1 Appendix: Python notebook

1.1 Prelim

1.1.1 Import stuff

[ ]:
```
# General
import os # change working directory
import pandas as pd # dataframes
import numpy as np # numpy
import math # square root
from matplotlib import pyplot as plt # plot figures
import copy # make actual copies

# Question a)
from sklearn.decomposition import FactorAnalysis # factor analysis
from scipy.stats import jarque_bera # jarque-bera statistic for normality
from statsmodels.regression.linear_model import OLS # OLS

# Simulation
from scipy.stats import lognorm # lognormal distribution
from scipy.stats import gamma # gamma distribution
from scipy.stats import norm # normal distributin
from typing import Callable # type hinting functions

os.getcwd()
```

1.1.2 Load Data

[ ]:
```
#os.chdir('C:/Users/pelpi/Documents/VSCode repositories/portfolio-management/
↪src')

# Load data on returns of 550 stocks between 2010 and 2016, and risk free rate
os.chdir("..") # Change workin directory to parent
return_df = pd.read_csv('data\RET.csv')
rf_df = pd.read_csv('data\F-F_Research_Data_Factors_daily.csv')
rf_df.rename({'Unnamed: 0': 'DATE'}, axis=1, inplace=True)
os.chdir(os.getcwd() + '\src') # Change working directory back

# Convert strings of dates to datetime objects
return_df['DATE'] = pd.to_datetime(return_df['DATE'])
rf_df['DATE'] = pd.to_datetime(rf_df['DATE'], format = '%Y%m%d')

# Change Fama-French to percentages
rf_df[rf_df.columns[1:]] = rf_df[rf_df.columns[1:]]/ 100
```

[ ]:
```
# Overview of what the data looks like
return_df.head()
```

[ ]:
```python
# Show the risk free rate at the start of the stock return sample
rf_df[rf_df['DATE'] > return_df.iloc[0]['DATE']].head()
```

[ ]:
```python
# Substract the risk free return from the returns
def substract_rf(returns_df, rf_df, date_col, rf_col):
    # Merge dataframes based on the 'date' column
    merged_df = pd.merge(returns_df, rf_df[[date_col, rf_col]], on=date_col,␣
↪how='inner')

    # Subtract risk-free rate from each stock return
    for stock_col in returns_df.columns[1:]:  # Assuming the first column is␣
↪'date'
        merged_df[stock_col] = merged_df[stock_col] - merged_df[rf_col]

    # Drop the 'risk_free_rate' column if you don't need it anymore
    merged_df = merged_df.drop(rf_col, axis=1)

    return merged_df

df = substract_rf(return_df, rf_df, 'DATE', 'RF')
```

[ ]:
```python
df.head()
```

1.2 Questions

1.2.1 Question a) Fit Factor Model

[ ]:
```python
# Derive factors
K = 1 # Number of latent factors
factor_model = FactorAnalysis(n_components=K)
factor_model.fit(df.drop('DATE', axis=1).T) # Take transpose so that the factors␣
↪differ per time instead of the stocks
factors = factor_model.components_[0]

# Obtain betas for all stocks
beta_dict_fa = dict()
for stock in df.columns[1:]:

    # Regress the factors on the returns to obtain beta, store in a dictionary
    beta = OLS(df[stock], factors).fit().params[0]
    beta_dict_fa[stock] = beta
```

[ ]:
```python
def quick_distribution_check(data, bins = 100, title='Distribution Check'):
    # Plot factors to see whether normality assumption makes sense
    plt.hist(data, density=True, bins=bins, edgecolor='black', alpha=0.7);
    plt.title(title)
    plt.xlabel('Values')
```

```
    plt.ylabel('Density')
    print(f'Jarque-Bera: {jarque_bera(data).statistic:0.2f}')

    # Find and print the mean and variance
    data_mean = data.mean()
    data_std = data.std()
    print(f'Mean: {data_mean}, Std: {data_std}')

    return data_mean, data_std

# Plot histogram and automatically estimate the normal parameters
factor_mean, factor_std = quick_distribution_check(factors,
                                                   title='Histogram and␣
 ↪theoretical distribution of factors')

# Add theoretical normal histogram
def add_theoretical_histogram(func, *args, **kwargs):
    xmin, xmax = plt.xlim()
    x = np.linspace(xmin, xmax, 100)
    p = func(x, *args, **kwargs)
    plt.plot(x, p, 'k',
             linewidth=2,
             color='red',
             label='Theoretical -- Distribution')

# Add the theoretical histogram under normality
add_theoretical_histogram(norm.pdf, factor_mean, factor_std)
```

1.2.2 Question a*) Fit Fama-French market factor

Get information from factor over entire sample (1926-07-01 - 2023-09-29)

[ ]:
```
mkt_mean, mkt_std = quick_distribution_check(rf_df['Mkt-RF'],
                                             bins=100,
                                             title='Histogram and theoretical␣
 ↪distribution of factors')
add_theoretical_histogram(norm.pdf, mkt_mean, mkt_std)
```

Fit betas

[ ]:
```
# Merge dataframes on the date column
df_merged = pd.merge(df, rf_df[['DATE', 'Mkt-RF']], on='DATE', how='inner')

# Obtain betas for all stocks
beta_dict_ff = dict()
std_dict_ff = dict()
for stock in df.columns[1:]:

    # Regress the factors on the returns to obtain beta, store in a dictionary
```

```
    model = OLS(df_merged[stock], df_merged['Mkt-RF']).fit()
    beta_dict_ff[stock] = model.params[0]

    # Save the scale of the model
    std_dict_ff[stock] = math.sqrt(model.scale)
```

1.2.3 Question b) Optimal Sharpe ratio under factor analysis factor

[ ]:
```
optimal_sharpe = (factor_mean) / factor_std
print(optimal_sharpe)

# Negative Sharpe, so the option to plot a histogram of the cumulative returns
quick_distribution_check(df[df.columns[1:]].sum(axis=1), bins = 100);
#quick_distribution_check((1+df[df.columns[1:]]).product(axis=1), bins = 100); #
 ↪Gives very strange results, so these returns probably are additive anyway??
```

1.2.4 Question b*) Optimal Sharpe ratio under Fama-French market factor

[ ]:
```
print(mkt_mean / mkt_std)
```

1.2.5 Simulation

Histogram of betas

[ ]:
```
# Plot histogram
beta_dict = beta_dict_ff # _ff for Fama-Fench, _fa for factor analysis
betas = np.array(list(beta_dict.values()))
quick_distribution_check(betas,
                         bins=50,
                         title='Histogram and theoretical distribution of betas')

# Estimate gamma parameters
beta_params = gamma.fit(betas)
print(beta_params)

# Plot theoretical histogram
add_theoretical_histogram(gamma.pdf, *beta_params)
```

Histogram of error standard deviations

Factor Analysis

[ ]:
```
# Plot histogram
quick_distribution_check(np.sqrt(factor_model.noise_variance_))

# Estimate parameters
std_params_fa = lognorm.fit(np.sqrt(factor_model.noise_variance_))
print(std_params_fa)
```

```
# Add theoretical histogram
add_theoretical_histogram(lognorm.pdf, *std_params_fa)
```

Fama-French factors

```
[ ]: quick_distribution_check(pd.Series(std_dict_ff.values()),
                             bins=50,
                             title='Histogram and theoretical distribution of std.␣
      ↪devs.')

     # Estimate parameters
     std_params_ff = lognorm.fit(np.sqrt(factor_model.noise_variance_))
     print(std_params_ff)

     # Add theoretical histogram
     add_theoretical_histogram(lognorm.pdf, *std_params_ff)
```

Check if betas and standard deviations are correlated

```
[ ]: plt.scatter(beta_dict_ff.values(), std_dict_ff.values())
```

Methods and Classes

```
[ ]: def make_drawer(function, *args, **kwargs):
         '''Method that wraps around a given function with given parameters to turn␣
      ↪it into a simple callable'''

         def drawer():
             return function(*args, **kwargs)

         return drawer

     class MarketSimulator:
         '''Class that creates a simulated market that is fully determined by a one␣
      ↪factor model'''

         def __init__(self, beta_drawer: Callable, std_drawer: Callable,␣
      ↪factor_drawer: Callable):
             self.beta_drawer = beta_drawer
             self.std_drawer = std_drawer
             self.factor_drawer = factor_drawer

         # Used for questions on simulated data
         def simulate(self, n_assets, n_observations):
             '''Method that simulates a MarketSimulator for a given number of assets␣
      ↪and observations'''

             # Draw the betas and factors
             betas = np.array([self.beta_drawer() for i in range(n_assets)])
```

```
        stds = np.array([abs(self.std_drawer()) for i in range(n_assets)])
        factors = np.array([self.factor_drawer() for t in range(n_observations)])

        # Create a matrix of the simulated returns
        Betas = np.vstack([betas] * n_observations)
        Factors = np.vstack([factors] * n_assets).T
        Errors = np.random.normal(0, stds, (n_observations, n_assets))

        simulated_returns = np.multiply(Betas, Factors) + Errors

        # Store the simulated returns in a dataframe
        cols = [f'{beta:0.2f}' for beta in betas]
        simulated_returns_df = pd.DataFrame(simulated_returns, columns = cols)

        # Return the dataframe as a MarketSimulation
        simulation = MarketSimulation(simulated_returns_df)
        simulation.betas = betas
        simulation.stds = stds
        simulation.factors = factors
        simulation.n_observations = n_observations

        return simulation

class MarketSimulation(pd.DataFrame):
    '''Extension of a pandas Dataframe that contains the results of
    a market simulation and useful methods to analyze portfolio
    strategy performances'''

    def __init__(self, *args, **kwargs):
        # use the __init__ method from DataFrame to ensure
        # that we're inheriting the correct behavior
        super(MarketSimulation, self).__init__(*args, **kwargs)
        self.betas = None
        self.stds = None
        self.factors = None
        self.n_observations = None

    @property
    # this method is makes it so our methods return an instance
    # of MarketSimulation, instead of a regular DataFrame
    def _constructor(self):
        return MarketSimulation

    # Used for question 2c)
    def get_equally_weighted_performance(self, in_sample_fraction, stop_fraction␣
↪= 1, return_returns = False):
```

```
        '''Method that gets the Sharpe ratio and turnover of an equally weighted␣
↪portfolio for a given in_sample_fraction of observations'''

        # Get the out-of-sample length
        out_of_sample_length = math.ceil((1-in_sample_fraction) * self.
↪n_observations)

        # Set up the easy weights matrix
        weights = np.ones((out_of_sample_length, len(self.betas))) / len(self.
↪betas)

        # Return the performance
        return self.get_performance(weights, in_sample_fraction, stop_fraction,␣
↪return_returns)

    # Used for a lot of questions
    def get_general_performance(self, weights_function, estimation_window,␣
↪in_sample_fraction, stop_fraction, return_returns=False):
        '''Method that gets the Sharpe ratio and turnover of for a given weights␣
↪function for a given in_sample_fraction of observations'''

        # Get the out-of-sample length
        out_of_sample_length = math.ceil((1-in_sample_fraction) * self.
↪n_observations)

        # Set up weights matrix
        weights = np.ones((out_of_sample_length, len(self.betas)))

        # Set the index where the out of sample period begins
        begin_index = math.floor(self.n_observations * in_sample_fraction)

        for t in range(out_of_sample_length):

            # Utilize last estimation window rows to determine sample covariance␣
↪matrix
            used_returns = self[begin_index + t - estimation_window:(begin_index␣
↪+ t)]
            inv_cov = np.linalg.inv(np.cov(used_returns.to_numpy().T)) #␣
↪Transpose so the covariance is calculated for the stocks instead of the dates
            mu = used_returns.mean().values

            # Add row to weights matrix using the provided weights function
            weights[t] = weights_function(inv_cov, mu)

        # Return the performance
```

```python
        return self.get_performance(weights, in_sample_fraction, stop_fraction,␣
↪return_returns)

    # Used for question d)
    def tangency_weights(self, inv_cov, mu):
        '''Method that calculates the tangency weights using the inverse sample␣
↪covariance matrix and mu'''
        return inv_cov @ mu / (np.ones(len(self.betas)).T @ inv_cov @ mu)

    def get_tangency_performance(self, estimation_window, in_sample_fraction,␣
↪stop_fraction = 1, return_returns=False):
        '''Method that gets the Sharpe ratio and turnover of a tangency␣
↪portfolio for a given in_sample_fraction of observations'''

        return self.get_general_performance(self.tangency_weights,␣
↪estimation_window, in_sample_fraction, stop_fraction, return_returns)

    # Used for question e)
    def get_tangency_performance_is(self, begin_fraction, end_fraction):
        '''Methd that obtains the in sample performance of a tangency␣
↪portfolio'''

        # Set begin and end indices
        begin_index = math.floor(self.n_observations * begin_fraction)
        end_index = math.floor(self.n_observations * end_fraction)

        # Calculate the weights that are used
        used_returns = self[begin_index:end_index]
        inv_cov = np.linalg.inv(np.cov(used_returns.to_numpy().T)) # Transpose␣
↪so the covariance is calculated for the stocks instead of the dates
        mu = used_returns.mean().values

        weights_row = inv_cov @ mu / (np.ones(len(self.betas)).T @ inv_cov @ mu)
        weights = np.vstack([weights_row] * (end_index - begin_index))

        # Return the performance
        return self.get_performance(weights, begin_fraction, end_fraction)

    # Used for question f)
    def unconstrained_mv_weights(self, inv_cov, _):
        '''Method that calculates the unconstrained minimum variance weights␣
↪using the inverse sample covariance matrix and mu'''
        iota = np.ones(len(self.betas))

        return inv_cov @ iota / (iota.T @ inv_cov @ iota)
```

```
    def get_unconstrained_mv_performance(self, estimation_window,␣
↪in_sample_fraction, stop_fraction = 1, return_returns=False):
        '''Method that gets the Sharpe ratio and turnover of a tangency␣
↪portfolio for a given in_sample_fraction of observations'''

        return self.get_general_performance(self.unconstrained_mv_weights,␣
↪estimation_window, in_sample_fraction, stop_fraction, return_returns)

    # Used for question g)
    def constrained_mv_weights(self, inv_cov, mu):
        '''Method that calculates the constrained minimum variance weights using␣
↪the inverse sample covariance matrix and mu'''

        # First calculate unconstrained weights
        unconstrained_weights = self.unconstrained_mv_weights(inv_cov, mu)

        # Force the negative weights to be zero
        b = np.array([1 if weight > 0 else 0 for weight in␣
↪unconstrained_weights])

        return inv_cov @ b / (b.T @ inv_cov @ b)

    def get_constrained_mv_performance(self, estimation_window,␣
↪in_sample_fraction, stop_fraction = 1, return_returns=False):
        '''Method that gets the Sharpe ratio and turnover of a tangency␣
↪portfolio for a given in_sample_fraction of observations'''

        return self.get_general_performance(self.constrained_mv_weights,␣
↪estimation_window, in_sample_fraction, stop_fraction, return_returns)

    # Used for question h)
    def get_oc_weights(self, inv_cov, mu):
        '''Method that calculates the optimal constrained weights using the␣
↪inverse sample covariance matrix and mu'''

        # Calculate the weights of relevant portfolios
        weights_1oN = np.ones(len(mu)) / len(mu)
        weights_mv = self.unconstrained_mv_weights(inv_cov, mu)
        weights_tan = self.tangency_weights(inv_cov, mu)

        # Caluclate the implied target returns
        mu_1oN = mu.T @ weights_1oN
        mu_mv = mu.T @ weights_mv
        mu_tan = mu.T @ weights_tan

        # Calculate and return the weighted return
```

```python
        weight = (mu_1oN - mu_mv) / (mu_tan - mu_mv)

        return weight * weights_tan + weights_mv * (1 - weight)

    def get_oc_performance(self, estimation_window, in_sample_fraction,␣
↪stop_fraction = 1, return_returns=False):
        '''Method that gets the Sharpe ratio and turnover of a tangency␣
↪portfolio for a given in_sample_fraction of observations'''

        return self.get_general_performance(self.get_oc_weights,␣
↪estimation_window, in_sample_fraction, stop_fraction, return_returns)

    # Used for a lot of questions
    def get_performance(self, weights, in_sample_fraction, stop_fraction,␣
↪return_returns=False):
        '''Method that obtains the Sharpe ratio and turnover of a given set of␣
↪weights and in sample fraction of observations'''

        # Set the index where the out of sample period begins
        begin_index = math.floor(self.n_observations * in_sample_fraction)
        stop_index = math.floor(self.n_observations * stop_fraction)

        if len(weights) != stop_index - begin_index:
            raise ValueError("The weights do not have the correct length")

        # Get an array of returns at each time
        returns = np.array([])

        for i in range(len(self[begin_index:stop_index])):

            row=self.iloc[i]
            row_array = row.to_numpy()

            returns = np.append(returns, row_array @ weights[i].T) # Add the␣
↪return

        # Calculate the sharpe ratio
        sharpe_ratio = returns.mean() / returns.std()

        # Calculate the turnover

        ## Calculate the portfolio weights at the end of the period
        weights_end = weights * (1 + self[begin_index:stop_index].to_numpy())

        ## Calculate the total portfolio value at the end of the period
        total_portfolio_value = np.sum(weights_end, axis=1)
```

```
        ## Normalize the portfolio weights to ensure they sum up to 1 at the end␣
↪of the period
        weights_end_normalized = weights_end / total_portfolio_value[:, np.
↪newaxis]

        ## get turnover
        turnover = np.sum(np.abs(weights_end_normalized[:-1] - weights[1:])) /␣
↪len(weights)

        if not return_returns:
            return sharpe_ratio, turnover
        else:
            return sharpe_ratio, turnover, returns
```

Simulate

[ ]:
```
# Parameters to use
std_params = std_params_ff

# Callables used to draw the betas and factors
beta_drawer = make_drawer(gamma.rvs, *beta_params)
std_drawer = make_drawer(lognorm.rvs, *std_params)
factor_drawer = make_drawer(np.random.normal, mkt_mean, mkt_std)
```

[ ]:
```
# Set seed
np.random.seed(1913)

# Simulate
n_observations = 20000
N = (10, 100)

simulations = dict()

for n in N:
    simulator = MarketSimulator(beta_drawer, std_drawer, factor_drawer)
    simulations[n] = simulator.simulate(n_assets = n, n_observations =␣
↪n_observations)
```

[ ]:
```
# Show the simulated MarketSimulation
simulations[10]
```

1.2.6 Question c)

[ ]:
```
simulations[10].get_equally_weighted_performance(0.6)
```

1.2.7 Question d)

[ ]:
```
simulations[10].get_tangency_performance(120, 0.6)
```

Question e)

[ ]:
```
simulations[10].get_tangency_performance_is(0, 0.6)
```

1.2.8 Question f)

[ ]:
```
simulations[10].get_unconstrained_mv_performance(120, 0.6)
```

1.2.9 Question g)

[ ]:
```
simulations[10].get_constrained_mv_performance(120, 0.6)
```

1.2.10 Question h)

[ ]:
```
simulations[10].get_oc_performance(120, 0.6)
```

1.2.11 Tables

[ ]:
```
# Sharpe and turnover table (Can take a long time)

N = [10, 100]
M = [120, 240, 3600]
split_ratio = 0.6
portfolios = ['1/N', 'Tangency out of sample', 'Tangency in sample',␣
↪'Unconstrained minmum variance', 'Constrained minimum variance', 'Optimal␣
↪constrained']

# Set up Dataframes with Sharpe ratios and turnovers
sharpe_df = pd.DataFrame(columns=[str(m) for m in M])
sharpe_df['portfolio'] = [f'{portfolio}_{n}' for portfolio in portfolios for n␣
↪in N]
sharpe_df.set_index('portfolio', inplace=True, drop=True)

turnover_df = copy.copy(sharpe_df)

for n in N:

    for portfolio in [portfolios[4]]:

        print(f'Now at n: {n}, portfolio: {portfolio}')

        if portfolio == '1/N':
            sharpe_turnover = {str(m): simulations[n].
↪get_equally_weighted_performance(split_ratio) for m in M}
```

```
        elif portfolio == 'Tangency out of sample':

            sharpe_turnover = {str(m): simulations[n].
↪get_tangency_performance(m, split_ratio) for m in M}

        elif portfolio == 'Tangency in sample':

            sharpe_turnover = {str(m): simulations[n].
↪get_tangency_performance_is(0, split_ratio) for m in M}

        elif portfolio == 'Unconstrained minmum variance':

            sharpe_turnover = {str(m): simulations[n].
↪get_unconstrained_mv_performance(m, split_ratio) for m in M}

        elif portfolio == 'Constrained minimum variance':

            sharpe_turnover = {str(m): simulations[n].
↪get_constrained_mv_performance(m, split_ratio) for m in M}

        elif portfolio == 'Optimal constrained':

            sharpe_turnover = {str(m): simulations[n].get_oc_performance(m,␣
↪split_ratio) for m in M}

        sharpe_row = {str(m): sharpe_turnover[str(m)][0] for m in M}
        turnover_row = {str(m): sharpe_turnover[str(m)][1] for m in M}

        # Change row
        sharpe_df.loc[f'{portfolio}_{n}'] = sharpe_row
        turnover_df.loc[f'{portfolio}_{n}'] = turnover_row
```

[ ]:
```
# Show the dataframes if you want to
turnover_df
```

1.2.12 Question k)

[ ]:
```
# Make excel file to use as Matlab input for the Sharpe significance tests

returns1a = simulations[10].get_oc_performance(120, 0.6, return_returns =␣
↪True)[2]
returns1b = simulations[10].get_equally_weighted_performance(0.6, return_returns␣
↪= True)[2]
returns2a = simulations[10].get_oc_performance(240, 0.6, return_returns =␣
↪True)[2]
```

```
returns2b = simulations[10].get_equally_weighted_performance(0.6, return_returns␣
 ↪= True)[2]
returns3a = simulations[10].get_oc_performance(3600, 0.6, return_returns =␣
 ↪True)[2]
returns3b = simulations[10].get_equally_weighted_performance(0.6, return_returns␣
 ↪= True)[2]

returns4a = simulations[100].get_oc_performance(120, 0.6, return_returns =␣
 ↪True)[2]
returns4b = simulations[100].get_equally_weighted_performance(0.6,␣
 ↪return_returns = True)[2]
returns5a = simulations[100].get_oc_performance(240, 0.6, return_returns =␣
 ↪True)[2]
returns5b = simulations[100].get_equally_weighted_performance(0.6,␣
 ↪return_returns = True)[2]
returns6a = simulations[100].get_oc_performance(3600, 0.6, return_returns =␣
 ↪True)[2]
returns6b = simulations[100].get_equally_weighted_performance(0.6,␣
 ↪return_returns = True)[2]
df = pd.DataFrame({
    'returns1a': returns1a,
    'returns1b': returns1b,
    'returns2a': returns2a,
    'returns2b': returns2b,
    'returns3a': returns3a,
    'returns3b': returns3b,
    'returns4a': returns4a,
    'returns4b': returns4b,
    'returns5a': returns5a,
    'returns5b': returns5b,
    'returns6a': returns6a,
    'returns6b': returns6b
})
```

[ ]:
```
# Export to Excel
df.to_excel('output_file.xlsx', index=False)
```