Download libraries

```
import math
import pickle
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf
import seaborn as sns
from statsmodels.tsa.stattools import acf
from scipy import stats
from scipy.stats import norm
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from arch import arch_model
```

Download: SP500 and stocks price data and convert into logarithmic daily returns, and 1-year US government bond yields and convert to daily yields

```
# Biggest public companies by market-cap as of 01/01/2010
stocks = ["XOM", "MSFT", "WMT", "GOOG", "AAPL", "JNJ", "PG", "JPM", "IBM", "CVX", "BRK-B", "WFC", "PFE", "CSCO",
"KO", "BAC", "GE", "T", "ORCL", "INTC"]

# Downloading data from 1y earlier - results will not consider 2009
data = yf.download(stocks, start="2009-01-01", end="2023-12-31")['Adj Close']
data = data.dropna()

log_returns = np.log(data / data.shift(1)).dropna()
log_returns_demean = log_returns.subtract(log_returns.mean(axis=0), axis=1)

sp500 = yf.download(["^GSPC"], start="2010-01-01", end="2023-12-31")['Adj Close']
sp500_returns = np.log(sp500 / sp500.shift(1)).dropna()
sp500_norm = 100*sp500/sp500.iloc[0]

risk_free = yf.download('^IRX', start="2009-01-01", end="2023-12-31")
risk_free = risk_free[['Close']].rename(columns={'Close': 'Daily Yield'})
risk_free = (1 + risk_free / 100) ** (1 / 365) - 1

log_returns_demean.index = log_returns_demean.index.date
risk_free.index = risk_free.index.date
sp500_returns.index = sp500_returns.index.date
sp500_norm.index = sp500_norm.index
```

Create three estimates of daily mean returns: historical mean, moving average, and exponentially-weighted moving average

```
# Unconditional Historical Means
gross_mean = pd.DataFrame(index = log_returns_demean.index, columns = log_returns.columns)
for n in range(1, (len(log_returns)-25)):
    mean_returns.iloc[25+n] = log_returns[:(24+n)].mean(axis=0)
gross_mean = mean_returns.dropna()

# 25-Trading Day Window Means
moving_average = pd.DataFrame(index = log_returns_demean.index, columns = log_returns.columns)
for n in range(1, (len(log_returns)-25)):
    mean_returns.iloc[25+n] = log_returns[(n-1):(24+n)].mean(axis=0)
moving_average = mean_returns.dropna()

# Exponentially-Weighted Moving Averages
ewma = pd.DataFrame(0.00, index = log_returns_demean.index, columns = log_returns.columns)
for n in range(1, (len(log_returns)-25)):
    lambda_ewma = 0.5
    w_1 = (1-lambda_ewma)/(1-lambda_ewma**(24+n))
    for i in range(1, 25+n):
        ewma.iloc[25+n] += log_returns.iloc[25+n-i] * w_1 * lambda_ewma ** (i-1)
```

Define functions to perform the principal component analysis (PCA)

```
def pca_rotation(data): # With a TxN dataframe (e.g. log_returns), it produces the NxN rotation matrix

    length = len(data)
    coln = len(stocks)
    pca = PCA(n_components = coln)
    principal_components = pca.fit(data)
    rotation_matrix = pca.components_
    column_names = [f'PC{i+1}' for i in range(coln)]
    df_rotation = pd.DataFrame(rotation_matrix, columns = column_names, index = stocks)
    explained_variance = pca.explained_variance_ratio_
    df_rotation.loc['Explained Variance'] = explained_variance

    return df_rotation

def pca_fit(data): # With a TxN dataframe (e.g. log_returns), it produces a TxN dataframe with the PCA beta 
estimates

    length = len(data)
    coln = len(stocks)
    pca = PCA(n_components=coln)
    column_names = [f'PC{i + 1}' for i in range(coln)]
    principal_components = pca.fit_transform(data)
    df_fit = pd.DataFrame(principal_components, columns=column_names, index=data.index)

    return df_fit
```

Define the univariate GARCH(1,1) estimation function

```
def uni_garch(data): # With a Tx1 vector (e.g. log_returns), it produces the Tx1 dataframe with univariate 
GARCH(1,1) estimates of conditional variance

    model = arch_model(data*100, mean = 'Zero', vol='Garch', p=1, q=1)
    garch_fit = model.fit(disp='warn')
    variance = (garch_fit.conditional_volatility ** 2) / (100**2)

    return variance
```

Define the orthogonal GARCH(1, 1) estimation function

```
def var_covar(data): # With a TxN dataframe, it produces the conditional variance-covariance matrix for date T+1

    rotation = pca_rotation(data)
    fit = pca_fit(data)
    columns_to_drop = []  # Using only significant factors

    for i in range(1, len(stocks) + 1):
        if rotation.loc['Explained Variance', f'PC{i}'] < 1/(2 * len(stocks)):
            columns_to_drop.append(f'PC{i}')
    rotation_significant = rotation.drop(columns=columns_to_drop)
    rotation_significant = rotation_significant.iloc[:len(stocks)]
    fit_significant = fit.drop(columns=columns_to_drop)
    significant_factors = fit_significant.columns

    errors = data - fit_significant.dot(rotation_significant.T)  # Finding error variances
    var_errors = pd.DataFrame(columns=stocks, index=data.index)
    for s in stocks:
        var_errors.loc[:, s] = uni_garch(errors.loc[:, s])

    var_factors = pd.DataFrame(columns=significant_factors, index=data.index)  # Finding factor variances
    for f in significant_factors:
        var_factors[f] = uni_garch(fit_significant[f])

    rotation_values = rotation_significant.values # Computing variance-covariance matrix for time t
    sigma_factors_diag = np.diag(var_factors.iloc[-1])
    var_covar_data = rotation_values.dot(sigma_factors_diag).dot(rotation_values.T) + np.diag(var_errors.iloc[-1])
    var_covar_t = pd.DataFrame(var_covar_data, columns = stocks, index = stocks)

    return var_covar_t
```

Define functions to generate portfolio mean returns and conditional volatility

```
def portfolio_return(t, portfolio, mean_returns): # Given date t, a vector portfolio defining the weights for each stock, and the mean returns estimates, it produces the portfolio's mean return

    weights = portfolio
    returns = mean_returns.loc[t]
    mean_return = weights.dot(returns.T)

    return mean_return

def portfolio_vol(sigma, portfolio): # Given a variance-covariance matrix (for date t) and a vector portfolio defining the weights for each stock, it produces the portfolio's conditional volatility

    weights = portfolio
    var = (weights).dot(sigma).dot(weights.T)
    vol = var ** (1/2)

    return vol
```

Define a function to run the Markowitz mean-variance optimization given some data inputs

```
def optimal_portfolio(data, risk_free, mean_returns): # Given a TxN dataframe (e.g. log_returns) and a 1x1 dataframe containing the daily risk-free rate for date T, it finds mean-variance optimal portfolio by simulation

    weights = np.random.uniform(0, 10, (10000, len(stocks))) # Generate 10,000 simulated portfolios
    weights /= weights.sum(axis=1)[:, None] # Standarize the weights such that the sum of all weights equals 1; individual weights are downwardly and upwardly bounded by 0 and 1
    portfolio_r_vol = pd.DataFrame(columns = ["Return", "Volatility", "Slope"], index = range(10000))

    sigma = var_covar(data)
    rf = risk_free.values
    t = data.index[-1]

    for i in range(10000):
        portfolio_r_vol.loc[i, "Return"] = portfolio_return(t, weights[i], mean_returns)
        portfolio_r_vol.loc[i, "Volatility"] = portfolio_vol(sigma, weights[i])
        portfolio_r_vol.loc[i, "Slope"] = (portfolio_r_vol.loc[i, "Return"] - rf) / portfolio_r_vol.loc[i, "Volatility"]
    optimal_weights = weights[portfolio_r_vol['Slope'].idxmax()]

    return optimal_weights
```

Run algorithm using all three different estimates of mean returns and save data

```
algo_gross_mean = pd.DataFrame(columns = stocks, index = log_returns_demean.index)
algo_moving_average = pd.DataFrame(columns = stocks, index = log_returns_demean.index)
algo_ewma = pd.DataFrame(columns = stocks, index = log_returns_demean.index)

for t in log_returns_demean.index:
    print(f"Finding optimal portfolio for {t}")
    try:
        data = log_returns_demean[:t]
        rf = risk_free.loc[t]
        algo_gross_mean.loc[t] = optimal_portfolio(data, rf, gross_mean[:t])
        algo_moving_average.loc[t] = optimal_portfolio(data, rf, moving_average[:t])
        algo_ewma.loc[t] = optimal_portfolio(data, rf, ewma[:t])
    except Exception as e:
        print(f"Risk-free rate not found for date {t}. Skipping this date.")
        continue

with open('algo_portfolio.pkl', 'wb') as f:
    pickle.dump(algo_gross_mean, f)

with open('algo_portfolio1.pkl', 'wb') as f:
    pickle.dump(algo_moving_average, f)

with open('algo_portfolio2.pkl', 'wb') as f:
    pickle.dump(algo_ewma, f)
```

Import data and use only from 2010-01-01

```
with open('algo_portfolio.pkl', 'rb') as f:
    gross_mean = pickle.load(f).dropna()

with open('algo_portfolio1.pkl', 'rb') as f:
    moving_average = pickle.load(f).dropna()

with open('algo_portfolio2.pkl', 'rb') as f:
    ewma = pickle.load(f).dropna()

algo_weights = [gross_mean, moving_average, ewma]
for x in algo_weights:
    x = x[pd.to_datetime(x.index) >= pd.to_datetime("2010-01-01")]
```

Find the returns of the algorithm-generated portfolios

```
gross_mean_r = pd.DataFrame(columns = ["Historical Mean"], index = gross_mean.index)
for t in gross_mean_r.index:
    weights = gross_mean.loc[t].values
    r = log_returns.loc[t].values
    gross_mean_r.loc[t, "Historical Mean"] = r.dot(weights.T)

moving_average_r = pd.DataFrame(columns = ["Moving Average"], index = moving_average.index)
for t in moving_average_r.index:
    weights = moving_average.loc[t].values
    r = log_returns.loc[t].values
    moving_average_r.loc[t, "Moving Average"] = r.dot(weights.T)

ewma_r = pd.DataFrame(columns = ["EWMA"], index = ewma.index)
for t in ewma_r.index:
    weights = ewma.loc[t].values
    r = log_returns.loc[t].values
    ewma_r.loc[t, "EWMA"] = r.dot(weights.T)

algo_portfolio = pd.concat([gross_mean_r, moving_average_r, ewma_r, sp500_returns], axis=1).dropna()
```

For every day, rank the performance of the different portfolios and the SP500 index

```
rank = algo_portfolio.rank(axis=1, method='dense', ascending=False).astype(int)
rank = 5 - rank
columns = ["Historical Mean", "Moving Average", "EWMA", "SP500"]
rank.columns = ["Historical Mean", "Moving Average", "EWMA", "SP500"]
```

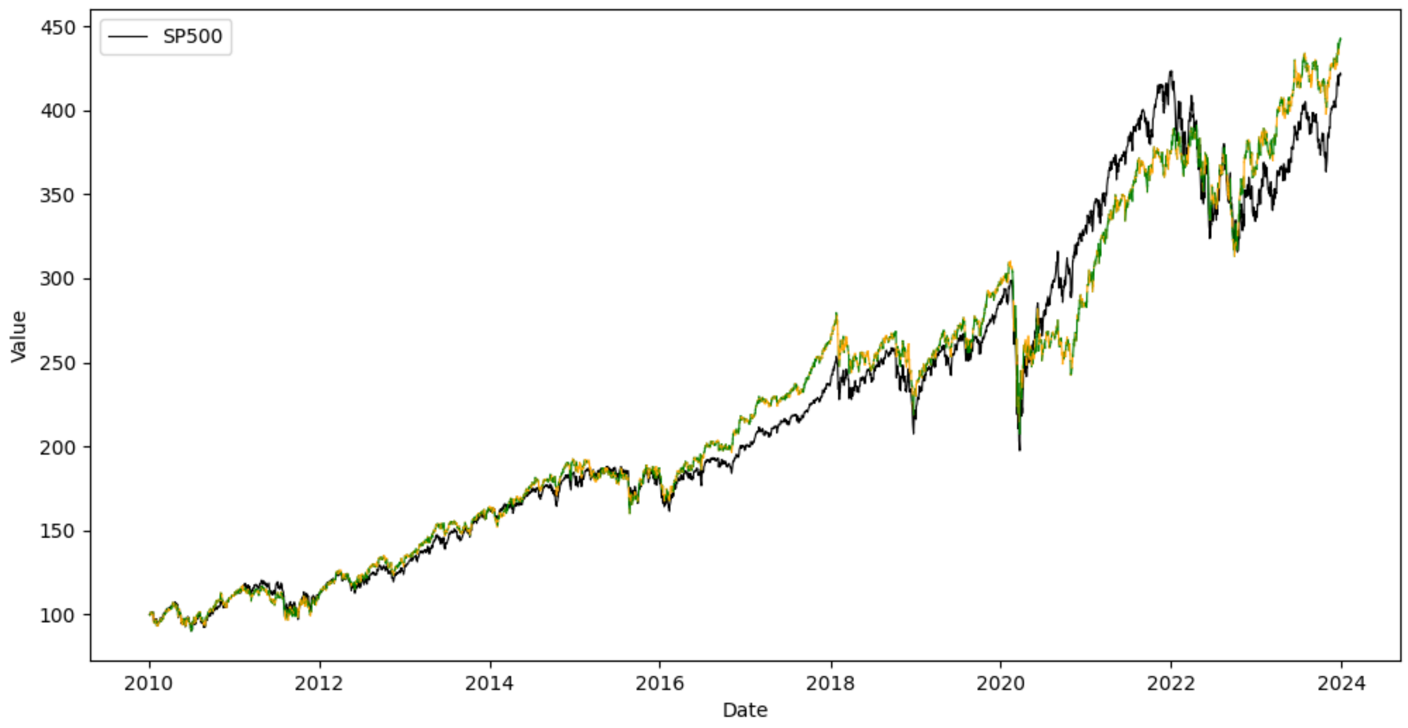
Convert daily returns into daily prices with base 2010-01-01 = 100

```
algo_returns = [gross_mean_r, moving_average_r, ewma_r]
algo_results = pd.DataFrame(columns = columns, index = algo_portfolio.index)
algo_results.iloc[0, 0:3] = 100

for i in range(1, len(algo_portfolio.index)):
    algo_results.iloc[i, 0] = algo_results.iloc[i-1, 0] * math.e ** algo_portfolio.iloc[i, 0]
    algo_results.iloc[i, 1] = algo_results.iloc[i-1, 1] * math.e ** algo_portfolio.iloc[i, 1]
    algo_results.iloc[i, 2] = algo_results.iloc[i - 1, 2] * math.e ** algo_portfolio.iloc[i, 2]
    algo_results.iloc[i, 3] = algo_results.iloc[i - 1, 3] * math.e ** algo_portfolio.iloc[i, 3]
```

Plot the value of the SP500 vs the different mean return estimates (e.g. EWMA)

```
plt.figure(figsize=(12, 6))
plt.plot(sp500_norm.index, sp500_norm, label='SP500', color='black', linewidth = 0.8)
for i in range(1, len(algo_results)):
    if rank["SP500"].iloc[i] > rank["EWMA"].iloc[i]:
        color = 'orange'  # Orange line if SP500 outperforms algorithm
    else:
        color = 'green'  # Green if algorithm outperforms SP500
    plt.plot(algo_results.index[i - 1:i + 1], algo_results["EWMA"].iloc[i - 1:i + 1], color=color, linewidth=0.8)
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.show()
```


SP500
Value
Date
450
400
350
300
250
200
150
100
2010
2012
2014
2016
2018
2020
2022
2024