# Time Series Forecasting on Stock Market

Osman Doğuhan Çiftçi
*Graduate School of Sciences & Engineering*
*Koc¸ University*
İstanbul, Turkey
ociftci14@ku.edu.tr

***Abstract*—Modern Portfolio Theory was invented by Harry Markowitz in 1952. It has been widely used even today because of its simplicity and performance. The theory is based on maximizing the expected return concerning lower diversity in the portfolios. This paper combines the Modern Portfolio Theory with Time Series Forecasting methods. Future values are predicted by the Time Series Forecasting and fetched to the Modern Portfolio model. An experiment is conducted with S&P500 stocks to illustrate the performance of this approach.**

***Index Terms*—Modern Portfolio Theory, Time Series Forecasting, Stock Market**

## I. Introduction

This project is designed for the DASC 591 course. The student will develop a project which will combine modern portfolio theory with time-series forecasting for the stock market. The project should be based on real stock market data, consisting of companies from several sectors such as technology, refinery, and automotive. The project will perform time-series forecasting by using the most recent 3 months data to predict the next week's stock value. Based on the predictions, the project will simulate buys and sells according to the modern portfolio theory. In the end, the student will analyze the project performance based on the Sharpe ratio.

## II. Implementation

### A. Defining Static Values

At the beginning of the implementation, the static variables are defined. The implementation stores the stock data in *history* and *experimental* variables. *History* variable holds the training data that is known before the experimentation. The boundaries of the training data are set by *historical_data_start_date* and *historical_data_end_date*. *The experimental* variable holds the test data that will be experimented on. The simulation will not use the *experimental* data before the investment. The time series forecasting will be applied to the training data to predict future values. Then, modern portfolio methods will be applied to the predicted values and investment will be placed according to the calculated weights of the portfolio theory. After the investment is placed, the real data will be retrieved from the *experimental* data. The boundaries of the test data are set by *experimental_data_start_date* and *experimental_data_end_date*. The *initial_balance* variable stores the amount of money that will be used in experimentation. The *default_lag_time* holds the initial value for the lag time parameter which is 1. The *stocks* variable is an array, and it holds the stocks that will be used in the experimentation. For the experimentation, AutoZone, Bank of America Corporation, Domino's Pizza Inc, Alphabet Inc, Microsoft Corporation, and Target Corporation stocks are selected from the S&P 500 stock market.

```
history = None
historical_data_start_date = '2021-01-01'
historical_data_end_date = '2021-09-30'

experimental = None
experimental_data_start_date = '2021-10-01'
experimental_data_end_date = '2021-12-31'

initial_balance = 100000
default_lag_time = 1

stocks = [
    'AZO',  # AutoZone
    'BAC',  # Bank of America Corporation
    'DPZ',  # Domino's Pizza, Inc.
    'GOOGL',  # Alphabet Inc.
    'MSFT',  # Microsoft Corporation
    'TGT',  # Target Corporation
]
```

### B. Data Collection

To collect the historical stock prices for history and experimental data, the ffn python library is used. Below get_data function retrieves and returns the stock prices data frame. On the first call to this function, the stock prices are collected from the internet and saved to a CSV file. The repetitive call for the same dataset is returned from the saved file.

```
def get_data(stocks, start, end, file):
    filename = file + '_' + start + '_' + end + '.csv'
    try:
        data = pd.read_csv(filename, index_col=0)
        data.index = pd.to_datetime(data.index)
        return data
    except FileNotFoundError:
        data = ffn.get(stocks, start=start, end=end)
        data.to_csv(filename)
        return data

history = get_data(','.join(stocks),  historical_data_start_date,
                   historical_data_end_date, 'history')
fig1 = history.rebase().plot()

experimental = get_data(','.join(stocks), experimental_data_start_date,
                        experimental_data_end_date, 'experimental')
fig2 = experimental.rebase().plot()
```

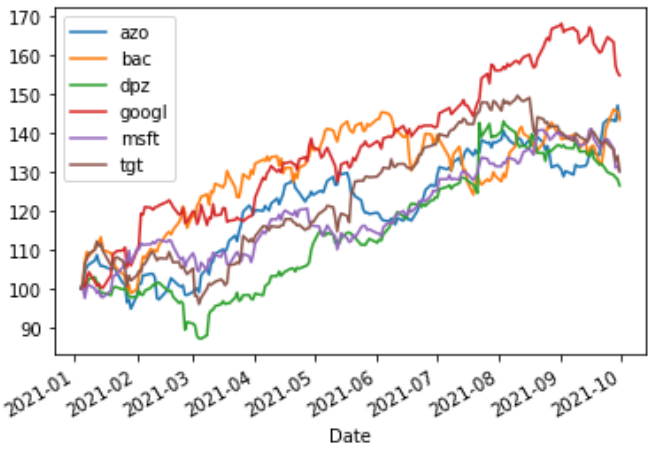


Fig. 1. Stock prices of the *history* dataset.

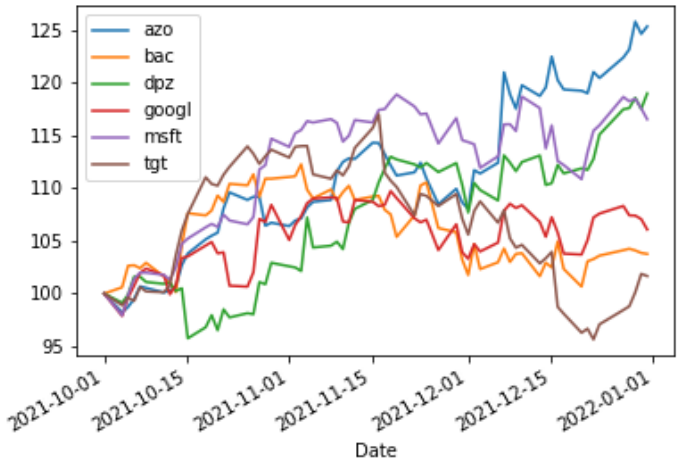


Fig. 2. Stock prices of the *experimental* dataset.

### C. Modern Portfolio Implementation

For Modern Portfolio Theory implementation, the PyPortfolioOpt library is used. PyPortfolioOpt library includes classical Efficient Frontier techniques and Black-Litterman allocation methods. Calculate_weights function calculates the expected returns and sample covariance for the given dataset. It performs the EfficientFrontier method to obtain the weights that yield in maximum Sharpe Ratio. If the *verbose* parameter is set to true, the function prints out the portfolio performance which is illustrated in Figure 3.

```
def calculate_weights(data, verbose=False):
    # Calculate expected returns and sample covariance
    mu = expected_returns.mean_historical_return(data)
    S = risk_models.sample_cov(data)

    # Optimize for maximal Sharpe ratio
    ef = EfficientFrontier(mu, S)
    raw_weights = ef.max_sharpe()
    cleaned_weights = ef.clean_weights()
    if verbose:
        print(ef.portfolio_performance(verbose=True))
    return cleaned_weights

example_weights = calculate_weights(history, True)
print('\nWeights:\n', example_weights)
```

```
Expected annual return: 65.6%
Annual volatility: 15.0%
Sharpe Ratio: 4.24
(0.6560549410998375, 0.15014515884013774, 4.236266730231753)

Weights:
 OrderedDict([('azo', 0.26368), ('bac', 0.20209), ('dpz', 0.15366), ('googl', 0.38057), ('msft', 0.0), ('tgt', 0.0)])
```

Fig. 3. Portfolio Performance and Stock Distribution

To simulate an investment, *invest* function is implemented which calculates how many stock shares to buy from each stock according to the weights. It performs the DiscreteAllocation method for calculating the allocation. This function returns the allocation dictionary and leftover balance. The *balance* parameter is updated with the leftover from the Discrete Allocation method. A sample output of this function is shown in Figure 4.

```
def invest(prices, weights, balance):


    da = DiscreteAllocation(weights, prices, total_portfolio_value=balance)
    allocation, leftover = da.greedy_portfolio()
    return allocation, leftover

example_balance = 100000
print(f'Initial Balance: {example_balance:.2f}$')
latest_prices = get_latest_prices(history)
example_allocation, example_balance = invest(latest_prices,
example_weights, example_balance)
print('After investment:')
print("\tAllocation:", example_allocation)
print(f'\tFinal Balance (leftover): {example_balance:.2f}$')
```

```
Initial Balance: 100000.00$
After investment:
        Allocation: {'googl': 14, 'azo': 15, 'bac': 478, 'dpz': 32}
        Final Balance (leftover): 1670.95$
```

Fig. 4. Stock share distribution for a particular investment

### D. Time Series Forecasting

A famous and widely used forecasting method for time-series forecasting is the AutoRegressive Integrated Moving Average (ARIMA) model. ARIMA models can capture a suite of different standard temporal structures in time-series data.

“AR” stands for Auto-Regressive, and it means that the model uses the dependent relationship between an observation and some predefined number of lagged observations (also known as “time lag” or “lag”).

“I” stands for Integrated, and it means that the model employs differencing of raw observations (e.g., it subtracts an observation from observation at the previous time step) to make the time-series stationery.

“MA” stands for Moving Average, and it means that the model exploits the relationship between the residual error and the observations.

The key point in Time Series Forecasting is selecting the right parameters for the given dataset. The model performs best when the given parameters are suitably selected according to the characteristics of the dataset. pmdarima library provides the auto_arima method which is the equivalent of R's auto.arima functionality This library tries different combinations of p, d, and q parameters, to obtain the perfect time-series forecaster for the given data. *get_arima_model* function trains the ARIMA model for the given dataset. It performs parameter tuning within the range of *start_{p,d,q}* and *max_{p,d,q}* parameters. The *trace* parameter is used to verbose logs of the auto_arima iterations. A sample run with verbose logs is shown in Figure 5.

```
def get_arima_model(data,
                    start_p=0, max_p=2,
                    start_d=0, max_d=2,
                    start_q=0, max_q=2,
                    trace=False
                    ):
    return pmd.auto_arima(
        data,
        start_p=start_p,  # initial guess for AR(p)
        start_d=start_d,  # initial guess for I(d)
        start_q=start_q,  # initial guess for MA(q)
        max_p=max_p,  # max guess for AR(p)
        max_d=max_d,  # max guess for I(d)
        max_q=max_q,  # max guess for MA(q)
        m=7,  # seasonal order
        start_P=0,  # initial guess for seasonal AR(P)
        start_D=0,  # initial guess for seasonal I(D)
        start_Q=0,  # initial guess for seasonal MA(Q)
        trend='c',
        information_criterion='aic',
        trace=trace,
        error_action='ignore'
                    )


example_model = get_arima_model(history[history.columns[0]], trace=True)

print(f'\nForecasted next value:\t{example_model.predict(10)[1]:.2f}')
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[7] intercept   : AIC=1643.631, Time=0.02 sec
 ARIMA(1,1,0)(1,0,0)[7] intercept   : AIC=1646.934, Time=0.16 sec
 ARIMA(0,1,1)(0,0,1)[7] intercept   : AIC=1646.955, Time=0.17 sec
 ARIMA(0,1,0)(0,0,0)[7]             : AIC=1643.631, Time=0.02 sec
 ARIMA(0,1,0)(1,0,0)[7] intercept   : AIC=1645.457, Time=0.10 sec
 ARIMA(0,1,0)(0,0,1)[7] intercept   : AIC=1645.473, Time=0.09 sec
 ARIMA(0,1,0)(1,0,1)[7] intercept   : AIC=1646.586, Time=0.23 sec
 ARIMA(1,1,0)(0,0,0)[7] intercept   : AIC=1645.066, Time=0.07 sec
 ARIMA(0,1,1)(0,0,0)[7] intercept   : AIC=1645.068, Time=0.06 sec
 ARIMA(1,1,1)(0,0,0)[7] intercept   : AIC=1646.559, Time=0.30 sec

Best model:  ARIMA(0,1,0)(0,0,0)[7]
Total fit time: 1.252 seconds

Forecasted next value:  1703.53
```

Fig. 5. ARIMA model parameter tuning

## III. Experimentation

In the experimentation part, Modern Portfolio Theory has been empowered by Time Series Forecasting. experiment function simulates trading with the real stock prices. Starting from the last of the training data, the Time series model is trained on the training data to forecast the future prices. Forecasted stock prices are inserted into the training data. Then, Modern Portfolio Theory is applied to the training data for maximizing the Sharpe Ratio. The balance is distributed to the stocks according to the MPT weights. Stocks are bought according to the current day's prices. Then, the implementation assumes that it waits for the lag time days and holds the stocks. After lag time days, stocks are sold according to the current day after lag time day's prices. One set of buying and selling operations is completed until now. The implementation replaces the forecasted values of training data with the real stock prices. Balance, weights, and other statistics about the iteration are stored in the log Data Frame.

```
experiment(training_data=history,
                real_data=experimental,
                balance=100000,
                lag_time_param=3,
                verbose=True)
```

A sample function call to the *experiment* function is shown above. With the static variables defined in the previous section, we can analyze a particular iteration of this program execution. *history* dataset contains stock prices between 2021-01-01 and 2021-09-30. The *experimental* dataset contains stock prices between 2021-10-01 and 2021-12-31. The initial balance 100.000$ and the lag time is 3. Starting from the 2021-09-30 (last day of history dataset), the Time Series Forecasting model is trained for each stock. Each model forecasts 3 future stock prices. Stock prices for 2021-10-01, 2021-10-04, and 2021-10-05 are forecasted. Note that the market was closed on 2021-10-02 and 2021-10-03. *history* dataset is updated with the new forecasted prices. *history* dataset now contains stock prices between 2021-01-01 and 2021-10-05. Modern Portfolio Theory is applied for the history dataset. The weights that maximize the Sharpe Ratio are calculated. 100.000$ is distributed among stocks according to the weights. The weights and allocations are shown in Figure 6.

| Stock | Price on 2021-09-30 | Weight | Allocation |
|---|---|---|---|
| azo | 1697.98 | 0.26398 | 15 |
| bac | 42.24 | 0.20218 | 478 |
| dpz | 476.12 | 0.15333 | 32 |
| googl | 2673.52 | 0.38051 | 14 |
| msft | 281.40 | 0.0 | 0 |
| tgt | 227.99 | 0.0 | 0 |

Fig. 6. Stock allocations that maxims the Sharpe Ratio.

After stocks are invested, the iteration assumes that it holds the stocks for 3 days. For the last 3 days' real stock prices, the *experimental* dataset is used. Stocks are cashed out according to the real stock prices on 2021-10-05. Balance is updated accordingly. Forecasted prices on the *history* dataset are replaced with real stock prices. The first 3 days are removed from the *experimental* dataset. History dataset now contains all real stock prices between 2021-01-01 and 2021-10-05. The *experimental* dataset now contains real stock prices between 2021-10-06 and 2021-12-31. Statistics are saved to the log data frame and iteration continues.

```
def experiment(training_data, real_data, balance, lag_time_param,verbose=False):
    lag_time = lag_time_param  # initialize lag_time

    # setting up the log dataframe columns
    log_columns = [f't + {lg + 1}' for lg in range(lag_time)]
    log_columns += ['Initial Balance', 'Lag Time', 'Final Balance']
    log_columns += stocks
    log_df = pd.DataFrame(index=pd.to_datetime([]), columns=log_columns)

    # Dataframes are copied before processing
    training_data = training_data.copy()
    real_data = real_data.copy()

    # Iterating through the experimental data
    while len(real_data) > 0:
        if len(real_data) < lag_time_param:  # if fewer stock prices than the lag time
            lag_time = len(real_data)  # update lag time to finish remaining days
        # current_data is the day that we calculate weights and invest our money
        current_date = training_data.index[-1]  # last day of the real_data
        # future_dates is the days that we will hold our stocks
        future_dates = real_data.index[:lag_time]
        # saving the prev balance for logging purposes
        prev_balance = balance
        # Time Series Forecasting for future prices
        forecasted_prices = []
        for stock in training_data.columns:
            # train a model for each stock
            model = get_arima_model(training_data[stock])
            # predict next lag_time stock prices
            forecasted_prices.append(model.predict(lag_time))

        # Update training_data with the forecasted prices
        for index, date in enumerate(future_dates):
            training_data.loc[date] = [values[index] for values in forecasted_prices]
        # Modern Portfolio Theory for maximizing Sharpe Ratio
        weights = calculate_weights(training_data, False)
        # Buy the stocks according to the current_date
        allocation, balance = invest(training_data.loc[current_date], weights, balance)
        ### Assume that we waited for a given lag_time
        ### After the given lag_time days:
        ##### We know the real stock prices
        ##### Sell the stocks with real prices according to the last day

        # real stock prices of current_date + lag_time
        future_real_values = real_data.loc[future_dates[-1]]

        for stock, share in allocation.items():
            balance += share * future_real_values[stock]

        # Replace forecasted prices with the real prices
        for date in future_dates:
            training_data.loc[date] = real_data.loc[date]

        # Remove processed values from the real_data
        real_data = real_data.iloc[lag_time:, :]

        log_record = [str(fd)[:10] for fd in future_dates] + [None] * (lag_time_param
- lag_time)
        log_record += [round(prev_balance, 2), lag_time, round(balance, 2)]
        log_record += [weights[s.lower()] for s in stocks]
        log_df.loc[current_date] = log_record
        if verbose:
            print(f'Current date:\t\t {str(current_date)[:10]}')
            print(f'Initial balance:\t {prev_balance:.2f}')
            print(f'Future dates:\t\t {", ".join([str(f)[:10] for f in future_dates])}')
            print(f'Weights:\t\t\t {weights}')
            print(f'Sell date:\t\t\t {str(future_dates[-1])[:10]}')
            print(f'Final balance:\t\t {balance:.2f}\n\n')
        # end of while loop

    # save log_df to a file
    log_df.to_csv(f'trading_log_{"verbosed_" if verbose else
""}{lag_time_param}.csv')
    return balance, log_df

# Sample run with lag_time = 15
example_balance, example_log_df = experiment(history, experimental,
initial_balance, 15, verbose=True)
```

## IV. Results

### A. *Reading the Logs*

Trading experimentation is simulated for different lag times. The below script is executed for the lag time between 1 and 30. Trading logs are saved to 'trading_log_x.csv' files. Since the experimentation is repeated 30 times, it takes some time to create log files. It is suggested to run the commented block, create the log files, and read the log data frames from the CSV files. Note that the *experiment* function saves the log data frame into a CSV file. Figures 7 and 8 show the log dataset for lag time 1 being 1 and 30, respectively.

```
"""
# Following loop simulates the trading with lag_time being 1 to 30.
Logs are saved to the csv files
for i in range(1, 31, 1):
    experiment(history, experimental, initial_balance, i)
"""
def read_logs():
    logs = []
    for i in range(1, 31, 1):
        current_log = pd.read_csv(f'logs/trading_log_{i}.csv',
index_col=0)
        current_log.index = pd.to_datetime(current_log.index)
        logs.append(current_log)
    return logs

trading_logs = read_logs()
trading_logs[0]['Lag Time'][0]
trading_logs[-1]['Lag Time'][0]
print(f'Trading logs are collected for simulation #{trading_logs[0]["Lag
Time"][0]} to simulation #{trading_logs[-1]["Lag Time"][0]}')
```

| | t + 1 | Initial Balance | Lag Time | Final Balance | AZO | BAC | DPZ | GOOGL | MSFT | TGT |
|---|---|---|---|---|---|---|---|---|---|---|
| **2021-09-30** | 2021-10-01 | 100000.00 | 1 | 100656.44 | 0.26331 | 0.20213 | 0.15453 | 0.38004 | 0.00000 | 0.0 |
| **2021-10-01** | 2021-10-04 | 100656.44 | 1 | 99405.52 | 0.22464 | 0.22092 | 0.15091 | 0.40352 | 0.00000 | 0.0 |
| **2021-10-04** | 2021-10-05 | 99405.52 | 1 | 100813.38 | 0.19738 | 0.25771 | 0.15784 | 0.38707 | 0.00000 | 0.0 |
| **2021-10-05** | 2021-10-06 | 100813.38 | 1 | 101678.04 | 0.18382 | 0.26757 | 0.15520 | 0.38204 | 0.01137 | 0.0 |
| **2021-10-06** | 2021-10-07 | 101678.04 | 1 | 102307.93 | 0.18383 | 0.25889 | 0.15942 | 0.37303 | 0.02483 | 0.0 |

Fig. 7. Log data set when lag time is equal to 1.

| | t + 1 | t + 2 | t + 3 | t + 4 | t + 5 | t + 6 | t + 7 | t + 8 | t + 9 | t + 10 | ... | t + 30 | Initial Balance | Lag Time | Final Balance | AZO | BAC | DPZ | GOOGL | MSFT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2021-09-30** | 2021-10-01 | 2021-10-04 | 2021-10-05 | 2021-10-06 | 2021-10-07 | 2021-10-08 | 2021-10-11 | 2021-10-12 | 2021-10-13 | 2021-10-14 | ... | 2021-11-11 | 100000.00 | 30 | 109584.78 | 0.26525 | 0.20301 | 0.15311 | 0.37863 | 0.00000 |
| **2021-11-11** | 2021-11-12 | 2021-11-15 | 2021-11-16 | 2021-11-17 | 2021-11-18 | 2021-11-19 | 2021-11-22 | 2021-11-23 | 2021-11-24 | 2021-11-26 | ... | 2021-12-27 | 109584.78 | 30 | 112591.68 | 0.24703 | 0.25176 | 0.10581 | 0.13726 | 0.25815 |
| **2021-12-27** | 2021-12-28 | 2021-12-29 | 2021-12-30 | 2021-12-31 | NaN | NaN | NaN | NaN | NaN | NaN | ... | NaN | 112591.68 | 4 | 112754.27 | 0.32316 | 0.16534 | 0.14192 | 0.17508 | 0.19450 |

Fig. 8. Log data set when lag time is equal to 30.

### B. *Balance Over Time*

Since logs are saved the lag time days, different lag times result in data frames with different columns. To obtain continuous balance information for each log data frame, the below script is implemented. It simply iterates through the t + columns of the log data frame and saves the balance information. Note that balance stays the same for x-1 days if lag time is x days. Figure 9 illustrates the *balances* data frame where rows are the dates and columns are the lag times.

```
trading_logs = read_logs()


# Creating indexes for the balances dataframe (only trading days
when market is open)
logs_indexes =
pd.DatetimeIndex(mcal.get_calendar('NYSE').valid_days(start_da
te='2021-09-30', end_date='2021-12-31')).tz_localize(None)

# balances dataframe for balance history for all simulations
balances = pd.DataFrame(index=logs_indexes, columns=[lt for lt
in range(1, 31, 1)])

# helper function to fill the dataframe
def place_with_index(df, indexes, column, value):
    indexer = df.index.get_indexer(pd.DatetimeIndex(indexes))
    df.iloc[indexer, column] = value


# Iterating through each simulation (1 to 30)
for lag_time, log in enumerate(trading_logs):
    # Iterating through each row (each trade)
    for index, row in log.iterrows():
        current_lag_time = row['Lag Time']
        current_initial_balance = row['Initial Balance']
        current_final_balance = row['Final Balance']

        # current_day to current_day + (lag_time - 1) => initial
balance
        place_with_index(balances, [index], lag_time,
current_initial_balance)
        place_with_index(balances, row[:current_lag_time - 1],
lag_time, current_initial_balance)

        # currentday + lag_time => final balance
        place_with_index(balances, [row[current_lag_time - 1]],
lag_time, current_final_balance)

balances
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2021-09-30 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | ... | 100000.0 | 100000.0 | 100000.0 | 100000.0 |
| 2021-10-01 | 100656.44 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | ... | 100000.0 | 100000.0 | 100000.0 | 100000.0 |
| 2021-10-04 | 99405.52 | 99393.3 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | ... | 100000.0 | 100000.0 | 100000.0 | 100000.0 |
| 2021-10-05 | 100813.38 | 99393.3 | 100722.87 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | ... | 100000.0 | 100000.0 | 100000.0 | 100000.0 |
| 2021-10-06 | 101678.04 | 101660.66 | 100722.87 | 101609.83 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | 100000.0 | ... | 100000.0 | 100000.0 | 100000.0 | 100000.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2021-12-27 | 112194.07 | 112464.07 | 112237.33 | 111899.28 | 112612.2 | 112002.71 | 107762.93 | 107213.54 | 110321.16 | 112337.95 | ... | 106306.66 | 106750.19 | 105825.27 | 108684.64 |
| 2021-12-28 | 112207.96 | 112464.07 | 112237.33 | 111899.28 | 112612.2 | 112002.71 | 107762.93 | 107213.54 | 110321.16 | 112337.95 | ... | 106306.66 | 106750.19 | 105825.27 | 108684.64 |

Fig. 9. Continuous balance history for each lag time

## C. *Profits By Lag Time*

In the previous part, the *balances* data frame is arranged to interpret the log data frames. Since rows are the dates, the final row of the data frame implies the final balances for each lag time. Subtracting the initial balance from the last row yields to the profits. Figure 10 shows the profit performance for each lag time.

```
final_balances = pd.to_numeric(balances.iloc[-1, :])

profits = final_balances - 100000 # substract initial money

profits.rebase().plot.bar(x='Lag Time', y='Final Balance', rot=0)

profits_sorted = profits.sort_values(ascending=False) # sort the
profits as descending
profits_sorted
```

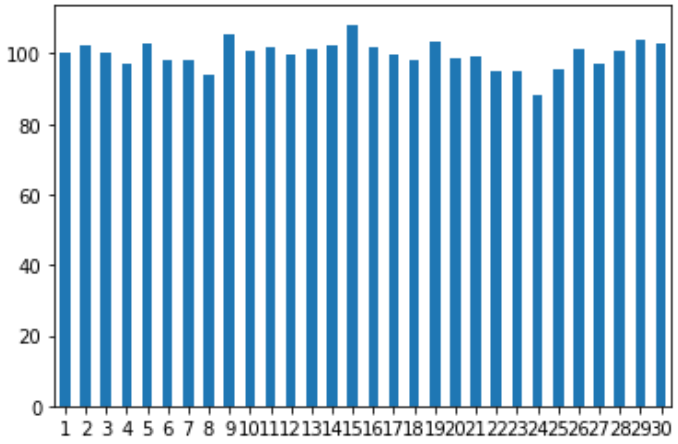


Fig. 10. Final profits for each lag time

## D. *Balance History – Most Profited*

From the balances data frame, the most profited experimentations have resulted when lag times are 15, 9, and 29. Figures 11, 12, and 13 show the balance over time in these experiments.

**Lag Time 15 - Profit 13420$**

```
# Lag_time = 15
ax1 = balances.iloc[:, 14:15].rebase().plot()
```

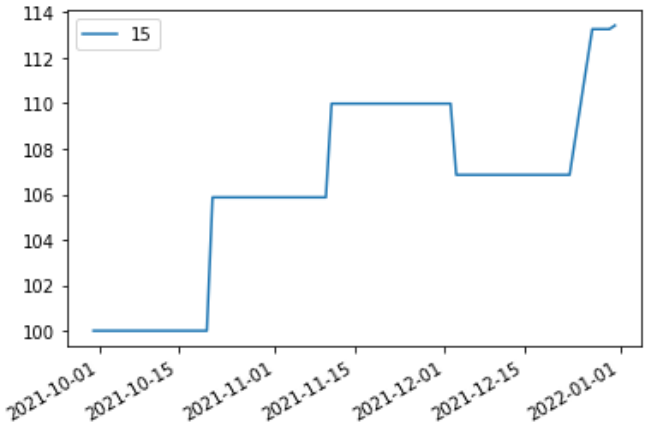


Fig. 11. Balance over time when lag time is 15

**Lag Time 9 - Profit 13059$**

```
# Lag_time = 9
ax1 = balances.iloc[:, 8:9].rebase().plot()
```

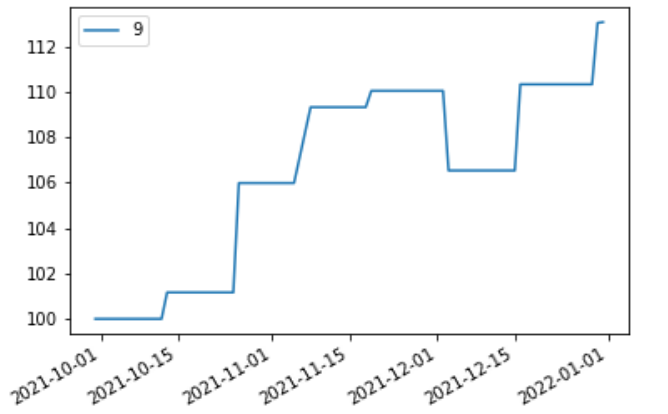


Fig. 12. Balance over time when lag time is 9

**Lag Time 29 - Profit 12878$**

```
# Lag_time = 29
ax1 = balances.iloc[:, 28:29].rebase().plot()
```

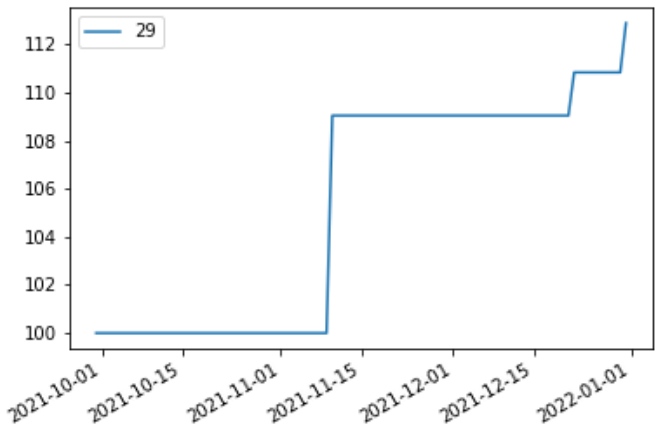


Fig. 13. Balance over time when lag time is 29

*E. Balance History – Common Lag times*

Although lag times being 15, 9, and 29 have resulted in the maximum profits, most of the traders do not use the lag times. In a normal stock market with no unexpected volatiles, the traders mostly prefer traditional behaviors such as daily, weekly, or monthly trading. Figures 14, 15, and 16 illustrate the balance over time for daily, weekly, and monthly lag times.

**Lag Time 1 (Daily Trading) - Profit 12403$**

```
# Lag_time = 1
ax1 = balances.iloc[:, 0:1].rebase().plot()
```

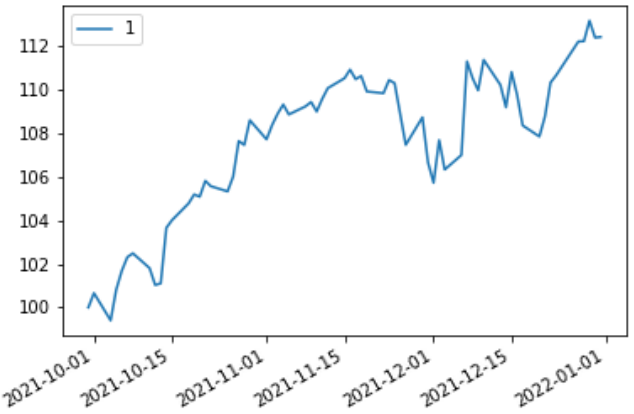


Fig. 14. Balance over time for daily trading

**Lag Time 5 (Weekly Trading) - Profit 12774$**

```
: # Lag_time = 5
ax1 = balances.iloc[:, 4:5].rebase().plot()
```

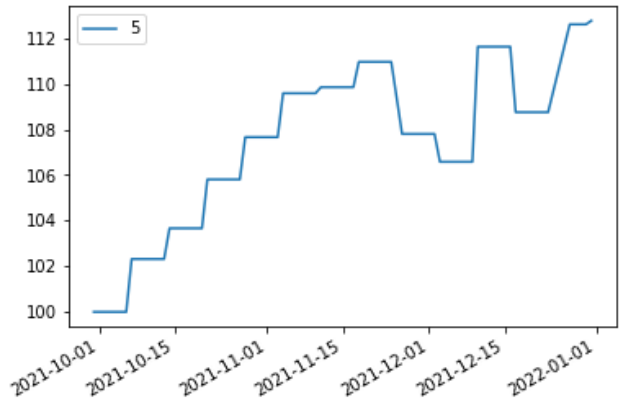


Fig. 15. Balance over time for weekly trading

**Lag Time 20 (Monthly Trading) - Profit 12255$**

```
# Lag_time = 20 # Assuming 20 worksday in a month
ax1 = balances.iloc[:, 20:21].rebase().plot()
```

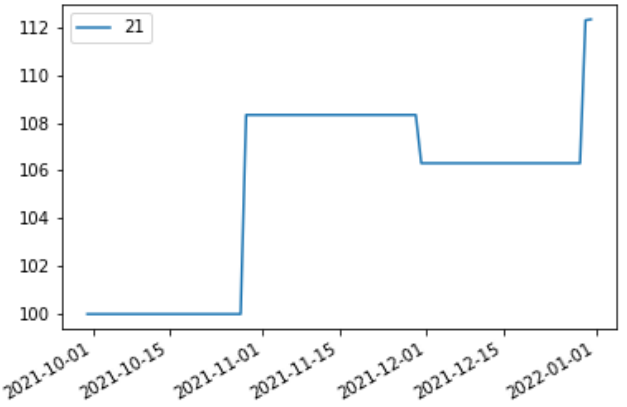


Fig. 16. Balance over time for monthly trading

### F. *Annual Projection*

At the beginning of the simulation (2021-09-30), the expected annual return is calculated by 65.6%. At the end of the simulation, the project profits 13.42% in 3 months period. If the project performs the same profit rate for the future, we can calculate the expected annual rate of the project approximately. The profit rate of 3 months trading is 1.1342018. In 2021-10-01, the portfolio's initial balance is 100.000$. After the first 3 months, the experimentation showed that the balance is increased to 113.420,18$ in 2022-01-01. If we assume that the project will perform the same performance over the next 3 months, the balance will go up to 128.641,37$ in 2022-04-01. For the next 3 months, the balance will be 145.905,28$ in 2022-07-01. Finally, the balance will be increased to 165.486,03$ in 2022-10-01 which is exactly one year later than the start date of the experimentation. According to this annual projection, the project profits 65.5% in a year which is very close to the expected annual return. The below script simulates the annual projection and plots the balance history in each quarter which is shown in Figure 17.

```
profit_rate_3_months = final_balances.max() / initial_balance
print(f'Profit rate of 3 months trading: {profit_rate_3_months}')
expected_annual_balance = initial_balance
expected_annual_balance_history = []
quarter_dates = ['2021-10-01',
                 '2022-01-01',
                 '2022-04-01',
                 '2022-07-01',
                 '2022-10-01']
for quarter in quarter_dates:
    print(f'{quarter} - {expected_annual_balance:.2f}')
    expected_annual_balance_history.append(expected_annual_balance)
    expected_annual_balance *= profit_rate_3_months

pd.DataFrame(expected_annual_balance_history, index=quarter_dates,
columns=['Projection']).rebase().plot()
```

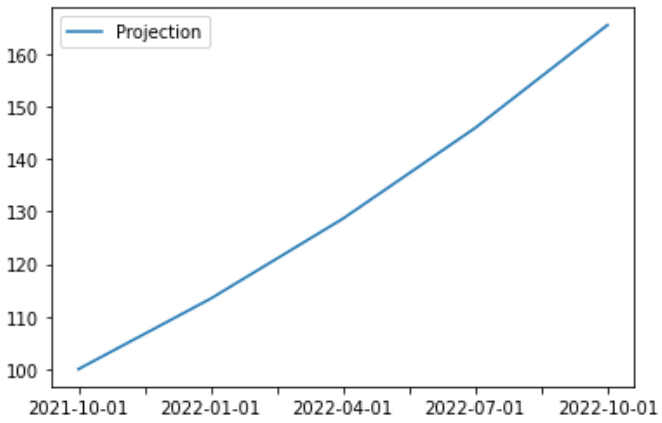


Fig. 17. Quarterly balances for annual projection

## V. Conclusion

In conclusion, this project implements a Modern Portfolio Theory with Time Series Forecasting. History dataset which contains stock prices between 2021-01-01 and 2021-09-30 is used as the training data. Along with the simulation, future prices are predicted with the ARIMA Time Series Forecasting method, and investment is optimized with the Modern Portfolio Theory. Given a lag time, forecasted values are replaced with the real stock prices so that Time Series Forecasting is performed under the latest known stock prices. 30 simulations are performed to evaluate the project performance with lag time-varying 1 to 30. The most profited lag time is 15 and it profits 13420$.

For evaluating the project performance, the portfolio performance can be evaluated with the *calculate_weights* function which verbose the performance when the *verbose* param is true.

The initial *history* contains the stock prices between 2021-01-01 and 2021-09-30. Figure 18 shows the portfolio performance for this dataset.

```
_ = calculate_weights(history, verbose=True)
```

```
Expected annual return: 65.6%
Annual volatility: 15.0%
Sharpe Ratio: 4.24
(0.6560549410998375, 0.15014515884013774, 4.236266730231753)
```

Fig. 18. Portfolio performance of *history* dataset

Additionally, we can get the S&P500 Index via the *ffn* library and index performance can be measured by fetching the *ffn* output into the *calculate_weights* function. Figure 19 shows the S&P500 index values over time and Figure 20 shows the S&P500 index performance.

```
sp500 = get_data('^GSPC',
                 historical_data_start_date,
                 historical_data_end_date,
                 'history_sp')
```

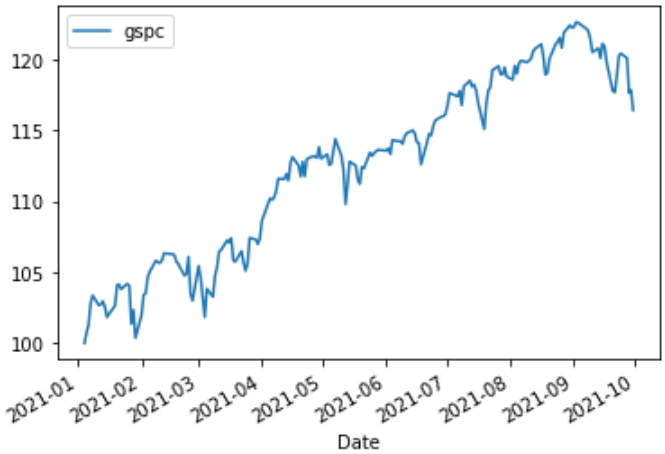


Fig. 19. S&P500 Index values over time.

```
_ = calculate_weights(sp500, verbose=True)
```

```
Expected annual return: 22.7%
Annual volatility: 12.7%
Sharpe Ratio: 1.62
(0.22708763085921957, 0.1274415280389866, 1.6249619260361337)
```

Fig. 20. S&P500 Index performance

## REFERENCES


[1]"GitHub - huseinzol05/Stock-Prediction-Models: Gathers machine learning and deep learning models for Stock forecasting including trading bots and simulations", GitHub, 2022. [Online]. Available: https://github.com/huseinzol05/Stock-Prediction-Models. [Accessed: 31- Jan- 2022].

[2]"GitHub - Poseyy/MarketAnalysis: Portfolio Theory, Options Theory, & Quant Finance", GitHub, 2022. [Online]. Available: https://github.com/Poseyy/MarketAnalysis. [Accessed: 31- Jan- 2022].

[3]P. Thuankhonrak, E. Rattagan and S. Phoomvuthisarn, "Machine Trading by Time Series Models and Portfolio Optimization", 2019 4th International Conference on Information Technology (InCIT), 2019. Available: 10.1109/incit.2019.8912015 [Accessed 31 January 2022].

[4]"Optimizing Portfolios with Modern Portfolio Theory Using Python", Medium, 2022. [Online]. Available: https://towardsdatascience.com/optimizing-portfolios-with-modern-portfolio-theory-using-python-60ce9a597808. [Accessed: 31- Jan- 2022].

[5]"Investment Portfolio Optimisation With Python - Python For Finance", Python For Finance, 2022. [Online]. Available: https://pythonforfinance.net/2017/01/21/investment-portfolio-optimisation-with-python/. [Accessed: 31- Jan- 2022].

[6]J. Brownlee, "What Is Time Series Forecasting?", Machine Learning Mastery, 2022. [Online]. Available: https://machinelearningmastery.com/time-series-forecasting/. [Accessed: 31- Jan- 2022].

[7]"Time-Series Forecasting: Predicting Stock Prices Using An ARIMA Model", Medium, 2022. [Online]. Available: https://towardsdatascience.com/time-series-forecasting-predicting-stock-prices-using-an-arima-model-2e3b3080bd70. [Accessed: 31- Jan- 2022].

[8]"A deep dive on ARIMA models", Mattsosna.com, 2022. [Online]. Available: https://mattsosna.com/ARIMA-deep-dive/. [Accessed: 31- Jan- 2022].

[9]Machinelearningplus.com, 2022. [Online]. Available: https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/. [Accessed: 31- Jan- 2022].