Market_Making_Model

July 5, 2024

1 Development and Implementation of an ML-Powered Market-Making Algorithm for Equity Markets

This notebook presents an advanced AI-powered algorithm designed to optimize market-making strategies in the stock market. By leveraging historical market data and sophisticated machine learning techniques, the algorithm aims to enhance trading performance through accurate price predictions, efficient portfolio management, and risk mitigation. The key components of the notebook include data collection, feature engineering, model training, backtesting, and performance evaluation.

The primary goal of this project is to demonstrate my skills and knowledge in strategy and model development, quantitative finance acumen, backtesting, and data science.

1.1 Table of Contents

1.2 Introduction

In the development of this AI-driven trading strategy, a robust foundation is crucial for identifying effective market signals. To this end, I have researched and utilized the seminal work of McLean and Pontiff (2016). Their comprehensive study provides an extensive list of predictive signals that can be categorized into four broad types: Event, Market, Valuation, and Fundamental.

McLean and Pontiff (2016) Signals:

Event	Market	Valuation	Fundamental
Change in Asset Turnover	52-Week High	Advertising/MV	Accruals
Change in Profit Margin	Age-Momentum	Analyst Value	Age

Event	Market	Valuation	Fundamental
Change in Recommendation	Amihud's Measure	Book-to-Market	Asset Growth
Chg. Forecast + Accrual	Beta	Cash Flow/MV	Asset Turnover
Debt Issuance	Bid/Ask Spread	Dividends	Cash Flow Variance
Dividend Initiation	Coskewness	Earnings-to-Price	Earnings Consistency
Dividend Omission	Idiosyncratic Risk	Enterprise Component of B/P	Forecast Dispersion
Dividends	Industry Momentum	Enterprise Multiple	G Index
Down Forecast	Lagged Momentum	Leverage Component of B/P	Gross Profitability
Exchange Switch	Long-term Reversal	Marketing/MV	G-Score
Growth in Inventory	Max	Org. Capital	G-Score 2
Growth in LTNOA	Momentum	R&D/MV	Herfindahl
IPO	Momentum and Long-term Reversal	Sales/Price	Investment
IPO + Age	Momentum-Ratings		Leverage
IPO no R&D	Momentum-Reversal		M/B and Accruals
Mergers	Price		NOA
Post Earnings Drift	Seasonality		Operating Leverage
R&D Increases	Short Interest		O-Score
Ratings Downgrades	Short-term Reversal		Pension Funding
Repurchases	Size		Percent Operating Accrual
Revenue Surprises	Volume		Percent Total Accrual
SEOs	Volume Trend		Profit Margin
Share Issuance 1-Year	Volume Variance		Profitability
Share Issuance 5-Year	Volume-Momentum		ROE
Spinoffs	Volume/MV		Sales Growth
Sustainable Growth			Tax
Total External Finance			Z-Score
Up Forecast			
ΔCapex - ΔIndustry CAPEX			
ΔNoncurrent Op. Assets			
ΔSales - ΔInventory			
ΔSales - ΔSG&A			
ΔWork. Capital			

We are particularly interested in market-moving indicators. Therefore, we will focus on the Event and Market columns for our research. Since the primary goal of this project is to demonstrate my quantitative finance skills, I chose to simplify the model development process by only considering those indicators from the two columns that can be directly obtained or derived from the data pulled through `yfinance`.

What wewill use, are the following:

Type	Metrics and Events
Price and Volume Data	
	52-Week High
	Beta
	Bid/Ask Spread
	Price
	Size (Market Capitalization)
	Volume
	Volume Trend (derived)
	Volume Variance (derived)
	Volume-Momentum (derived)
	Volume/MV (derived)
Financial Metrics and Events	
	Dividends
	IPO Information
	Share Issuance 1-Year/5-Year (requires processing historical data)
Derived Metrics	
	Momentum
	Lagged Momentum
	Seasonality (based on historical prices)
	Long-term Reversal
	Short-term Reversal
	Idiosyncratic Risk (requires advanced calculations)
	Industry Momentum (requires sector analysis)
	Age-Momentum (requires additional historical data processing)

[1]:
```
# Uncomment the line below before running
# pip install -U yfinance PyPortfolioOpt xgboost
```

[2]:
```
import yfinance as yf
import pandas as pd
import numpy as np
from statsmodels.regression.linear_model import OLS
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from sklearn.compose import ColumnTransformer
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import RandomizedSearchCV
from sklearn.feature_selection import RFECV, RFE
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.model_selection import KFold
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.tree import DecisionTreeRegressor
import xgboost as xgb
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
import seaborn as sns
import cvxpy as cp
from pypfopt import expected_returns, risk_models, EfficientFrontier, plotting
from pypfopt.discrete_allocation import DiscreteAllocation, get_latest_prices
from pypfopt.plotting import plot_efficient_frontier
import joblib
import warnings

warnings.filterwarnings('ignore')
```

1.3 Data Collection and Preprocessing

We will collect historical trade data, order book data, and market news using `yfinance`. The data will be cleaned and preprocessed to ensure it is ready for analysis.

[3]:
```
tickers = ["AAPL", "MSFT", "GOOG", "NVDA"] # Arbitrary
period = "5y"
start_date = "2019-07-04"
end_date = "2024-07-03"

def get_stock_data(ticker, start_date, end_date):
    stock = yf.Ticker(ticker)
    hist = stock.history(start=start_date, end=end_date)
    return hist

stock_data = {ticker: get_stock_data(ticker, start_date, end_date) for ticker␣
 ↪in tickers}

stock_data['AAPL'].head()
```

[3]:
```
                                 Open       High        Low      Close  \
Date
2019-07-05 00:00:00-04:00  49.063133  49.480536  48.954556  49.275452
2019-07-08 00:00:00-04:00  48.450294  48.592645  47.871238  48.259689
2019-07-09 00:00:00-04:00  48.061837  48.619180  47.967741  48.554039
```

```
2019-07-10 00:00:00-04:00  48.701220  49.154813  48.631248  49.034176
2019-07-11 00:00:00-04:00  49.053477  49.314054  48.667440  48.677090

                              Volume  Dividends  Stock Splits
Date
2019-07-05 00:00:00-04:00   69062000        0.0           0.0
2019-07-08 00:00:00-04:00  101354400        0.0           0.0
2019-07-09 00:00:00-04:00   82312000        0.0           0.0
2019-07-10 00:00:00-04:00   71588400        0.0           0.0
2019-07-11 00:00:00-04:00   80767200        0.0           0.0
```

1.4 3. Feature Engineering

We have identified key features that impact market movements above. Some of them require derivation and processing.

Feature Definitions

- **52_Week_High**: The highest closing price of the stock over the past 52 weeks.
- **Momentum**: The percentage change in the stock's closing price over the past 21 trading days.
- **Lagged_Momentum**: Momentum shifted by 21 days to capture past performance trends.
- **Volume_Trend**: The 21-day moving average of the stock's trading volume.
- **Volume_Variance**: The variance of the stock's trading volume over the past 21 days.
- **Volume_Momentum**: The percentage change in the stock's trading volume over the past 21 days.
- **Volume_MV**: The ratio of the stock's trading volume to its market capitalization.
- **Long_Term_Reversal**: The percentage change in the stock's closing price over the past 252 trading days, indicating mean reversion tendencies.
- **Short_Term_Reversal**: The percentage change in the stock's closing price over the past 5 trading days, indicating short-term mean reversion.
- **Beta**: A measure of the stock's volatility relative to the market.
- **Market_Cap**: The total market value of the company's outstanding shares.
- **Seasonality**: The average closing price of the stock for each month, capturing seasonal trends.
- **Idiosyncratic_Risk**: The standard deviation of the residuals from the regression of the stock's returns against market returns, indicating risk specific to the stock.
- **Sector_Momentum**: The percentage change in the closing price of the sector ETF (e.g., Technology Select Sector SPDR Fund) over the past 21 trading days, capturing the momentum of the sector to which the stock belongs.

These features provide a comprehensive view of various aspects of the stock's performance and are used to enhance the predictive power of our models.

[4]:
```python
def generate_features(hist, market_cap, beta):
    hist['52_week_high'] = hist['Close'].rolling(window=252).max()
    hist['momentum'] = hist['Close'] / hist['Close'].shift(21) - 1
    hist['lagged_momentum'] = hist['momentum'].shift(21)
```

```
    hist['volume_trend'] = hist['Volume'].rolling(window=21).mean()
    hist['volume_variance'] = hist['Volume'].rolling(window=21).var()
    hist['volume_momentum'] = hist['Volume'] / hist['Volume'].shift(21) - 1
    hist['volume_mv'] = hist['Volume'] / market_cap
    hist['long_term_reversal'] = hist['Close'].shift(252) / hist['Close'] - 1
    hist['short_term_reversal'] = hist['Close'].shift(5) / hist['Close'] - 1
    hist['beta'] = beta
    hist['market_cap'] = market_cap
    hist['seasonality'] = hist['Close'].groupby(hist.index.month).
↪transform('mean')
    # Idiosyncratic Risk (assuming S&P 500 as market proxy)
    market_returns = yf.Ticker('^GSPC').history(period=period)['Close'].
↪pct_change().dropna()
    stock_returns = hist['Close'].pct_change().dropna()
    combined = pd.concat([stock_returns, market_returns], axis=1).dropna()
    combined.columns = ['Stock', 'Market']
    from statsmodels.regression.linear_model import OLS
    model = OLS(combined['Stock'], combined['Market']).fit()
    hist['idiosyncratic_risk'] = model.resid.std()
    # Industry Momentum
    sector_ticker = yf.Ticker('XLK')  # Technology Select Sector SPDR Fund
    sector_hist = sector_ticker.history(period=period)
    sector_hist['sector_momentum'] = sector_hist['Close'] /␣
↪sector_hist['Close'].shift(21) - 1
    hist = hist.join(sector_hist['sector_momentum'], rsuffix='_sector')
    return hist

# Additional information for beta and market cap
def get_additional_info(ticker):
    stock = yf.Ticker(ticker)
    summary = stock.info
    market_cap = summary['marketCap']
    beta = summary['beta']
    return market_cap, beta

for ticker in tickers:
    market_cap, beta = get_additional_info(ticker)
    stock_data[ticker] = generate_features(stock_data[ticker], market_cap, beta)

stock_data['AAPL'].head()
```

[4]:

Date	Open	High	Low	Close	\
2019-07-05 00:00:00-04:00	49.063133	49.480536	48.954556	49.275452	
2019-07-08 00:00:00-04:00	48.450294	48.592645	47.871238	48.259689	
2019-07-09 00:00:00-04:00	48.061837	48.619180	47.967741	48.554039	
2019-07-10 00:00:00-04:00	48.701220	49.154813	48.631248	49.034176	

```
2019-07-11 00:00:00-04:00  49.053477  49.314054  48.667440  48.677090

                             Volume  Dividends  Stock Splits  52_week_high  \
Date
2019-07-05 00:00:00-04:00   69062000        0.0           0.0           NaN
2019-07-08 00:00:00-04:00  101354400        0.0           0.0           NaN
2019-07-09 00:00:00-04:00   82312000        0.0           0.0           NaN
2019-07-10 00:00:00-04:00   71588400        0.0           0.0           NaN
2019-07-11 00:00:00-04:00   80767200        0.0           0.0           NaN

                           momentum  lagged_momentum  …  volume_variance  \
Date                                                  …
2019-07-05 00:00:00-04:00       NaN              NaN  …              NaN
2019-07-08 00:00:00-04:00       NaN              NaN  …              NaN
2019-07-09 00:00:00-04:00       NaN              NaN  …              NaN
2019-07-10 00:00:00-04:00       NaN              NaN  …              NaN
2019-07-11 00:00:00-04:00       NaN              NaN  …              NaN

                           volume_momentum  volume_mv  long_term_reversal  \
Date
2019-07-05 00:00:00-04:00              NaN   0.000020                 NaN
2019-07-08 00:00:00-04:00              NaN   0.000030                 NaN
2019-07-09 00:00:00-04:00              NaN   0.000024                 NaN
2019-07-10 00:00:00-04:00              NaN   0.000021                 NaN
2019-07-11 00:00:00-04:00              NaN   0.000024                 NaN

                           short_term_reversal  beta     market_cap  \
Date
2019-07-05 00:00:00-04:00                  NaN  1.25  3397269848064
2019-07-08 00:00:00-04:00                  NaN  1.25  3397269848064
2019-07-09 00:00:00-04:00                  NaN  1.25  3397269848064
2019-07-10 00:00:00-04:00                  NaN  1.25  3397269848064
2019-07-11 00:00:00-04:00                  NaN  1.25  3397269848064

                           seasonality  idiosyncratic_risk  sector_momentum
Date
2019-07-05 00:00:00-04:00   127.089238            0.012146              NaN
2019-07-08 00:00:00-04:00   127.089238            0.012146              NaN
2019-07-09 00:00:00-04:00   127.089238            0.012146              NaN
2019-07-10 00:00:00-04:00   127.089238            0.012146              NaN
2019-07-11 00:00:00-04:00   127.089238            0.012146              NaN

[5 rows x 21 columns]
```

For this project, I will **predict the Next Day's Closing Price**.

Understanding:

- **Foreward Looking Relevance:** Useful for traders to make future buy/sell/hold decisions.
- **Feasibility:** Influenced by short-term factors captured in daily trading data.
- **Simplicity:** It offers a straightforward evaluation of model performance, as actual next day prices are easily obtainable.
- **Short-Term Trading:** It aligns well with short-term trading and market making activities, providing immediate insights for trading decisions.

[5]:
```
for ticker in stock_data.keys():
    stock_data[ticker]['Next_Close'] = stock_data[ticker]['Close'].shift(-1)

stock_data = {ticker: data.dropna() for ticker, data in stock_data.items()}

combined_data = pd.concat(stock_data.values(), keys=stock_data.keys())
stock_data['AAPL'].head()
```

[5]:

Date	Open	High	Low	Close	\
2020-07-06 00:00:00-04:00	90.336900	91.748109	90.305159	91.276894	
2020-07-07 00:00:00-04:00	91.657782	92.441513	90.881374	90.993683	
2020-07-08 00:00:00-04:00	91.977632	93.144687	91.889732	93.112946	
2020-07-09 00:00:00-04:00	94.011412	94.065126	92.458598	93.513344	
2020-07-10 00:00:00-04:00	93.105614	93.735535	92.490349	93.676933	

Date	Volume	Dividends	Stock Splits	52_week_high	\
2020-07-06 00:00:00-04:00	118655600	0.0	0.0	91.276894	
2020-07-07 00:00:00-04:00	112424400	0.0	0.0	91.276894	
2020-07-08 00:00:00-04:00	117092000	0.0	0.0	93.112946	
2020-07-09 00:00:00-04:00	125642800	0.0	0.0	93.513344	
2020-07-10 00:00:00-04:00	90257200	0.0	0.0	93.676933	

Date	momentum	lagged_momentum	…	volume_momentum	\
2020-07-06 00:00:00-04:00	0.159872	0.086142	…	0.355129	
2020-07-07 00:00:00-04:00	0.124254	0.105669	…	-0.180881	
2020-07-08 00:00:00-04:00	0.143676	0.100819	…	0.224115	
2020-07-09 00:00:00-04:00	0.113434	0.109180	…	-0.149409	
2020-07-10 00:00:00-04:00	0.087405	0.120092	…	-0.458408	

Date	volume_mv	long_term_reversal	short_term_reversal	\
2020-07-06 00:00:00-04:00	0.000035	-0.460154	-0.054086	
2020-07-07 00:00:00-04:00	0.000033	-0.469637	-0.029274	
2020-07-08 00:00:00-04:00	0.000034	-0.478547	-0.043449	
2020-07-09 00:00:00-04:00	0.000037	-0.475645	-0.049346	
2020-07-10 00:00:00-04:00	0.000027	-0.480373	-0.051006	

```
                           beta     market_cap  seasonality  \
Date
2020-07-06 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-07 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-08 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-09 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-10 00:00:00-04:00  1.25  3397269848064   127.089238

                           idiosyncratic_risk  sector_momentum  Next_Close
Date
2020-07-06 00:00:00-04:00            0.012146         0.086987   90.993683
2020-07-07 00:00:00-04:00            0.012146         0.048044   93.112946
2020-07-08 00:00:00-04:00            0.012146         0.059820   93.513344
2020-07-09 00:00:00-04:00            0.012146         0.058672   93.676933
2020-07-10 00:00:00-04:00            0.012146         0.040994   93.244774

[5 rows x 22 columns]
```

[6]:
```
def calculate_missing_percentage(df):
    total_rows = len(df)
    missing_rows = df.isnull().any(axis=1).sum()
    missing_percentage = (missing_rows / total_rows) * 100
    return total_rows, missing_rows, missing_percentage

missing_summary = {ticker: calculate_missing_percentage(data) for ticker, data␣
 ↪in stock_data.items()}
missing_summary_df = pd.DataFrame(missing_summary, index=['Total Rows',␣
 ↪'Missing Rows', 'Missing Percentage']).T
missing_summary_df
```

[6]:
```
      Total Rows  Missing Rows  Missing Percentage
AAPL      1004.0           0.0                 0.0
MSFT      1004.0           0.0                 0.0
GOOG      1004.0           0.0                 0.0
NVDA      1004.0           0.0                 0.0
```

[7]:
```
stock_data = {ticker: data.dropna() for ticker, data in stock_data.items()}
stock_data['AAPL'].head()
```

[7]:
```
                                 Open       High        Low      Close  \
Date
2020-07-06 00:00:00-04:00  90.336900  91.748109  90.305159  91.276894
2020-07-07 00:00:00-04:00  91.657782  92.441513  90.881374  90.993683
2020-07-08 00:00:00-04:00  91.977632  93.144687  91.889732  93.112946
2020-07-09 00:00:00-04:00  94.011412  94.065126  92.458598  93.513344
2020-07-10 00:00:00-04:00  93.105614  93.735535  92.490349  93.676933
```

```
                              Volume  Dividends  Stock Splits  52_week_high  \
Date
2020-07-06 00:00:00-04:00  118655600        0.0           0.0     91.276894
2020-07-07 00:00:00-04:00  112424400        0.0           0.0     91.276894
2020-07-08 00:00:00-04:00  117092000        0.0           0.0     93.112946
2020-07-09 00:00:00-04:00  125642800        0.0           0.0     93.513344
2020-07-10 00:00:00-04:00   90257200        0.0           0.0     93.676933

                           momentum  lagged_momentum  …  volume_momentum  \
Date                                                  …
2020-07-06 00:00:00-04:00  0.159872         0.086142  …         0.355129
2020-07-07 00:00:00-04:00  0.124254         0.105669  …        -0.180881
2020-07-08 00:00:00-04:00  0.143676         0.100819  …         0.224115
2020-07-09 00:00:00-04:00  0.113434         0.109180  …        -0.149409
2020-07-10 00:00:00-04:00  0.087405         0.120092  …        -0.458408

                           volume_mv  long_term_reversal  short_term_reversal  \
Date
2020-07-06 00:00:00-04:00   0.000035           -0.460154            -0.054086
2020-07-07 00:00:00-04:00   0.000033           -0.469637            -0.029274
2020-07-08 00:00:00-04:00   0.000034           -0.478547            -0.043449
2020-07-09 00:00:00-04:00   0.000037           -0.475645            -0.049346
2020-07-10 00:00:00-04:00   0.000027           -0.480373            -0.051006

                           beta     market_cap  seasonality  \
Date
2020-07-06 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-07 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-08 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-09 00:00:00-04:00  1.25  3397269848064   127.089238
2020-07-10 00:00:00-04:00  1.25  3397269848064   127.089238

                           idiosyncratic_risk  sector_momentum  Next_Close
Date
2020-07-06 00:00:00-04:00            0.012146         0.086987   90.993683
2020-07-07 00:00:00-04:00            0.012146         0.048044   93.112946
2020-07-08 00:00:00-04:00            0.012146         0.059820   93.513344
2020-07-09 00:00:00-04:00            0.012146         0.058672   93.676933
2020-07-10 00:00:00-04:00            0.012146         0.040994   93.244774

[5 rows x 22 columns]
```

Since we are working with a range of signals of different magnitudes, they are considered to be on different scales. For instance, `Volume`, with its much higher magnitude, can tend to influence our model more than `52_week_high`. To ensure that all signals have equal weightage in our model, we will scale them.

[8]:
```
numeric_features = stock_data['AAPL'].select_dtypes(include=['float64',␣
↪'int64']).columns.drop('Next_Close')

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features)
    ]
)

def preprocess_data(df):
    df[numeric_features] = preprocessor.fit_transform(df[numeric_features])
    return df

stock_data = {ticker: preprocess_data(data) for ticker, data in stock_data.
↪items()}
stock_data['AAPL'].head()
```

[8]:

Date	Open	High	Low	Close	Volume	Dividends	Stock Splits	52_week_high	momentum	lagged_momentum	…	volume_momentum	volume_mv	long_term_reversal	short_term_reversal	beta
2020-07-06 00:00:00-04:00	-2.419776	-2.424778	-2.356480	-2.383529	0.809735	-0.12703	-0.031575	-2.970239	1.688998	0.783114	…	0.591630	0.809735	-1.296796	-1.278726	0.0
2020-07-07 00:00:00-04:00	-2.368597	-2.397976	-2.334181	-2.394484	0.657193	-0.12703	-0.031575	-2.970239	1.253362	1.021446	…	-0.528859	0.657193	-1.345333	-0.650422	0.0
2020-07-08 00:00:00-04:00	-2.356204	-2.370795	-2.295160	-2.312511	0.771458	-0.12703	-0.031575	-2.899660	1.490906	0.962242	…	0.317756	0.771458	-1.390938	-1.009373	0.0
2020-07-09 00:00:00-04:00	-2.277403	-2.335217	-2.273146	-2.297024	0.980784	-0.12703	-0.031575	-2.884269	1.121028	1.064293	…	-0.463070	0.980784	-1.376086	-1.158704	0.0
2020-07-10 00:00:00-04:00	-2.312499	-2.347957	-2.271918	-2.290697	0.114532	-0.12703	-0.031575	-2.877980	0.802683	1.197473	…	-1.109009	0.114532			

```
2020-07-10 00:00:00-04:00          -1.400284           -1.200743   0.0

                           market_cap  seasonality  idiosyncratic_risk  \
Date
2020-07-06 00:00:00-04:00         0.0    -1.322234        1.734723e-18
2020-07-07 00:00:00-04:00         0.0    -1.322234        1.734723e-18
2020-07-08 00:00:00-04:00         0.0    -1.322234        1.734723e-18
2020-07-09 00:00:00-04:00         0.0    -1.322234        1.734723e-18
2020-07-10 00:00:00-04:00         0.0    -1.322234        1.734723e-18

                           sector_momentum  Next_Close
Date
2020-07-06 00:00:00-04:00         1.124061   90.993683
2020-07-07 00:00:00-04:00         0.481021   93.112946
2020-07-08 00:00:00-04:00         0.675479   93.513344
2020-07-09 00:00:00-04:00         0.656509   93.676933
2020-07-10 00:00:00-04:00         0.364615   93.244774

[5 rows x 22 columns]
```

We will split our data into training and testing sets. Since we are working with time series data, we will not shuffle the splits. Instead, we will maintain the chronological order and perform a 75:25 split.

[9]:
```
numeric_features = combined_data.select_dtypes(include=['float64', 'int64']).
 ↪columns.drop('Next_Close')

X = combined_data[numeric_features]
y = combined_data['Next_Close']

split_index = int(len(X) * 0.75)

X_train, X_test = X.iloc[:split_index], X.iloc[split_index:]
y_train, y_test = y.iloc[:split_index], y.iloc[split_index:]
```

[10]:
```
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (3012, 21)
X_test shape: (1004, 21)
y_train shape: (3012,)
y_test shape: (1004,)
```

We are working with a large number of indicators, and we aim to eliminate redundant ones. To achieve this, I perform feature selection with cross-validation to prevent overfitting and data leakage. To capture the complex nature of our dataset, we will ensure that at least 5 predictors are selected.

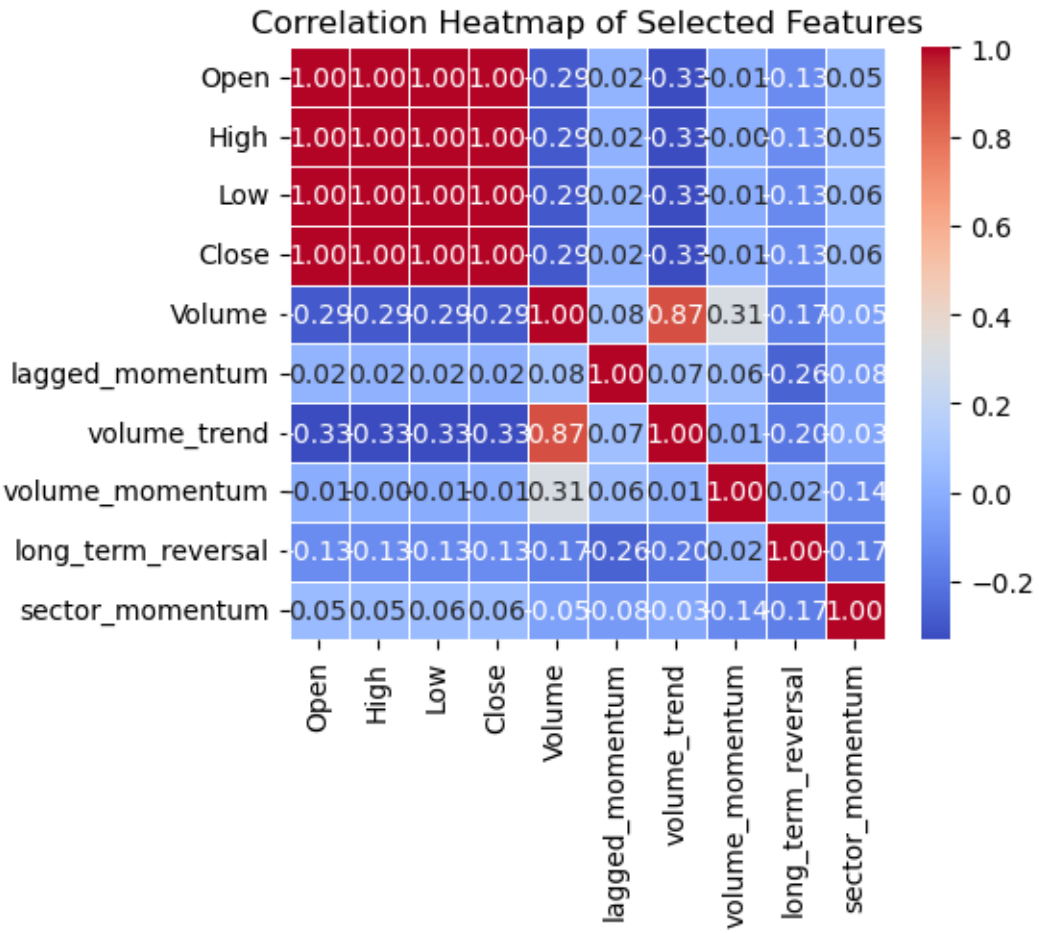
I chose DecisionTreeRegressor as the base model due to its ability to handle non-linear relationships and interactions between features effectively.

[11]:
```
base_model = DecisionTreeRegressor(random_state=42)
cv = KFold(n_splits=5, shuffle=True, random_state=42)

rfecv = RFECV(estimator=base_model, step=1, cv=cv, scoring='r2', n_jobs=-1,␣
 ↪min_features_to_select=5)
rfecv.fit(X_train, y_train)

ranking = rfecv.ranking_
selected_features = X_train.columns[rfecv.support_]

print("RFECV Rankings:", ranking)
print("Selected features:", selected_features)
```

```
RFECV Rankings: [ 1  1  1  1  1  8 12  6  5  1  1  3  1  4  1  2  9 11  7 10  1]
Selected features: Index(['Open', 'High', 'Low', 'Close', 'Volume',
'lagged_momentum',
       'volume_trend', 'volume_momentum', 'long_term_reversal',
       'sector_momentum'],
      dtype='object')
```

[12]:
```
X_train_selected = X_train[selected_features]
X_test_selected = X_test[selected_features]
```

[13]:
```
correlation_matrix = X_train_selected.corr()

plt.figure(figsize=(5, 4))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm',␣
 ↪linewidths=0.5)
plt.title('Correlation Heatmap of Selected Features')
plt.show()
```



The model above selected more than 5 predictors, some of which are perfectly correlated. Ideally, we aim to prevent multicollinearity by eliminating redundant indicators. However, in our specific use case, correlated predictors can still be effective for predicting the next day's closing price. This is because financial markets are influenced by various factors that often move together. For example, `Open`, `High`, `Low`, and `Close` prices are naturally correlated because they are derived from the same trading day.

In this context, correlation can imply causation as these features are not only related but also influence each other directly. For instance, a high opening price can lead to higher closing prices, and high trading volumes can indicate market sentiment that drives prices. Including these correlated features enhances the model's ability to capture these causal relationships and the intricate patterns within the data, thus improving its predictive power.

Even though we eventually use a linear model, which is generally sensitive to multicollinearity, the inclusion of correlated predictors in our feature selection phase helps in identifying the most significant signals. During the training of the linear model, techniques such as regularization (Ridge regression) can mitigate the impact of multicollinearity, allowing us to retain the predictive benefits of correlated features while ensuring model stability and performace. .

1.5 Model Development

We will develop machine learning models to predict market trends using the engineered features. The models will be trained and validated using historical data.

We will train and hyperparameter tune our model. I have setup data pipelines to prevent any data leakages from the test data into the training process. I also make sure that the `RandomSearchCV` hyperparameter tuner utilizes cross-validation when fitting and experimenting with model parameters. This way I can ensure that I prevent overfitting and I do not get overly optimistic results.

[14]:
```
def train_and_evaluate_model(X_train, y_train, X_test, y_test, model,␣
↪param_dist, cv, scoring='r2', model_filename='best_model.pkl'):
    pipeline = Pipeline(steps=[
        ('regressor', model)
    ])

    random_search = RandomizedSearchCV(
        pipeline,
        param_distributions=param_dist,
        n_iter=100,
        cv=cv,
        scoring=scoring,
        random_state=42,
        n_jobs=-1  # Use all available cores
    )

    random_search.fit(X_train, y_train)
    best_params = random_search.best_params_
    best_score = random_search.best_score_
    print(f"Best parameters for {model.__class__.__name__}:", best_params)
    print(f"Best cross-validation score for {model.__class__.__name__}:",␣
↪best_score)

    y_pred = random_search.best_estimator_.predict(X_test)
    test_score = r2_score(y_test, y_pred)
    test_mse = mean_squared_error(y_test, y_pred)

    print(f"Test set R² score for {model.__class__.__name__}:", test_score)
    print(f"Test set MSE score for {model.__class__.__name__}:", test_mse)
    print("\n")

    joblib.dump(random_search.best_estimator_, model_filename)

    return random_search.best_estimator_, best_params, best_score, test_score
```

[15]:
```
param_dist_lr = {}
param_dist_ridge = {
    'regressor__alpha': [0.01, 0.1, 1.0, 10.0, 100.0]
```

```
}

param_dist_rf = {
    'regressor__n_estimators': [1, 2, 3, 4, 5],
    'regressor__max_depth': [3, 5, 7]
}

param_dist_gb = {
    'regressor__n_estimators': [5, 10, 20, 25],
    'regressor__learning_rate': [0.01, 0.1, 0.15],
    'regressor__max_depth': [3, 5, 7]
}

param_dist_xgb = {
    'regressor__n_estimators': [5, 10, 20, 25],
    'regressor__learning_rate': [0.01, 0.1, 0.15],
    'regressor__max_depth': [3, 5, 7]
}


# Linear Regression model
best_lr, best_params_lr, best_score_lr, test_score_lr =␣
 ↪train_and_evaluate_model(
    X_train_selected, y_train, X_test_selected, y_test, LinearRegression(),␣
 ↪param_dist_lr, cv, model_filename='best_lr_model.pkl'
)

# Ridge Regression model
best_ridge, best_params_ridge, best_score_ridge, test_score_ridge =␣
 ↪train_and_evaluate_model(
    X_train_selected, y_train, X_test_selected, y_test, Ridge(random_state=42),␣
 ↪param_dist_ridge, cv, model_filename='best_ridge_model.pkl'
)

# RandomForestRegressor model
best_rf, best_params_rf, best_score_rf, test_score_rf =␣
 ↪train_and_evaluate_model(
    X_train_selected, y_train, X_test_selected, y_test,␣
 ↪RandomForestRegressor(random_state=42), param_dist_rf, cv,␣
 ↪model_filename='best_rf_model.pkl'
)

# GradientBoostingRegressor model
best_gb, best_params_gb, best_score_gb, test_score_gb =␣
 ↪train_and_evaluate_model(
```

```
    X_train_selected, y_train, X_test_selected, y_test,␣
 ↪GradientBoostingRegressor(random_state=42), param_dist_gb, cv,␣
 ↪model_filename='best_gb_model.pkl'
)

# XGBRegressor model
best_xgb, best_params_xgb, best_score_xgb, test_score_xgb =␣
 ↪train_and_evaluate_model(
    X_train_selected, y_train, X_test_selected, y_test, xgb.
 ↪XGBRegressor(objective='reg:squarederror', random_state=42), param_dist_xgb,␣
 ↪cv, model_filename='best_xgb_model.pkl'
)
```

```
Best parameters for LinearRegression: {}
Best cross-validation score for LinearRegression: 0.9983687065897395
Test set R² score for LinearRegression: 0.9972457296538665
Test set MSE score for LinearRegression: 1.7239942644652835


Best parameters for Ridge: {'regressor__alpha': 100.0}
Best cross-validation score for Ridge: 0.9983715306557774
Test set R² score for Ridge: 0.997302990618498
Test set MSE score for Ridge: 1.6881526214177534


Best parameters for RandomForestRegressor: {'regressor__n_estimators': 5,
'regressor__max_depth': 7}
Best cross-validation score for RandomForestRegressor: 0.9979965895222609
Test set R² score for RandomForestRegressor: -2.750388378017309
Test set MSE score for RandomForestRegressor: 2347.4994247734685


Best parameters for GradientBoostingRegressor: {'regressor__n_estimators': 25,
'regressor__max_depth': 5, 'regressor__learning_rate': 0.15}
Best cross-validation score for GradientBoostingRegressor: 0.9978655730706679
Test set R² score for GradientBoostingRegressor: -3.118081322660167
Test set MSE score for GradientBoostingRegressor: 2577.6513154687705


Best parameters for XGBRegressor: {'regressor__n_estimators': 25,
'regressor__max_depth': 5, 'regressor__learning_rate': 0.15}
Best cross-validation score for XGBRegressor: 0.9977752788655601
Test set R² score for XGBRegressor: -3.1597071332802633
Test set MSE score for XGBRegressor: 2603.7063680758993
```

- T#est set MSE score: 2610.42

1.5.1 Understanding the Results

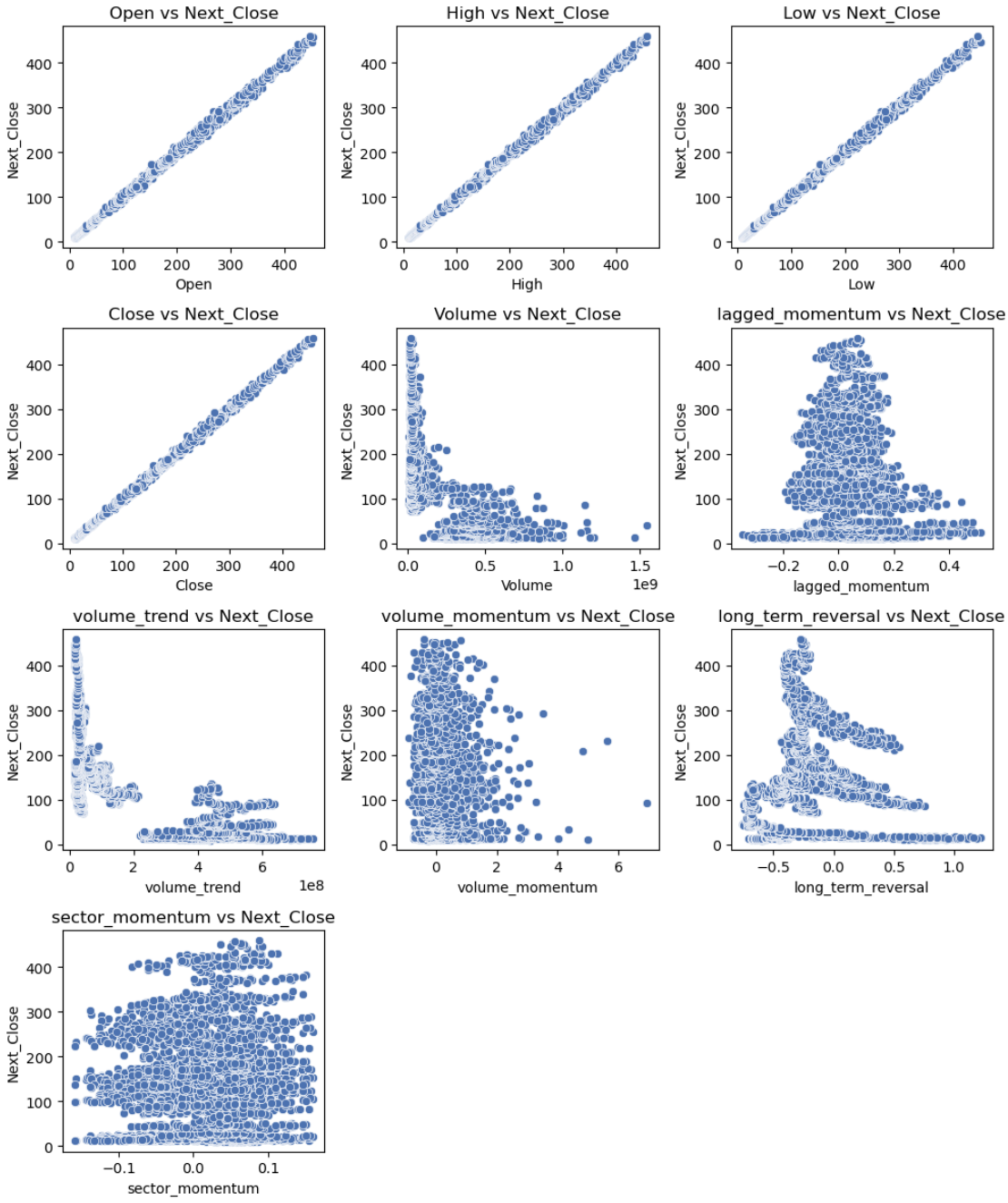
The Linear Regression and Ridge Regression models perform exceptionally well, achieving high cross-validation and test set R^2 scores with low MSE scores. This suggests that the linear models are sufficient for capturing the relationships in our data.

On the other hand, the more complex models (Random Forest, Gradient Boosting, and XGB Regressors) exhibit negative R^2 scores on the test set and significantly higher MSE scores. This indicates overfitting on the training data and poo#r generalization to unseen The Ridge Regression model was chosen for its balance of high predictive power and regularization capabilities: - **Performance**: High test R^2 (0.997) and low MSE (1.68) indicate excellent generalization. - **Regularization**: Ridge's L2 regularization helps mitigate overfitting. - **Simplicity**: Linear models are computationally efficient and interpretable.

Complex models (Random Forest, Gradient Boosting, XGB) overfit the training data, evidenced by negative test R^2 scores and high MSEs, making them unsuitable for this dataset.

The Ridge model's ability to handle multicollinearity while maintaining simplicity and computational efficiency makes it the optimal choice for our predictive task. making it the optimal choice for this project.

[16]:
```
def plot_features_grid(features, data, target='Next_Close', cols=3):
    rows = (len(features) + cols - 1) // cols
    fig, axes = plt.subplots(rows, cols, figsize=(10, 3 * rows))
    axes = axes.flatten()

    for i, feature in enumerate(features):
        sns.scatterplot(ax=axes[i], data=data, x=feature, y=target)
        axes[i].set_title(f'{feature} vs {target}')
        axes[i].set_xlabel(feature)
        axes[i].set_ylabel(target)

    # Hiding any unused subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()

plot_features_grid(selected_features, combined_data)
```



The scatter plots above illustrate the relationships between each selected feature and the next day's closing price, affirming our choice of a linear model.

- **Open, High, Low, and Close Prices**: These features show a near-perfect linear relationship with the next day's closing price, indicating that the current day's prices are strong predictors of the next day's close. This is a clear reason why linear models outperform complex models in this scenario; the linear dependencies are easily captured by simpler models without the risk of overfitting.

- **52_week_high**: This feature also shows a relatively strong linear correlation with the next day's closing price, further supporting the efficacy of linear models.
- **Volume**: Volume exhibits an inverse correlation with the next day's close, where higher volumes are associated with lower next-day closing prices. This relationship, although not as strong as price-based features, is still effectively captured by linear models.
- **Volume Trend**: Higher volume trends indicate sustained interest in the stock, which can inversely correlate with the next day's closing price. Linear models handle this feature well because the relationship, though inverse, is consistent and doesn't require complex interactions to capture its effect.
- **Volume Momentum**: Higher volume momentum suggests increased trading activity, which can inversely impact the next day's closing price.
- **Lagged Momentum**: This feature appears to approach a normal distribution with a spread around zero, suggesting that recent momentum tends to revert to the mean. This pattern is well-suited for linear modeling.
- **Long Term Reversal**: This indicator shows inverse correlation patterns at different mean reversal levels, indicating that stocks with higher long-term reversals tend to have lower next-day closing prices.
- **Sector Momentum**: This feature is more spread out, showing less clear linearity.

These observations suggest that the relationships between our selected features and the target variable are largely linear or can be approximated linearly. Consequently, linear models perform better than complex models in this context, as they capture the main trends without overfitting.

1.6 Backtesting

To evaluate our model's performance, I implemented a backtesting strategy using historical data. Here's a concise overview:

1. **Model Loading**: We load the trained model (`best_ridge_model.pkl`) using `joblib`.
2. **Feature Selection and Prediction**: For each stock, we select the relevant features and predict the next day's closing price.
3. **Trading Strategy**:
    - **Buy Setup**: If the predicted next day's close is higher than the current close, we simulate a buy. Return is calculated as `(next_close - current_close) / current_close`.
    - **Short Setup**: If the predicted next day's close is lower than the current close, we simulate a short. Return is `(current_close - next_close) / current_close`.
    - **No Trade**: If the predicted next day's close equals the current close, no trade is executed.
    - **Note**: Each day, we engage in two trades. We square off the trade executed on the day before, and we place a new trade today based on the Next Day's Close Price. To accomodate the two trade nature, I square off my position as I trade since I already have access to the Next Day's Close Price.
4. **Return Calculation**: Total returns from all trades are accumulated and annualized by dividing by the number of years (5 years).
5. **Multi-Stock Backtesting**: The process is repeated for each stock. Annualized returns for each stock are stored and reviewed.

Why This Approach?

- **Real-World Evaluation**: Backtesting with historical data evaluates the model's performance in real market conditions.
- **Trading Strategy Assessment**: Simulating buy and short setups helps assess the strategy's profitability and risk.
- **Standardized Metrics**: Annualized returns provide a clear comparison of performance across different stocks.

[17]:
```
def backtest_model(data, selected_features, model_filename='best_ridge_model.
↪pkl'):

    best_model = joblib.load(model_filename)
    X = data[selected_features]
    y_pred = best_model.predict(X)

    total_return = 0.0

    for i in range(len(data) - 1):  # Avoiding the last row as we need the next␣
↪day's actual close
        current_close = data['Close'].iloc[i]
        next_close = data['Close'].iloc[i + 1]
        predicted_next_close = y_pred[i]

        if predicted_next_close > current_close: # Buy Setup
            trade_return = (next_close - current_close) / current_close
        elif predicted_next_close < current_close: # Short Setup
            trade_return = (current_close - next_close) / current_close
        else:
            trade_return = 0

        total_return += trade_return

    annualized_return = total_return / 5

    return annualized_return

def perform_backtesting_on_all_stocks(data, selected_features, model_filename):
    annualized_returns = {}

    for stock in data.index.get_level_values(0).unique():
        stock_data = data.loc[stock].copy()
        annualized_return = backtest_model(stock_data, selected_features,␣
↪model_filename)
        annualized_returns[stock] = annualized_return
        print(f"{stock} - Annualized Return: {annualized_return}")

    return annualized_returns
```

```
annualized_returns = perform_backtesting_on_all_stocks(combined_data,␣
↪selected_features, 'best_ridge_model.pkl')
```

```
AAPL - Annualized Return: 0.2033017546368717
MSFT - Annualized Return: 0.18657324628458774
GOOG - Annualized Return: 0.30539242825201185
NVDA - Annualized Return: 0.33221935563643895
```

1.7 Portfolio Construction and Performance Evaluation

We will construct and simulate a weighted portfolio using the `PyPortfolioOpt` library. This involves:

1. **Simulating Portfolios**: Using annualized stock returns and the covariance matrix for risk assessment.
2. **Efficient Frontier**: Charting an efficient frontier to identify optimal portfolios in the risk-return space.
3. **Optimal Portfolio**: Selecting the portfolio with the highest Sharpe ratio.
4. **Benchmark Comparison**: Comparing our optimized portfolio against the S&P 500 on return, volatility, and Sharpe ratio.

Inputs required: - **Annualized Returns**: Series, list, or array of annualized returns. - **Covariance Matrix**: Derived from stock price data for risk effectively.

[18]:
```
def get_close_prices(ticker, start_date, end_date):
    stock = yf.Ticker(ticker)
    hist = stock.history(start=start_date, end=end_date)
    return hist['Close']

close_prices = pd.DataFrame({ticker: get_close_prices(ticker, start_date,␣
↪end_date) for ticker in tickers})

close_prices.head()
```

[18]:

Date	AAPL	MSFT	GOOG	NVDA
2019-07-05 00:00:00-04:00	49.275452	130.713547	56.515186	3.982759
2019-07-08 00:00:00-04:00	48.259689	130.618179	55.754055	3.907692
2019-07-09 00:00:00-04:00	48.554039	130.141312	56.177574	3.910178
2019-07-10 00:00:00-04:00	49.034176	131.466965	56.959183	3.978534
2019-07-11 00:00:00-04:00	48.677090	131.991470	57.145470	4.133142

[26]:
```
volatility_matrix = risk_models.semicovariance(close_prices, benchmark=((1.
↪0392**(1/252))-1))
annualized_returns_array = np.array(list(annualized_returns.values()))

ef = EfficientFrontier(annualized_returns_array, volatility_matrix)
```

```
raw_weights = ef.max_sharpe()
cleaned_weights = ef.clean_weights()
```

[20]:
```
custom_return, custom_volatility, custom_sharpe_ratio = ef.
↪portfolio_performance(verbose=True)
```

```
Expected annual return: 30.5%
Annual volatility: 21.9%
Sharpe Ratio: 1.30
```

[21]:
```
stock = yf.Ticker("^GSPC")
hist = stock.history(start=start_date, end=end_date)
close_prices_sp500 = hist['Close']

close_prices_sp500.head()
```

[21]:
```
Date
2019-07-05 00:00:00-04:00    2990.409912
2019-07-08 00:00:00-04:00    2975.949951
2019-07-09 00:00:00-04:00    2979.629883
2019-07-10 00:00:00-04:00    2993.070068
2019-07-11 00:00:00-04:00    2999.909912
Name: Close, dtype: float64
```

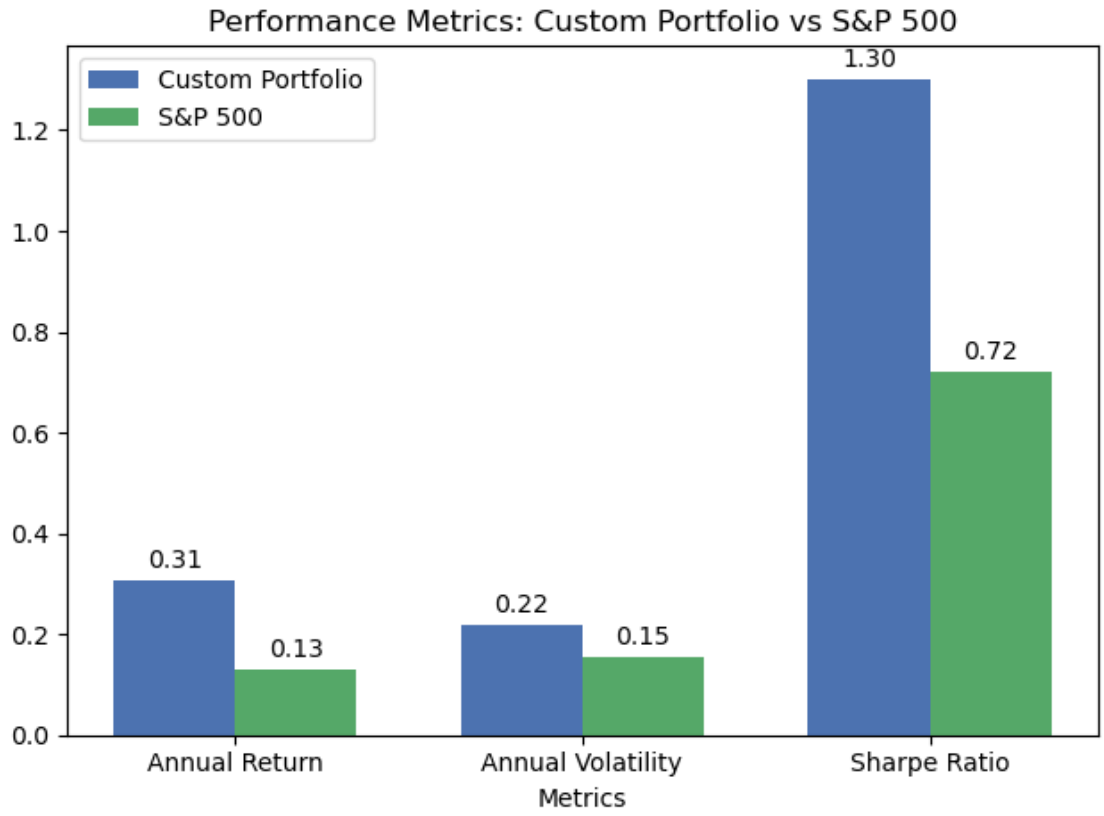
[22]:
```
annualized_returns_sp500 = expected_returns.
↪mean_historical_return(close_prices_sp500)
volatility_matrix_sp500 = risk_models.semicovariance(close_prices_sp500,␣
↪benchmark=((1.0392**(1/252))-1))

ef_sp500 = EfficientFrontier(annualized_returns_sp500, volatility_matrix_sp500)

sp_raw_weights = ef_sp500.max_sharpe()
sp_cleaned_weights = ef_sp500.clean_weights()
```

[23]:
```
sp_return, sp_volatility, sp_sharpe_ratio = ef_sp500.
↪portfolio_performance(verbose=True)
```

```
Expected annual return: 13.0%
Annual volatility: 15.3%
Sharpe Ratio: 0.72
```

[24]:
```
# Custom Portfolio Metrics
custom_metrics = {
    'Annual Return': custom_return,
    'Annual Volatility': custom_volatility,
    'Sharpe Ratio': custom_sharpe_ratio
}
```

```
# S&P 500 Metrics
sp500_metrics = {
    'Annual Return': sp_return,
    'Annual Volatility': sp_volatility,
    'Sharpe Ratio': sp_sharpe_ratio
}

metrics = list(custom_metrics.keys())
custom_values = list(custom_metrics.values())
sp500_values = list(sp500_metrics.values())

x = np.arange(len(metrics))
width = 0.35

fig, ax = plt.subplots()

rects1 = ax.bar(x - width/2, custom_values, width, label='Custom Portfolio')
rects2 = ax.bar(x + width/2, sp500_values, width, label='S&P 500')

ax.set_xlabel('Metrics')
ax.set_title('Performance Metrics: Custom Portfolio vs S&P 500')
ax.set_xticks(x)
ax.set_xticklabels(metrics)
ax.legend()

def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.2f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

fig.tight_layout()
plt.show()
```



Performance Analysis: Custom Portfolio vs S&P 500

- **Annual Return**: The custom portfolio achieves a significantly higher annual return (31%) compared to the S&P 500 (13%). This indicates the custom portfolio's strong performance in generating returns.
- **Annual Volatility**: The custom portfolio has higher volatility (22%) compared to the S&P 500 (15%), reflecting increased risk.
- **Sharpe Ratio**: The custom portfolio's Sharpe Ratio (1.30) is notably higher than the S&P 500's (0.72), indicating better risk-adjusted returns. Despite higher volatility, the custom portfolio provides superior returns per unit of risk.

Alpha and Beta under CAPM

- Alpha measures the active return on an investment compared to a market index or benchmark, indicating the value added (or subtracted) by an investment manager.
- Beta measures the sensitivity of a portfolio's returns to market movements, indicating its systematic risk.

Formula:

$$\alpha = R_i - \left(R_f + \beta \times (R_m - R_f)\right)$$

Legend: - α: Alpha (Excess Returns) - R_i: Return on my Portfolio - R_f: Risk-free rate of return

- β: Beta - R_m: Return of S&P 500 index

[25]:
```
risk_free_rate = 0.025 # Assumption

def get_stock_beta(ticker):
    stock = yf.Ticker(ticker)
    summary = stock.info
    return summary['beta']

stock_betas = {ticker: get_stock_beta(ticker) for ticker in tickers}
portfolio_beta = sum(cleaned_weights[ticker] * stock_betas[ticker] for ticker␣
 ↪in tickers)
alpha = custom_return - (risk_free_rate + portfolio_beta * (sp_return -␣
 ↪risk_free_rate))

print(f"Portfolio Beta: {portfolio_beta:.4f}")
print(f"Portfolio Alpha: {alpha:.4f}")
```

```
Portfolio Beta: 1.0460
Portfolio Alpha: 0.1701
```

- **Portfolio Beta (1.0460)**: A beta close to 1 indicates that the portfolio's systematic risk is similar to the market's risk. In this case, the custom portfolio is 1.046 times more volatile than the market. Thus, for every 1% move by the market, the portfolio moves approximately 1.046%. This suggests that the portfolio's risk is comparable to the market's risk.
- **Portfolio Alpha (0.1701)**: The custom portfolio has an alpha of 0.1701, suggesting that it has outperformed the benchmark (S&P 500) by approximately 17.01% on a risk-adjusted basis. This positive alpha indicates that the investment strategy has added value beyond what would be expected based on its beta.

1.8 Conclusion and Next Steps

In this project, I have demonstrated various skills essential for a Quantitative Equity Analyst, Researcher, or Trader.

- **Quantitative Research**: Extensive use of quantitative techniques to analyze financial data and derive meaningful insights.
- **Model Development and Validation**: Building and validating predictive models to forecast stock prices using historical data and various market signals.
- **Portfolio Construction**: Applying advanced portfolio optimization techniques to construct portfolios that maximize returns while managing risk.
- **Backtesting Strategies**: Implementing rigorous backtesting methodologies to evaluate the performance of trading strategies over historical data.
- **Financial Acumen**: A deep understanding of financial concepts such as alpha, beta, and the efficient frontier, and their application in portfolio management.
- **Programming Proficiency**: Utilizing Python and libraries such as pandas, sklearntatsmodels, and PyPortfolioOpt to execute complex financial analyses and modelinThe primary focus of this project was to demonstrate a wide range of quantitative skills rather than to produce

the most accurate predictive results. However, there are several areas where I would like to make improvements:improvement.

Areas for Improvement: - **Utilize Multiple Data Sources**: Incorporate additional data sources to assess all market and event signals from the McLean and Pontiff (2016) research. - **Implement Sentiment Analysis**: Use Natural Language Processing (NLP) to analyze sentiment from news sources to enhance signal assessment. - **Refine Model Training and Testing**: Train the model on a 3-year period and test it on a 5-year period for more robust performance evaluation. - **Develop Trading Strategies**: Devise trading strategies using the selected signals and predicted close prices. The predicted close prices can serve as price levels for trading scenarios.

[ ]: