaded into a SQL database and merged into a unified table.The merged table was then queried back into Jupyter to address missing values resulting from the combination of datasets with varying lengths. To address this, rows were dropped as necessary, and 'NaN' values were either filled with static values or forward-filled based on the most recent observation.

Five visualizations were created to enhance understanding of the relationships between variables. The first visualization featured a correlation heat map displaying major features in the dataset. Expected correlations were confirmed, while unexpected correlations also surfaced. Notably, a correlation between daily volume and consumer confidence emerged, a relationship not commonly known.The second visualization showcased a multiaxis line plot illustrating the connection between SPY price, job vacancies, and the federal target interest rate. It became evident that the market typically

experienced a drop following the initial interest rate hike, followed by an upward trajectory during the remainder of the cycle. The steepest drop occurred during interest rate reductions. Meanwhile, job vacancies appeared to correlate with SPY price movements, albeit as a lagging indicator.

The third visualization featured a scatter plot comparing bond yields to federal target rates. The trendlines highlighted an inversion in bond yields as the federal target rate increased, often indicating an impending recession.The fourth visualization employed a box plot to compare intra-day price changes with after-hours price changes. Surprisingly, it revealed that the most significant price movements tended to occur during trading hours rather than after hours, contrary to common belief. Lastly, a bar chart depicted the average price movements for each month of the year. This visualization illustrated that, on average, the worst-performing months for the market were June, September, and December, While the best-performing months were February, March, and November.

In conclusion, the project has significantly expanded my understanding of various concepts. While also reinforcing commonly held but untested knowledge. This newfound knowledge will undoubtedly prove invaluable as I continue to delve deeper into market variables and work on developing predictive models in the future.