



University of l'Aquila

Department of Information Engineering and Computer Science
and Mathematics

FINAL PROJECT

***Optimized portfolio using machine
learning models and CSP***

Students

Gea Viozzi
Lorenzo Berretta

Abstract

This documentation presents a project that uses machine learning algorithms and CSP techniques to build an optimized portfolios for five financial assets, including AMD, CSCO, QCOM, SBUX, and TSLA.

Its scope involves the implementation of machine learning models for individual asset predictions, followed by the utilization of CSP for constructing diversified portfolios based on user preferences.

The structure of the report includes chapters on Materials and Methods, Experiments and Results, and Conclusion. The Materials and Methods chapter provides insights into the dataset, machine learning models, and CSP formulation. The Experiments and Results chapter shows the outcomes, and the performance metrics. Finally, the Conclusion chapter summarizes findings and discusses limitations.

Contents

1	Introduction	1
1.1	Scope of the Project	1
1.2	How to use	2
1.3	GitHub Repository	2
2	Materials and Methods	3
2.1	Implementation and Tools	3
2.2	Exploratory Data Analysis (EDA)	4
2.3	Libraries	4
2.4	ml.py	5
2.4.1	Data Collection and Preprocessing	5
2.4.2	Feature Engineering and Feature Selection	6
2.4.3	Machine Learning Models	8
2.4.4	Modelling	9
2.5	Constraint Satisfaction Problem (CSP)	12
2.5.1	Volatility Constraint	13
2.5.2	Solver	14
2.6	Function <code>main</code>	14
2.6.1	<code>load_model</code> function	18
2.6.2	<code>data_visualization</code> function	19
2.6.3	<code>get_features</code> function	21
2.7	<code>app.py</code>	22
3	Experiments and Results	23
3.1	Results as a User	23
3.2	Performance Metrics	25
3.2.1	Metrics in the <code>modelling</code> function	25
3.2.2	Performance visualization	26
3.2.3	Performance values	26
4	Conclusion	30

Chapter 1

Introduction

In this contemporary era, the financial sector has witnessed significant shifts with the emergence of machine learning, particularly in data analysis. Thanks to this technology, studying stock market trends and crafting personalized portfolios has become achievable for everybody.

The project's foundation lies in the aspiration to utilize machine learning algorithms and CSP methodologies to create optimized portfolios for various assets. By using past financial data and predictive modelling, the goal is to improve decision-making in the ever-changing financial markets.

The primary **objectives** of this project include:

- Utilizing machine learning models for predicting future asset prices.
- Integrating CSP techniques to formulate constraints and optimize portfolio allocation.
- Developing a cohesive and adaptable system capable of handling multiple financial instruments.
- Evaluating the performance and effectiveness of the proposed methodology.

1.1 Scope of the Project

This project focuses on a set of financial assets, namely [AMD, CSCO, QCOM, SBUX, TSLA]. The scope encompasses the development and implementation of machine learning models for individual asset prediction and the subsequent utilization of CSP for constructing diversified portfolios, taking into account the user's preferences. It is important to note that our machine

learning model was a didactic, toy model built for educational purposes, thus it does not provide an accurate prediction of future values, instead it is a pure simulation.

1.2 How to use

If is the first time running the program make sure to have all the libraries in the file "requirements.txt" already installed in your environment. Besides, the directory "models" should already have 5 pre-trained models inside, if that is not the case, we would advise to run the file *ml.py* file, before taking any further step, so the next phases will take significantly less time to compile.

- Execute the *app.py* file (make sure that the correct file is selected as the current file).
- Open any browser you want and type "127.0.0.1:5000" in the search bar. In alternative, if running the code through PyCharm or any other IDE that provides an output console, you can click on the link that appears in the console.
- Type the values in the input spaces.(investment limits must multiples of 10 and risk factor must be between 0.35 and 1)
- Wait for the algorithm to finish and redirect you to the output page.

1.3 GitHub Repository

We have created a GitHub repository for this project, thus you can visit this [GitHub repository](#) for more details.

Chapter 2

Materials and Methods

2.1 Implementation and Tools

The entire project was implemented using Python. The project code is organized into modular scripts and directories for better maintainability.

Amongst the modules we can find:

- **data_visualization.ipynb** - A jupyter notebook that we used for the Exploratory Data Analysis. This was a useful way to understand what the data looked like.
- **ml.py** - The script where we are implementing our machine learning component.
- **CSP_generics.py and CSP_solver.py** - Here we have the specifics of how a CSP problem is defined and the algorithms for the arc consistency and the CSP solver. These modules were discussed and given in class, but we added a new function (*solve_all*) that gives us all the solutions to the CSP.
- **csp_predictions.py** - In this module we defined our CSP problem and we incorporated the main function of our project, which uses both the machine learning and CSP component.
- **app.py** - In this module we have implemented the flask app.

And amongst the directories:

- **features** - Here we can find five text files containing the features selected to train the models for the stocks. These are generated during the modelling process.

- **models** - In this directory there are five pickle files where the pre-trained models are stored.
- **performance** - Here we have five text files containing the error metrics of the five different models.
- **static** - In this directory we are saving the graphs that are shown in our app.
- **stocks** - In this directory we saved the csv files of our chosen stocks.
- **templates** - In this directory we have our html templates, used for our web app.

2.2 Exploratory Data Analysis (EDA)

This Jupyter Notebook *data_visualization.ipynb* was used for exploratory data analysis (EDA) during the project's initial stages. Despite not being used in any significant part of our project, it provided us with insights into the structure and characteristics of the dataset.

2.3 Libraries

In this section we are going to briefly list all the libraries that are included in our project.

```
1 # app.py
2 from flask import Flask, render_template, request, redirect,
   url_for
3
4 # ml.py
5 import warnings
6 import pandas as pd
7 import ta
8 from sklearn.neighbors import KNeighborsRegressor
9 from ta.momentum import RSIIndicator
10 from sklearn.model_selection import train_test_split,
   cross_val_score
11 from sklearn.linear_model import LinearRegression
12 from sklearn.metrics import mean_squared_error, r2_score
13 import pickle
14 import os
15 import numpy as np
16 from sklearn.feature_selection import RFE
17 from sklearn.ensemble import RandomForestRegressor
```

```
18 import matplotlib.pyplot as plt
19
20 # csp_predictions.py
21 import warnings
22 from datetime import timedelta
23 import pandas as pd
24 import numpy as np
25 import statistics
26 import pickle
27 import os
28 import matplotlib.pyplot as plt
29 import plotly.graph_objects as go
30 import random
```

2.4 ml.py

2.4.1 Data Collection and Preprocessing

The datasets used in this project consist of historical financial data for selected assets, in particular we chose the companies:

- Starbucks, Inc.(SBUX)
- Cisco Systems, Inc.(CSCO)
- QUALCOMM Incorporated (QCOM)
- Tesla, Inc. (TSLA)
- Advanced Micro Devices, Inc.(AMD)

The dataset were selected from the website [www.nasdaq.com], we chose the 5-year long datasets, updated up to February 3rd 2024. The data presents relevant features such as opening and closing prices, high and low prices, trading volumes, and dates.

The datasets were preprocessed through the *data_preparation* function using the following steps:

- Conversion of dates to the YYYY-MM-DD format.
- Renaming of columns.
- Removal of non-numeric characters and conversion of relevant columns to float.
- Forward-filling to handle missing data.

2.4.2 Feature Engineering and Feature Selection

The function *feature_eng* aims to create informative features from the raw financial data. The following features were considered:

- **Technical Indicators:** Included indicators such as Exponential Moving Average (calculated on windows of 12 and 26 days), Relative Strength Index (RSI), On-Balance Volume (OBV), and Simple Moving Average (calculated on windows of 14, 30 and 100 days).
- **Statistical Metrics:** Mean Absolute Deviation (MAD), Commodity Channel Index (CCI), and Moving Average Convergence Divergence (MACD).

```

1 def feature_eng(ds_name):
2     """
3     :param ds_name: takes a csv file name as an input
4     :return: dataframe which includes new significant values
5     """
6
7     #building path to dataset
8     ds_name = 'stocks/' + ds_name
9
10    # pre-processing data
11    df = data_preparation(ds_name)
12
13
14    # Relative Strength Index (RSI)
15    df['RSI'] = ta.momentum.rsi(df['Close'])
16    # On Balance Volume (OBV)
17    df['OBV'] = ta.volume.OnBalanceVolumeIndicator(close=df['Close'], volume=df['Volume']).on_balance_volume()
18    # Take Profit
19    df['TP'] = (df['High'] + df['Low'] + df['Close']) / 3
20
21    # Simple Moving Average
22    # 14 days
23    df['sma14'] = ta.trend.sma_indicator(df['Close'], window=14)
24    # 30 days
25    df['sma30'] = ta.trend.sma_indicator(df['Close'], window=30)
26    # 100 days
27    df['sma100'] = ta.trend.sma_indicator(df['Close'], window=100)
28
29    # Mean Absolute Deviation

```

```

30     df['mad'] = df['TP'].rolling(14).apply(lambda x: (pd.
31         Series(x) - pd.Series(x).mean()).abs().mean())
32     # Commodity Channel Index (30 days window)
33     df['CCI'] = (df['TP'] - df['sma30']) / (0.015 * df['mad']
34         ])
35
36     # Moving Average Convergence-Divergence
37     # Calculating Exponential Moving Average for 12 and 26
38     periods
39     df['ema_12'] = df['Close'].ewm(span=12, adjust=False).
40     mean()
41     df['ema_26'] = df['Close'].ewm(span=26, adjust=False).
42     mean()
43     # Calculating Moving Average Convergence-Divergence (MACD
44     )
45     df['macd'] = df['ema_12'] - df['ema_26']
46
47     # drops na values (caused by values calculated over a
48     window of time)
49     df = df.dropna()
50
51     return df

```

The `feature_selection` function is designed to perform feature selection on the training and test sets to improve the efficiency and performance of the model, by identifying and retaining the most relevant features for prediction.

```

1 def feature_selection(X_train, y_train, X_test):
2     """
3     Takes train and test groups as inputs.
4     Returns the X groups (train and test) with only selected
5     features and the names of the selected features.
6     """
7     # We prefer not to consider banal columns (Open, High,
8     Low)
9     columns_to_drop = ['Open', 'High', 'Low']
10    X_train = X_train.drop(columns=columns_to_drop, errors='
11    ignore')
12    X_test = X_test.drop(columns=columns_to_drop, errors='
13    ignore')
14
15    # Initialize a linear regression model for feature
16    selection
17    model = LinearRegression()
18
19    # Initialize Recursive Feature Elimination (RFE) for
20    feature selection
21    rfe = RFE(model, n_features_to_select=5)

```

```

16
17     # Train and transform the training set
18     X_train_selected = rfe.fit_transform(X_train, y_train)
19
20     # Transform the test set
21     X_test_selected = rfe.transform(X_test)
22
23     # Get the names of the selected features from RFE
24     rfe_feature_names = X_train.columns[rfe.support_]
25
26     return X_train_selected, X_test_selected,
        rfe_feature_names

```

- **Column Removal:** The function starts by excluding certain columns ('Open', 'High', 'Low') that may be considered banal for feature selection.
- **Linear Regression Model:** It uses a linear regression model for feature selection, aiming to retain the most informative features for prediction.
- **Recursive Feature Elimination (RFE):** RFE is employed to recursively remove features and rank them based on their importance. The number of features to select is set to 5.
- **Transformation of Sets and Feature Names:** The function transforms both the training and test sets based on the selected features using RFE, which are returned along with the names of the selected features.

2.4.3 Machine Learning Models

For individual asset prediction, machine learning models were trained using the prepared datasets.

```

1     models = {
2         'Linear Regression': LinearRegression(),
3         'Random Forest': RandomForestRegressor(),
4         'K Neighbors Regressor' : KNeighborsRegressor(),
5     }

```

The following models were considered:

- **Random Forest Regressor**
- **Linear Regression**

- **K Neighbors Regressor**

```

1      # iterates over the models
2      for name, possible_model in models.items():
3          # fits the model to the train groups
4          possible_model.fit(X_train, y_train)
5          # predicts the target values of the x_test group
6          y_pred = possible_model.predict(X_test)
7          # calculates accuracy of predictions using mean
squared error
8          mse = mean_squared_error(y_test, y_pred)
9
10         # finding model with lowest mse
11         if mse < best_mse:
12             best_mse = mse
13             model = possible_model

```

In the `choose_model` function, models were evaluated based on Mean Squared Error (MSE), in fact, function selects the model with the lowest MSE score.

```

1          # writes in the file which model was chosen
2          print(f"For the dataset {ds_name}, {name} was
chosen.", file=file)
3          #draws accuracy scatter for selected model
4          accuracy_scatter(y_test, y_pred, ds_name)

```

The name of the model chosen for a given asset is going to be saved in a file in the *performance*, along with a scatter plot that visualizes the model accuracy.

2.4.4 Modelling

The `modelling` function is responsible for training machine learning models using the specified dataset. The function follows a series of steps, including feature engineering, feature selection, model training, and performance evaluation.

- **Feature Engineering:** The function begins by performing feature engineering on the dataset to prepare the data for modeling.

```

1      #performs feature engineering
2      df = feature_eng(dataset)
3
4      # sets date as index
5      df.set_index('Date', inplace=True)

```

- **Feature Selection:** It then splits the dataset, performs feature selection, and saves the selected features to a text file.

```

1      # close price is target value
2      X = df.drop('Close', axis=1)
3      Y = df['Close']
4
5      # splitting dataset for feature selection
6      X_train, X_test, y_train, y_test = train_test_split(X
7      , Y, test_size=0.2, random_state=42)
8
9      #performs feature selection
10     X_train, X_test, features = feature_selection(X_train
11     , y_train, X_test)
12
13     #saving the features in a txt file
14     # (if the model is linear regressor it does not have
15     a "feature_name" attribute)
16     os.makedirs('features', exist_ok=True)
17     dataset_name = dataset.replace('.csv', '')
18     with open(f'features/{dataset_name}_features.txt', 'w')
19     as file:
20         for name in features:
21             print(f"{name}", file=file)

```

- **Model Training:** The machine learning model is chosen using the `choose_model` function, and the model is trained on the training set.

```

1      # splitting train group for cross validation
2      X_train, X_val, y_train, y_val = train_test_split(
3      X_train, y_train, test_size=0.2, random_state=42)
4
5      # needed dataset names without extensions to name .
6      pkl and .txt files
7      dataset_name = os.path.splitext(dataset)[0]
8
9      # choosing model
10     model = choose_model(X_train, y_train, X_test, y_test
11     , dataset_name)
12
13     #indicating feature_names
14     model.feature_names = features
15
16     # training the model
17     model.fit(X_train, y_train)
18
19     # predicting
20     y_pred = model.predict(X_test)

```

- **Performance Evaluation:** The model's performance is evaluated on the test set, and various metrics, such as Mean Squared Error (MSE) and R-squared (R^2), are calculated. Cross-validation is also performed to assess the model's generalization performance. These metrics are then saved in a text file in the *performance* directory.

```

1      # calculating mse
2      mse = mean_squared_error(y_test, y_pred)
3      # calculating r2 scored
4      r2 = r2_score(y_test, y_pred)
5      # calculating residual standard deviation
6      std_residual = np.sqrt(mse)
7
8      # cross_validation
9      cross_val_scores = cross_val_score(model, X, Y, cv=5,
10                                         scoring='neg_mean_squared_error')
11      # calculating average cross validation
12      avg_cross_val_mse = -cross_val_scores.mean()
13
14      # saving performance data
15      with open(f'performance/{dataset_name}_performances.
16                txt', 'a') as file:
17          print("STATISTICS:", file=file)
18          print(
19              f"MSE: {mse}\nR : {r2}\nResidual Standard
20              Deviation: {std_residual}\nAverage mse with cross
21              validation: {avg_cross_val_mse}",
22              file=file)

```

- **Model Saving:** The trained model is saved in *models* directory in a binary (.pkl) file for future use.

```

1      #saving models in a directory
2      with open(f'{models_directory}/model_{dataset_name}.
3                pkl', 'wb') as file:
4          # dumps pre-trained model in the .pkl file
5          pickle.dump(model, file)
6          print(f"Saved and trained model for {dataset}!")

```

2.5 Constraint Satisfaction Problem (CSP)

The portfolio optimization process was formulated as a CSP, whose solutions are dictionaries of the form **asset : allocation**. The objective was to find an optimal allocation of investments across assets.

Our variables were the Assets and due to the long compiling time of the CSP solver, we decided to scale the domains accordingly to the maximum investment desired:

```

1 def build_portfolio_csp(ds_names, min_investment,
2   max_investment, risk_factor, asset_volatilities):
3     # the increment in the domain changes as the max
4     # investment changes
5     # done to reduce waiting times during CSP
6     if max_investment <= 100:
7         incr = 10
8     elif max_investment > 100 and max_investment <= 400:
9         incr = 50
10    else:
11        incr = 100
12
13    # setting domain for CSP variables
14    domain = np.arange(0, max_investment + incr, incr)
15
16    # creating variables
17    variables = []
18    for name in ds_names:
19        variables.append(Variable(name, domain))

```

We also created three constraints:

```

1 constraints = []
2
3 # investment limits constraints
4 constraints.append(Constraint(scope=variables, condition=
5   lambda *values: sum(values) <= max_investment))
6 constraints.append(Constraint(scope=variables, condition=
7   lambda *values: sum(values) >= min_investment))
8
9 # volatility constraint
10 constraints.append(
11     Constraint(scope=variables, condition=lambda *values:
12       calculate_portfolio_volatility(*values) <= risk_factor))

```

- **Investment Limits:** Ensuring that the total investment does not exceed predefined minimum and maximum thresholds.
- **Risk Factor Constraint:** Limiting the portfolio's volatility to a specified risk factor. This is achieved through the following lambda function

that calculates the weighted average volatility of the portfolio:

```

1  def calculate_portfolio_volatility(*values):
2      """
3      :return: average volatility of the portfolio
4      """
5
6      # calculating the sum of weighted volatilities
7      weighted_volatilities_sum = sum(
8          value * asset_volatilities[asset] for value,
9          asset in zip(values, asset_volatilities))
10
11     # calculating the total sum of investments
12     total_investment = sum(values)
13
14     # checking for division by zero
15     if total_investment == 0:
16         return 0
17
18     # calculating the portfolio volatility
19     portfolio_volatility = weighted_volatilities_sum /
20     total_investment
21
22     return portfolio_volatility

```

The dictionary *asset_volatilities* is an input of the *build_portfolio_csp* function and is calculated as described in the following subsection.

2.5.1 Volatility Constraint

In this project, the volatility of selected assets, was calculated using the following procedure:

The relative price was calculated as the proportional change in stock price between a day and the day before. For each trading day i , the relative price (*Price relative*) was computed using the formula:

$$\text{Price relative}_i = \frac{\text{Close}_i}{\text{Close}_{i-1}}$$

Simultaneously, the proportional change in stock price, referred to as the *Daily Return*, was computed using the logarithmic return formula:

$$\text{Daily Return}_i = \ln \left(\frac{\text{Close}_i}{\text{Close}_{i-1}} \right)$$

The daily volatility (*DailyVolatility*) was then calculated as the standard

deviation of the `Daily Return` series. This metric provides a measure of the day-to-day variability in stock prices.

Then, the daily volatility was annualized. The annualized daily volatility (`AnnualizedDailyVolatility`) was computed using the formula:

$$\text{AnnualizedDailyVolatility} = \text{DailyVolatility} \times \sqrt{365}$$

The square root of 365 accounts for the number of trading days in a year. The resulting value represents the expected annualized volatility measured in calendar days.

The process was repeated for multiple assets, and the annualized daily volatility values were stored in a dictionary (`asset_volatilities`). These values were later used in the portfolio optimization process, in particular, in *risk factor* constraint.

2.5.2 Solver

The CSP was solved using the Arc Consistency algorithm provided in class, to which we added a new `solve_all` function. This function serves the purpose of returning all possible solutions.

`solve_all`

The ‘`solve_all`’ method in the `Arc_Consistency` class (module `CSP_solver`) implements an arc consistency solver for Constraint Satisfaction Problems. The solver aims to find all solutions to the given CSP by iteratively applying arc consistency and domain splitting.

It uses arc consistency to make the CSP arc-consistent. It then explores the search space by selecting a variable with more than one possible value in its domain. If such a variable is found, the domain is split into two, and the solver proceeds recursively by considering both domains. The process continues until either all solutions are found or a domain with no solution is encountered.

2.6 Function main

The `main` function orchestrates the entire workflow, thus it includes data preparation, predictions, portfolio construction, and visualization.

- **Loading and Preparing Data:**

- Loads the data from CSV files corresponding to different asset datasets like AMD, CSCO, etc.
- Utilize the `calculate_volatility` function to compute asset volatilities for the datasets.

```

1  # List of dataset names
2  dataset_names = ["AMD", "CSCO", "QCOM", "SBUX", "TSLA"]
3
4  # Appends '.csv' to each dataset name
5  datasets = [dataset + '.csv' for dataset in dataset_names]
6
7  # Calculates volatilities for assets
8  asset_volatilities = calculate_volatility(datasets)
9
10 # Dictionary to store prediction data for each dataset
11 prediction_data = {}
12
13 # Dictionary to store the last 'Close' values of the
14 # training set for each dataset
15 last_y_train_values = {}
16
17 # Dictionary to store the last 'Close' values of the
18 # training set for each dataset
19 datasets = []

```

• Modelling and Prediction:

- For each dataset, it loads a model if available; otherwise, if the model is not available, perform modeling using the `modelling` function.
- It performs feature engineering on the dataset and uses the `get_features` function to make sure the features on this dataframe are the same as the ones that were used during the training phase.
- It makes predictions on the test data using the loaded model.

```

1  if model is not None:
2      print(f"Model loaded successfully for {dataset_name}")
3  else:
4      modelling(dataset)
5      model = load_model(dataset_name)

```

```

6         print(f"Model loaded successfully for {
dataset_name}")
7
8         # Append dataset name to the list
9         datasets.append(dataset)
10
11        # Feature engineering
12        df = feature_eng(dataset)
13        df['Date'] = pd.to_datetime(df['Date'])
14        df = df.dropna()
15        df.set_index('Date', inplace=True)
16
17        # Splits the dataset into training and test sets
18        split_date = df.index.max() - timedelta(days=365)
19        train = df[df.index <= split_date]
20        test = df[df.index > split_date]
21
22        # Separate independent variables (X) from
23        dependent variables (Y)
24        X_train = train.drop('Close', axis=1)
25        y_train = train['Close']
26        # We need the same features that the model has
27        been trained on
28        X_test = get_features(dataset_name, test)
29        # Makes predictions on the test set (future data
30        )
31        y_pred = model.predict(X_test)

```

- **Portfolio Construction:**

- generate a portfolio constraint satisfaction problem (CSP) using asset datasets, minimum and maximum investment limits, risk factor, and asset volatilities.
- solve the CSP using the `csp_solver` function.

```

1        # Builds the CSP for portfolio optimization
2        csp = build_portfolio_csp(datasets, min_investment,
3        max_investment, risk_factor, asset_volatilities)
4
5        # Solve the CSP to obtain all possible portfolio
6        solutions
7        solutions = csp_solver(csp)

```

- **Evaluation and Visualization:**

- Evaluates different solutions obtained from the CSP solver.

- Selects the solution with the highest expected return. We simulated this taking the price of the asset during the last day of training (hypothetical day when the asset is bought) and the price of the asset during the last day of the predictions(hypothetical selling day). Then we calculated the expected return as the ratio of the allocation and the price value at the time the asset was bought, multiplied by the predicted value, minus the allocation invested.
- Creates graphs for the visualization of the selected solution thanks to the `data_visuaization` function.

```

1      # Records the last 'Close' value of the training
      set for each dataset
2      last_y_train_values[dataset] = y_train.iloc
      [-1]
3
4      # Records the last 'Close' value of the
      prediction set for each dataset
5      prediction = y_pred[-1]
6      prediction_data[dataset] = prediction
7  # Finds the best portfolio solution with the highest
      expected return
8      best_return = 0
9      best_solution = None
10     for solution in solutions:
11         #initializing total return of the portfolio
12         total_return = 0
13
14         # Calculates expected return for each asset
15         in the portfolio
16         for variable, allocation in solution.items():
17             dataset_name = variable.name
18             prediction = prediction_data[dataset_name]
19
20             last_asset_value = last_y_train_values[
21                 dataset_name]
22
23             #represents an estimate of the return
24             expected from the current asset
25             expected_return = prediction * (
26                 allocation / last_asset_value) - allocation
27
28             total_return += expected_return
29
30         #picks the solution with the highest expected
31         return
32         if total_return > best_return:
33             best_return = total_return
34             best_solution = solution

```

- **Returns:**

- Returns the final selected solution or None if no optimal solution is found.

```

1         # Visualizes and prints the best portfolio
      solution if found
2     if best_solution is not None:
3         final_solution = {}
4         for x, y in best_solution.items():
5             if y!=0:
6                 final_solution[x] = y
7
8
9         data_visualization(final_solution)
10
11
12         best_solution_str = {key: round(value, 2) for key
, value in final_solution.items()}
13         print(str(best_solution_str) + '. Expected return
: ' + str(best_return))
14
15
16         return final_solution
17

```

2.6.1 load_model function

The load_model function is responsible for loading a pre-trained model corresponding to a given dataset.

```

1     models_directory = 'models'
2     model_filename = f'model_{dataset_name}.pkl'
3     model_path = os.path.join(models_directory,
model_filename)

```

Here we are building the path to the pre-trained model stored in a pickle file.

```

1     if os.path.exists(model_path):
2         # loads the model from the file
3         with open(model_path, 'rb') as file:
4             model = pickle.load(file)
5         return model
6     else:
7         # prints a message if the model file is not found
8         print(f"Model file not found for {dataset_name}")
9         return None

```

Then we check if the model file exists: if it does, the output of the function is said model, otherwise the function returns None.

2.6.2 data_visualization function

```
1 def data_visualization(solution):
```

The `data_visualization` function is responsible for generating interactive time series plots, pie charts and polar charts representing asset allocations. It takes as a parameter a dictionary containing the selected solution with asset names as keys and their corresponding allocations as values.

```
1 #lists for pie chart and polar chart
2     assets = []
3     allocations = []
4
5
6     for var, y in solution.items():
7         file_name = var.name
8
9         #time series plot
10        df = data_preparation('stocks/'+file_name)
11        color = random.choice(colors)
12        fig.add_trace(go.Scatter(x=df['Date'], y=df['High'],
13                                mode='lines', name=file_name.replace('.csv', ''), marker=
14                                dict(color=color)))
15        colors.remove(color)
16
17        #for pie chart and polar chart
18        file_name = file_name.replace(".csv", "")
19        assets.append(file_name)
20        allocations.append(y)
```

For each asset in the given solution, the function:

- Prepares the data for time series plots using the `data_preparation` function.
- Adds a trace for the time series plot with distinct colors.
- Generates data for pie and polar charts, where assets and their allocations are added to the corresponding lists.
- Creates a polar chart using Matplotlib to represent asset allocations in a circular format.

```

1 #time series
2     fig.update_layout(title='Assets trends',
3                       xaxis_title='Date',
4                       yaxis_title='Price',
5                       plot_bgcolor='#ffffff')
6
7     if os.path.exists("static/time_series_plot_interactive.
8 html"):
9         os.remove("static/time_series_plot_interactive.html")
10    fig.write_html("static/time_series_plot_interactive.html"
11 )

```

After creating the time series plot, the function saves it as an interactive HTML file and generates a pie chart using Matplotlib.

```

1     #pie chart
2     colors = ['#6f3cff', '#fca778', '#c2bcff', '#ffc282', '#7
3 f50ff']
4     plt.pie(allocations, colors=colors, labels=assets)
5     plt.axis('equal')
6
7     # saves pie chart as an image file
8     if os.path.exists("static/piegraph.jpg"):
9         os.remove("static/piegraph.jpg")
10    plt.savefig("static/piegraph.jpg", dpi=300)
11    plt.cla()
12    plt.clf()
13    plt.close()

```

The pie chart is saved as an image file.

```

1     # Polar chart
2     theta = np.linspace(0, 2 * np.pi, len(assets), endpoint=
3 False)
4
5     plt.figure(figsize=(10, 6))
6     plt.subplot(polar=True)
7
8     # Plots allocations on polar chart
9     plt.plot(theta, allocations)
10    plt.fill(theta, allocations, 'b', alpha=0.2)
11
12    # Adds legend and title
13    plt.legend(labels=('Allocations',), loc=1)
14    plt.title("Asset Allocations")
15
16    # Saves polar chart as an image file
17    if os.path.exists("static/polargraph.jpg"):
18        os.remove("static/polargraph.jpg")
19    plt.savefig("static/polargraph.jpg", dpi=300)

```

```

20 plt.cla()
21 plt.clf()
22 plt.close()

```

Additionally, the function creates a polar chart using Matplotlib to represent asset allocations in a circular format. The polar chart is saved as an image file. All files are saved in the 'static' directory

2.6.3 get_features function

The `get_features` function is responsible for filtering the features in a DataFrame based on a features file corresponding to a specific dataset. The features file contains the features that were previously used to train the model for the dataset.

```

1     with open(f'features/{dataset}_features.txt', 'r') as
      file:
2         lines = file.readlines()
3
4         # Create a list to store feature names from the file
5         column_names_in_file = []
6
7         # Populate the list with feature names from the file
8         for line in lines:
9             line = line.strip()
10            column_names_in_file.append(line)
11            For each asset in the given solution, the function:

```

The function reads the feature names from the features file associated with the given dataset. It then creates a list (`column_names_in_file`) to store these feature names. The list is populated by iterating through the lines of the features file.

```

1     # Creates a set of column names in the DataFrame X
2     column_names_X = set(X.columns)
3
4     # Drops columns from X that are not present in the
      features file
5     for col in column_names_X:
6         if col not in column_names_in_file:
7             X = X.drop(columns=col, errors='ignore')

```

Next, the function creates a set (`column_names_X`) of column names present in the DataFrame X. It iterates through the column names in X, and if a column is not present in the features file, it is dropped from X using the `drop` method with `errors='ignore'`.


```
1 return X
```

Finally, the function returns the DataFrame `X` with only the features specified in the features file.

2.7 app.py

As for the visualization and input of the data, we decided to create a web app utilizing Flask. After importing the libraries, `render_template`, `redirect`, `url_for` to manage the HTTP requests and the module `csp_predictions` for the prediction of the data, we defined three routes:

- `('/')`: is in charge of managing the requests from the homepage and returns the content of the form `index.html`
- `('/handledata')`: it is configured to accept only POST requests and manage the data received from the user by sending them to the function `csp_predictions.main` this function returns a prediction and creates the `polargraph` and the `piegraph` in the form of PNG pictures contained in the folder `./static`
- `('/output')`: after cleaning the output of the function `csp`, it sends the data to the page `output.html` for visualization of the graphs and data

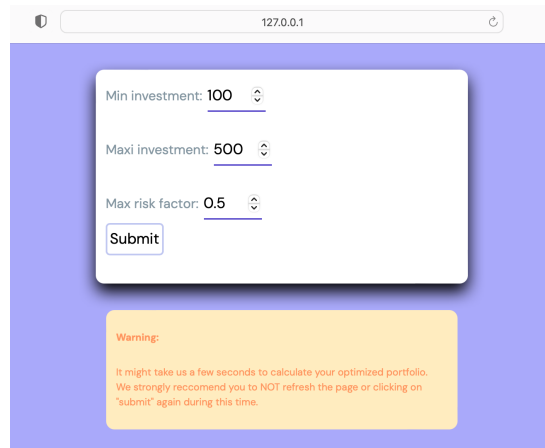
Chapter 3

Experiments and Results

This chapter illustrates how the final result looks through the user interface, providing an example of use, and reports the performance metrics that were use to evaluate our model

3.1 Results as a User

Once a user correctly opens the server, this input page will appear.

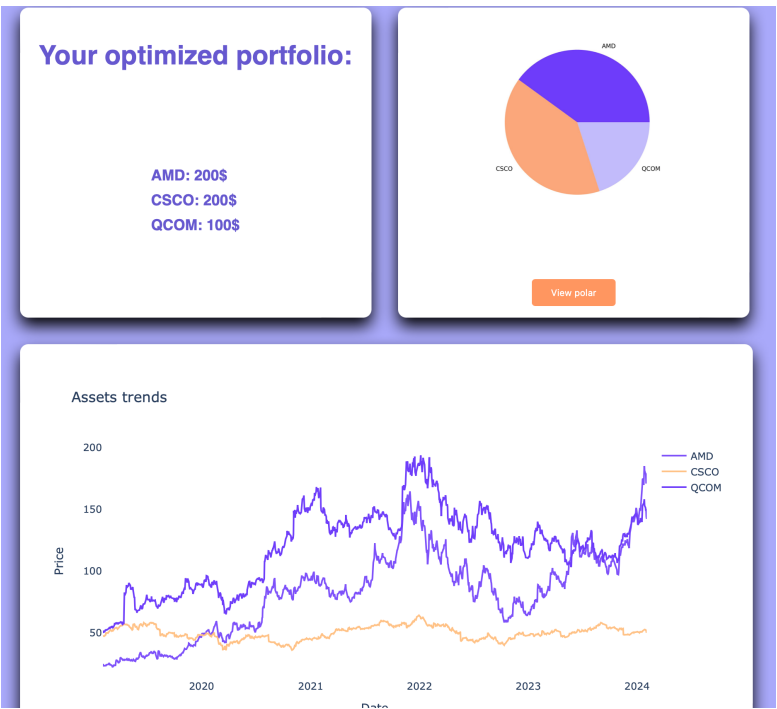


The screenshot shows a web browser window with the address bar displaying '127.0.0.1'. The main content area has a light blue background. In the center, there is a white rounded rectangle containing three input fields with numerical values and a 'Submit' button. Below this rectangle is a yellow warning box with orange text.

Field	Value
Min investment:	100
Maxi investment:	500
Max risk factor:	0.5

Warning:
It might take us a few seconds to calculate your optimized portfolio.
We strongly recommend you to NOT refresh the page or clicking on 'submit' again during this time.

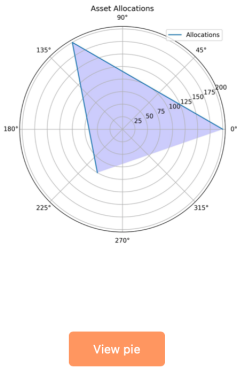
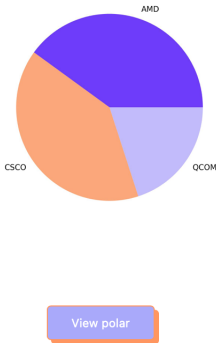
For this example we'll be using 100 as minimum investement, 500 as maximum investment, and 0.5 as maximum risk factor. After clickling on *submit*, the CSP solver will take a few seconds to compile, then the user will be redirected to the output page.



In the output page is possible to view:

- The names of the assets in which the user should invest and how much he or she should invest
- Pie and polar charts of the assets distribution.
- A graph representing the trends of the selected assets throughout time.

It is also possible to switch between the pie and the polar charts by clicking on the orange button.



3.2 Performance Metrics

The performance of the machine learning models is assessed using various metrics to measure their predictive accuracy and generalization capabilities.

3.2.1 Metrics in the modelling function

The following metrics are generated in the `modelling` function of the *ml.py* module:

```
1 # calculating mse
2 mse = mean_squared_error(y_test, y_pred)
```

The **mean squared error** is a measure of the average squared differences between the actual and predicted values. Lower MSE values indicate better performance.

```
1 # calculating r2 scored
2 r2 = r2_score(y_test, y_pred)
```

R-squared measures the proportion of the variance in the dependent variable (close prices) that is predictable from the independent variables (features). A higher R^2 value suggests a better fit of the model to the data.

```
1 # calculating residual standard deviation
2 std_residual = np.sqrt(mse)
```

The **residual standard deviation** provides an estimate of the differences between actual and predicted values. A lower standard deviation indicates a more accurate model.

```
1 # cross_validation
2 cross_val_scores = cross_val_score(model, X, Y, cv=5,
3   scoring='neg_mean_squared_error')
4 # calculating average cross validation
4 avg_cross_val_mse = -cross_val_scores.mean()
```

Cross-validation is a technique used to check how well the model works on new data. It works by dividing the dataset into several parts, training the model on different subsets, and then testing its performance. The average mean squared error obtained from cross-validation gives us a reliable measure of how well the model performs.

These metrics are then saved in the performance directories, and we will show the actual values in **section 3.2.3**.

3.2.2 Performance visualization

In addition to the numerical metrics, scatter plots to visually inspect the accuracy of the predicted values are created for each dataset in the function `accuracy_scatter`. These plots illustrate the correlation between the actual values (y_{test}) and the predicted values (y_{pred}). Each point in the scatter plot represents an observation, allowing for a quick visual assessment of the model's performance.

The scatter plots are saved as PNG images in the "performance" directory. The actual graphs will be shown in the next section.

3.2.3 Performance values

In this section we will be looking at the actual values of the performance metrics and the accuracy graphs of each asset.

AMD

For the dataset AMD, Linear Regression was chosen.

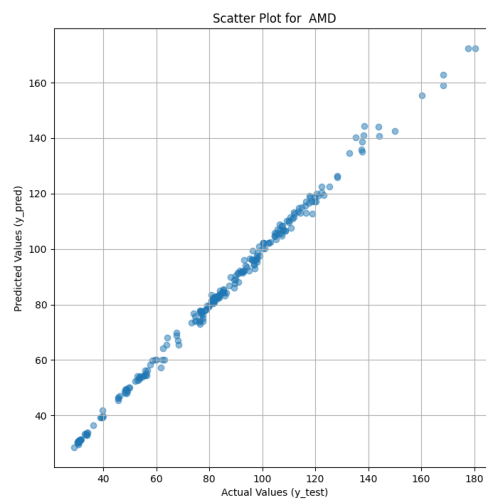
STATISTICS:

MSE: 0.8128050860798128

R^2 : 0.999146446208997

Residual Standard Deviation: 0.9015570342911273

Average mse with cross validation: 2.0888606291576623e-23



CSCO

For the dataset CSCO, Linear Regression was chosen.

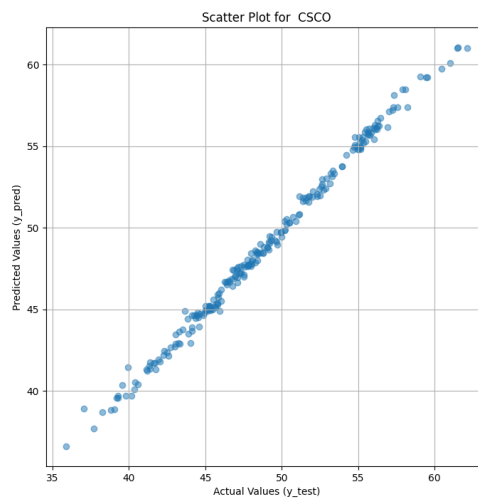
STATISTICS:

MSE: 0.04726269066167061

R^2 : 0.9984290495032537

Residual Standard Deviation: 0.21739984052816277

Average mse with cross validation: 8.699528919100035e-22

**QCOM**

For the dataset QCOM, Linear Regression was chosen.

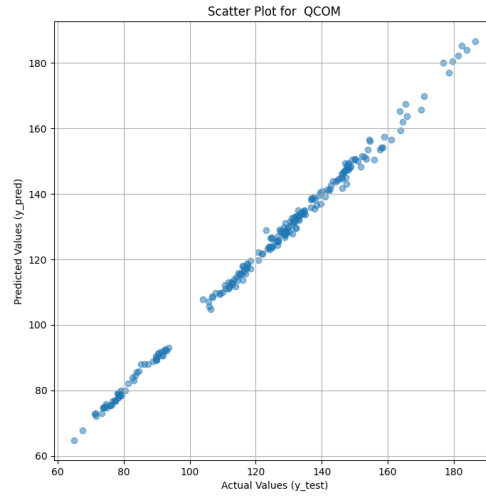
STATISTICS:

MSE: 0.711486471298663

R^2 : 0.9990694221848393

Residual Standard Deviation: 0.8434965745624952

Average mse with cross validation: 3.840006800295034e-20



SBUX

For the dataset SBUX, Linear Regression was chosen.

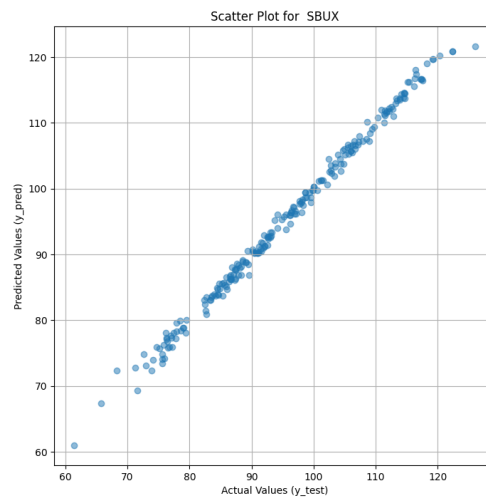
STATISTICS:

MSE: 0.21314719074488495

R^2 : 0.9987828637309117

Residual Standard Deviation: 0.46167866611408953

Average mse with cross validation: 5.2831708838053343e-20



TSLA

For the dataset TSLA, Linear Regression was chosen.

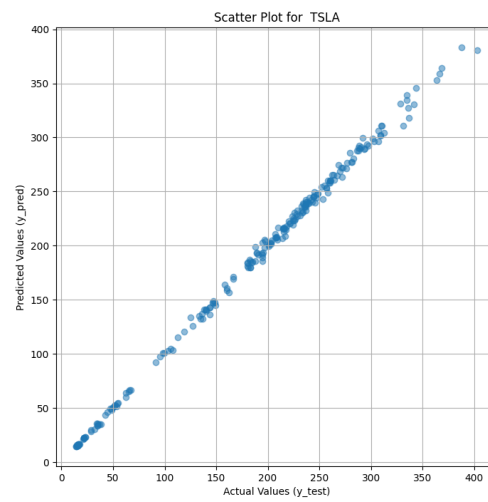
STATISTICS:

MSE: 5.639649264595771

R^2 : 0.9994076195409831

Residual Standard Deviation: 2.3747945731359104

Average mse with cross validation: 2.4498484324992504e-20



Chapter 4

Conclusion

This project aimed to demonstrate the cooperation between machine learning and CSP methodologies in financial portfolio management. Using historical financial data and predictive modeling, the system predicts asset prices and creates customized portfolios based on user preferences. The experiments show that the methodology is effective in generating well-balanced and risk-aware investment portfolios.

The machine learning models, including Random Forest Regressor, Linear Regression, and K Neighbors Regressor, are chosen and used for individual asset prediction. Feature engineering and selection processes enhance the models' performance by considering relevant features. Additionally, CSP is integrated to formulate portfolio construction as a constraint satisfaction problem, considering risk tolerance, and investment limits.

The user interface allows seamless interaction, enabling users to input parameters like minimum and maximum investment limits and risk factors. The CSP solver efficiently explores the solution space to provide optimized portfolio recommendations. Visualization tools, such as time series plots and interactive charts, enhance the user experience and aid in understanding the recommended portfolios.

Finally, we would like to remind the readers that it's important to note that our machine learning model was a didactic, toy model for educational purposes, thus it does not provide an accurate prediction of future values, instead it is a pure simulation.