Programming Project

Francesco Fantuzzo, Jacopo Liberati, Oleksandr Pryimak

June 21, 2024

Contents

1 Introduction

The main assumption of the strategy is the Market Efficiency Hypothesis (MEH) which suggests that financial markets reflect all available information, making it impossible for investors to consistently outperform the market without taking on additional risk. In other words, the theory suggests that attempts to beat the market using fundamental or technical analysis useless in the long run (Fama, 1970). Therefore, what we came up with is a simple strategy that requires two things: S&P 500 asset allocation weights and holdings of the seven US-based biggest by assets under management (AUM) Financial Institutions (FIs) (ADV Ratings, n.d.). By subtracting the index weights from the combined FIs portfolio of the same stocks, we assign a position attribute - which we will later use for trading - of "SELL" to stocks whose weight difference is negative, meaning they are overweighted by the FIs, and "BUY" if it is positive. Thus, we bet against the big FIs since we expect them to revert to the index (market) portfolio.

In addition, we add another level of complexity to improve the reasoning behind the strategy by relying on the Fama-French (FF) 5-factor model, which aims to explain stock returns by incorporating factors related to market risk, size, value, profitability, and investment, providing a more comprehensive framework that captures variations in stock performance due to diverse economic influences. Specifically, the model accounts for the empirical observations, such as that companies with high book-to-market ratios (value stocks) tend to outperform the market on a risk-adjusted basis, thereby addressing anomalies that simpler models could not explain. (Fama & French, 2015). Hence, we estimate expected prices from the predicted returns of the model and compare those to the prices of the stocks we weeded out during the initial model. If the actual price is higher, then the stock is overpriced, meaning that we are willing to "SELL", and the opposite works for underpriced stocks.

Therefore, by combining the two pieces our strategy compares the over- and underweightness of the stocks by FIs with their over- or underpricing according to the FF 5-factor model. If the two match we separate them from the rest and pick up to 5 most over- and undervalued assets to form portfolios, short for over- and long for under-. To complete the portfolio, we weigh chosen stocks by normalizing the weight difference compared to the S&P 500, which we obtain from the scraping part of the code. Finally, we set up a code that would buy and sell stocks at the market and take an opposite position at a limit order given the predicted prices of the FF model. Here, we use the assumption of the FF that in the long run, prices will tend to come back to their model-estimated true value. In a year, we will run the last part of the code, which will cancel all the unsatisfied positions and liquidate them at the market price, after which we can repeat the process of portfolio formation.

2 Outsourcing

2.1 Overview

The `Program.cs` file contains a C# program that performs web scraping to extract data about companies listed in the S&P 500 index. The program is structured to perform the following tasks:

- Fetch the list of S&P 500 companies and their weights from `www.slickcharts.com/sp500`.
- Extract CUSIP numbers for each company from `quantumonline.com`.
- Retrieve CIK values for financial institutions from a configuration file.
- Scrape holdings data for each financial institution.
- Compare S&P500 weights with the weights of the SP portfolio assembled from the portfolios of big FIs.
- Generate and save the results, namely, ticker, weight and position according to the model in an Excel file.

2.2 Main Function

The `Main` function is the entry point of the program. It initializes an HTTP client and calls various asynchronous functions to perform web scraping and data extraction.

```
private static async Task Main(string[] args)
{
    var httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("User-Agent","USI pryimousi.ch");

    // Get list of companies from S&P500
    var snpList = await getSnPDataAsync(httpClient);

    var excelFilesPath = Path.Combine(Directory.GetCurrentDirectory(),"ExcelFiles");

    #region CUSIP_Extraction

    var cusipData = new Dictionary<string,string>();

    // Extract CUSIP for each S&P500 company from quantumonline.com
    Console.WriteLine("CUSIP extraction started");
```

```
foreach(var ticker in snpList)
{
    var url = ConfigurationManager.AppSettings["quantumonlineUrl"].Replace("{tick

    // HTTP request
    var htmlString = await httpClient.GetStringAsync(url);

    // Regex expression to find CUSIP in HTTP response
    string pattern = @"CUSIP:\s*([A-Z0-9]+)";
    Match match = Regex.Match(htmlString, pattern);
    if(match.Success)
    {
        string cusip = match.Groups[1].Value;
        cusipData.Add(ticker.Symbol, cusip);
    }
    else
    {
        Console.WriteLine($"Couldn't find CUSIP for: {ticker.Symbol}");
    }
}

Console.WriteLine("CUSIP extraction finished");

# endregion

// Get CIK value for each financial institution from configuration file
var companiesConfig = ConfigurationManager.GetSection("FinInst_CIK") as NameValue

var companiesHoldings = new List<List<StockRecord>>();

// Web scraping for each financial institution
# region Get_FinancialInst_Holdings

foreach(string key in companiesConfig.AllKeys)
{
    var cik = companiesConfig[key];
    // Further code for scraping financial institution holdings
}
```

```
    # endregion

    // Further processing and saving to Excel
}
```

2.3 CUSIP Extraction

This part of the code iterates over each company in the S&P 500 list, constructs the URL for `quantumonline.com`, performs an HTTP request, and uses a regular expression to extract the CUSIP number through Regex function. The regex looks for "CUSIP:", then skips over any spaces and finally it captures the sequence of letters and numbers that comes next (this is the CUSIP number).

```
#region CUSIP_Extraction

var cusipData = new Dictionary<string,string>();

Console.WriteLine("CUSIP extraction started");
foreach(var ticker in snpList)
{
    var url =
    ↪ ConfigurationManager.AppSettings["quantumonlineUrl"].Replace("{ticker_placeho
      ,ticker.Symbol);

    var htmlString = await httpClient.GetStringAsync(url);

    string pattern = @"CUSIP:\s*([A-Z0-9]+)";
    Match match = Regex.Match(htmlString, pattern);
    if(match.Success)
    {
        string cusip = match.Groups[1].Value;
        cusipData.Add(ticker.Symbol, cusip);
    }
    else
    {
        Console.WriteLine($"Couldn't find CUSIP for: {ticker.Symbol}");
    }
}

Console.WriteLine("CUSIP extraction finished");
```

```
#endregion
```

2.4 Configuration and Financial Institution Holdings

The program retrieves CIK values from a configuration file and performs web scraping to get the holdings data for each financial institution. This data is then processed and saved into Excel files.

```
var companiesConfig = ConfigurationManager.GetSection("FinInst_CIK") as NameValueColl

var companiesHoldings = new List<List<StockRecord>>();

// Web scraping for each financial institution
#region Get_FinancialInst_Holdings

foreach(string key in companiesConfig.AllKeys)
{
    var cik = companiesConfig[key];
    // Further code for scraping financial institution holdings
}

#endregion
```

2.5 Compare weights

The code first compiles the list of holdings from various financial institutions and filters to include only those stocks that appear in at least 6 reports. It then calculates the percentage weight for each stock in the newly formed portfolio of holdings and compares it to the corresponding weight in the SP 500. Based on the weight difference, it generates a decision to "BUY" or "SELL" and exports the results to an Excel file.

```
#region Compare_weights_w/S&P500

        Console.WriteLine("Weights calculation started.");

        //Flatten the list of lists
        var allHoldings = companiesHoldings.SelectMany(company =>
        ↪  company);
```

```
//Get only stocks that exist in more than 5 financial
↪  institutions reports
var filteredCUSIPs = allHoldings
    .GroupBy(holding => holding.CUSIP)
    .Where(group => group.Count() > 5)
    .Select(group => group.Key);

// Group by CUSIP and calculate the average value for each
↪  stock
var avgPercentByCUSIP = allHoldings
    .Where(holding => filteredCUSIPs.Contains(holding.CUSIP))
    .GroupBy(holding => holding.CUSIP)
    .Select(group => new
    {
        CUSIP = group.Key,
        AvgValue = group.Average(holding => holding.Percentage)
    });

var resultList = new List<ResultRow>();

//Calculate weight difference for each stock
foreach(var stock in avgPercentByCUSIP)
{
    var ticker = cusipData.FirstOrDefault(x => x.Value ==
    ↪  stock.CUSIP).Key;

    var snpRecord = snpList.FirstOrDefault(x => x.Symbol ==
    ↪  ticker);

    var weightDiff = snpRecord.Percentage - stock.AvgValue;

    //Create list of objects to be populated into a result
    ↪  table
    resultList.Add(new ResultRow
    {
        Ticker = ticker,
        Decision = weightDiff >= 0 ? "BUY" : "SELL",
        WeightDiff = weightDiff,
```

```
        });
    }

    Console.WriteLine("Weights calculation finished.");
    #endregion

    //Export results to an excel table
    ExportToExcel(resultList,excelFilesPath);

    Console.ReadLine();
}
```

2.6 Export to Excel

The code creates a worksheet and adds its headers "Ticker", "Decision", and "WeightDiff" to the first row. The method then iterates through the provided list of ResultRow objects, populating the worksheet with the ticker symbol, decision (BUY/SELL), and weight difference for each stock. For each entry, it sets the background colour of the "Decision" cell to green for "BUY" and red for "SELL". The Excel package is then saved to a file named "Results.xlsx" in the specified folder path.

```
//Create an Excel package
using(var package = new ExcelPackage())
{
    //Add a worksheet to the Excel package
    ExcelWorksheet worksheet =
    ↪ package.Workbook.Worksheets.Add("Results");

    //Add headers
    worksheet.Cells[1,1].Value = "Ticker";
    worksheet.Cells[1,2].Value = "Decision";
    worksheet.Cells[1,3].Value = "WeightDiff";

    //Populate data from the list
    int row = 2;
    foreach(var result in list)
    {
        worksheet.Cells[row,1].Value = result.Ticker;
        worksheet.Cells[row,2].Value = result.Decision;
        worksheet.Cells[row,3].Value = result.WeightDiff;
```

```
            if(result.Decision == "BUY")
            {
                worksheet.Cells[row,2].Style.Fill.PatternType =
                ↪ OfficeOpenXml.Style.ExcelFillStyle.Solid;

                ↪ worksheet.Cells[row,2].Style.Fill.BackgroundColor.SetColor(Color.
            } else
            {
                worksheet.Cells[row,2].Style.Fill.PatternType =
                ↪ OfficeOpenXml.Style.ExcelFillStyle.Solid;

                ↪ worksheet.Cells[row,2].Style.Fill.BackgroundColor.SetColor(Color.
            }
            row++;
        }

        // Save the Excel package to a file
        if(!Directory.Exists("folderPath"))
            Directory.CreateDirectory(folderPath);

        FileInfo excelFile = new
        ↪ FileInfo(Path.Combine(folderPath,"Results.xlsx"));
        package.SaveAs(excelFile);
    }
    Console.WriteLine("Excel file created successfully.");
}
```

3 Fama French Model

3.1 Overview

The file `Final.ipynb` contains a jupyter notebook with a code written in Python that will use the file `Results.xlsx` from the previous part to compute the prediction of fair prices we will use to set the limit order prices in the algotrading code.

3.2 Fetching the Data

The first step is to scrape the Fama-French 5 factors data using URL and fetch the content using the 'requests' library.

```
# URL for FF5 factors from Fama-French website
url = 'https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/F-
    F_Research_Data_5_Factors_2x3_CSV.zip'

# Fetch data from the URL
response = requests.get(url)

# Extract the content of the zip file
zip_content = response.content
```

Next, the code extracts the content of the zip file and opens the archive to inspect the files within.

```
# Open the zip archive
with zipfile.ZipFile(BytesIO(zip_content)) as myzip:
    # List all files in the archive for inspection
    file_list = myzip.namelist()

    # Find the correct file name (adjust based on what's listed)
    for file in file_list:
        if '5_Factors' in file: # Adjust the condition to match the actual
            file name or pattern
            correct_file_name = file
            break
```

After identifying the file within the archive, the code reads the data into a pandas DataFrame.

```
    # Read the correct file into a pandas DataFrame
```

```
    with myzip.open(correct_file_name) as myfile:
        ff5_data = pd.read_csv(myfile, skiprows=3)
```

The 'Unnamed: 0' column, which contains the dates, is used to clean the file by leaving only past 20 years of observation. Also, we convert the table to a datetime format.

```
# Function to clean and convert date
def clean_and_convert_date(x):
    try:
        return pd.to_datetime(str(x)[:6], format='%Y%m') # Convert x to
            string and then slice first 6 characters
    except ValueError:
        return pd.NaT # Return NaT for invalid dates

# Clean and convert the 'Unnamed: 0' column
ff5_data['Unnamed: 0'] = ff5_data['Unnamed: 0'].apply(
    clean_and_convert_date)

# Filter out rows with NaT (invalid dates)
ff5_data = ff5_data.dropna()
```

```
# Filter data for the last 20 years (from 2024 back to 2004)
current_year = pd.Timestamp.now().year
start_year = current_year - 20
end_year = pd.Timestamp.now().year

fixed_ff5_data = ff5_data[(ff5_data['Unnamed: 0'].dt.year >= start_year)
    & (ff5_data['Unnamed: 0'].dt.year <= end_year)]
```

Finally, the code resets the index of the DataFrame, renames the 'Unnamed: 0' column to 'Date', and displays the first few rows of the processed DataFrame.

```
# Reset index starting from 0 and rename 'Unnamed: 0' to 'Date'
fixed_ff5_data.reset_index(drop=True, inplace=True)
fixed_ff5_data.rename(columns={'Unnamed: 0': 'Date'}, inplace=True)

# Display the first few rows of the fixed dataframe with updated index
    and column name
fixed_ff5_data
```

Next, import the tickers S&P500 from the file `Results.xlsx` and create a list containing all the tickers was created.

```
#IMPORTING Tickers S&P500

df_tickers = pd.read_excel('Scraping_Updated\Scraping_Updated\ExcelFiles\
    Results.xlsx')
#df_tickers.head()

list_tickers = df_tickers['Ticker'].tolist()
print(list_tickers)
```

Based on this list of tickers, monthly data for each stock for the period between 20 years ago and today was downloaded from Yahoo Finance. We considered only "Adj Close" and based on this, the monthly returns for each stock in the considered period were calculated.

```
# Get today's date
end_date = datetime.today()

# Calculate the date 20 years ago from today
try:
    start_date = end_date.replace(year=end_date.year - 20)
except ValueError:
    # Handle the case for leap years (if today is Feb 29 and 20 years ago
        is not a leap year)
    start_date = end_date.replace(year=end_date.year - 20, day=28)

# Format dates as strings in the required format
end_date_str = end_date.strftime('%Y-%m-%d')
start_date_str = start_date.strftime('%Y-%m-%d')

# Download stock prices from 20 years ago until today
data = yf.download(tickers, start=start_date_str, end=end_date_str,
    interval='1mo')
```

```
# Assuming monthly_returns and fixed_ff5_data are your DataFrames
# Convert 'Date' columns to datetime format
monthly_returns['Date'] = pd.to_datetime(monthly_returns['Date'])
fixed_ff5_data['Date'] = pd.to_datetime(fixed_ff5_data['Date'])

# Identify the common dates (values) in 'Date' column
```

```
common_dates = set(monthly_returns['Date']).intersection(set(
    fixed_ff5_data['Date']))

# Filter both DataFrames based on common dates
monthly_returns = monthly_returns[monthly_returns['Date'].isin(
    common_dates)]
fixed_ff5_data = fixed_ff5_data[fixed_ff5_data['Date'].isin(common_dates)
    ]

# Set 'Date' columns as index (if needed)
monthly_returns.set_index('Date', inplace=True)
fixed_ff5_data.set_index('Date', inplace=True)

# Concatenate the DataFrames
df_conc = pd.concat([monthly_returns, fixed_ff5_data], axis=1)

# Replace NaN values with 0
df_conc = df_conc.fillna(0)

# Display the resulting DataFrame
df_conc
```

```
# Take only Adj Close
close_prices = data['Adj Close']

# Calculate monthly returns for each ticker
monthly_returns = close_prices.pct_change()
```

Then we ensure that the date columns in both data tables (`monthly_returns` and `fixed_ff5_data`) are properly formatted as dates. Once the dates are standardized, it identifies the dates that are common to both tables. This intersection of dates is used to filter each table, retaining only the rows that share the same dates.

After filtering, the date columns are set as the index for both tables.

Next, the two tables are combined side-by-side, creating a new table where each row corresponds to a common date and includes data from both original tables. Any missing values in this combined table are then replaced with zeroes to ensure there are no gaps in the data.

```

# Assuming monthly_returns and fixed_ff5_data are your DataFrames
```

```
# Convert 'Date' columns to datetime format
monthly_returns['Date'] = pd.to_datetime(monthly_returns['Date'])
fixed_ff5_data['Date'] = pd.to_datetime(fixed_ff5_data['Date'])

# Identify the common dates (values) in 'Date' column
common_dates = set(monthly_returns['Date']).intersection(set(
    fixed_ff5_data['Date']))

# Filter both DataFrames based on common dates
monthly_returns = monthly_returns[monthly_returns['Date'].isin(
    common_dates)]
fixed_ff5_data = fixed_ff5_data[fixed_ff5_data['Date'].isin(common_dates)
    ]

# Set 'Date' columns as index (if needed)
monthly_returns.set_index('Date', inplace=True)
fixed_ff5_data.set_index('Date', inplace=True)

# Concatenate the DataFrames
df_conc = pd.concat([monthly_returns, fixed_ff5_data], axis=1)

# Replace NaN values with 0
df_conc = df_conc.fillna(0)

# Display the resulting DataFrame
df_conc
```

The obtained data was placed within the "monthly_returns" data frame, using the same row indices used for the Fama French factors to facilitate the subsequent concatenation of the data frames.

```
monthly_returns.reset_index(inplace=True)
monthly_returns.rename(columns={'index': 'Date'}, inplace=True)
monthly_returns
```

Subsequently, the data frame related to the Fama French factors and the one related to the monthly returns of each stock were concatenated. Consequently, the missing values "NaN" were replaced with the value "0".

```
# Concatenate the DataFrames
df_conc = pd.concat([monthly_returns, fixed_ff5_data], axis=1)
```

```
# Replace NaN values with 0
df_conc = df_conc.fillna(0)

# Display the resulting DataFrame
df_conc
```

The columns to be retained for the Excess Returns data frame were then selected from the previously defined `df_conc`, specifically the column related to the Date and the columns related to the monthly returns of the stocks. The column related to the risk-free rate (the last column of `df_conc`) was then defined and subtracted from the monthly returns of the stocks to obtain the monthly excess returns for each considered stock. Finally, the resulting data frame, containing the Date column and the monthly excess returns for each stock, was printed.

```
# Initialize df_excess_returns with the same structure as monthly_returns
df_excess_returns = monthly_returns.copy()

# Get the 'RF' column (last column)
rf_column = df_conc.iloc[:, -1]

# Convert rf_column to numeric
rf_column = pd.to_numeric(rf_column, errors='coerce')



# Calculate the excess returns
for column in df_excess_returns.columns: # Excluding Date and the last 6
    columns (Factors)
    mask = ~df_excess_returns[column].isna()
    df_excess_returns.loc[mask, column] = df_excess_returns.loc[mask,
        column] - rf_column[mask]

df_excess_returns = df_excess_returns.fillna(0)
df_excess_returns
```

Consequently, the code performs a series of linear regressions to estimate alpha and the beta coefficients for each ticker in the list. The previously calculated excess returns are used as the dependent variable while the independent variables are the 5 Fama French factors identifiable within the previously defined `df_conc`.

```
    # List of the tickers
```

```
tickers = list_tickers

# Dictionary in which to save the regression results:
regression_results_ff = {}

# Convert every column in the DataFrame to float
df_conc = df_conc.astype(float)

# Iterate over all tickers
for tickers_of_interest in tickers:
    # Dependent Variable: tickers excess return
    y_ff = df_excess_returns[tickers_of_interest]

    # Independent Variable: market excess return, SMB, HML with first
       column of ones
    X_ff = sm.add_constant(df_conc[['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA'
       ]])

    # Solve the equation using Matrix Algebra
    beta_ff = np.linalg.inv(X_ff.T @ X_ff) @ X_ff.T @ y_ff

    # Extract Alfas, Betas for Mkt-RF, Betas for SMB, Betas for HML,
       Betas for RMW, Betas for CMA
    alfa_ff = beta_ff[0]
    beta_mkt_rf_ff = beta_ff[1]
    beta_smb_ff = beta_ff[2]
    beta_hml_ff = beta_ff[3]
    beta_rmw_ff = beta_ff[4]
    beta_cma_ff = beta_ff[5]


    # Save the results in the Dictionary
    regression_results_ff[tickers_of_interest] = {
        'Alfa': alfa_ff,
        'Beta_Mkt-RF': beta_mkt_rf_ff,
        'Beta_SMB': beta_smb_ff,
        'Beta_HML': beta_hml_ff,
        'Beta_RMW': beta_rmw_ff,
        'Beta_CMA': beta_cma_ff
```

```
    }

# Print the results (alphas and betas) for each ticker
for tickers_of_interest, results in regression_results_ff.items():
    print(f'Results for portfolio {tickers_of_interest}:')
    print(f'Alfa (): {results["Alfa"]}')
    print(f'Beta_Mkt-RF (): {results["Beta_Mkt-RF"]}')
    print(f'Beta_SMB (): {results["Beta_SMB"]}')
    print(f'Beta_HML (): {results["Beta_HML"]}')
    print(f'Beta_RMW (): {results["Beta_RMW"]}')
    print(f'Beta_CMA (): {results["Beta_CMA"]}')
    print()
```

3.3 Price Today

Similarly to what was done in the initial part of the code, data related to each ticker present in the list was then downloaded from Yahoo Finance for the pre-last month of the FF dataset to match the prediction of the model with the actual timing. Only the Adj Close prices were retained.

```
# Assuming df_predicted_data_ff is your DataFrame
date_list = df_predicted_data_ff.index.tolist()
print(date_list)

# List of the tickers
tickers = list_tickers

end_date = date_list[-1] # Second-to-last date
start_date = date_list[-2] # Date before the second-to-last date

# Download data for each stock from Yahoo Finance
data = yf.download(tickers, start=start_date, end=end_date, interval='1mo
    ')

# Take only today's Adj Close
close_prices = data['Adj Close']

print(close_prices)
```

After replacing NaN values with 0, the transpose was performed to obtain a column

vector containing all the previously downloaded stock prices. This way, a single data frame was then created to visualize the price last month and the return for this month. The columns were then renamed for clarity.

```
# Replace NaN values with 0
close_pricess = close_prices.fillna(0)
# Keep the last predicted return for each stock
last_return = df_predicted_data_ff.iloc[-1]
# Make transpose
close_pricesss = close_pricess.transpose()
# Data Frame with close prices from month ago compared to the last
   predicted return for each stock and the last return
merged_df = pd.concat([close_pricesss, last_return], axis=1)
# Rename columns
new_names_columns = ['Month_Ago_Price','Last_ff5_returns']
merged_df.columns = new_names_columns
print(merged_df)
```

3.4 Finding Today Price as predicted by Fama French 5 factors

```
   Example of the logic
   RET(2024/04) = P(2024/04) / P(2024/03) - 1
   P(2024/04) = P(2024/03) * (1 + RET(2024/04))
```

The objective was then to calculate the predicted price from the Fama French Factor for today. Consequently, the formula mentioned above was followed, in which the predicted monthly returns from the Fama French 5 Factor Model were multiplied by prices on last month downloaded from Yahoo Finance. By then adding the prices of last month, the predicted prices for today were obtained.

```
# Calculate the predicted price for each stock on 28/03/2024
merged_df['Next_Month'] = merged_df['Month_Ago_Price'] * (1 + merged_df['
   Last_ff5_returns'])

# Keep only the next month column, containing the predicted price for
   each stock
limit_prices_2024 = merged_df['Next_Month']
print(limit_prices_2024)
```

```
# List of the tickers
tickers = list_tickers
```

```
latest_date = max(date_list) # Find the latest date in date_list
end_date = latest_date + relativedelta(months=1) # Add one month to the
    latest date
start_date = date_list[-1] # Date before the second-to-last date

# Download data for each stock from Yahoo Finance
data = yf.download(tickers, start=start_date, end=end_date, interval='1mo
    ')

# Take only Adj Close 28/03/2024
close_prices1 = data['Adj Close']

print(close_prices1)
```

Today's prices are then downloaded from Yahoo Finance, following the same approach used previously, to compare them with the prices predicted by the FF model for the last month of the FF dataset. Based on this analysis, we assign the action "BUY" if the predicted price from the model is higher than the real price. Conversely, we assigned the action "SELL" if the predicted price from the model is lower than the real price.

```
# Rename the columns (real price and fama french predicted price
    28/03/2024)
merged_df1.columns = ['real_price', 'predicted_price']

# Determine "buy" or "sell" on the basis of real price and predicted
    price for today
def determine_action(row):
    if row['predicted_price'] < row['real_price']:
        return "sell"
    else:
        return "buy"

# Apply the fuction to each row of the DataFrame (each stock)
merged_df1['action'] = merged_df1.apply(determine_action, axis=1)

# Print the DataFrame with the action for each stock
print(merged_df1[['real_price', 'predicted_price', 'action']])
```

Finally, a data frame is created with the action to take for each stock (BUY / SELL) and the corresponding limit price to use for limit orders, which is the price predicted for

the last month of the FF dataset.

3.5 Final results

At this point, we want to find up to 5 most undervalued companies and long them and up to 5 most overvalued and short them.

The code begins by importing scraping results from the Excel file named `Results.xlsx` and setting the `Ticker` column as the index of the DataFrame.

```
merged_df = df_results.join(final_dataframe, how='inner')
```

The `df_results` DataFrame is joined with another DataFrame named `final_dataframe`, keeping only the common indexes.

```
filtered_merged_df = merged_df[
    ((merged_df['Decision'] == 'SELL') & (merged_df['action'] == 'sell'))
        |
    ((merged_df['Decision'] == 'BUY') & (merged_df['action'] == 'buy'))
]
```

Rows where the `Decision` and `action` columns match (i.e., both are 'SELL' or both are 'BUY') are filtered and stored in `filtered_merged_df`.

```
ticker = 'AAPL'
row_data = merged_df.loc[ticker]
```

Data for a specific ticker (in this case, 'AAPL') is extracted from `merged_df`.

```
top_5_high = filtered_merged_df.nlargest(5, 'WeightDiff')
bottom_5_low = filtered_merged_df.nsmallest(5, 'WeightDiff')
```

The first 5 rows with the highest and lowest values in the `WeightDiff` column are selected.

```
sum_top_5_high = top_5_high['WeightDiff'].sum()
sum_bottom_5_low = bottom_5_low['WeightDiff'].sum()
top_5_high['WeightDiff_normalized'] = top_5_high['WeightDiff'] /
   sum_top_5_high
bottom_5_low['WeightDiff_normalized'] = bottom_5_low['WeightDiff'] /
   sum_bottom_5_low
```

The weights for the top and bottom 5 rows are normalized by dividing by the sum of their respective `WeightDiff` values.

```
top_5_high['Decision'] = top_5_high['Decision']
top_5_high['price2025/03/28'] = top_5_high['price2025/03/28']
bottom_5_low['Decision'] = bottom_5_low['Decision']
```

```
bottom_5_low['price2025/03/28'] = bottom_5_low['price2025/03/28']
```

The `Decision` and `price2025/03/28` columns are added to both the top and bottom 5 rows.

```
longtable = top_5_high[['WeightDiff', 'WeightDiff_normalized', 'Decision'
    , 'price2025/03/28']]
shorttable = bottom_5_low[['WeightDiff', 'WeightDiff_normalized', '
    Decision', 'price2025/03/28']]
merged_table = pd.concat([longtable, shorttable])
merged_table.to_csv('final_dataframe.csv')
```

The final data frame, containing the top and bottom 5 rows with the normalized weights and additional columns, is concatenated.

4 Algotrading

4.1 Overview

The algotrading script is designed to automate the trading process by integrating with an Interactive Brokers (IB) paper trading account. It consists of two parts:

- We buy the portfolio. It is located in the same code.
- We cancel the remaining positions at the end of the year. It is done with a separate code called "Cancellation of the Portfolio"

4.2 Formation of the portfolio

The script starts reading the CSV file `final_dataframe.csv` and renaming the columns for clarity. The `fair_prices` column values are rounded to two decimal places.

```
data = pd.read_csv("final_dataframe.csv")
data = data.rename(columns={data.columns[0]: "ticker",data.columns[4]: "
    fair_prices"})
data['fair_prices'] = data['fair_prices'].round(2)
```

The script then establishes a connection to the IB paper trading account and retrieves the account summary. The total cash value (TCV) of the portfolio at the beginning of the trading period is extracted and converted to a numeric format. Here it is important to manually copy and save the initial value of the portfolio to later use it in the Cancellation code when estimating yearly return.

```
ib = IB()
ib.connect('127.0.0.1', 7497, clientId=0)
myAccount = ib.accountSummary()
myportfoliovalue_begin_tcv = pd.to_numeric(myAccount[29].value)
```

The quantity of each stock to be traded is calculated based on the normalized weights and the fair prices. The quantities are then rounded and stored in the data frame. Figure 1 shows the initial page of the IB.

```
data['quantity'] = round(abs((myportfoliovalue_begin_tcv / 2) * data['
    WeightDiff_normalized'] / data['fair_prices']))
```

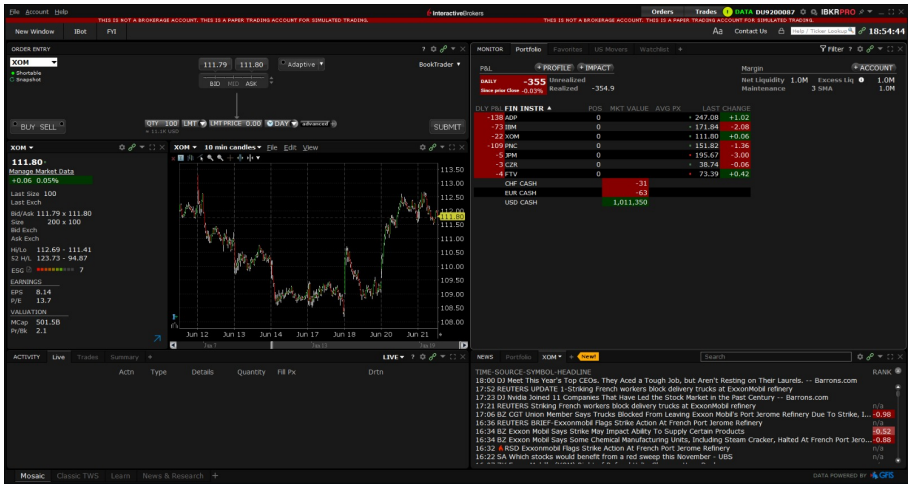

Figure 1: IB account before trades

The script proceeds to create and place market orders for each stock in the data frame. For each stock, a contract object is created, and qualified, and a market order is placed based on the `Decision` column (either 'BUY' or 'SELL') and the calculated quantity.

```
for i in range(len(data)):
    contract = Stock(symbol=data['ticker'].iloc[i], exchange='SMART',
        currency='USD')
    ib.qualifyContracts(contract)
    marketOrder = MarketOrder(action=data['Decision'].iloc[i],
        totalQuantity=data['quantity'].iloc[i])
    trade = ib.placeOrder(contract, marketOrder)
    print(f"Placed order for {data['ticker'].iloc[i]}: {data['Decision'].
        iloc[i]} {data['quantity'].iloc[i]} shares.")
```

After placing the market orders, the script sets up limit orders to automatically execute trades at predefined prices. These limit orders are the opposite actions of the initial market orders and are set to the fair prices stored in the data frame. The script ensures that these orders are correctly qualified and placed. Figure 2 shows the IB page after the portfolio was purchased by the algorithm.

```
for i in range(len(data)):
    contract = Stock(symbol=data.loc[i, 'ticker'], exchange='SMART',
        currency='USD')
    opposite_action = 'SELL' if data.loc[i, 'Decision'] == 'BUY' else '
        BUY'
    limit_order = LimitOrder(action=opposite_action, totalQuantity=data.
        loc[i, 'quantity'], lmtPrice=data.loc[i, 'fair_prices'])
    qualified_contract = ib.qualifyContracts(contract)[0]
```

```
    trade = ib.placeOrder(qualified_contract, limit_order)
    print(f"Setting up limit order for {data.loc[i, 'ticker']}: {
        opposite_action} {data.loc[i, 'quantity']} shares at limit price $
        {data.loc[i, 'fair_prices']}")
```

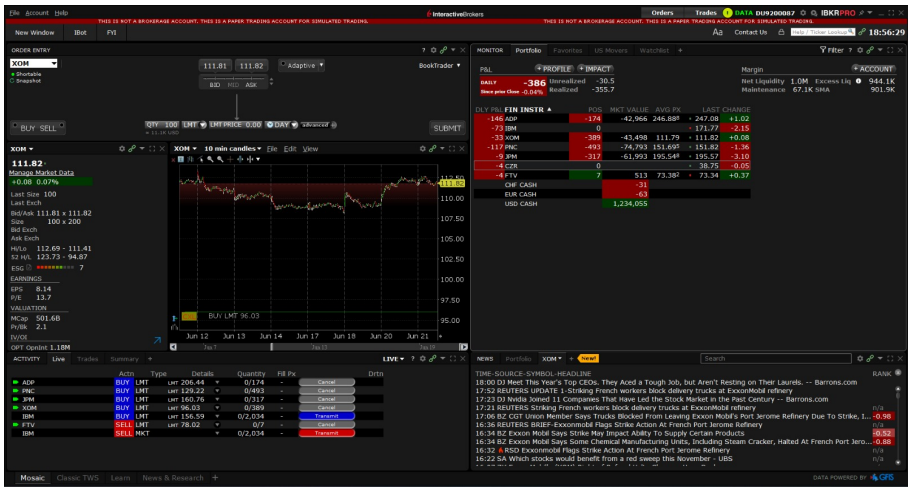

Figure 2: IB account after trades

4.3 Cancellation of the Portfolio

The script cancels any open limit orders and fetches the portfolio holdings. Functions are defined to fetch the current portfolio holdings and to execute sell and buy orders to close all positions at the end of the trading period. Figure 3 shows how the IB page looks after selling the portfolio.

```
    # Fetch all open orders
open_orders = ib.openOrders()

# Cancel all open orders
for order in open_orders:
    ib.cancelOrder(order)

# Wait briefly to ensure cancellation completes
ib.sleep(1)
```

```
# Function to fetch portfolio holdings
def fetch_portfolio():
    positions = ib.reqPositions()
    portfolio = {p.contract.symbol: p.position for p in positions}
    return portfolio
```

```
# Function to execute sell and buy orders to close positions
def close_positions(portfolio):
    for symbol, quantity in portfolio.items():
        if quantity > 0:
            # Closing long position (sell)
            contract = Contract()
            contract.symbol = symbol
            contract.exchange = 'SMART'
            contract.currency = 'USD'
            contract.secType = 'STK'

            sell_order = MarketOrder(action='SELL', totalQuantity=quantity
                )
            ib.placeOrder(contract, sell_order)
            print(f"Placed market sell order for {symbol}: {quantity}
                shares")

        elif quantity < 0:
            # Closing short position (buy)
            contract = Contract()
            contract.symbol = symbol
            contract.exchange = 'SMART'
            contract.currency = 'USD'
            contract.secType = 'STK'

            buy_order = MarketOrder(action='BUY', totalQuantity=-quantity)
                # Ensure quantity is positive for buy order
            ib.placeOrder(contract, buy_order)
            print(f"Placed market buy order to cover short for {symbol}:
                {-quantity} shares")

# Fetch portfolio holdings
portfolio = fetch_portfolio()

# Execute sell and buy orders to close positions
close_positions(portfolio)
```

The script finally calculates the portfolio value at the end of the year and computes

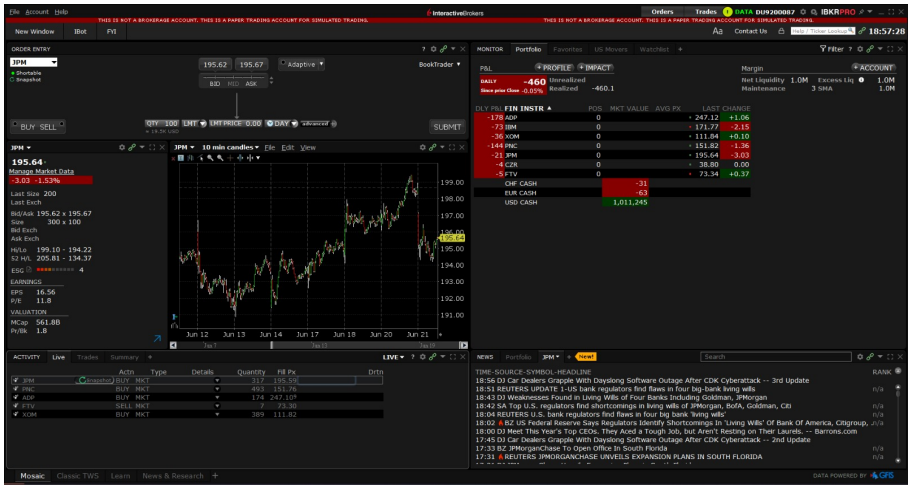

Figure 3: IB account after selling everything

the yearly return by comparing the end value to the initial TCV. The yearly return is printed as the output. Here the beginning value should be inserted from the Formation code.

```
# Portfolio Value at the end of the year
myportfoliovalue_end_af = [myAccount[29].value]

# Convert to numeric (float)
myportfoliovalue_end_af = pd.to_numeric(myportfoliovalue_end_af)
myportfoliovalue_begin_tcv = 1005763.51

# Calculate yearly return
myyearlyreturn = myportfoliovalue_end_af / myportfoliovalue_begin_tcv - 1
print(myyearlyreturn)
```

5 Conclusion

Unfortunately, we were unable to backtest our model due to the inability to obtain the yearly SP 500 asset allocation. By having it we could have used the 13F reports up to 2013 (due to stylistic differences in the application of the report that occurred in the year making it impossible to scrape the data before), run the model, estimate portfolios, and estimate their yearly returns given the prices during the year and limit prices according to FF 5 model. Since we do not observe the reality, we want to make an educated guess about the results of the model, namely, why we believe that it is supposed to perform poorly.

First of all, there are market actors called high-frequency traders (HFTs) who use advanced algorithms to quickly detect and exploit market imbalances. They process information within seconds and pay high fees for access to order flows, which are too costly to individual investors (Mandes, 2016). This rapid trading environment supports the Market Efficiency Hypothesis (MEH), which suggests that market prices adjust swiftly to reflect all available information (Fama, 1970). This means that our approach forces us to trade on information that is already incorporated into market prices, which puts our entire strategy on par with random stock picking.

Another reason might be that institutions bet on the movements of stocks due to overall analyses of the market, including fundamentals such as betas, balance sheet analyses such as Dupont decomposition and macroeconomic data such as interest rate policy prediction – everything that we as students cover in our classes in finance. In addition, the FIs are frequently large shareholders of the companies they hold, meaning that they take an active part in the decision-making and guidance of the big corporations, which provides them with a deeper understanding of the companies' performance, or in other words, their actual value (Zeckhauser Pound, 1990).

Finally, from 13F reports, we observed that the institutional holdings include not only stocks but also bonds, derivatives, real estate and loans. Hence, the complexity of their portfolios brings benefits to their investors in the form of additional diversification, coming from a greater variety of assets, reduced downside effect on returns due to hedging, and satisfaction of the specific clients' requests such as Environment Sustainability and Governance (ESG) or high risk – high reward strategies.

References

[1] ADV Ratings. (n.d.). Top Asset Management Firms. Retrieved June 21, 2024, from https://www.advratings.com/top-asset-management-firms

[2] Fama, E. F., & French, K. R. (2015). "A Five-Factor Asset Pricing Model." Journal of Financial Economics, 116(1), 1-22. Retrieved from https://www.sciencedirect.com/science/article/abs/pii/S0304405X14002323#preview-section-abstract

[3] Fama, E. F. (1970). Efficient Capital Markets: A Review of Theory and Empirical Work. The Journal of Finance, 25(2), 383–417. https://doi.org/10.2307/2325486

[4] Mandes, A. (2016). Algorithmic and high-frequency trading strategies: A literature review. University of Marburg. Retrieved from https://www.econstor.eu/bitstream/10419/144690/1/860290824.pdf

[5] Zeckhauser, R., & Pound, J. (1990). Are Large Shareholders Effective Monitors? An Investigation of Share Ownership and Corporate Performance. In R. G. Hubbard (Ed.), Asymmetric Information, Corporate Finance, and Investment. University of Chicago Press. Retrieved from https://www.nber.org/system/files/chapters/c11471/c11471.pdf