Does it pay to be active?
Norwegian mutual fund performance from 1991 to 2019

Knut Mehl
MSc in Business with Finance - QTEM

Henrik Reitan
MSc in Business with Finance - QTEM

Codebook

An attachment to our Master's thesis

Supervisor: Bruno Gerard

Department of Finance

BI Norwegian Business School

Spring 2020

Table of Contents

Introduction

BI Norwegian Business School uses DigEx for submission of the Master's thesis. The system limits the attachments to three files and do not allow Python files. Our data manipulations consists of ~10 Python files, 2 Matlab files, and ~10 Excel files which cannot be delivered in a sensible way in DigiEx. We therefore merge and report all Python and Matlab codes in this document and refer to the link below for our full data library.

Click here to access our data library.

Please reach out to knut_mehl@live.no and henrik.a.reitan@gmail.com if the link is not working.

Overview of figures and tables

The following table shows which file(s) we use to create the figures and tables in our thesis.

Figure / Table	Code file
Figure 3.1	02_empirical methods_v0.3.py
Figure 3.2	02_empirical methods_v0.3.py
Table 4.1	N/A
Table 4.2	03_data_v2.4.py
Figure 4.1	03_data_v2.4.py
Table 4.3	03_data_v2.4.py
Table 4.4	03_data_v2.4.py
Table 4.5	03_data_v2.4.py
Table 4.6	03_data_v2.4.py
Table 4.7	03_data_v2.4.py
Table 4.8	03_data_v2.4.py
Figure 4.2	03_data_v2.4.py
Figure 4.3	03_data_v2.4.py
Table 5.1	04_traditional_v4.0.py
Table 5.2	04_traditional_v4.0.py
Table 5.3	04_traditional_v4.0.py
Figure 5.1	04_additional_charts_v1.1.py
Figure 5.2	04_additional_charts_v1.1.py
Figure 5.3	04_additional_figure03_v2.1.py
Table 5.4	05_SSD_test_v1.0.py
Table 5.5	04_simulation_A_v1.1.py
Figure 5.4	04_simulation_A_v1.1.py
Table 5.6	04_simulation_B_v1.2.py
Figure 5.4	04_simulation_B_v1.2.py
Appendix I	None
Appendix II	None
Appendix III	03_data_v2.4.py
Appendix IV	03_data_v2.4.py
Appendix V	03_data_v2.4.py
Appendix VI	04_traditional_v4.0.py
Appendix VII	04_additional_charts_v1.1.py
Appendix VIII	06_SSD_Crane&Crotty_v0.1.py and m06_SSD_CraneCrotty_v0_4
Appendix IX	04_simulation_A_v1.1.py
Appendix X	04_simulation_B_v1.2.py
Appendix A	None
Appendix B	None
Appendix C	None
Appendix D	05_OtherMktRet_v2.0.py
Appendix E	None
Appendix F	None
Appendix G	m05_SD_matlab_v3_1.m
Appendix H	04_simulation_B_inc_fees_v1.2.py

Code files

01_cleaning v6.8.py

```
#%% Self-created functions

# Clear all variables from variable explorer
def clear_all():
    gl = globals().copy()
    for var in gl:
        if var[0] == '_': continue
        if 'func' in str(globals()[var]): continue
        if 'module' in str(globals()[var]): continue

        del globals()[var]

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Clear workspace
clear_all()

#%% Import packages and set global variables

if True: # If statement so section can be folded

    # Import packages
    import pandas as pd
    import os as os
    import sys
    import math
    import numpy as np
    import time
    import datetime
    import calendar
    import MScThesis_v4 as msc
    import re

    # Global variables
    file_name =  os.path.basename(sys.argv[0])  # Store name of this file
    file_path =  os.path.realpath(file_name)    # Store filepath of this file
    file_path =  file_path.strip(file_name)
    imp_path  =  file_path + "/01_uncleaned_data"
    out_path  =  file_path + "/01_cleaning_output"

    # Global settings
    ExportMode = False # If this is set to true, the codes will OVERWRITE all output tables

    # Last version of "MS_Overview of funds_v*.*.xlsx"
    MS_version = "v6.5"

#%% Import files
if True: # If statement so section can be folded

    # Create overview file
    colNames = ["Source"]
    fileOverview = pd.DataFrame(columns=colNames)

    # File 00 - Fund info Oslo Børs
    imp_name = file_path + "/01_cleaning_input/MS_Overview of funds_" + MS_version + ".xlsx"
    funds_OSE = pd.read_excel(imp_name, sheet_name='python_OB')
    fileOverview.loc['funds_OB', 'Source'] = "Oslo Stock Exchange - email"

    # File 01 - Fund info Morningstar
    imp_name = file_path + "/01_cleaning_input/MS_Overview of funds_" + MS_version + ".xlsx"
    funds_MS = pd.read_excel(imp_name, sheet_name='python_MS')
    fileOverview.loc['funds_MS', 'Source'] = "Morningstar Direct"

    # File 02 - MS Net Returns
    imp_name = imp_path + "/00 MS data/02 MS monthly net returns_NOK.xlsx"
    ret_raw_MS = pd.read_excel(imp_name)
    fileOverview.loc['ret_raw_MS', 'Source'] = "Morningstar Direct"

    # File 03 - OBI Net returns
    imp_name = imp_path + "/01 Returns_Other sources/monthly_fund_returns_ose_inc2019.csv"
    ret_raw_OBI = pd.read_csv(imp_name, sep='delimiter', header=None, engine='python')
    # Note: Falling back to the 'python' engine because the 'c' engine does not support regex 
separators (separators > 1 char and different from '\s+' are interpreted as regex); you can 
avoid this warning by specifying engine='python'.
    fileOverview.loc['ret_raw_OBI', 'Source'] = "Bernt Ødegaard/OBI - large data"

    # File 04 - Market returns #1
    imp_name = imp_path + "/02 Market Returns/02 Market returns_v3.xlsx"
    mkt_raw = pd.read_excel(imp_name, sheet_name='python_mkt')
    fileOverview.loc['mkt_raw', 'Source'] = "Bernt Ødegaard/OBI - small data & Bernt 
Ødegaard/Mkt returns"

    # File 05 - Risk free
    imp_name = imp_path + "/03 Risk free 1month_Ødegaard_v2.xlsx"
    rf_raw = pd.read_excel(imp_name)
    fileOverview.loc['rf_raw', 'Source'] = "Bernt Ødegaard"

    # File 06 - Factors
    imp_name = imp_path + "/04 Factor Data Norwegian Equities_v2.xlsx"
    factors_raw = pd.read_excel(imp_name)
    fileOverview.loc['factors_raw', 'Source'] = "Bernt Ødegaard"

    # File 07 - NAV
    imp_name = imp_path + "/00 MS data/04 MS Net Assets - share class (Monthly).xlsx"
    NAV_raw_MS = pd.read_excel(imp_name)
    fileOverview.loc['NAV_raw_MS', 'Source'] = "Morningstar Direct"

    # File 08 - Minimum investment
    imp_name = "01_cleaning_input/MS_Overview of min investment_v0.1.xlsx"
    data_min_investment = pd.read_excel(imp_name, sheet_name='Output')
    fileOverview.loc['data_min_investment', 'Source'] = "Morningstar Direct"

#%% Clear variables not to be used later
clear_variables(["colNames", "imp_name", 'file_name'])

#%% Create overview dataset
if True: # If statement so section can be folded

    # Create main dataset and copy OB funds to it
    data_overview = funds_OSE[['Symbol', 'Security name', 'ISIN', 'Index fund', \
                               'Include', 'Extract data from', 'Short tag']].copy()
    data_overview['Source'] = 'Oslo Stock Exchange Information'
    data_overview.insert(1,'MS_SecID', 'nan')

    # Create temporary copy of MS funds so we may clean the data without changing the original 
dataframe
    tmp_funds_MS = funds_MS.copy()

    # Append MS funds to main dataset
    for index in tmp_funds_MS.index:
        # If statement to only append funds we want to append
        if tmp_funds_MS.loc[index,'Oslo Børs - Aktive og delistede'] == False:
            data_overview = data_overview.append({ \
                'Symbol': 'nan', \
                'Security name' : tmp_funds_MS.loc[index, 'Name'],\
                'ISIN' : tmp_funds_MS.loc[index, 'ISIN'],\
                'Include' : tmp_funds_MS.loc[index, 'Include'],\
                'Extract data from' : "ISIN exists in MS data",\
                'Short tag' : "MS_ISIN",\
                'Source' : "Morningstar Direct",\
                'MS_SecID' : tmp_funds_MS.loc[index, 'SecId'],\
                } , ignore_index=True)


    # Set "Index fund" column to boolean
    for index in data_overview.index:

        # Populate "Index fund" column for MS funds
        if data_overview.loc[index,'Source'] == "Morningstar Direct":
            indexMS = list(tmp_funds_MS['ISIN']).index(data_overview.loc[index,'ISIN'])
            if tmp_funds_MS.loc[indexMS, 'Index Fund'] == "Yes":
                data_overview.loc[index,'Index fund']= True
            elif tmp_funds_MS.loc[indexMS, 'Index Fund'] == "No":
                data_overview.loc[index,'Index fund']= False

        # Convert "Index fund" column for OSE funds
        if data_overview.loc[index,'Source'] == "Oslo Stock Exchange Information":
            if data_overview.loc[index,'Index fund'] == "Yes":
                data_overview.loc[index,'Index fund'] = True
            elif data_overview.loc[index,'Index fund'] == "No":
                data_overview.loc[index,'Index fund'] = False

    # Set missing ISINs to nan
    for index in data_overview.index:
        if data_overview.loc[index,'ISIN'] == 0:
            data_overview.loc[index,'ISIN'] = 'nan'

    # Fill missing MS_SecID for MS funds
    for index in data_overview.index:
        if data_overview.loc[index,'Short tag'] == "MS_ISIN" and
data_overview.loc[index,'MS_SecID'] == 'nan':
            data_overview.loc[index,'MS_SecID'] = list(funds_MS.loc[funds_MS['ISIN'] ==
data_overview.loc[index,'ISIN'] , 'SecId'])[0]

    # Rename columns
    data_overview = data_overview.rename(columns={'Symbol':'OSE_Ticker', 'Security name':'Fund
name'})

    # Add column with unique names (which also works as unique identifiers)
    data_overview.insert(3, 'Unique fund name', 'nan')
    for index in data_overview.index:   # Loop through data_overview and fill unique name
        temp_fund_name = data_overview.loc[index, 'Fund name']
        temp_counter = list(data_overview.loc[0:index, 'Fund name']).count(temp_fund_name)
        if temp_counter == 1:
            data_overview.loc[index, 'Unique fund name'] = data_overview.loc[index, 'Fund
name']
        else:
            data_overview.loc[index, 'Unique fund name'] = data_overview.loc[index, 'Fund
name'] + "__" + str(temp_counter)

#%% Clear variables not to be used later
clear_variables(["index", "indexMS", "tmp_funds_MS", "temp_fund_name", "temp_counter"])

#%% Clean OBI return data
'''This section uses around 60 seconds'''
start_time = time.time() # Time section
colNames = ['OBI_ID', 'ISIN', 'Ticker', 'Fund_name', 'Date', 'MonthlyReturn']

temp_list = []

for i in range(ret_raw_OBI.shape[0]):
    temp_list.append(ret_raw_OBI[0][i].split(";"))

ret_clean_OBI = pd.DataFrame(temp_list,columns=colNames)
ret_clean_OBI = ret_clean_OBI[1:]                        # Remove first row with headers
ret_clean_OBI.reset_index(inplace = True, drop = True)  # Reset index (because first row was
removed)

# Remove duplicates
ret_clean_OBI['uniqueID'] = ret_clean_OBI['Ticker']+ret_clean_OBI['Date']
ret_clean_OBI = ret_clean_OBI.drop_duplicates(subset='uniqueID', keep='first')

# Drop redundant columns
ret_clean_OBI = ret_clean_OBI.drop(['OBI_ID', 'uniqueID'], axis=1)

# Rename columns
ret_clean_OBI = ret_clean_OBI.rename(columns={'Ticker':'OSE_Ticker', 'Fund_name':'Fund name',
'Date':'Month', 'MonthlyReturn':'Return'})

timer_00 = str(datetime.timedelta(seconds=(time.time()-start_time)))    # End timer

#%% Clear variables not to be used later
clear_variables(["temp_list", "colNames", "i", "start_time", "timer_00"])

#%% Clean MS Returns
ret_clean_MS = ret_raw_MS.copy()
ret_clean_MS = ret_clean_MS.set_index('SecId')
ret_clean_MS.drop(ret_clean_MS.iloc[:, 0:12], inplace=True, axis=1)
```

```
# Remove all non-numeric characters from column names
[N_rows, N_cols] = ret_clean_MS.shape   # Get size of dataframe
newColNames = []                        # List with cleaned column names
for i in range(N_cols):                 # Loop to append clean column names to list
    newColNames.append(re.sub("[^0-9]", "", ret_clean_MS.columns[i]))
ret_clean_MS.columns = newColNames      # Set new column names

#%% Clear variables not to be used later
clear_variables(["N_rows", "N_cols", "newColNames", "i"])

#%% Create large dataset (i.e. data per fundmonth)
'''This section uses around 8 seconds'''
start_time = time.time() # Time section

# Create dataframe
colNames = ["OSE_Ticker", "MS_SecID", "Fund name", "Unique fund name", "ISIN", "Index fund",
"Month", "Return"]
data_fundmonths = pd.DataFrame(columns=colNames)

# Populate return data for OSE funds
for index in data_overview.index: # For all funds
# for index in range(25):           # For temporary testing
     if data_overview.loc[index,'Short tag'] == "OBI_large_Ticker" and
data_overview.loc[index,'Include'] == True:
         temp_ticker = data_overview.loc[index,'OSE_Ticker']
         temp_indexFund = data_overview.loc[index,'Index fund']
         temp_ret_OBI = ret_clean_OBI[ret_clean_OBI['OSE_Ticker']==temp_ticker].copy()
         temp_ret_OBI['Index fund'] = data_overview.loc[index,'Index fund']
         temp_ret_OBI['Unique fund name'] = data_overview.loc[index,'Unique fund name']
         data_fundmonths = data_fundmonths.append(temp_ret_OBI, ignore_index=True)  # Append
to data_fundmonths

# Populate return data for MS funds
for index in data_overview.index: # For all funds
# for index in range(158,163):      # For temporary testing
    if data_overview.loc[index,'Short tag'] == "MS_ISIN" and
data_overview.loc[index,'Include'] == True:
        MS_secID = data_overview.loc[index,'MS_SecID']
        MS_date_inception = list(funds_MS.loc[funds_MS['SecId'] == MS_secID,'Inception
Date'])[0] # Get the funds inception date
        temp_ret_MS = ret_clean_MS.copy().loc[MS_secID].dropna().to_frame()
        temp_ret_MS.columns = ['Return']
        temp_ret_MS['Return'] = temp_ret_MS['Return']/100 # Convert returns to percentage
        temp_ret_MS['MS_SecID'] = MS_secID
        temp_ret_MS['Fund name'] =
list(data_overview.loc[data_overview['MS_SecID']==MS_secID]['Fund name'])[0]
        temp_ret_MS['Unique fund name'] = data_overview.loc[index,'Unique fund name']
        temp_ret_MS['ISIN'] =
list(data_overview.loc[data_overview['MS_SecID']==MS_secID]['ISIN'])[0]
        temp_ret_MS['Index fund'] =
list(data_overview.loc[data_overview['MS_SecID']==MS_secID]['Index fund'])[0]
        temp_ret_MS['Month'] = temp_ret_MS.index
        temp_ret_MS['Month_value'] = 'nan'
        for i in range(len(list(temp_ret_MS['Month']))):  # Set months to correct format
            temp_month = temp_ret_MS['Month'][i]
            temp_y = int(temp_month[0:4])
            temp_m = int(temp_month[4:6])
            temp_d = calendar.monthrange(temp_y, temp_m)[1]
            temp_ret_MS['Month'][i] = str(temp_y) + str(temp_m).zfill(2) + str(temp_d)
            temp_ret_MS['Month_value'][i] = datetime.datetime(temp_y,temp_m,temp_d)

        temp_ret_MS = temp_ret_MS[temp_ret_MS['Month_value']>MS_date_inception] # Filter out
returns from before inception date
        del temp_ret_MS['Month_value']                                          # Remove
helping column

        data_fundmonths = data_fundmonths.append(temp_ret_MS, ignore_index=True)  # Append to
data_fundmonths

timer_01 = str(datetime.timedelta(seconds=(time.time()-start_time)))    # End timer

#%% Clear variables not to be used later
clear_variables(["start_time", "colNames", "index", "temp_ticker", "temp_indexFund",
"temp_ret_OBI"])
clear_variables(["MS_secID", "MS_date_inception", "temp_ret_MS", "i", "temp_month", "temp_y",
"temp_m"])
clear_variables(["temp_d", "timer_01"])
```

```
#%% Fill additional variables (except NAV)

# Convert months in data_fundmonths to numeric
data_fundmonths['Month'] = pd.to_numeric(data_fundmonths['Month'])

# Fill market returns
del mkt_raw['Source']
del mkt_raw['Index used']
data_fundmonths = data_fundmonths.join(mkt_raw.set_index('date'), on='Month')

# Clean and fill rf
rf_raw.columns = rf_raw.iloc[0]
rf_raw = rf_raw[1:]
rf_raw = rf_raw.rename(columns={'Rf(1m)':'Rf'})
data_fundmonths = data_fundmonths.join(rf_raw.set_index('date'), on='Month')

# Fill factor returns from Ødegaard
data_fundmonths = data_fundmonths.join(factors_raw.set_index('date'), on='Month')

# Convert columns to numeric
data_fundmonths['Rf'] = pd.to_numeric(data_fundmonths['Rf'])
data_fundmonths['Return'] = pd.to_numeric(data_fundmonths['Return'])

# Fix fund names
data_fundmonths['Fund name'] = data_fundmonths['Fund name'].str.replace('"','')

#%% Clean MS NAV
NAV_clean_MS = NAV_raw_MS.copy()
NAV_clean_MS = NAV_clean_MS.set_index('SecId')
NAV_clean_MS.drop(NAV_clean_MS.iloc[:, 0:12], inplace=True, axis=1)

# Remove all non-numeric characters from column names
[N_rows, N_cols] = NAV_clean_MS.shape   # Get size of dataframe
newColNames = []                        # List with cleaned column names
for i in range(N_cols):                 # Loop to append clean column names to list
    newColNames.append(re.sub("[^0-9]", "", NAV_clean_MS.columns[i]))
NAV_clean_MS.columns = newColNames      # Set new column names

# Remove obvious data errors
np.array(1*(NAV_clean_MS < 2000)).sum() # 1048 observations are below 2000 NOK
NAV_clean_MS[NAV_clean_MS < 2000] = np.nan

#%% Fill MS NAV

# Create NAV df on fundmonth format
NAV_clean_MS_fundmonths = pd.DataFrame(columns=['MS_SecID', 'Month', 'NAV'])
for index in data_overview.index: # For all funds
# for index in range(158,163):    # For temporary testing
    if data_overview.loc[index,'Short tag'] == "MS_ISIN" and
data_overview.loc[index,'Include'] == True:
        MS_secID = data_overview.loc[index,'MS_SecID']
        temp_NAV_MS = NAV_clean_MS.copy().loc[MS_secID].dropna().to_frame()
        temp_NAV_MS.columns = ['NAV']
        temp_NAV_MS['MS_SecID'] = MS_secID
        temp_NAV_MS['Month'] = temp_NAV_MS.index
        for i in range(len(list(temp_NAV_MS['Month']))):  # Set months to correct format
            temp_month = temp_NAV_MS['Month'][i]
            temp_y = int(temp_month[0:4])
            temp_m = int(temp_month[4:6])
            temp_d = calendar.monthrange(temp_y, temp_m)[1]
            temp_NAV_MS['Month'][i] = str(temp_y) + str(temp_m).zfill(2) + str(temp_d)
        NAV_clean_MS_fundmonths = NAV_clean_MS_fundmonths.append(temp_NAV_MS,
ignore_index=True)

# Convert Month column to numeric
NAV_clean_MS_fundmonths['Month'] = pd.to_numeric(NAV_clean_MS_fundmonths['Month'])

# Move NAVs to main fundmonth dataset
data_fundmonths = pd.merge(data_fundmonths, NAV_clean_MS_fundmonths, how='left', \
                     left_on=['MS_SecID', 'Month'], right_on=['MS_SecID', 'Month'])

# Here, we need to conduct the following clean as the below-mentioned funds share secID
    # (which make them duplicated in the data; two entries per fundmonth)
    # The duplicates are created by the codes in this section.
    # Funds:
        # F0000110TN; DNB Norge Selektiv (III) R; DNB Norge Selektiv R
        # F00000ZFFZ; DNB SMB R, DNB SMB R__2
data_fundmonths = data_fundmonths.drop_duplicates()

#%% Interpolate NAV data

# Check NAV data quality
len(list(data_fundmonths['NAV']))                                       # OUTDATED: 22 838
observations in total
len(list(data_fundmonths['NAV']))-sum(1*data_fundmonths['NAV'].isna())  # OUTDATED: 13 788 obs
have NAV data (before interpolation)

# Create dfs in df
data_dfs = msc.create_dfs(data_overview, data_fundmonths, 'skip')

# Loop through funds
for fund_name in data_dfs.keys():

    # Get fund data
    fund_data = data_dfs[fund_name]

    # If fund has NAV data
    if sum(fund_data['NAV']>0)!=0:

        # Interpolate
        fund_data['NAV']=fund_data['NAV'].interpolate(method ='linear', limit_direction
='forward')

# Move dfs back to data_fundmonths
data_fundmonths = msc.dfs_to_fundmonths(data_dfs)

len(list(data_fundmonths['NAV']))-sum(1*data_fundmonths['NAV'].isna()) # OUTDATED: 15 171 obs
have NAV data (after interpolation)

#%% Clear variables not to be used later
clear_variables(["N_rows", "N_cols", "newColNames", "i", "NAV_clean_MS_fundmonths", "index"])
clear_variables(["MS_secID", "temp_NAV_MS", "temp_month", "temp_y", "temp_m", "temp_d"])
clear_variables(["fund_name"])

#%% Create summary of returns and NAV in data_overview
data_overview = msc.summarize_fundmonths(data_overview, data_fundmonths)

#%% Clean fundmonths with returns = 0 (total of 71 fundmonths)

# Create dfs with zero returns
data_dfs_zeroRet = msc.create_dfs(data_overview,
data_fundmonths[data_fundmonths['Return']==0], 0)

# Fokus Barnespar (6 returns)
delName = 'Fokus Barnespar'
delRets = [np.int64(20000430), np.int64(20000531), np.int64(20000630), np.int64(20000731),
np.int64(20000831), np.int64(20000930)]
for i in range(len(delRets)):
    data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']!=delRets[i]) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# DNB Norge (I) (5 returns)
delName = 'DNB Norge (I)'
delRets = [np.int64(19850430), np.int64(19850531), np.int64(19850630), np.int64(19850731),
np.int64(19850831)]
for i in range(len(delRets)):
    data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']!=delRets[i]) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Nordea Avkastning (29 returns)
delName = 'Nordea Avkastning'
delBefore = np.int64(19830801)
data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']>delBefore) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Nordea Vekst (28 returns)
delName = 'Nordea Vekst'
delBefore = np.int64(19830701)
```

```
data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']>delBefore) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Danske Invest Aktiv Formuesf. A (2 returns)
delName = 'Danske Invest Aktiv Formuesf. A'
delRets = [np.int64(20060630), np.int64(20060731)]
for i in range(len(delRets)):
    data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']!=delRets[i]) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Nordea Kapital (1 return)
delName = 'Nordea Kapital'
delRet = np.int64(19950228)
data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']!=delRet) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Create dfs with zero returns (to verify deletion process)
data_dfs_zeroRet2 = msc.create_dfs(data_overview,
data_fundmonths[data_fundmonths['Return']==0], 0)

#%% Clear variables not to be used later
clear_variables(["data_dfs_zeroRet", "data_dfs_zeroRet2", "delName", "delRets", "i",
"delBefore", "delRet"])

#%% Clean Atlas Norge / Vibrand Norden which has been a Norwegian fund, but had its investment
profile changed

# Vibrand Norden (delete 15 returns)
delName = 'Vibrand Norden'
delAfter = np.int64(20171001)
data_fundmonths = data_fundmonths[ ((data_fundmonths['Month']<delAfter) &
(data_fundmonths['Unique fund name']==delName)) |  (data_fundmonths['Unique fund
name']!=delName)  ]

# Vibrand Norden (rename to Atlas Norge)
newName = 'Atlas Norge'
OSE_Ticker = 'NR-NORGE'
data_overview.loc[data_overview['OSE_Ticker']==OSE_Ticker,'Fund name'] = newName
data_overview.loc[data_overview['OSE_Ticker']==OSE_Ticker,'Unique fund name'] = newName
data_fundmonths.loc[data_fundmonths['OSE_Ticker']==OSE_Ticker,'Fund name'] = newName
data_fundmonths.loc[data_fundmonths['OSE_Ticker']==OSE_Ticker,'Unique fund name'] = newName

#%% Clear variables not to be used later
clear_variables(["delName", "delAfter", "newName", "OSE_Ticker"])

#%% Count number of missing values per regression variable
data_missing_values = msc.count_missing_values(data_fundmonths)

#%% Add minimum investment variable

# Get relevant columns
data_min_investment = data_min_investment[['Unique fund name', 'Min investment']]

# Merge minimum investment variable into datasets
data_overview = pd.merge(data_overview, data_min_investment, how='left', on=['Unique fund
name'])
data_fundmonths = pd.merge(data_fundmonths, data_min_investment, how='left', on=['Unique fund
name'])

#%% Remove excluded funds from Data Overview
data_overview = data_overview[data_overview['Include']==True]

#%% Dealing with incubation bias (305 observations)

data_fundmonths_new2 = pd.DataFrame(columns=data_fundmonths.columns)

for fund in list(data_overview['Unique fund name']):

    # Get fund data
    data_fm_temp = data_fundmonths [data_fundmonths['Unique fund name']==fund].copy()

    # Create new column with month on datetime format
    data_fm_temp['Month_format'] = pd.to_datetime(data_fm_temp['Month'].astype(str),
format='%Y%m%d')
```

```
    # Get the funds ISIN (skip to next if no data for the fund)
    try:
        temp_ISIN = list(data_fm_temp['ISIN'])[0]
    except:
        continue

    # Skip if ISIN is not in OSE dataset (i.e. we do not have listed date)
    if sum(1*(funds_OSE['ISIN'] == temp_ISIN)) == 0:
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp)
        continue

    # If ISIN is 'nan', keep all (all clases with incubation problems have ISIN; verified
manually in Excel)
    if temp_ISIN=='':
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp)
    else:

        # Get the funds listing date
        temp_listed_from = list(funds_OSE[funds_OSE['ISIN']==temp_ISIN]['List member
from'])[0]

        # Create indicator for returns to be kept
        data_fm_temp['inc_bias_keep'] = (data_fm_temp['Month_format'] > temp_listed_from)

        # Get data to be kept
        data_fm_temp_keep = data_fm_temp[data_fm_temp['inc_bias_keep']]

        # Store data
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp_keep)

# Delete temporary columns
del data_fundmonths_new2['Month_format']
del data_fundmonths_new2['inc_bias_keep']

# Update main dataset
data_fundmonths = data_fundmonths_new2.copy()
del data_fundmonths_new2

#%% Mitigating the "share class problem" (i.e. keeping only one class per fund)

# Rename current return and NAV column (works as a failsafe after we fixed the share class
problem)
data_fundmonths = data_fundmonths.rename(columns={'Return': 'Return_Raw', 'NAV': 'NAV_Raw'})

# Import overview of fund classes
imp_name = "01_cleaning_input/Overview of fund classes_v0.1.xlsx"
data_fund_classes = pd.read_excel(imp_name, sheet_name='python')
fileOverview.loc['data_fund_classes', 'Source'] = "Our research"

# Merge fund class information into data overview and data_fundmonths
data_overview = pd.merge(data_overview, data_fund_classes, how='left', on=['Unique fund
name'])
data_fundmonths = pd.merge(data_fundmonths, data_fund_classes, how='left', on=['Unique fund
name'])

# Get list of funds with classes
funds_with_classes = msc.unique(data_overview['Class of fund'])
funds_with_classes = [x for x in funds_with_classes if str(x) != 'nan']

# Define minimum investment groups
min_inv_groups = {}
min_inv_groups['Small'] = 999
min_inv_groups['Medium'] = 99999
min_inv_groups['Large'] = 300000000

# Create columns group returns and NAV
    # Set to -999999 to make errors visible
for group in list(min_inv_groups.keys()) + ['All']:
    data_fundmonths['NAV_' + group] = -999999
    data_fundmonths['Return_' + group] = -999999

# Create new output dataset
data_fundmonths_new = pd.DataFrame(columns=data_fundmonths.columns)

# Fill group returns and NAV for funds with a share class
for temp_fund in funds_with_classes:
    # Get fund data and months
    temp_data_fm = data_fundmonths.copy()
    temp_data_fm = temp_data_fm[temp_data_fm['Class of fund'] == temp_fund]
    # temp_data_fm = temp_data_fm.sort_values(by=['Month'], ascending=True)
    months = msc.unique(temp_data_fm['Month'])
    months.sort()

    # Loop through months
    for month in months:

        # Get data for the month
        temp_data_fm_month = temp_data_fm[temp_data_fm['Month'] == month]

        # Get the fund class to be kept
        temp_data_fm_month_keep = temp_data_fm_month[temp_data_fm_month['Main shareclass'] ==
True].copy()

        # DNB Norge: Treat return series manually
            # DNB Norge (I) is the oldest class and has returns for 1981/11-1985/3 and 1985/9-
2014/2
            # We use the following returns:
                # DNB Norge (Avanse I)  for 1985/4 - 1985/8
                # DNB Norge             for 2014/3 - 2019/8
                # DNB Norge D           for 2019/9 - 2019/12
        if len(temp_data_fm_month_keep['Unique fund name']) != 1 and temp_fund == 'DNB Norge
(I)':

            # Use DNB Norge (Avanse I)
            if month >= 19850430 and month <= 19850831:
                temp_data_fm_month_keep = temp_data_fm_month[temp_data_fm_month['Unique fund
name'] == 'DNB Norge (Avanse I)'].copy()
                temp_data_fm_month_keep['Main shareclass'] = True
                temp_data_fm_month['Main shareclass'] = True

            # Use DNB Norge
            if month >= 20140331 and month <= 20190831:
                temp_data_fm_month_keep = temp_data_fm_month[temp_data_fm_month['Unique fund
name'] == 'DNB Norge'].copy()
                temp_data_fm_month_keep['Main shareclass'] = True

            # Use DNB Norge D
            if month >= 20190831 and month <= 20191231:
                temp_data_fm_month_keep = temp_data_fm_month[temp_data_fm_month['Unique fund
name'] == 'DNB Norge D'].copy()
                temp_data_fm_month_keep['Main shareclass'] = True

        # Check that a fund was kept
        if len(temp_data_fm_month_keep['Unique fund name']) != 1:
            print("")
            print("NOTE! Did not process the following:")
            print("Fund: " + str(temp_fund))
            print("Month: " + str(month))
            print("Cause: Either month before listing date or something else.")

        # If no NAV's recorded, use oldest share class, els discard NAV's missing
        if sum(1*(temp_data_fm_month['NAV_Raw']>0)) == 0:
            temp_data_fm_month = temp_data_fm_month[temp_data_fm_month['Main shareclass'] ==
True]

            temp_data_fm_month['NAV_Raw'] = -1 # Set to "any" amount as it is the only one
        else:
            temp_data_fm_month = temp_data_fm_month[temp_data_fm_month['NAV_Raw'] > 0]

        # Compute TNAV and Return for groups
        for group in list(min_inv_groups.keys()) + ['All']:

            # Get group data
            if group == 'All':
                temp_data = temp_data_fm_month
            else:
                temp_boolean = (temp_data_fm_month['Min investment'] < min_inv_groups[group])
                temp_data = temp_data_fm_month[temp_boolean]

            # Write NAV and Return if returns for the group, else nan
            if len(temp_data['Unique fund name']) > 0:
                temp_data_fm_month_keep['NAV_' + group] = sum(temp_data['NAV_Raw'])
                temp_data_fm_month_keep['Return_' + group] =
sum(np.multiply(temp_data['Return_Raw'], temp_data['NAV_Raw']/sum(temp_data['NAV_Raw'])))
            else:
                temp_data_fm_month_keep['NAV_' + group] = 'nan'
                temp_data_fm_month_keep['Return_' + group] = 'nan'

        # Count shareclasses
        temp_data_fm_month_keep['Share class count'] = len(temp_data_fm_month['Main
shareclass'])

        # Merge to main dataset
        data_fundmonths_new = data_fundmonths_new.append(temp_data_fm_month_keep)

# Get list of funds without classes
funds_without_classes = list(data_overview[data_overview['Class of fund'].isna()]['Unique fund
name'])

# Fill group returns and NAV for funds without a share class
for fund in funds_without_classes:

    # Get fund data and months
    temp_data_fm = data_fundmonths[data_fundmonths['Unique fund name'] == fund]
    months = msc.unique(temp_data_fm['Month'])
    months.sort()

    # Loop through each month
    for month in months:

        # Get data for the month
        temp_data_fm_month = temp_data_fm[temp_data_fm['Month'] == month]

        # Only one class per month (so a simpler operation than for funds with share classes)
        temp_data_fm_month_keep = temp_data_fm_month.copy()

        # Compute TNAV and Return for groups
        for group in list(min_inv_groups.keys()) + ['All']:

            # Get group data
            if group == 'All':
                temp_data = temp_data_fm_month
            else:
                temp_boolean = (temp_data_fm_month['Min investment'] < min_inv_groups[group])
                temp_data = temp_data_fm_month[temp_boolean]

          # Write NAV and Return if returns for the group, else nan
            if len(temp_data['Unique fund name']) > 1:
                print("ERROR! More than one observation for fundmonth with one class!")
            elif len(temp_data['Unique fund name']) == 1:
                temp_data_fm_month_keep['NAV_' + group] = sum(temp_data['NAV_Raw'])
                temp_data_fm_month_keep['Return_' + group] = sum(temp_data['Return_Raw']) #
One class, so no need to weight
            else:
                temp_data_fm_month_keep['NAV_' + group] = 'nan'
                temp_data_fm_month_keep['Return_' + group] = 'nan'

        # Count shareclasses
        temp_data_fm_month_keep['Share class count'] = len(temp_data_fm_month['Main
shareclass'])

        # Merge to main dataset
        data_fundmonths_new = data_fundmonths_new.append(temp_data_fm_month_keep)

# Check for -999999's
temp = 1*(data_fundmonths_new == -99999)
print("Count of -99 999: ", temp.to_numpy().sum())

# Check for duplicate fundmonths
temp = data_fundmonths_new[data_fundmonths_new.duplicated(['Month', 'Unique fund name'])]
print("Duplicate entries: ", len(temp['Unique fund name']))

# Rename min investment column (not supposed to be used anymore)
data_fundmonths_new = data_fundmonths_new.rename(columns={'Min investment': 'OLD_Min
investment'})

# Update main dataset
data_fundmonths = data_fundmonths_new.copy()
del data_fundmonths_new
```

```
# Clean -1 and convert new columns to numeric
data_fundmonths = data_fundmonths.replace(-1,'nan')
temp_cols = ['NAV_Small', 'Return_Small', 'NAV_Medium', 'Return_Medium', 'NAV_Large',
'Return_Large', 'NAV_All', 'Return_All']
data_fundmonths[temp_cols] = data_fundmonths[temp_cols].apply(pd.to_numeric, errors='coerce')

#%% Export files

# Perform export
if ExportMode:
    msc.export_data('Minor change', data_overview, data_fundmonths, 'skip') # 'Overwrite',
'Minor change' or 'Version'
```

02_empirical methods_v0.3.py

```
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

import numpy as np
import MScThesis_v4 as msc
import matplotlib.pyplot as plt
from pylab import *

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Get garamond font
garamond = msc.import_garamond()

#%%
def create_S2(Xi_s, zs): # Trolig raskere med np.array
    CDF = [sum((1*(Xi_s<=z_val))*(z_val-Xi_s))/len(Xi_s)*100 for z_val in zs]
    return CDF

import MScThesis_v4 as msc
garamond = msc.import_garamond()

dx = 0.01

# SSD plot 2
G = np.random.normal(0, 1, 5000)
F = np.random.normal(-0.5, 3, 5000)

exportMode = False
fig_name = 'SSD02_XX'

# Create chart
F.sort()
G.sort()

Y  = exp(-F ** 2)
X  = exp(-G ** 2)

# Normalize the data to a proper PDF
Y /= (dx * Y).sum()
X /= (dx * X).sum()

# Compute the CDF
CY = np.cumsum(Y * dx)
CX = np.cumsum(X * dx)

# Plot the PDF
plt.figure()
plt.plot(G,X, label='G', color='k')
plt.plot(F,Y, label='F', color='k', linestyle='dashed')
plt.legend(prop=garamond, loc='upper left')
plt.title('Comparison of PDFs')
plt.ylabel('Probability')
ax = plt.gca()
ax.set_xlim(-6,6)
msc.chart_garamond(plt, ax)
if exportMode:
    plt.savefig('02_empirical methods/figure_' + fig_name + '_PDF.eps', format='eps')

# Plot the CDF
plt.figure()
plt.plot(G, CX, label='G', color='k')
plt.plot(F, CY, label='F', color='k', linestyle='dashed')
plt.legend(prop=garamond, loc='upper left')
plt.title('Comparison of CDFs')
plt.ylabel('Probability')
ax = plt.gca()
ax.set_xlim(-6,6)
msc.chart_garamond(plt, ax)
if exportMode:
    plt.savefig('02_empirical methods/figure_' + fig_name + '_CDF.eps', format='eps')

# Plot the S function
SY = np.cumsum(CY * dx)
SX = np.cumsum(CX * dx)

SY = SY/max(SY)
SX = SX/max(SX)

plt.figure()
plt.plot(G, SX, label='G', color='k')
plt.plot(F, SY, label='F', color='k', linestyle='dashed')
plt.legend(prop=garamond, loc='upper left')
plt.title('Comparison of S functions')
ax = plt.gca()
ax.set_xlim(-6,6)
msc.chart_garamond(plt, ax)
if exportMode:
    plt.savefig('02_empirical methods/figure_' + fig_name + '_S.eps', format='eps')
```

03_data_v2.4.py

```python
#%% Self-created functions

# Clear all variables from variable explorer
def clear_all():
    """Clears all the variables from the workspace of the spyder application."""
    gl = globals().copy()
    for var in gl:
        if var[0] == '_': continue
        if 'func' in str(globals()[var]): continue
        if 'module' in str(globals()[var]): continue

        del globals()[var]

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

# Export regression as picture
def export_regression(export_mode, model_object,relative_file_path):
    import matplotlib.pyplot as plt
    plt.figure()
    plt.rc('figure', figsize=(6, 4))
    plt.text(0.01, 0.05, str(model_object.summary()), {'fontsize': 10}, fontproperties =
'monospace') # approach improved by OP -> monospace!
    plt.axis('off')
    plt.tight_layout()
    if export_mode:
        plt.savefig(relative_file_path + '.eps', format='eps')

#%% Import and prepare

# Import packages
import pandas as pd
import os as os
import sys
import numpy as np
import MScThesis_v4 as msc
import statsmodels.api as sm
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import matplotlib.dates as mdates

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Get old data_fundmonths file
# Reason: Counting distinct funds which cannot be done in the new data fundmonths format
temp_path = os.path.realpath(os.path.basename(sys.argv[0])) + "/01_cleaning_output/First
round_data_fundmonths_v8.0 (Used in Data code).xlsx"
data_fundmonths_v8 = pd.read_excel(temp_path, index_col=0)
del temp_path

#%% Get distinct funds dummy
''' Note: Must update file version of distinct list, if changed! '''
data_distinct = pd.read_excel(os.path.realpath(os.path.basename(sys.argv[0])) +
"/01_cleaning_input/Overview of fund classes_v0.1.xlsx", sheet_name='python')
data_distinct = data_distinct[['Unique fund name', 'Main shareclass']]

data_overview = pd.merge(data_overview, data_distinct, how='left', on=['Unique fund name'])
data_fundmonths_v8 = pd.merge(data_fundmonths_v8, data_distinct, how='left', on=['Unique fund
name'])

#%% Deal with incubation bias for data_fundmonths_v8

# Get fund info Oslo Børs
MS_version = "v6.5"
```

```
imp_name = os.path.realpath(os.path.basename(sys.argv[0])) + "/01_cleaning_input/MS_Overview
of funds_" + MS_version + ".xlsx"
funds_OSE = pd.read_excel(imp_name, sheet_name='python_OB')

data_fundmonths_new2 = pd.DataFrame(columns=data_fundmonths_v8.columns)

for fund in list(data_overview['Unique fund name']):

    # Get fund data
    data_fm_temp = data_fundmonths_v8 [data_fundmonths_v8['Unique fund name']==fund].copy()

    # Create new column with month on datetime format
    data_fm_temp['Month_format'] = pd.to_datetime(data_fm_temp['Month'].astype(str),
format='%Y%m%d')

    # Get the funds ISIN (skip to next if no data for the fund)
    try:
        temp_ISIN = list(data_fm_temp['ISIN'])[0]
    except:
        continue

    # Skip if ISIN is not in OSE dataset (i.e. we do not have listed date)
    if sum(1*(funds_OSE['ISIN'] == temp_ISIN)) == 0:
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp)
        continue

    # If ISIN is 'nan', keep all (all clases with incubation problems have ISIN; verified
manually in Excel)
    if temp_ISIN=='':
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp)
    else:

        # Get the funds listing date
        temp_listed_from = list(funds_OSE[funds_OSE['ISIN']==temp_ISIN]['List member
from'])[0]

        # Create indicator for returns to be kept
        data_fm_temp['inc_bias_keep'] = (data_fm_temp['Month_format'] > temp_listed_from)

        # Get data to be kept
        data_fm_temp_keep = data_fm_temp[data_fm_temp['inc_bias_keep']]

        # Store data
        data_fundmonths_new2 = data_fundmonths_new2.append(data_fm_temp_keep)

# Delete temporary columns
del data_fundmonths_new2['Month_format']
del data_fundmonths_new2['inc_bias_keep']

# Update main dataset
data_fundmonths_v8 = data_fundmonths_new2.copy()

#%% Clear variables not to be used later
clear_variables(["data_fundmonths_new2", "data_fm_temp", "temp_ISIN", "temp_listed_from",
"data_fm_temp_keep"])
clear_variables(["fund", "MS_version", "imp_name"])

#%% Convert months to month format
data_fundmonths['YearMonth'] = [str(x)[:-2] for x in data_fundmonths['Month']]
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

data_fundmonths_v8['YearMonth'] = [str(x)[:-2] for x in data_fundmonths_v8['Month']]
data_fundmonths_v8['Month'] = pd.to_datetime(data_fundmonths_v8['Month'].astype(str),
format='%Y%m%d')

# Create datasets for distinct funds
data_fm_distinct = data_fundmonths#data_fundmonths_v8[data_fundmonths_v8['Main
shareclass']==True]
## The data_fm_distinct variable is now redundant (just equal to data_fundmonths)
## and can be removed. It was used before the share class problem was fixed.

#%% Create CMA and RMW

data_fundmonths['Rm-Rf'] = data_fundmonths['Rm'] - data_fundmonths['Rf']

# Import 5-factor for Europe
```

```
KF_eur5f = pd.read_csv("01_uncleaned_data/04 Europe_5_Factors.csv", sep=',', skiprows=lambda
x: x in [0, 2])
KF_eur5f = KF_eur5f.iloc[0:354]
temp_cols = list('eur5f_' + KF_eur5f.columns)
temp_cols[0] = 'YearMonth'
KF_eur5f.columns = temp_cols
KF_eur5f['YearMonth'] = KF_eur5f['YearMonth'].str.replace(' ', '')
KF_eur5f[list(KF_eur5f.columns)[1:]] =
KF_eur5f[list(KF_eur5f.columns)[1:]].apply(pd.to_numeric, errors='coerce', axis=1)/100

# Import Bernt Ødegaard factors
factors_raw = pd.read_excel("01_uncleaned_data/04 Factor Data Norwegian Equities_v2.xlsx")
factors_raw['YearMonth'] = factors_raw.apply(lambda x: str(x['date'])[:-2], axis = 1)
factors_raw['YearMonth'] = [str(x)[:-2] for x in factors_raw['date']]

# Merge datasets
KF_eur5f = pd.merge(left=KF_eur5f, right=data_fundmonths[['YearMonth', 'Rm-
Rf']].drop_duplicates(), on='YearMonth', how='left')
KF_eur5f = pd.merge(KF_eur5f, factors_raw, on='YearMonth', how='left')

# Regress RMW on the other three factors to create our RMW
Y = KF_eur5f['eur5f_RMW']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['RMW'] = model.resid + model.params['const']

# Regress CMA on the other three factors to create our CMA
Y = KF_eur5f['eur5f_CMA']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['CMA'] = model.resid + model.params['const']

# Merge CMA and RMW into data_fundmonths
data_fundmonths = pd.merge(data_fundmonths, KF_eur5f[['YearMonth', 'RMW', 'CMA']],
on='YearMonth', how='left')

#%% Clear variables not to be used later
clear_variables(["temp_cols", "factors_raw", "Y", "X", "model"])

#%%

# Define variables
periods = [[1981, 2019], [1981, 1990], [1991, 2005], [2006, 2019]]

# Full data
[periods_names, periods_dfs, periods_dfs_index, periods_dfs_active] =
msc.create_period_dfs(data_fundmonths, periods)

# Distinct fund data
[d_period_names, d_periods_dfs, d_periods_dfs_index, d_periods_dfs_active] =
msc.create_period_dfs(data_fm_distinct, periods)

# Full data v8
[periods_names_v8, periods_dfs_v8, periods_dfs_index_v8, periods_dfs_active_v8] =
msc.create_period_dfs(data_fundmonths_v8, periods)

#%% Clear variables not to be used later
clear_variables(["period", "period_start", "period_end", "period_data", "period_name"])

#%% Import Garamond
garamond = msc.import_garamond()

#%% Figure 01: Descriptive table of our funds

# Create dataset
rowNames = ["Number of distinct funds", \
            "Number of distinct index funds", \
            "Number of distinct active funds", \
            "Number of distinct fund classes", \
            "Number of monthly returns", \
            "Number of monthly returns for index funds", \
            "Number of monthly returns for active funds", \
            "Average number of observations per month"]

figure01 = pd.DataFrame(index=rowNames, columns=periods_names)
```

```
# Fill values
for period in periods_names:
    figure01.loc["Number of distinct funds"][period] =
len(msc.unique(d_periods_dfs[period]["Unique fund name"]))
    figure01.loc["Number of distinct index funds"][period] =
len(msc.unique(d_periods_dfs_index[period]["Unique fund name"]))
    figure01.loc["Number of distinct active funds"][period] =
len(msc.unique(d_periods_dfs_active[period]["Unique fund name"]))
    figure01.loc["Number of distinct fund classes"][period] =
len(msc.unique(periods_dfs_v8[period]["Unique fund name"]))
    figure01.loc["Number of monthly returns"][period] = len(periods_dfs[period]["Unique fund
name"])
    figure01.loc["Number of monthly returns for index funds"][period] =
len(periods_dfs_index[period]["Unique fund name"])
    figure01.loc["Number of monthly returns for active funds"][period] =
len(periods_dfs_active[period]["Unique fund name"])
    figure01.loc["Average number of observations per month"][period] =
len(periods_dfs[period]["Unique fund name"]) /
len(msc.unique(periods_dfs_active[period]["Month"]))

# Save figure
if exportMode:
    figure01.to_excel('03_data/figure01.xlsx')

#%% Clear variables not to be used later
clear_variables(["rowNames"])


#%% Create frame with months
[d_monthly_overview, d_monthly_dfs] = msc.create_monthly_dfs(data_fm_distinct)

#%% Count observations per fund class and find average returns

'''NOTE: This section computes per fund class (not per distinct fund)'''

# Create columns for counts
d_monthly_overview['Count_total'] = 'nan'
d_monthly_overview['Count_active'] = 'nan'
d_monthly_overview['Count_index'] = 'nan'

# Create columns for average returns
d_monthly_overview['Avg_ret_total'] = 'nan'
d_monthly_overview['Avg_ret_active'] = 'nan'
d_monthly_overview['Avg_ret_index'] = 'nan'

# Loop through data and populate
for index in d_monthly_overview.index:
    # print(index)

    # Create sub-frames
    temp_active = d_monthly_dfs[index][d_monthly_dfs[index]['Index fund']==False]
    temp_index = d_monthly_dfs[index][d_monthly_dfs[index]['Index fund']==True]

    # Count number of fund classes
    d_monthly_overview.loc[index]['Count_total'] = len(list(d_monthly_dfs[index]['Index
fund']))
    d_monthly_overview.loc[index]['Count_active'] = len(list(temp_active['Index fund']))
    d_monthly_overview.loc[index]['Count_index'] = len(list(temp_index['Index fund']))

    # Compute average returns
    d_monthly_overview.loc[index]['Avg_ret_total'] = d_monthly_dfs[index]['Return_All'].mean()
    d_monthly_overview.loc[index]['Avg_ret_active'] = temp_active['Return_All'].mean()
    d_monthly_overview.loc[index]['Avg_ret_index'] = temp_index['Return_All'].mean()

#%% Figure 02: Stacked area chart of distinct funds - active vs index

colors = ['black', 'gainsboro']      # Coloers [active, index]
alpha = 0.75                         # Strenght of colors

# Compute fractions
d_monthly_overview['%_index'] = d_monthly_overview['Count_index'] /
d_monthly_overview['Count_total']
d_monthly_overview['%_active'] = d_monthly_overview['Count_active'] /
d_monthly_overview['Count_total']

# Create plot
```

```
fig = plt.figure()
ax = fig.add_subplot(111)

y = np.array(d_monthly_overview[['%_active', '%_index']], dtype=float).transpose() * 100
ax.stackplot(d_monthly_overview.index, y, colors=colors, edgecolor='black', linewidth=0.5,
alpha=alpha)

# Adjust x-axis
ax.set_xlim([pd.to_datetime(np.int64(19810731),
format='%Y%m%d'),pd.to_datetime(np.int64(20200101), format='%Y%m%d')])

# Fix layout
ax.set_ylabel('Percent of distinct funds (%)')
plt.legend(['Active', 'Index'], loc='lower right', prop=garamond)
ax.margins(0, 0) # Set margins to avoid "whitespace"
msc.chart_garamond(plt,ax)

# Save figure
if exportMode:
    plt.savefig('03_data/figure02.eps', format='eps')

#%% Clear variables not to be used later
clear_variables(["fig","y", "ax"])

#%% Figure 03: Descriptive for returns (for fund classes, not distinct funds)

# Include 1991 to 2019 for figure 03
periods_f03 = [[1981, 2019], [1991, 2019], [1981, 1990], [1991, 2005], [2006, 2019]]
[periods_names_f03, periods_dfs_f03, periods_dfs_index_f03, periods_dfs_active_f03] =
msc.create_period_dfs(data_fundmonths, periods_f03)


def figure03(periods_dfs, periods_names):

    # Create dataset
    colNames = ["Obs", "Mean", "Min", "Max", "Std", "Skew", "Kurt"]
    figure03 = pd.DataFrame(index=periods_names, columns=colNames)

    # Fill values
    for period in periods_names:
        figure03.loc[period]["Obs"] = len(periods_dfs[period]["Return_All"])
        figure03.loc[period]["Mean"] = periods_dfs[period]["Return_All"].mean() *100
        figure03.loc[period]["Min"] = periods_dfs[period]["Return_All"].min()   *100
        figure03.loc[period]["Max"] = periods_dfs[period]["Return_All"].max()   *100
        figure03.loc[period]["Std"] = periods_dfs[period]["Return_All"].std()   *100
        figure03.loc[period]["Skew"] = periods_dfs[period]["Return_All"].skew() # Not *100
        figure03.loc[period]["Kurt"] = periods_dfs[period]["Return_All"].kurt() # Not *100

    return figure03

# Generate figures
figure03_all        =   figure03(periods_dfs_f03, periods_names_f03)
figure03_active     =   figure03(periods_dfs_active_f03, periods_names_f03)
figure03_index      =   figure03(periods_dfs_index_f03, periods_names_f03)

# Save figure
if exportMode:
    figure03_all.to_excel('03_data/figure03_all.xlsx')
    figure03_active.to_excel('03_data/figure03_active.xlsx')
    figure03_index.to_excel('03_data/figure03_index.xlsx')

#%% Clear variables not to be used later
clear_variables(["colNames","period"])

#%% Figure 05: Descriptives for factors (not returns)

periods_dfs_factors = {}

# Update periods
periods = [[1991, 2019], [1981, 2019], [1981, 1990], [1991, 2005], [2006, 2019]]
# [periods_names, periods_dfs, __, __] = create_period_dfs(data_fundmonths, periods)
[periods_names, periods_dfs, periods_dfs_index, periods_dfs_active] =
msc.create_period_dfs(data_fundmonths, periods)


# Create dfs
for period in periods_names:
    # Get unique months (sorted)
    tmp_data = periods_dfs[period]
    tmp_months = msc.unique(tmp_data['Month'])
    tmp_months.sort()

    # Create dataframe with unique months as index
    tmp_df = pd.DataFrame()
    tmp_df['Month'] = tmp_months

    # Merge in data for factors fra fm data
    tmp_data_red = tmp_data[["Month", "Rm", "Rf", "SMB", "HML", "PR1YR", "RMW", "CMA"]]
    tmp_df = pd.merge(tmp_df, tmp_data_red, how='left', on='Month')
    tmp_df["Rm-Rf"] = tmp_df['Rm'] - tmp_df['Rf']
    periods_dfs_factors[period] = tmp_df

# Method that creates figure
def gen_figure05(period):
    colNames = ["Mean", "Min", "Max", "Std", "Skew", "Kurt"]
    rowNames = ["Rm", "Rf", "Rm-Rf", "SMB", "HML", "PR1YR", "RMW", "CMA"]
    figure05 = pd.DataFrame(index=rowNames, columns=colNames)

    # Fill figure
    tmp_data = periods_dfs_factors[period]
    for row_name in rowNames:
        figure05.loc[row_name]['Mean'] = tmp_data[row_name].mean()  *100
        figure05.loc[row_name]['Min'] = tmp_data[row_name].min()    *100
        figure05.loc[row_name]['Max'] = tmp_data[row_name].max()    *100
        figure05.loc[row_name]['Std'] = tmp_data[row_name].std()    *100
        figure05.loc[row_name]['Skew'] = tmp_data[row_name].skew()  # Not *100
        figure05.loc[row_name]['Kurt'] = tmp_data[row_name].kurt()  # Not *100

    return figure05

# Create figure per perid
figure05 = {}

# Create figures
for period in periods_names:
    figure05[period] = gen_figure05(period)

# Save figures
if exportMode:
    for period in periods_names:
        figure05[period].to_excel('03_data/figure05_' + period + '.xlsx')

#%% Clear variables not to be used later
clear_variables(["period","row_name", "rowNames", "colNames", "tmp_data_red", "tmp_df"])

#%% Prepare monthly data

# Create monthly overview for non-distinct funds
[monthly_overview, monthly_dfs] = msc.create_monthly_dfs(data_fundmonths)

# Create columns for counts
monthly_overview['Count_total'] = 'nan'
monthly_overview['Count_active'] = 'nan'
monthly_overview['Count_index'] = 'nan'

# Create columns for average returns
monthly_overview['Avg_ret_total'] = 'nan'
monthly_overview['Avg_ret_active'] = 'nan'
monthly_overview['Avg_ret_index'] = 'nan'

monthly_overview['Avg_ret_active_TNAVW'] = 'nan'

# Create columns for factors and RF
monthly_overview['Rm'] = 'nan'
monthly_overview['Rf'] = 'nan'
monthly_overview['SMB'] = 'nan'
monthly_overview['HML'] = 'nan'
monthly_overview['PR1YR'] = 'nan'
monthly_overview['RMW'] = 'nan'
monthly_overview['CMA'] = 'nan'

# Loop through data and populate
for index in monthly_overview.index:
    # Create sub-frames
    temp_active = monthly_dfs[index][monthly_dfs[index]['Index fund']==False]
    temp_index = monthly_dfs[index][monthly_dfs[index]['Index fund']==True]

    # Count number of fund classes
    monthly_overview.loc[index]['Count_total'] = len(list(monthly_dfs[index]['Index fund']))
    monthly_overview.loc[index]['Count_active'] = len(list(temp_active['Index fund']))
    monthly_overview.loc[index]['Count_index'] = len(list(temp_index['Index fund']))

    # Compute average returns
    monthly_overview.loc[index]['Avg_ret_total'] = monthly_dfs[index]['Return_All'].mean()
    monthly_overview.loc[index]['Avg_ret_active'] = temp_active['Return_All'].mean()
    monthly_overview.loc[index]['Avg_ret_index'] = temp_index['Return_All'].mean()

    # Compute TNAV-W returns (left as 'nan' if no NAV observations)
    temp_active.dropna(subset=['NAV_All'], how='all', inplace=True)
    if len(temp_active['Return_All'])>0:
        monthly_overview.loc[index]['Avg_ret_active_TNAVW'] = 
sum(np.multiply(temp_active['Return_All'],
temp_active['NAV_All']/sum(temp_active['NAV_All'])))

    # Get factor data
    monthly_overview.loc[index]['Rm'] = monthly_dfs[index]['Rm'].mean()
    monthly_overview.loc[index]['Rf'] = monthly_dfs[index]['Rf'].mean()
    monthly_overview.loc[index]['SMB'] = monthly_dfs[index]['SMB'].mean()
    monthly_overview.loc[index]['HML'] = monthly_dfs[index]['HML'].mean()
    monthly_overview.loc[index]['PR1YR'] = monthly_dfs[index]['PR1YR'].mean()
    monthly_overview.loc[index]['RMW'] = monthly_dfs[index]['RMW'].mean()
    monthly_overview.loc[index]['CMA'] = monthly_dfs[index]['CMA'].mean()

# Convert columns to numeric
temp_cols = ['Avg_ret_total', 'Avg_ret_active', 'Avg_ret_index', 'Avg_ret_active_TNAVW', 'Rm',
'Rf', 'SMB', 'HML', 'PR1YR', 'RMW', 'CMA']
monthly_overview[temp_cols] = monthly_overview[temp_cols].apply(pd.to_numeric,
errors='coerce')

# Clear variables not to be used later
clear_variables(["index","temp_active", "temp_index", "temp_cols"])

#%% Figure 06: Cross-correlations

figure06 = {}


# Get period data
for period in periods:

    variables = ["Ri-Rf (EW Active)", "Ri-Rf (TNAVW Active)", "Ri-Rf (EW Index)", "Rm-Rf",
"SMB", "HML", "PR1YR", "RMW", "CMA"]

    temp = pd.DataFrame(index=variables,columns=variables)

    # Get period data
    period_start = pd.to_datetime(str(period[0]) + "01" + "01", format='%Y%m%d')
    period_end = pd.to_datetime(str(period[1]) + "12" + "31", format='%Y%m%d')
    temp_m_overview = monthly_overview[ (monthly_overview.index >= period_start ) &
(monthly_overview.index <= period_end ) ]

    # All funds
    temp.loc["Rm-Rf"]["Rm-Rf"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['Rm']-temp_m_overview['Rf'])[0][1]
    temp.loc["Rm-Rf"]["SMB"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['SMB'])[0][1]
    temp.loc["Rm-Rf"]["HML"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['HML'])[0][1]
    temp.loc["Rm-Rf"]["PR1YR"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['PR1YR'])[0][1]
    temp.loc["Rm-Rf"]["RMW"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['RMW'])[0][1]
    temp.loc["Rm-Rf"]["CMA"] = np.corrcoef(temp_m_overview['Rm']-temp_m_overview['Rf'],
temp_m_overview['CMA'])[0][1]

    # Active funds EW
    temp.loc["Ri-Rf (EW Active)"]["Ri-Rf (EW Active)"] =
np.corrcoef(temp_m_overview['Avg_ret_active']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_active']-temp_m_overview['Rf'])[0][1]
```

```
    temp.loc["Ri-Rf (EW Active)"]["Ri-Rf (TNAVW Active)"] =
np.corrcoef(temp_m_overview['Avg_ret_active']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["Ri-Rf (EW Index)"] =
np.corrcoef(temp_m_overview['Avg_ret_active']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_index']-temp_m_overview['Rf'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["SMB"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['SMB'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["HML"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['HML'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["PR1YR"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['PR1YR'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["RMW"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['RMW'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["CMA"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['CMA'])[0][1]
    temp.loc["Ri-Rf (EW Active)"]["Rm-Rf"] = np.corrcoef(temp_m_overview['Avg_ret_active']-
temp_m_overview['Rf'], temp_m_overview['Rm'] - temp_m_overview['Rf'])[0][1]

    # Active funds TNAVW
    temp.loc["Ri-Rf (TNAVW Active)"]["Ri-Rf (TNAVW Active)"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["Ri-Rf (EW Index)"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_index']-temp_m_overview['Rf'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["SMB"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['SMB'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["HML"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['HML'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["PR1YR"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['PR1YR'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["RMW"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['RMW'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["CMA"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['CMA'])[0][1]
    temp.loc["Ri-Rf (TNAVW Active)"]["Rm-Rf"] =
np.corrcoef(temp_m_overview['Avg_ret_active_TNAVW']-temp_m_overview['Rf'],
temp_m_overview['Rm'] - temp_m_overview['Rf'])[0][1]

    # Index funds EW
    temp.loc["Ri-Rf (EW Index)"]["Ri-Rf (EW Index)"] =
np.corrcoef(temp_m_overview['Avg_ret_index']-temp_m_overview['Rf'],
temp_m_overview['Avg_ret_index']-temp_m_overview['Rf'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["SMB"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['SMB'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["HML"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['HML'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["PR1YR"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['PR1YR'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["RMW"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['RMW'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["CMA"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['CMA'])[0][1]
    temp.loc["Ri-Rf (EW Index)"]["Rm-Rf"] = np.corrcoef(temp_m_overview['Avg_ret_index']-
temp_m_overview['Rf'], temp_m_overview['Rm'] - temp_m_overview['Rf'])[0][1]

    # Factors
    temp.loc["SMB"]["SMB"] = np.corrcoef(temp_m_overview['SMB'], temp_m_overview['SMB'])[0][1]
    temp.loc["SMB"]["HML"] = np.corrcoef(temp_m_overview['SMB'], temp_m_overview['HML'])[0][1]
    temp.loc["SMB"]["PR1YR"] = np.corrcoef(temp_m_overview['SMB'],
temp_m_overview['PR1YR'])[0][1]
    temp.loc["SMB"]["RMW"] = np.corrcoef(temp_m_overview['SMB'], temp_m_overview['RMW'])[0][1]
    temp.loc["SMB"]["CMA"] = np.corrcoef(temp_m_overview['SMB'], temp_m_overview['CMA'])[0][1]

    temp.loc["HML"]["HML"] = np.corrcoef(temp_m_overview['HML'], temp_m_overview['HML'])[0][1]
    temp.loc["HML"]["PR1YR"] = np.corrcoef(temp_m_overview['HML'],
temp_m_overview['PR1YR'])[0][1]
    temp.loc["HML"]["RMW"] = np.corrcoef(temp_m_overview['HML'], temp_m_overview['RMW'])[0][1]
    temp.loc["HML"]["CMA"] = np.corrcoef(temp_m_overview['HML'], temp_m_overview['CMA'])[0][1]
    temp.loc["PR1YR"]["PR1YR"] = np.corrcoef(temp_m_overview['PR1YR'],
temp_m_overview['PR1YR'])[0][1]
    temp.loc["PR1YR"]["RMW"] = np.corrcoef(temp_m_overview['PR1YR'],
temp_m_overview['RMW'])[0][1]
    temp.loc["PR1YR"]["CMA"] = np.corrcoef(temp_m_overview['PR1YR'],
temp_m_overview['CMA'])[0][1]

    temp.loc["RMW"]["RMW"] = np.corrcoef(temp_m_overview['RMW'], temp_m_overview['RMW'])[0][1]
    temp.loc["RMW"]["CMA"] = np.corrcoef(temp_m_overview['RMW'], temp_m_overview['CMA'])[0][1]

    temp.loc["CMA"]["CMA"] = np.corrcoef(temp_m_overview['CMA'], temp_m_overview['CMA'])[0][1]

    # Transpose
    temp = temp.transpose()

    # Add dictionary
    period_name = str(period[0]) + "-" + str(period[1])
    figure06[period_name] = temp

# Save figures
if exportMode:
    for period in figure06.keys():
        figure06[period].to_excel('03_data/figure06_' + period + '.xlsx')

#%% Clear variables not to be used later
clear_variables(["variables","period", "temp", "period_name", "period_start", "period_end",
"temp_m_overview"])

#%% Figure 08: Cumulative returns on factors

# Settings for chart
month_start = 19901231 # First valid month is 19810731!
month_end = 20191231
before_1991 = (month_start < 19901231) # Ignore RMW and CMA before 1990 (no data on them)

''' Prepare data '''

# Import factor data for SMB, HML and PR1YR
file_name =  os.path.basename(sys.argv[0])  # Store name of this file
file_path =  os.path.realpath(file_name)    # Store filepath of this file
file_path =  file_path.strip(file_name)
# imp_name = file_path + "/01_uncleaned_data/04 Factor Data Norwegian Equities.xlsx"
factors_raw = pd.read_excel(file_path + "/01_uncleaned_data/04 Factor Data Norwegian
Equities_v2.xlsx" )

# Define chart data
start_value = 100

month_start = pd.to_datetime(np.int64(month_start), format='%Y%m%d') # First month of index
fund returns
month_end = pd.to_datetime(np.int64(month_end), format='%Y%m%d')

# Clean factors
factors_clean = factors_raw.copy()
factors_clean['YearMonth'] = [str(x)[:-2] for x in factors_clean['date']]
factors_clean = pd.merge(factors_clean, KF_eur5f[['YearMonth', 'CMA', 'RMW']], on='YearMonth',
how='left')
# del factors_clean['YearMonth']

factors_clean['date'] = pd.to_datetime(factors_clean['date'].astype(str), format='%Y%m%d')
factors_clean = factors_clean[ (factors_clean['date']>=month_start) &
(factors_clean['date']<=month_end)]

# Compute cumulative returns for SMB, HML and PR1YR
if before_1991:
    factors_cum = pd.DataFrame(index = factors_clean['date'], columns=['Rm-Rf', 'SMB', 'HML',
'PR1YR'])
else:
    factors_cum = pd.DataFrame(index = factors_clean['date'], columns=['Rm-Rf', 'SMB', 'HML',
'PR1YR', 'RMW', 'CMA'])


factors_cum.loc[month_start][:] = 100
for i in range(1, len(factors_clean['date'])):
    for col in factors_cum.columns:
        if not col=='Rm-Rf':
            factors_cum.iloc[i][col] = factors_cum.iloc[i-1][col]*(1+(factors_clean.iloc[i-
1][col]))

# Import Rm and Rf data
Rm_raw = pd.read_excel(file_path + "/01_uncleaned_data/02 Market Returns/02 Market 
returns_v2.xlsx")
rf_raw = pd.read_excel(file_path + "/01_uncleaned_data/03 Risk free 1month_Ødegaard_v2.xlsx")
rf_raw.columns = rf_raw.iloc[0]
rf_raw = rf_raw[2:] # So it's the same period as mkt

# Clean data
factor_mkt = pd.merge(Rm_raw, rf_raw, on='date', how='left')
del factor_mkt['New source']
factor_mkt['date'] = pd.to_datetime(factor_mkt['date'].astype(str), format='%Y%m%d')
factor_mkt = factor_mkt.set_index('date')
factor_mkt = factor_mkt[factor_mkt.index>=month_start]
factor_mkt['Rm-Rf'] = factor_mkt['Rm'] - factor_mkt['Rf(1m)']

# Cumpute cumulative returns for Rm-Rf
col = 'Rm-Rf'
for i in range(1, len(factors_cum.index)):
    factors_cum.iloc[i][col] = factors_cum.iloc[i-1][col]*(1+(factor_mkt.iloc[i-1][col]))

# Clear variables
clear_variables(["file_name","file_path", "imp_name"])

''' Create plot '''

# Create plot
plt.figure()

# Set plot characteristics
lw = 0.5                                        # Linewidth
plt.rcParams.update({'font.size': 8})    # Font size
plt.rcParams['axes.linewidth'] = 0.5     # Chart border

# Create plot
plt.plot(factors_cum.index, factors_cum['Rm-Rf']-100, label = 'Rm-Rf', color='blue', 
linewidth=lw)
plt.plot(factors_cum.index, factors_cum['SMB']-100, label = 'SMB', color='red', linewidth=lw)
plt.plot(factors_cum.index, factors_cum['HML']-100, label = 'HML', color='green', 
linewidth=lw)
plt.plot(factors_cum.index, factors_cum['PR1YR']-100, label = 'PR1YR', color='black', 
linewidth=lw)
if not before_1991:
    plt.plot(factors_cum.index, factors_cum['RMW']-100, label = 'RMW', color='orange', 
linewidth=lw)
    plt.plot(factors_cum.index, factors_cum['CMA']-100, label = 'CMA', color='silver', 
linewidth=lw)

# Set title and legend
ax = plt.axes()
# plt.title("Cumulative average returns")
legend = ax.legend()
plt.setp(legend.texts,fontproperties=garamond)
legend.get_frame().set_linewidth(lw)
legend.get_frame().set_edgecolor("black")

# Set axis names
plt.ylabel('Cumulative return (%)')
# plt.xlabel('Year')

# Format axis ticks
a=plt.gca()
a.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
a.yaxis.set_major_formatter(mtick.FormatStrFormatter("% .0f"))

# Set garamond style
msc.chart_garamond(plt,ax)

# Save chart
if exportMode:
    plt.savefig('03_data/figure08_' + month_start.strftime('%d%m%Y') + '_' + 
month_end.strftime('%d%m%Y') + '.eps', format='eps')

#%% Generate survivorship dataset
```

```
# If statement so this part only runs once on the data
# (the merging would destroy the dataset if runned more than once)
table_generated = False
if table_generated == False:

    # Create dummy for alive or dead
    data_fundmonths['NAV'] = data_fundmonths['NAV_All']
    data_overview = msc.summarize_fundmonths(data_overview, data_fundmonths)
    del data_fundmonths['NAV']
    data_overview['Alive'] = data_overview['last_return'] ==
pd.to_datetime(np.int64(20191231), format='%Y%m%d')
    data_overview_alive = data_overview[['Unique fund name', 'Alive']]
    data_fundmonths = pd.merge(data_fundmonths, data_overview_alive, how='left', on='Unique
fund name')
    msc.unique(data_fundmonths['Alive']) # Check; Only True and False

    # Get monthly files
    [s_monthly_overview, s_monthly_dfs_alive] = msc.create_monthly_dfs(data_fundmonths)

    table_generated = True

# Create columns for returns
s_monthly_overview['Count_alive'] = 'nan'
s_monthly_overview['Count_dead'] = 'nan'
s_monthly_overview['Count_total'] = 'nan'

s_monthly_overview['Return_alive'] = 'nan'
s_monthly_overview['Return_dead'] = 'nan'
s_monthly_overview['Return_all'] = 'nan'

# Loop through data and populate
for index in s_monthly_overview.index:

    # Create sub-frames
    tmp_alive = s_monthly_dfs_alive[index][s_monthly_dfs_alive[index]['Alive']==True]
    tmp_dead = s_monthly_dfs_alive[index][s_monthly_dfs_alive[index]['Alive']==False]

    # Count number of fund classes
    s_monthly_overview.loc[index]['Count_alive'] = len(list(tmp_alive['Return_All']))
    s_monthly_overview.loc[index]['Count_dead'] = len(list(tmp_dead['Return_All']))
    s_monthly_overview.loc[index]['Count_total'] =
len(list(s_monthly_dfs_alive[index]['Return_All']))

    # Compute average returns
    s_monthly_overview.loc[index]['Return_alive'] = tmp_alive['Return_All'].mean()
    s_monthly_overview.loc[index]['Return_dead'] = tmp_dead['Return_All'].mean()
    s_monthly_overview.loc[index]['Return_all'] =
s_monthly_dfs_alive[index]['Return_All'].mean()

# Clear variables
clear_variables(['tmp_alive', 'tmp_dead'])

#%% [FIGURE 09]: Generate table for survivorship bias

# Create dataframes for the given periods
def create_period_dfs_survivor(data, periods):
    '''Note: This method has the same implementation as msc.create_period_dfs,
    but it is modified to fit the survivorship dataset'''
    periods_dfs = {}
    periods_names = [] # Need this to keep the order (the dictionary changes the order)

    # Collect data
    for period in periods:

        # Get data
        period_start = pd.to_datetime(str(period[0]) + "01" + "01", format='%Y%m%d')
        period_end = pd.to_datetime(str(period[1]) + "12" + "31", format='%Y%m%d')
        period_data = data_fundmonths[ (data_fundmonths['Month'] >= period_start ) &
(data_fundmonths['Month'] <= period_end ) ]

        # Insert data to main dictionary
        period_name = str(period[0]) + "-" + str(period[1])
        periods_names.append(period_name)
        periods_dfs[period_name] = period_data

    return periods_names, periods_dfs
```

```
# Get dictionary with data per period
s_monthly_overview['Month'] = s_monthly_overview.index
[periods_names, periods_dfs_survivor] = create_period_dfs_survivor(s_monthly_overview,
periods)
del s_monthly_overview['Month']

# Create output tables
rowNames = ["Avg_ret_Alive", \
            "Avg_ret_Dead", \
            "Avg_ret_All", \
            "Unique_count_Alive",\
            "Unique_count_Dead",\
            "Unique_count_All",\
            "Avg_count_Alive", \
            "Avg_count_Dead",\
            "Avg_count_All"]

figure09 = pd.DataFrame(index=rowNames, columns=periods_names)

# Populate output table
for period in periods_names:

    # Store data temporarily
    tmp_alive = periods_dfs_survivor[period][periods_dfs_survivor[period]['Alive']==True]
    tmp_dead = periods_dfs_survivor[period][periods_dfs_survivor[period]['Alive']==False]

    # Average returns
    figure09.loc["Avg_ret_Alive"][period] = tmp_alive['Return_All'].mean()*100
    figure09.loc["Avg_ret_Dead"][period] = tmp_dead['Return_All'].mean()*100
    figure09.loc["Avg_ret_All"][period] =
periods_dfs_survivor[period]['Return_All'].mean()*100

    # Unique counts
    figure09.loc["Unique_count_Alive"][period] = len(msc.unique(tmp_alive['Unique fund
name']))
    figure09.loc["Unique_count_Dead"][period] = len(msc.unique(tmp_dead['Unique fund name']))
    figure09.loc["Unique_count_All"][period] =
len(msc.unique(periods_dfs_survivor[period]['Unique fund name']))

    # Average counts
    figure09.loc["Avg_count_Alive"][period] =
len(tmp_alive['Month'])/len(msc.unique(tmp_alive['Month']))
    figure09.loc["Avg_count_Dead"][period] =
len(tmp_dead['Month'])/len(msc.unique(tmp_dead['Month']))
    figure09.loc["Avg_count_All"][period] =
len(periods_dfs_survivor[period]['Month'])/len(msc.unique(periods_dfs_survivor[period]['Month'
]))

# Transpose figure
figure09 = figure09.transpose()

# Save figure
if exportMode:
    figure09.to_excel('03_data/figure09.xlsx')

#%% [Figure 10]: Stacked area chart

colors = ['black', 'gainsboro']    # Coloers [active, index]
alpha = 0.75                 # Strenght of colors

# Compute fractions
s_monthly_overview['%_alive'] = s_monthly_overview['Count_alive'] /
s_monthly_overview['Count_total']
s_monthly_overview['%_dead'] = s_monthly_overview['Count_dead'] /
s_monthly_overview['Count_total']

# Create plot
fig = plt.figure()
ax = fig.add_subplot(111)
y = np.array(s_monthly_overview[['%_alive', '%_dead']], dtype=float).transpose() * 100
ax.stackplot(s_monthly_overview.index, y, colors=colors, edgecolor='black', linewidth=0.5,
alpha=alpha)
ax.set_ylabel('Fund classes (%)')
plt.legend(['Alive', 'Dead'], loc='lower right', prop=garamond)
ax.margins(0, 0) # Set margins to avoid "whitespace"
ax.set_xlim([pd.to_datetime(np.int64(19830831),
format='%Y%m%d'),pd.to_datetime(np.int64(20200101), format='%Y%m%d')])
```

```
# Fix fonts (except legend)
def chart_garamond(plt, ax):
    ax.xaxis.get_label().set_fontproperties(garamond)
    ax.yaxis.get_label().set_fontproperties(garamond)
    ax.title.set_fontproperties(garamond)
    for label in (ax.get_xticklabels() + ax.get_yticklabels()):
        label.set_fontproperties(garamond)

chart_garamond(plt,ax)

# Save figure
if exportMode:
    plt.savefig('03_data/figure10.eps', format='eps')

#%% [FIGURE 11]: Compute data for chart

# Define chart data
start_value = 100
month_start = pd.to_datetime(np.int64(19830831), format='%Y%m%d') # First month of index fund
returns
month_end = pd.to_datetime(np.int64(20191231), format='%Y%m%d')

# Create dataframe
cum_avg_ret = s_monthly_overview[ (s_monthly_overview.index >= month_start) &
(s_monthly_overview.index <= month_end)]

cum_avg_ret = cum_avg_ret[['Return_alive', 'Return_dead']].copy()

# Compute cumulative average returns
cum_avg_ret['Cum_avg_ret_alive'] = 'nan'
cum_avg_ret['Cum_avg_ret_dead'] = 'nan'
cum_avg_ret.loc[month_start]['Cum_avg_ret_alive'] = 100
cum_avg_ret.loc[month_start]['Cum_avg_ret_dead'] = 100
for i in range(1, len(cum_avg_ret.index)):
    cum_avg_ret.iloc[i]['Cum_avg_ret_alive'] = cum_avg_ret.iloc[i-1]['Cum_avg_ret_alive'] *
(1+ cum_avg_ret.iloc[i]['Return_alive'])
    cum_avg_ret.iloc[i]['Cum_avg_ret_dead'] = cum_avg_ret.iloc[i-1]['Cum_avg_ret_dead'] * (1+
cum_avg_ret.iloc[i]['Return_dead'])

# Subtract 100
cum_avg_ret['Cum_avg_ret_alive'] = cum_avg_ret['Cum_avg_ret_alive']-100
cum_avg_ret['Cum_avg_ret_dead'] = cum_avg_ret['Cum_avg_ret_dead']-100

#%% [FIGURE 11]: Plot cumulative average returns data

import matplotlib.dates as mdates

# Set format of dates
dates = cum_avg_ret.index

# Set plot characteristics
lw = 0.5                                # Linewidth
plt.rcParams.update({'font.size': 8})   # Font size
plt.rcParams['axes.linewidth'] = 0.5    # Chart border

# Create plot
plt.plot(dates, cum_avg_ret['Cum_avg_ret_alive'], label = 'Alive', color='black',
linewidth=lw)
plt.plot(dates, cum_avg_ret['Cum_avg_ret_dead'], label = 'Dead', color='black',
linestyle='dotted', linewidth=lw)

# Set title and legend
ax = plt.axes()
legend = ax.legend()
plt.setp(legend.texts,fontproperties=garamond)
legend.get_frame().set_linewidth(lw)
legend.get_frame().set_edgecolor("black")

# Set axis names
plt.ylabel('Cumulative return (%)')
# plt.xlabel('Year')

# Format axis ticks
a=plt.gca()
a.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
a.yaxis.set_major_formatter(mtick.FuncFormatter(lambda x, p: format(int(x), ' ')))
```

```
# Format x axis limits
ax.set_xlim([pd.to_datetime(np.int64(19780101),
format='%Y%m%d'),pd.to_datetime(np.int64(20220101), format='%Y%m%d')])

# Set garamond style
msc.chart_garamond(plt,ax)

# Save chart
if exportMode:
    plt.savefig('03_data/figure11.eps', format='eps')

#%% [FIGURE 12]: Summary of additional data

# Update periods_dfs
[periods_names, periods_dfs, periods_dfs_index, periods_dfs_active] =
msc.create_period_dfs(data_fundmonths, periods)

# Create dataset
rowNames = ["Total_Number of monthly returns"    , \
            "NAV_Number of monthly returns"      , \
            "NAV_%_of_total"              , \
            "min_inv_Number of monthly returns" , \
            "min_inv_%_of_total"]

figure12 = pd.DataFrame(index=rowNames, columns=periods_names)

# Fill values
for period in periods_names:
    figure12.loc["Total_Number of monthly returns"][period] =
len(periods_dfs_active[period]["Unique fund name"])

    figure12.loc["NAV_Number of monthly returns"][period] = figure12.loc["Total_Number of
monthly returns"][period] - sum(1*periods_dfs_active[period]["NAV_All"].isna())
    figure12.loc["NAV_%_of_total"][period] = figure12.loc["NAV_Number of monthly
returns"][period] / figure12.loc["Total_Number of monthly returns"][period]

    figure12.loc["min_inv_Number of monthly returns"][period] = figure12.loc["Total_Number of
monthly returns"][period] - sum(1*periods_dfs_active[period]["OLD_Min investment"].isna())
    figure12.loc["min_inv_%_of_total"][period] = figure12.loc["min_inv_Number of monthly
returns"][period] / figure12.loc["Total_Number of monthly returns"][period]

# Transpose
figure12 = figure12.transpose()

# Save figure
if exportMode:
    figure12.to_excel('03_data/figure12.xlsx')

#%% [FIGURE 13]: Summary of minimum investment groups

# Create dataset
min_inv_groups = {}
min_inv_groups["Small"] = 999
min_inv_groups["Medium"] = 99999
min_inv_groups["Large"] = 300000000
min_inv_groups["All"] = 'All'

# Create figure
figure13 = pd.DataFrame(index=min_inv_groups.keys(), columns=periods_dfs.keys())

# Loop through data and populate figure
for period in periods_dfs.keys():

    for min_inv_group in min_inv_groups.keys():

        temp_data_fm = periods_dfs_active[period]

        temp_data_fm = temp_data_fm[temp_data_fm['Return_' + min_inv_group].isna()==False]

        figure13.loc[min_inv_group][period] = len(temp_data_fm['Return_' + min_inv_group])

# Save figure
if exportMode:
    figure13.to_excel('03_data/figure13.xlsx')
```

04_additional_charts_v1.1.py

```python
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import numpy as np
import MScThesis_v4 as msc
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.ticker as mticker

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Get garamond font
garamond = msc.import_garamond()

#%% Fix for "share class problem", see Cleaning file for more info
# This file only uses 'All', so we can use a simple fix
data_fundmonths['Return'] = data_fundmonths['Return_All']
data_fundmonths['NAV'] = data_fundmonths['NAV_All']

#%% Figure 01
ret_active = data_fundmonths[data_fundmonths['Index fund'] == False][['Month', 'Return']]
ret_index = data_fundmonths[data_fundmonths['Index fund'] == True][['Month', 'Return']]

# Create chart
fig = plt.figure()
plt.scatter(ret_active['Month'], ret_active['Return']*100, s=0.5, c='b', label='Active funds')
plt.scatter(ret_index['Month'], ret_index['Return']*100, s=0.5, c='r', label = 'Index funds')

# Add line on y=0
plt.axhline(y=0, color='black', linestyle='-', linewidth=0.5)

# Format chart
ax = plt.axes()
plt.legend(prop=garamond)
msc.chart_garamond(plt, ax)
ax.set_ylim([-40, 40])
ax.yaxis.set_major_formatter(mticker.PercentFormatter())
plt.ylabel('Return')

# Save chart
if exportMode:
    plt.savefig('04_additional/figure01.eps', format='eps')

#%% Prepare monthly data

# Create monthly overview for non-distinct funds
[monthly_overview, monthly_dfs] = msc.create_monthly_dfs(data_fundmonths)

# Create columns for counts
monthly_overview['Count_total'] = 'nan'
monthly_overview['Count_active'] = 'nan'
monthly_overview['Count_index'] = 'nan'

# Create columns for average returns
```

```
monthly_overview['Avg_ret_total'] = 'nan'
monthly_overview['Avg_ret_active'] = 'nan'
monthly_overview['Avg_ret_index'] = 'nan'

monthly_overview['Avg_ret_active_TNAVW'] = 'nan'

# Loop through data and populate
for index in monthly_overview.index:

    # Create sub-frames
    temp_active = monthly_dfs[index][monthly_dfs[index]['Index fund']==False]
    temp_index = monthly_dfs[index][monthly_dfs[index]['Index fund']==True]

    # Count number of fund classes
    monthly_overview.loc[index]['Count_total'] = len(list(monthly_dfs[index]['Index fund']))
    monthly_overview.loc[index]['Count_active'] = len(list(temp_active['Index fund']))
    monthly_overview.loc[index]['Count_index'] = len(list(temp_index['Index fund']))

    # Compute average returns
    monthly_overview.loc[index]['Avg_ret_total'] = monthly_dfs[index]['Return'].mean()
    monthly_overview.loc[index]['Avg_ret_active'] = temp_active['Return'].mean()
    monthly_overview.loc[index]['Avg_ret_index'] = temp_index['Return'].mean()

    # Compute TNAV-W returns (left as 'nan' if no NAV observations)
    temp_active.dropna(subset=['NAV'], how='all', inplace=True)
    if len(temp_active['Return'])>0:
        monthly_overview.loc[index]['Avg_ret_active_TNAVW'] =
sum(np.multiply(temp_active['Return'], temp_active['NAV']/sum(temp_active['NAV'])))


# Clear variables not to be used later
clear_variables(["index","temp_active", "temp_index"])

#%% Figure 02 [METHODS]

def myround(x, base=5):
    return base * round(x/base)

# Plot cumulative returns
def plt_cum_ret(dates, data_active, data_index, data_active_TNAVW, fig_name, period,
exportMode):

    # Get period name
    period_name = str(period[0])+  '-' + str(period[1])

    # Set plot characteristics
    lw = 0.5                                    # Linewidth
    plt.figure()
    plt.rcParams.update({'font.size': 8})       # Font size
    plt.rcParams['axes.linewidth'] = 0.5        # Chart border

    # Create plot
    plt.plot(dates, data_active, label = 'Active funds (EW)', color='k', linestyle='dashed',
linewidth=lw)
    plt.plot(dates, data_active_TNAVW, label = 'Active funds (TNAV-W)', color='darkgrey',
linestyle='dashed', linewidth=lw)
    plt.plot(dates, data_index, label = 'Index funds', color='k', linewidth=lw)

    # Set title and legend
    ax = plt.axes()
    legend = ax.legend()
    plt.setp(legend.texts,fontproperties=garamond)
    legend.get_frame().set_linewidth(lw)
    legend.get_frame().set_edgecolor("black")

    # Define x ticks
    x_ticks = []
    year = myround(period[0] - 1)
    while year <= myround(period[1]+1):
        x_ticks.append(pd.to_datetime(np.int64(str(year) + "0101"), format='%Y%m%d'))
        year = year + 5

    if period[1]==2005:
        x_ticks.append(pd.to_datetime(np.int64(str(2008) + "0101"), format='%Y%m%d'))

    # Set axis names
    plt.ylabel('Cumulative return (%)')
    plt.xlabel('Year')

    # Format axis ticks
    a=plt.gca()
    a.set_xticks(x_ticks)
    a.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
    a.yaxis.set_major_formatter(mticker.FormatStrFormatter("%.0f"))


    # Set garamond style
    msc.chart_garamond(plt,ax)

    # Save chart
    if exportMode:
        plt.savefig('04_additional/' + fig_name + '_' + period_name + '.eps', format='eps')

#%% Figure 02: Cumulative mean return

# Set periods
periods = [[1991, 2019], [1991, 2005], [2006, 2019]]

# Define starting value for charts chart data
start_value = 100

# Loop through periods
for period in periods:

    # Get first and last year
    year_start = period[0]
    year_end = period[1]

    # Define period
    month_start = pd.to_datetime(np.int64(str(year_start) + "0131"), format='%Y%m%d') # First
month of index fund returns
    month_end = pd.to_datetime(np.int64(str(year_end) + "1231"), format='%Y%m%d')

    # Create dataframe
    cum_avg_ret = monthly_overview[ (monthly_overview.index >= month_start) &
(monthly_overview.index <= month_end)]
    cum_avg_ret = cum_avg_ret[['Avg_ret_active', 'Avg_ret_index',
'Avg_ret_active_TNAVW']].copy()

    # Compute cumulative average returns
    cum_avg_ret['Cum_avg_ret_active'] = 'nan'
    cum_avg_ret['Cum_avg_ret_index'] = 'nan'
    cum_avg_ret['Cum_avg_ret_active_TNAVW'] = 'nan'
    cum_avg_ret.loc[month_start]['Cum_avg_ret_active'] = start_value
    cum_avg_ret.loc[month_start]['Cum_avg_ret_index'] = start_value
    cum_avg_ret.loc[month_start]['Cum_avg_ret_active_TNAVW'] = start_value
    for i in range(1, len(cum_avg_ret.index)):
        # print(i)
        cum_avg_ret.iloc[i]['Cum_avg_ret_active'] = cum_avg_ret.iloc[i-
1]['Cum_avg_ret_active'] * (1+ cum_avg_ret.iloc[i]['Avg_ret_active'])
        cum_avg_ret.iloc[i]['Cum_avg_ret_index'] = cum_avg_ret.iloc[i-1]['Cum_avg_ret_index']
* (1+ cum_avg_ret.iloc[i]['Avg_ret_index'])
        cum_avg_ret.iloc[i]['Cum_avg_ret_active_TNAVW'] = cum_avg_ret.iloc[i-
1]['Cum_avg_ret_active_TNAVW'] * (1+ cum_avg_ret.iloc[i]['Avg_ret_active_TNAVW'])

    # Create plot
    fig_name = 'figure02'
    dates = cum_avg_ret.index
    data_active = cum_avg_ret['Cum_avg_ret_active']- start_value
    data_index = cum_avg_ret['Cum_avg_ret_index'] - start_value
    data_active_TNAVW = cum_avg_ret['Cum_avg_ret_active_TNAVW']- start_value
    plt_cum_ret(dates, data_active, data_index, data_active_TNAVW, fig_name, period,
exportMode)

# Clear variables not to be used later
clear_variables(["i","start_value"])
```

04_additional_figure03_v2.1.py

```python
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import numpy as np
import MScThesis_v4 as msc
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker


# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Get garamond font
garamond = msc.import_garamond()

#%% Fix for share class problem
# This file only uses 'All', so we can use a simple fix
data_fundmonths['Return'] = data_fundmonths['Return_All']

#%% Self-created functions

def create_CDF2(returns, z):
    CDF = [sum((returns<=z_val)*1)/len(returns)*100 for z_val in z]
    return CDF

def create_S2(Xi_s, zs): # Trolig raskere med np.array
    CDF = [sum((1*(Xi_s<=z_val))*(z_val-Xi_s))/len(Xi_s)*100 for z_val in zs]
    return CDF

#%% Prepare

# Define periods
reg_periods = [[1991, 2019], [1991, 2005], [2006, 2019]]

# Create periods dfs
[periods_names, __, periods_dfs_index, periods_dfs_active] =
msc.create_period_dfs(data_fundmonths, reg_periods)

#%% FIGURE 03: PDFs

def create_PDF(data, bins):
    y,binEdges=np.histogram(data,bins=bins)
    bincenters = 0.5*(binEdges[1:]+binEdges[:-1])
    y = y/sum(y)
    return bincenters, y

def create_f03_PDF(period, exportMode):

    data_active = periods_dfs_active[period]['Return']*100
    data_index = periods_dfs_index[period]['Return']*100

    # Set number of bins for histograms
    n_bins_hist = 90

    # Create chart
    plt.figure()

    plt.hist(data_active,n_bins_hist,normed=True, histtype='step', label='Active funds',
color='k', linestyle='dashed')
    plt.hist(data_index,50,normed=True, histtype='step', label='Index funds', color='k')
        # Only 50 bins for index

    # Format x axis
    ax = plt.axes()
    xlim = ax.get_xlim()
    min_abs_xlim = max(abs(xlim[0]), abs(xlim[1]))
    new_xlim = [-min_abs_xlim, +min_abs_xlim]
    ax.set_xlim(new_xlim)
    ax.xaxis.set_major_formatter(mticker.PercentFormatter())

    # Format y axis
    ylim = ax.get_ylim()
    min_abs_ylim = max(abs(ylim[0]), abs(ylim[1]))
    new_ylim = [-0.005, +min_abs_ylim]
    ax.set_ylim(new_ylim)

    # Format chart
    msc.chart_garamond(plt, ax)
    plt.legend(prop=garamond, loc='upper left')

    # Set axis names
    plt.ylabel('Probability')

    # Save plot
    if exportMode:
        plt.savefig('04_additional/figure03_PDF_' + period + '.eps', format='eps')

# Generate charts
for period in periods_names:
    create_f03_PDF(period,exportMode)

#%% FIGURE 03 [METHODS]

def create_f03_CDF(period,exportMode):

    '''Create arrays of the CDFs'''

    # Get data
    data_active = periods_dfs_active[period]['Return']*100
    data_index = periods_dfs_index[period]['Return']*100

    # Create the x bins
    n_bins = 500
    data_all = list(data_index) + list(data_active) # Merge data to find min and max across
all funds
    x_min = min(data_all)
    x_max = max(data_all)
    x_bins = np.arange(x_min, x_max, (x_max-x_min)/n_bins )

    CDF_active = np.array(create_CDF2(data_active, x_bins))/100
    CDF_index = np.array(create_CDF2(data_index, x_bins))/100

    '''Create a dictionary with the background areas to be colored'''

    # Create an indicator array for wheter the active CDF is furthest to the left (i.e. worse
than index)
    CDF_active_lm = []
    for l1,l2 in zip(CDF_active,CDF_index):
        if l1 < l2:
            CDF_active_lm.append(False)
        elif l1 > l2:
            CDF_active_lm.append(True)
        elif l1 == l2:
            CDF_active_lm.append(True)

    # Create temporary lists for the x-values for the start and end of all areas to be colored
    starts = []
    ends = []

    # Set first entrance manually
    if CDF_active_lm[0] == True:
        starts.append(x_bins[0])
    # Loop through and find all starts and ends
    for i in range(1, len(CDF_active_lm)): # Loop from i=1, first is set manually

        # If this one is true and the previous is false -> a start
        if CDF_active_lm[i]==True and CDF_active_lm[i-1]==False:
            starts.append(x_bins[i])

        # If this one is false and the previous is true -> an end
        if CDF_active_lm[i]==False and CDF_active_lm[i-1]==True:
            ends.append(x_bins[i])

    # Collect starts and ends in a dictionary
    x_fill = {}
    for i in range(len(starts)):
        x_fill[starts[i]]=ends[i]

    '''Create plot'''
    plt.figure()
    plt.plot(x_bins, CDF_active, label='Active funds', color='k', linestyle='dashed')
    plt.plot(x_bins, CDF_index, label='Index funds', color='k')
    ax = plt.axes()

    # Fill background
    for x_start in x_fill.keys():
        x_area = [x_start, x_fill[x_start]]
        ax.fill_between(x_area, -0.05*100, 1.05*100, color='lightgrey')

    # Format chart
    msc.chart_garamond(plt, ax)
    plt.legend(prop=garamond, loc='upper left')
    ax.set_ylim([-0.05,1.05])

    # Set axis names
    plt.ylabel('Probability')

    # Format axis ticks
    ax.xaxis.set_major_formatter(mticker.PercentFormatter())
    ax.yaxis.set_major_formatter(mticker.StrMethodFormatter("{x:.2f}"))

    #  Make chart symmetric
    temp_xlim = ax.get_xlim()
    max_xlim =  max(abs(temp_xlim[0]), abs(temp_xlim[1]))
    ax.set_xlim([-max_xlim,max_xlim])

    # Save plot
    if exportMode:
        plt.savefig('04_additional/figure03_CDF_' + period + '.eps', format='eps')


for period in periods_names:
    create_f03_CDF(period,exportMode)


#%% FIGURE 03 [METHODS]: S-functions

# Create the CDFss
def create_CDFs(data_active, data_index):
    # Note: This can probably be done by using np.histogram(density=True)

    # Create the x bins
    n_bins = 500
    data_all = list(data_index) + list(data_active) # Merge data to find min and max across
all funds
    x_min = min(data_all)
    x_max = max(data_all)
    x_bins = np.arange(x_min, x_max, (x_max-x_min)/n_bins )

    # Create lists for the CDFs (for each x bin)
    CDF_active = []
    CDF_index = []
    for i in range(len(x_bins)):
        CDF_active.append( sum(data_active<x_bins[i])/len(data_active) )
        CDF_index.append( sum(data_index<x_bins[i])/len(data_index) )

    return CDF_active, CDF_index, x_bins
```

```
# Get "dominance area" in which the background should be filled
def get_dom_area(x_bins, y1, y2):

    # Create an indicator array for whether y1 is furthest to the left (i.e. worse than y2)
    y1_lm = []
    for l1,l2 in zip(y1,y2):
        if l1 < l2:
            y1_lm.append(False)
        elif l1 > l2:
            y1_lm.append(True)
        elif l1 == l2:
            y1_lm.append(True)

    # Create temporary lists for the x-values for the start and end of all areas to be colored
    starts = []
    ends = []

    # Set first entrance manually
    if y1_lm[0] == True:
        starts.append(x_bins[0])

    # Loop through and find all starts and ends
    for i in range(1, len(y1_lm)): # Loop from i=1, first is set manually

        # If this one is true and the previous is false -> a start
        if y1_lm[i]==True and y1_lm[i-1]==False:
            starts.append(x_bins[i])

        # If this one is false and the previous is true -> an end
        if y1_lm[i]==False and y1_lm[i-1]==True:
            ends.append(x_bins[i])

    # Collect starts and ends in a dictionary
    x_fill = {}
    for i in range(len(starts)):
        x_fill[starts[i]]=ends[i]

    return x_fill # The area where y1 is furthest to the left

# Generate charts
def create_f04(period,exportMode):
    # Get data
    # period = periods_names[0]
    data_active = periods_dfs_active[period]['Return']*100
    data_index = periods_dfs_index[period]['Return']*100

    # Create CDFs
    [CDF_active, CDF_index, x_bins] = create_CDFs(data_active, data_index)

    '''Compute and plot S-function (i.e. the integral of the CDFs)'''

    # Create the x bins
    n_bins = 500
    data_all = list(data_index) + list(data_active) # Merge data to find min and max across
all funds
    x_min = min(data_all)
    x_max = max(data_all)
    x_bins = np.arange(x_min, x_max, (x_max-x_min)/n_bins )

    int_active = np.array(create_S2(data_active, x_bins))/100
    int_index = np.array(create_S2(data_index, x_bins))/100

    # Create plot
    plt.figure()
    plt.plot(x_bins, int_active, label='Active funds', color='k', linestyle='dashed')
    plt.plot(x_bins, int_index, label='Index funds', color='k')

    # Get area where dominance occurs
    x_fill = get_dom_area(x_bins, int_active, int_index)

    # Fill background
    ax = plt.axes()
    temp_ylim = ax.get_ylim()
    for x_start in x_fill.keys():
        x_area = [x_start, x_fill[x_start]]
        ax.fill_between(x_area, temp_ylim[0]-0.1, temp_ylim[1]+0.1, color='lightgrey')
```

```python
    # Format chart
    ax.set_ylim(temp_ylim)
    msc.chart_garamond(plt, ax)
    plt.legend(prop=garamond, loc='upper left')

    # Set axis names
    plt.xlabel('Return')

    # Format axis ticks
    ax.xaxis.set_major_formatter(mticker.PercentFormatter())

    #  Make chart symmetric
    temp_xlim = ax.get_xlim()
    max_xlim =  max(abs(temp_xlim[0]), abs(temp_xlim[1]))
    ax.set_xlim([-max_xlim,max_xlim])

    # Save plot
    if exportMode:
        plt.savefig('04_additional/figure03_Sfunc_' + period + '.eps', format='eps')

# Generate charts
for period in periods_names:
    create_f04(period,exportMode)
```

04_simulation_A_v1.1.py

```
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import math
import numpy as np
import MScThesis_v4 as msc
import matplotlib.pyplot as plt
import scipy.stats as stats
import matplotlib.dates as mdates

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Get garamond font
garamond = msc.import_garamond()

#%% [HISTORICAL SIMULATION #1] Forward looking cumulative returns (EW and TNAV-W)

def simulation01(data_fm, min_inv, name, exportMode, plt_hold_per=60):

    ## min_inv: Variable not in use (after solving the share class problem)

    ## Prepare data (due to share classes causing several return and NAV col's) ##

    # Prepare index fund data
    temp_index = data_fm[data_fm['Index fund']==True].copy()
    temp_index['Return'] = temp_index['Return_All']
    temp_index['NAV'] = temp_index['NAV_All']

    # Prepare active fund data
    temp_active = data_fm[data_fm['Index fund']==False].copy()
    temp_active = temp_active[temp_active['Return_' + name].isna()==False] # Remove nan's for
min_inv group
    temp_active['Return'] = temp_active['Return_' + name]
    temp_active['NAV'] = temp_active['NAV_' + name]

    # Update data_fm
    data_fm = temp_index.copy()
    data_fm = data_fm.append(temp_active)

    ## End: Prepare data    ##

    # Define period
    first_month = pd.to_datetime(19910101, format='%Y%m%d')
    last_month = pd.to_datetime(20191231, format='%Y%m%d')

    # Create dataframes per month
    data_fm_sim1 = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <=
last_month)]
    [monthly_overview_sim1, monthly_dfs_sim1] = msc.create_monthly_dfs(data_fm_sim1)

    # Get rf (for Sharpe ratio computations)
    monthly_rf = pd.merge(left=monthly_overview_sim1, right=data_fundmonths[['Month',
'Rf']].drop_duplicates(), left_index=True, right_on='Month')
```

```
    ''' Step 1: Compute EW and TNAV-W returns'''

    # Create columns for average returns
    monthly_overview_sim1['EW_Ret_Active'] = 'nan'
    monthly_overview_sim1['EW_Ret_Index'] = 'nan'
    monthly_overview_sim1['TNAVW_Ret_Active'] = 'nan'
    monthly_overview_sim1['TNAVW_Ret_Index'] = 'nan'

    # Loop through data and populate EW and TNAV-W returns
    for index in monthly_overview_sim1.index:

        # Create temporary dataframes
        temp_active = monthly_dfs_sim1[index][monthly_dfs_sim1[index]['Index fund']==False]
        temp_index = monthly_dfs_sim1[index][monthly_dfs_sim1[index]['Index fund']==True]

        # Compute EW returns
        monthly_overview_sim1.loc[index]['EW_Ret_Active'] = temp_active['Return'].mean()
        monthly_overview_sim1.loc[index]['EW_Ret_Index'] = temp_index['Return'].mean()

        # Compute TNAV-W returns for active funds
        temp_active = temp_active[temp_active['NAV'].notna()]
        monthly_overview_sim1.loc[index]['TNAVW_Ret_Active'] =
sum(np.multiply(temp_active['Return'],temp_active['NAV']/sum(temp_active['NAV'])))

        # Compute TNAV-W returns for index funds (NOTE: Not used for anything; always EW for
index funds!)
        if len(temp_index['Unique fund name']) == 1:  # If only one observation for index,
then this return is the NAV-W
            monthly_overview_sim1.loc[index]['TNAVW_Ret_Index'] = sum(temp_index['Return'])
        else:                                           # Otherwise compute the TNAV-W
            temp_index = temp_index[temp_index['NAV'].notna()]
            monthly_overview_sim1.loc[index]['TNAVW_Ret_Index'] =
sum(np.multiply(temp_index['Return'],temp_index['NAV']/sum(temp_index['NAV'])))

    monthly_overview_sim1['Rf'] = np.array(monthly_rf['Rf'])

    ''' Step 2: Compute EW and TNAV-W cumulative returns '''

    # Set variables
    holding_periods = [12, 24, 36, 48, 60] # Must be whole years for indices to match (see
pd.DateOffset below)

    # Create dictionary to store the data
    # The dictionary is on the form (dict/cumret -> dict/holding period -> dict/start month ->
dataframe/returns)
    cumret = {}

    # Loop through holding periods
    for holding_period in holding_periods:

        # Create a new dictionary for every holding period
        cumret[holding_period] = {}

        # Loop through the months in the data
        for start_month in monthly_overview_sim1.index:

            # Compute the end month given the start month and the holding period
            end_month = start_month + pd.DateOffset(months=holding_period)

            # If end month is withing the sample period
            if end_month <= pd.to_datetime(20191231, format='%Y%m%d'):

                # Copy relevant data from monthly_overview_sim1 to a new dataframe
                cumret[holding_period][start_month] = monthly_overview_sim1[(
monthly_overview_sim1.index>=start_month) & (monthly_overview_sim1.index<end_month)].copy()

                # Compute cumulative returns based on the relevant data
                cumret[holding_period][start_month]['EW_CumRet_Active'] = (1 +
cumret[holding_period][start_month]['EW_Ret_Active']).cumprod()
                cumret[holding_period][start_month]['EW_CumRet_Index'] = (1 +
cumret[holding_period][start_month]['EW_Ret_Index']).cumprod()
                cumret[holding_period][start_month]['TNAVW_CumRet_Active'] = (1 +
cumret[holding_period][start_month]['TNAVW_Ret_Active']).cumprod()
                cumret[holding_period][start_month]['TNAVW_CumRet_Index'] = (1 +
cumret[holding_period][start_month]['TNAVW_Ret_Index']).cumprod()
                cumret[holding_period][start_month]['CumRet_Rf'] = (1 +
cumret[holding_period][start_month]['Rf']).cumprod()
```

```
    ''' Step 3: Compute summary statistics '''

    # Create figure for full sample
    colNames = ["Holding period", \
                "Weighting" ,\
                "p_active_preferred", \
                "Avg_diff_ret", \
                "Median_diff_ret", \
                "Std_diff_ret", \
                "Sharpe_ratio_diff",\
                "Observations"]
    figureS01 = pd.DataFrame(columns=colNames)

    # Create figure for sub samples
    sub_samples = {1991:2005,2006:2019}
    sub_samples_names = []   # Store name of period
    sub_samples_dt = {}      # Store start:end as datetime
    figureS01_sub = {}       # Store figure for each subperiod
    for start_year in sub_samples.keys():

        # Store name of sub period
        temp_name = str(start_year) + "-" + str(sub_samples[start_year])
        sub_samples_names.append(temp_name)

        # Add datetime-format to dictionary
        sub_samples_dt[start_year] = pd.to_datetime(str(start_year)+"0101", format='%Y%m%d')
        sub_samples_dt[sub_samples[start_year]] =
pd.to_datetime(str(sub_samples[start_year])+"1231", format='%Y%m%d')

        # Initiate output table
        figureS01_sub[temp_name] = pd.DataFrame(columns=colNames)

    # Loop through data and populate output tables
    for holding_period in holding_periods:

        # Collect all cumulative return differences
        hold_per_diff_ret_EW = np.empty(0)
        hold_per_diff_ret_TNAVW = np.empty(0)
        sub_hold_per_diff_ret_EW = {}
        sub_hold_per_diff_ret_TNAVW = {}

        # .. and Sharpe ratios
        hold_per_diff_sharpe_EW = np.empty(0)
        hold_per_diff_sharpe_TNAVW = np.empty(0)
        sub_hold_per_diff_sharpe_EW = {}
        sub_hold_per_diff_sharpe_TNAVW = {}

        for sub_name in sub_samples_names:
            sub_hold_per_diff_ret_EW[sub_name] = np.empty(0)
            sub_hold_per_diff_ret_TNAVW[sub_name] = np.empty(0)

            sub_hold_per_diff_sharpe_EW[sub_name] = np.empty(0)
            sub_hold_per_diff_sharpe_TNAVW[sub_name] = np.empty(0)

        # Loop through  all months in the data
        for start_month in monthly_overview_sim1.index:

            # Compute the end month given the start month and the holding period
            end_month = start_month + pd.DateOffset(months=holding_period)

            # If end month is within the sample period
            if end_month <= pd.to_datetime(20191231, format='%Y%m%d'):

                # Get last row of the dataframe (i.e. cumulative returns for the entire
holding period)
                end_month_data = cumret[holding_period][start_month].iloc[-1]

                # Compute difference in cumulative returns (Note: We always use EW for index)
                hold_per_diff_ret_EW = np.append(hold_per_diff_ret_EW,
end_month_data['EW_CumRet_Active']-end_month_data['EW_CumRet_Index'])
                hold_per_diff_ret_TNAVW = np.append(hold_per_diff_ret_TNAVW,
end_month_data['TNAVW_CumRet_Active']-end_month_data['EW_CumRet_Index'])

                # Compute difference in Sharpe ratio
                tmp_rf_mean =
stats.mstats.gmean(list(1+cumret[holding_period][start_month]['Rf']))-1
```

```
                tmp_index_mean = 
stats.mstats.gmean(list(1+cumret[holding_period][start_month]['EW_Ret_Index']))-1
                tmp_index_std = 
stats.gstd(list(1+cumret[holding_period][start_month]['EW_Ret_Index']))-1
                tmp_sharpe_index_EW = (tmp_index_mean - tmp_rf_mean) / tmp_index_std

                tmp_active_mean = 
stats.mstats.gmean(list(1+cumret[holding_period][start_month]['EW_Ret_Active']))-1
                tmp_active_std = 
stats.gstd(list(1+cumret[holding_period][start_month]['EW_Ret_Active']))-1
                tmp_sharpe_active_EW = (tmp_active_mean - tmp_rf_mean) / tmp_active_std
                hold_per_diff_sharpe_EW = np.append(hold_per_diff_sharpe_EW, 
tmp_sharpe_active_EW-tmp_sharpe_index_EW)

                tmp_active_mean = 
stats.mstats.gmean(list(1+cumret[holding_period][start_month]['TNAVW_Ret_Active']))-1
                tmp_active_std = 
stats.gstd(list(1+cumret[holding_period][start_month]['TNAVW_Ret_Active']))-1
                tmp_sharpe_active_TNAVW = (tmp_active_mean - tmp_rf_mean) / tmp_active_std
                hold_per_diff_sharpe_TNAVW = np.append(hold_per_diff_sharpe_TNAVW, 
tmp_sharpe_active_TNAVW-tmp_sharpe_index_EW)

            # Loop through sub-samples and do the same
            for sub_start in sub_samples.keys():
                temp_sub_start_month = sub_samples_dt[sub_start]
                temp_sub_end_month = sub_samples_dt[sub_samples[sub_start]]

                # Check that both start and end month is within the sub sample period
                if (start_month >= temp_sub_start_month) and (end_month <= 
temp_sub_end_month):
                    # print(start_smonth)
                    temp_sub_name = str(sub_start) + "-" + str(sub_samples[sub_start])

                    # Compute difference in cumulative returns (Note: We always use EW for 
index)
                    sub_hold_per_diff_ret_EW[temp_sub_name] = 
np.append(sub_hold_per_diff_ret_EW[temp_sub_name], end_month_data['EW_CumRet_Active']-
end_month_data['EW_CumRet_Index'])
                    sub_hold_per_diff_ret_TNAVW[temp_sub_name] = 
np.append(sub_hold_per_diff_ret_TNAVW[temp_sub_name], end_month_data['TNAVW_CumRet_Active']-
end_month_data['EW_CumRet_Index'])

                    # Compute difference in Sharpe ratio
                    sub_hold_per_diff_sharpe_EW[temp_sub_name] = 
np.append(sub_hold_per_diff_sharpe_EW[temp_sub_name], tmp_sharpe_active_EW-
tmp_sharpe_index_EW)
                    sub_hold_per_diff_sharpe_TNAVW[temp_sub_name] = 
np.append(sub_hold_per_diff_sharpe_TNAVW[temp_sub_name], tmp_sharpe_active_TNAVW-
tmp_sharpe_index_EW)

        # Compute and append results for full period
        figureS01 = figureS01.append({ \
                    'Holding period' : holding_period,\
                    'Weighting' : 'EW' ,\
                    'p_active_preferred' : (1*(hold_per_diff_ret_EW>0)).mean()*100 ,\
                    # 'Avg_diff_ret' : hold_per_diff_ret_EW.mean()*100/(holding_period/12) ,\
                    'Avg_diff_ret' : (( 
(1+hold_per_diff_ret_EW.mean())**(1/(holding_period/12)) ) -1) *100 ,\
                    'Median_diff_ret' : (( 
(1+np.median(hold_per_diff_ret_EW))**(1/(holding_period/12)) ) -1) *100 ,\
                    'Std_diff_ret' : 
hold_per_diff_ret_EW.std()*100/math.sqrt(holding_period/12) ,\
                    'Sharpe_ratio_diff' : hold_per_diff_sharpe_EW.mean(),\
                    'Observations' : hold_per_diff_ret_EW.size ,\
                    }, ignore_index=True)

        figureS01 = figureS01.append({ \
                    'Holding period' : holding_period,\
                    'Weighting' : 'TNAV-W' ,\
                    'p_active_preferred' : (1*(hold_per_diff_ret_TNAVW>0)).mean()*100 ,\
                    # 'Avg_diff_ret' : hold_per_diff_ret_TNAVW.mean()*100/(holding_period/12) 
,\
                    'Avg_diff_ret' : (( 
(1+hold_per_diff_ret_TNAVW.mean())**(1/(holding_period/12)) ) -1) *100 ,\
                    'Median_diff_ret' : (( 
(1+np.median(hold_per_diff_ret_TNAVW))**(1/(holding_period/12)) ) -1) *100 ,\
                    'Std_diff_ret' :
hold_per_diff_ret_TNAVW.std()*100/math.sqrt(holding_period/12) ,\
                    'Sharpe_ratio_diff' : hold_per_diff_sharpe_TNAVW.mean(),\
                    'Observations' : hold_per_diff_ret_TNAVW.size ,\
                    }, ignore_index=True)

        # Compute and append results for sub periods
        for sub_name in sub_samples_names:

            figureS01_sub[sub_name] = figureS01_sub[sub_name].append({ \
                'Holding period' : holding_period,\
                'Weighting' : 'EW' ,\
                'p_active_preferred' : (1*(sub_hold_per_diff_ret_EW[sub_name]>0)).mean()*100
,\
                # 'Avg_diff_ret' :
sub_hold_per_diff_ret_EW[sub_name].mean()*100/(holding_period/12) ,\
                'Avg_diff_ret' : ((
(1+sub_hold_per_diff_ret_EW[sub_name].mean())**(1/(holding_period/12)) ) -1) *100 ,\
                'Median_diff_ret' : ((
(1+np.median(sub_hold_per_diff_ret_EW[sub_name]))**(1/(holding_period/12)) ) -1) *100 ,\
                'Std_diff_ret' :
sub_hold_per_diff_ret_EW[sub_name].std()*100/math.sqrt(holding_period/12) ,\
                'Sharpe_ratio_diff' : sub_hold_per_diff_sharpe_EW[sub_name].mean(),\
                'Observations' : sub_hold_per_diff_ret_EW[sub_name].size ,\
                }, ignore_index=True)

            figureS01_sub[sub_name] = figureS01_sub[sub_name].append({ \
                'Holding period' : holding_period,\
                'Weighting' : 'TNAV-W' ,\
                'p_active_preferred' :
(1*(sub_hold_per_diff_ret_TNAVW[sub_name]>0)).mean()*100 ,\
                # 'Avg_diff_ret' :
sub_hold_per_diff_ret_TNAVW[sub_name].mean()*100/(holding_period/12) ,\
                'Avg_diff_ret' : ((
(1+sub_hold_per_diff_ret_TNAVW[sub_name].mean())**(1/(holding_period/12)) ) -1) *100 ,\
                'Median_diff_ret' : ((
(1+np.median(sub_hold_per_diff_ret_TNAVW[sub_name]))**(1/(holding_period/12)) ) -1) *100 ,\
                'Std_diff_ret' :
sub_hold_per_diff_ret_TNAVW[sub_name].std()*100/math.sqrt(holding_period/12) ,\
                'Sharpe_ratio_diff' : sub_hold_per_diff_sharpe_TNAVW[sub_name].mean(),\
                'Observations' : sub_hold_per_diff_ret_TNAVW[sub_name].size ,\
                }, ignore_index=True)


    ''' Step 4: Plot cumulative returns '''

    # Only for 'All'
    if name=='All':

        # Define holding_period for plot
        holding_period = plt_hold_per

        # Set linewidth for charts
        lw = 0.5

        # Initiate figures
        plt.figure("EW")
        plt.figure("TNAVW")

        # Collect min and max return (to optimize the plot)
        min_ret = 0
        max_ret = 0

        # Loop through  all months in the data
        for start_month in monthly_overview_sim1.index:

            # Compute the end month given the start month and the holding period
            end_month = start_month + pd.DateOffset(months=holding_period)

            # If end month is within the sample period
            if end_month <= pd.to_datetime(20191231, format='%Y%m%d'):

                # Add new line with 1 as starting return
                cumret[holding_period][start_month].loc[start_month-pd.DateOffset(months=1)] =
[0, 0, 0, 0, 0, 1, 1, 1, 1, 0]
                cumret[holding_period][start_month] =
cumret[holding_period][start_month].sort_index(ascending=True)
```

```
                # Compute difference in cumulative returns
                cumret[holding_period][start_month]['EW_CumRet_Diff'] = 
cumret[holding_period][start_month]['EW_CumRet_Active'] - 
cumret[holding_period][start_month]['EW_CumRet_Index']
                cumret[holding_period][start_month]['TNAVW_CumRet_Diff'] = 
cumret[holding_period][start_month]['TNAVW_CumRet_Active'] - 
cumret[holding_period][start_month]['TNAVW_CumRet_Index']

                # Update max and min
                min_ret = 
min(min_ret,min(cumret[holding_period][start_month]['EW_CumRet_Diff']),
min(cumret[holding_period][start_month]['TNAVW_CumRet_Diff']))
                max_ret = 
max(max_ret,max(cumret[holding_period][start_month]['EW_CumRet_Diff']),
max(cumret[holding_period][start_month]['TNAVW_CumRet_Diff']))

                # Add to plots
                plt.figure("EW")

plt.plot(cumret[holding_period][start_month].index,cumret[holding_period][start_month]['EW_Cum
Ret_Diff']*100, linewidth=lw)

                plt.figure("TNAVW")

plt.plot(cumret[holding_period][start_month].index,cumret[holding_period][start_month]['TNAVW_
CumRet_Diff']*100, linewidth=lw)

        # Define x ticks
        x_ticks = [pd.to_datetime(np.int64(19900101), format='%Y%m%d'), 
pd.to_datetime(np.int64(19950101), format='%Y%m%d'),\
                   pd.to_datetime(np.int64(20000101), format='%Y%m%d'), 
pd.to_datetime(np.int64(20050101), format='%Y%m%d'),\
                   pd.to_datetime(np.int64(20100101), format='%Y%m%d'), 
pd.to_datetime(np.int64(20150101), format='%Y%m%d'),\
                   pd.to_datetime(np.int64(20200101), format='%Y%m%d')]

        # Define y ticks
        def round_down(num, divisor):
            return num - (num%divisor)

        def round_base(x, base=5):
            return base * round(x/base)
        y_min = round_base(round_down(min_ret,1/5),1/5)*100
        y_max = round_base(max_ret,1/5)*100
        y_ticks = np.linspace(y_min, y_max, int((y_max-y_min)/(100/5)+1))

        # Loop through figures and format
        for figure_name in ["EW", "TNAVW"]:

            # Get figure
            plt.figure(figure_name)

            # Format x axis
            ax = plt.axes()
            ax.set_xticks(x_ticks)
            ax.set_xlim([pd.to_datetime(np.int64(19880101), 
format='%Y%m%d'),pd.to_datetime(np.int64(20220101), format='%Y%m%d')])
            ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))

            # Format y-axis
            plt.ylabel('Difference in cumulative returns (%)')
            ax.set_ylim(y_min+0.05, max_ret+0.05)
            ax.set_yticks(y_ticks)

            # Add line at y=0
            plt.axhline(y=0.0, color='k', linestyle='-', linewidth=1)

            # Set Garamond font
            msc.chart_garamond(plt,ax)

    ''' Step 5: Export tables and plots '''
    if exportMode:

        # Figures
        if name=='All':
            plt.figure("EW")
            plt.savefig('04_simulations/simulation_A_holdper_' + str(plt_hold_per) + '_EW_' + 
name + '.eps', format='eps')
            plt.figure("TNAVW")
            plt.savefig('04_simulations/simulation_A_holdper_' + str(plt_hold_per) + '_TNAVW_' 
+ name + '.eps', format='eps')

        # Tables
        figureS01.to_excel('04_simulations/simulation_A_All_' + name + '.xlsx')
        for sub_figure in figureS01_sub.keys():
            figureS01_sub[sub_figure].to_excel('04_simulations/simulation_A_' +  sub_figure + 
'_' + name + '.xlsx')

    plt.close('All')

    return figureS01, figureS01_sub

#%%
exportMode=False
figS01_small = simulation01(data_fundmonths, 999, 'Small', exportMode)
figS01_medium = simulation01(data_fundmonths, 99999, 'Medium', exportMode)
figS01_large = simulation01(data_fundmonths, 300000000, 'Large', exportMode)
figS01_all = simulation01(data_fundmonths, 'All', 'All', exportMode)

#%% For plots
for plt_hold_per in [12, 24, 36, 48, 60]:
    figS01_plot = simulation01(data_fundmonths, 'All', 'All', exportMode, plt_hold_per)
```

04_simulation_B_inc_fees_v1.2.py

```python
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:             pass

#%% Import and prepare

# Import packages
import pandas as pd
import math
import numpy as np
import MScThesis_v4 as msc

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Get garamond font
garamond = msc.import_garamond()

#%% Import and prepare fees

import os
import sys

# Global variables
file_name =  os.path.basename(sys.argv[0])  # Store name of this file
file_path =  os.path.realpath(file_name)    # Store filepath of this file
file_path =  file_path.strip(file_name)

# Import file
imp_name = file_path + "/04_simulations_inc_fees/Redemption-and-subscription-
fees_Finansportalen_v2.0.xlsx"
data_fees = pd.read_excel(imp_name, sheet_name='python')

# Create dictionaries
cost_sub = dict(zip(data_fees['Name in our data'], data_fees['subscriptionFee']))
cost_red = dict(zip(data_fees['Name in our data'], data_fees['redemptionFee']))

cost_sub_avg_index = data_fees[data_fees['Index fund']==True]['subscriptionFee'].mean()
cost_sub_avg_active = data_fees[data_fees['Index fund']==False]['subscriptionFee'].mean()

cost_red_avg_index = data_fees[data_fees['Index fund']==True]['redemptionFee'].mean()
cost_red_avg_active = data_fees[data_fees['Index fund']==False]['redemptionFee'].mean()

#%% Methods used in this file

def draw_random_fund(data, month):

    # Get number of funds in the month
    n_funds = len(data[month].keys())

    # If only one fund, return the fund name
    if n_funds==1:
        return list(data[month].keys())[0]

    # If not one fund, get a random fund name
    rnd_id = np.random.randint(0, n_funds-1)
    rnd_fund_name = list(data[month].keys())[rnd_id]

    return rnd_fund_name

#%% [HISTORICAL SIMULATION #2]
```

```
def simulation02(data_fm, year_start, year_end, name, N_sims, holding_periods, exportMode):

    # min_inv = 'All'
    data_fm = data_fundmonths.copy()

    # Define period
    first_month = pd.to_datetime(str(year_start) + "0101", format='%Y%m%d')
    last_month = pd.to_datetime(str(year_end) + "1231", format='%Y%m%d')

    ''' Step 1: Prepare data '''

    ## Prepare data (due to share classes causing several return and NAV col's) ##

    # Prepare index fund data
    temp_index = data_fm[data_fm['Index fund']==True].copy()
    temp_index['Return'] = temp_index['Return_All']
    temp_index['NAV'] = temp_index['NAV_All']

    # Prepare active fund data
    temp_active = data_fm[data_fm['Index fund']==False].copy()
    temp_active = temp_active[temp_active['Return_' + name].isna()==False] # Remove nan's for
min_inv group
    temp_active['Return'] = temp_active['Return_' + name]
    temp_active['NAV'] = temp_active['NAV_' + name]

    # Update data_fm
    data_fm = temp_index.copy()
    data_fm = data_fm.append(temp_active)

    ## End: Prepare data    ##

    # Get period data
    data_fm_sim2 = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <=
last_month)]

    # Create dataframes per month
    [monthly_overview_sim2, monthly_dfs_sim2] = msc.create_monthly_dfs(data_fm_sim2)

    # Create data structure
    data_active = {}
    data_index = {}

    for month in monthly_dfs_sim2.keys():

        # Active funds
        temp_data_active = monthly_dfs_sim2[month][monthly_dfs_sim2[month]['Index
fund']==False]
        data_active[month] = dict(zip(temp_data_active['Unique fund name'],
temp_data_active['Return']))

        # Index funds
        temp_data_index = monthly_dfs_sim2[month][monthly_dfs_sim2[month]['Index fund']==True]
        data_index[month] = dict(zip(temp_data_index['Unique fund name'],
temp_data_index['Return']))

    ''' Step 2: Simulate and store results '''

    # Collect simulation results
    sim_data_all = {}
    sim_res_all = {}
    monthly_overview_sim2['p_active_best'] = 'nan'
    monthly_overview_sim2['Avg_Diff_Ret'] = 'nan'

    # Create figure for full sample
    colNames = ["Holding period", \
                "p_active_preferred", \
                "Avg_diff_ret", \
                "Median_diff_ret", \
                "Std_diff_ret"]
    figureS02 = pd.DataFrame(columns=colNames)

    # Loop through holding periods
    for holding_period in holding_periods:

        # Create dictionary for the holding period
        sim_data_all[holding_period] = {}
        sim_res_all[holding_period] = {}

        # Get months for the given holding_period
        start_months =  monthly_overview_sim2.index[
(monthly_overview_sim2.index>=first_month)&\
                            (monthly_overview_sim2.index<= last_month-
pd.DateOffset(months=holding_period))]

        # Collect difference in returns
        diff_ret = np.empty(0)

        # Define period
        for start_month in start_months:
            end_month = start_month + pd.DateOffset(months=holding_period) +
pd.DateOffset(days=5)
            ## Plus 5 days to always capture heap years

            # Create dictionary for the start month
            sim_data_all[holding_period][start_month] = {}
            sim_res_all[holding_period][start_month] = {}

            # Collect data for chart
            diff_ret_month = np.empty(0)
            active_best_month = np.empty(0)

            # Loop for n simulations
            for sim_i in range(0, N_sims):

                # Prepare dataframe
                sim_data = monthly_overview_sim2[ (monthly_overview_sim2.index>=start_month) &
(monthly_overview_sim2.index<=end_month) ]
                sim_data['Ret_active'] = 'nan'
                sim_data['Ret_index'] = 'nan'
                sim_data['Name_active'] = 'nan'
                sim_data['Name_index'] = 'nan'
                sim_data['CumRet_active'] = 'nan'
                sim_data['CumRet_index'] = 'nan'


                # Set names to null (so the try-catch fails for the first iteration)
                sim_active_name = 0
                sim_index_name = 0

                for month in sim_data.index:

                    # Try to get return for a given fund name, or draw new fund on exception
                    try:
                        sim_active_return = data_active[month][sim_active_name]
                    except:
                        sim_active_name = draw_random_fund(data_active, month)
                        sim_active_return = data_active[month][sim_active_name]

                    try:
                        sim_index_return = data_index[month][sim_index_name]
                    except:
                        sim_index_name = draw_random_fund(data_index, month)
                        sim_index_return = data_index[month][sim_index_name]

                    # Store result
                    sim_data.loc[month]['Ret_active'] = sim_active_return
                    sim_data.loc[month]['Name_active'] = sim_active_name
                    sim_data.loc[month]['Ret_index'] = sim_index_return
                    sim_data.loc[month]['Name_index'] = sim_index_name

                # Compute cumulative returns
                sim_data['CumRet_active'] = (1 + sim_data['Ret_active']).cumprod()
                sim_data['CumRet_index'] = (1 + sim_data['Ret_index']).cumprod()

                ######
                ''' Treat fees '''

                # Get subscription fees
                try:
                    tmp_active_sub = cost_sub[sim_data.iloc[0]['Name_active']]
                except:
                    tmp_active_sub = cost_sub_avg_active
                try:
                    tmp_index_sub = cost_sub[sim_data.iloc[0]['Name_index']]
                except:
                    tmp_index_sub = cost_sub_avg_index

                # Get redemption fees
                try:
                    tmp_active_red = cost_red[sim_data.iloc[-1]['Name_active']]
                except:
                    tmp_active_red = cost_red_avg_active
                try:
                    tmp_index_red = cost_red[sim_data.iloc[-1]['Name_index']]
                except:
                    tmp_index_red = cost_red_avg_index

                # Divide by 100
                tmp_active_sub = tmp_active_sub/100
                tmp_index_sub = tmp_index_sub/100
                tmp_active_red = tmp_active_red/100
                tmp_index_red = tmp_index_red/100

                # Adjust cumulative returns (using a simple rule of multiplication)
                sim_data.iloc[-1]['CumRet_active'] = sim_data.iloc[-1]['CumRet_active'] * (1-
tmp_active_sub) * (1-tmp_active_red)
                sim_data.iloc[-1]['CumRet_index'] = sim_data.iloc[-1]['CumRet_index'] * (1-
tmp_index_sub) * (1-tmp_index_red)

                ######


                # Store result
                end_month_data = sim_data.iloc[-1]
                diff_ret = np.append(diff_ret, end_month_data['CumRet_active']-
end_month_data['CumRet_index'])
                sim_data['CumRet_Diff'] = sim_data['CumRet_active'] - sim_data['CumRet_index']

                sim_data_all[holding_period][start_month][sim_i] = sim_data.copy()
                sim_res_all[holding_period][start_month][sim_i] =
end_month_data['CumRet_active']-end_month_data['CumRet_index']

                # Collect data for chart
                diff_ret_month = np.append(diff_ret_month, diff_ret)
                active_best_month = (diff_ret>0)

            # Collect data for chart
            monthly_overview_sim2.loc[month]['p_active_best'] = np.mean(active_best_month*1)
            monthly_overview_sim2.loc[month]['Avg_Diff_Ret'] = np.mean(diff_ret_month)

            # Print status
            print("Month completed: ", start_month)

        # Compute and append results to output table
        figureS02 = figureS02.append({ \
                    'Holding period' : holding_period,\
                    'p_active_preferred' : (1*(diff_ret>0)).mean()*100 ,\
                    'Avg_diff_ret' : diff_ret.mean()*100/(holding_period/12) ,\
                    'Median_diff_ret' : np.median(diff_ret)*100/(holding_period/12) ,\
                    'Std_diff_ret' : diff_ret.std()*100/math.sqrt(holding_period/12) ,\
                    }, ignore_index=True)

        # Print status
        print("\n\n Holding period completed: ", holding_period)

    return figureS02, sim_data_all, monthly_overview_sim2

#%% Perform simulations

np.random.seed(2504)

# Initiate variables
N_sims = 100
holding_periods = [12, 24, 36, 48, 60]
min_invs = ['Small', 'Medium', 'Large', 'All']
period_starts = [1991, 2006]
period_ends =   [2005, 2019]

sim02 = {}
```

```
# Keep track of progress
counter = 0
no_sims = len(period_starts) * len(min_invs)

# Loop through periods
for period_start, period_end in zip(period_starts, period_ends):

    # Loop through minimum investment
    for min_inv in min_invs:
        name = str(period_start) + '-' + str(period_end) + '_' + str(min_inv)
        sim02[name] = simulation02(data_fundmonths, period_start, period_end, min_inv, N_sims,
holding_periods, exportMode)
        counter = counter + 1
        print("\n\n\nSimulation " + str(counter) + " done (of " + str(no_sims) + ")!\n\n\n")

''' Export results '''

# Store in one output table
colNames = ["Sample period",\
            "Minimum investment", \
            "Holding period", \
            "p_active_preferred", \
            "Avg_diff_ret", \
            "Median_diff_ret", \
            "Std_diff_ret"]
figureS02 = pd.DataFrame(columns=colNames)

# Loop through periods and append tables to output figure
for period_start, period_end in zip(period_starts, period_ends):

    # Loop through minimum investment
    for min_inv in min_invs:
        name = str(period_start) + '-' + str(period_end) + '_' + str(min_inv)
        temp = sim02[name][0]
        temp['Minimum investment'] = min_inv
        temp['Sample period'] = str(period_start) + '-' + str(period_end)

        figureS02 = figureS02.append(temp)

# Export table
if exportMode:
    figureS02.to_excel('04_simulations_inc_fees/simulation_B_Nsims' + str(N_sims) + '.xlsx')
```

04_simulation_B_v1.2.py

```
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:             pass

#%% Import and prepare

# Import packages
import pandas as pd
import math
import numpy as np
import MScThesis_v4 as msc
import matplotlib.pyplot as plt
import scipy.stats as stats
import matplotlib.dates as mdates

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Get garamond font
garamond = msc.import_garamond()

#%% Disable pandas warnings
pd.options.mode.chained_assignment = None  # default='warn'

#%% Methods used in this file

def draw_random_fund(data, month):

    # Get number of funds in the month
    n_funds = len(data[month].keys())

    # If only one fund, return the fund name
    if n_funds==1:
        return list(data[month].keys())[0]

    # If not one fund, get a random fund name
    rnd_id = np.random.randint(0, n_funds-1)
    rnd_fund_name = list(data[month].keys())[rnd_id]

    return rnd_fund_name

#%% [HISTORICAL SIMULATION #2]

def simulation02(data_fm, year_start, year_end, name, N_sims, holding_periods, exportMode):

    # min_inv = 'All'
    data_fm = data_fundmonths.copy()

    # Define period
    first_month = pd.to_datetime(str(year_start) + "0101", format='%Y%m%d')
    last_month = pd.to_datetime(str(year_end) + "1231", format='%Y%m%d')

    ''' Step 1: Prepare data '''

    ## Prepare data (due to share classes causing several return and NAV col's) ##

    # Prepare index fund data
    temp_index = data_fm[data_fm['Index fund']==True].copy()
    temp_index['Return'] = temp_index['Return_All']
    temp_index['NAV'] = temp_index['NAV_All']
    # Prepare active fund data
    temp_active = data_fm[data_fm['Index fund']==False].copy()
    temp_active = temp_active[temp_active['Return_' + name].isna()==False] # Remove nan's for
min_inv group
    temp_active['Return'] = temp_active['Return_' + name]
    temp_active['NAV'] = temp_active['NAV_' + name]

    # Update data_fm
    data_fm = temp_index.copy()
    data_fm = data_fm.append(temp_active)

    ## End: Prepare data    ##

    # Get period data
    data_fm_sim2 = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <=
last_month)]

    # Create dataframes per month
    [monthly_overview_sim2, monthly_dfs_sim2] = msc.create_monthly_dfs(data_fm_sim2)

    # Create data structure
    data_active = {}
    data_index = {}

    for month in monthly_dfs_sim2.keys():

        # Active funds
        temp_data_active = monthly_dfs_sim2[month][monthly_dfs_sim2[month]['Index
fund']==False]
        data_active[month] = dict(zip(temp_data_active['Unique fund name'],
temp_data_active['Return']))

        # Index funds
        temp_data_index = monthly_dfs_sim2[month][monthly_dfs_sim2[month]['Index fund']==True]
        data_index[month] = dict(zip(temp_data_index['Unique fund name'],
temp_data_index['Return']))

    ''' Step 2: Simulate and store results '''

    # Collect simulation results
    sim_data_all = {}
    sim_res_all = {}
    monthly_overview_sim2['p_active_best'] = 'nan'
    monthly_overview_sim2['Avg_Diff_Ret'] = 'nan'

    # Get rf (for Sharpe ratio computations)
    monthly_rf = pd.merge(left=monthly_overview_sim2, right=data_fundmonths[['Month',
'Rf']].drop_duplicates(), left_index=True, right_on='Month')

    # Create figure for full sample
    colNames = ["Holding period", \
                "p_active_preferred", \
                "Avg_diff_ret", \
                "Median_diff_ret", \
                "Std_diff_ret",\
                "Sharp_ratio_diff",\
                "%_reject_alt_hyp_active_best",\
                "%_reject_alt_hyp_index_best"]
    figureS02 = pd.DataFrame(columns=colNames)

    # Loop through holding periods
    for holding_period in holding_periods:

        # Create dictionary for the holding period
        sim_data_all[holding_period] = {}
        sim_res_all[holding_period] = {}

        # Get months for the given holding_period
        start_months =  monthly_overview_sim2.index[
(monthly_overview_sim2.index>=first_month)&\
                            (monthly_overview_sim2.index<= last_month-
pd.DateOffset(months=holding_period))]

        # Collect difference in returns
        diff_ret = np.empty(0)
        diff_sharpe_ratio = np.empty(0)
        # Collect t-stats ('active_best' means alternative hypothesis is active>index)
        t_stats_active_best = np.empty(0)
        t_stats_index_best = np.empty(0)

        # Loop through months
        for start_month in start_months:
            end_month = start_month + pd.DateOffset(months=holding_period) +
pd.DateOffset(days=5)
                # Plus 5 days to always capture heap years

            # Create dictionary for the start month
            sim_data_all[holding_period][start_month] = {}
            sim_res_all[holding_period][start_month] = {}

            # Collect data for chart
            diff_ret_month = np.empty(0)
            active_best_month = np.empty(0)

            # Loop for n simulations
            for sim_i in range(0, N_sims):

                # Prepare dataframe
                sim_data = monthly_overview_sim2[ (monthly_overview_sim2.index>=start_month) &
(monthly_overview_sim2.index<=end_month) ]
                sim_data['Ret_active'] = 'nan'
                sim_data['Ret_index'] = 'nan'
                sim_data['Name_active'] = 'nan'
                sim_data['Name_index'] = 'nan'
                sim_data['CumRet_active'] = 'nan'
                sim_data['CumRet_index'] = 'nan'


                # Set names to null (so the try-catch fails for the first iteration)
                sim_active_name = 0
                sim_index_name = 0

                # Loop through holding period
                for month in sim_data.index:

                    # Try to get return for a given fund name, or draw new fund on exception
                    try:
                        sim_active_return = data_active[month][sim_active_name]
                    except:
                        sim_active_name = draw_random_fund(data_active, month)
                        sim_active_return = data_active[month][sim_active_name]

                    try:
                        sim_index_return = data_index[month][sim_index_name]
                    except:
                        sim_index_name = draw_random_fund(data_index, month)
                        sim_index_return = data_index[month][sim_index_name]

                    # Store result
                    sim_data.loc[month]['Ret_active'] = sim_active_return
                    sim_data.loc[month]['Name_active'] = sim_active_name
                    sim_data.loc[month]['Ret_index'] = sim_index_return
                    sim_data.loc[month]['Name_index'] = sim_index_name

                # Compute cumulative returns
                sim_data['CumRet_active'] = (1 + sim_data['Ret_active']).cumprod()
                sim_data['CumRet_index'] = (1 + sim_data['Ret_index']).cumprod()

                # Store result
                end_month_data = sim_data.iloc[-1]
                diff_ret = np.append(diff_ret, end_month_data['CumRet_active']-
end_month_data['CumRet_index'])
                sim_data['CumRet_Diff'] = sim_data['CumRet_active'] - sim_data['CumRet_index']

                sim_data_all[holding_period][start_month][sim_i] = sim_data.copy()
                sim_res_all[holding_period][start_month][sim_i] =
end_month_data['CumRet_active']-end_month_data['CumRet_index']

                # Collect data for chart
                diff_ret_month = np.append(diff_ret_month, diff_ret)
                active_best_month = (diff_ret>0)

                # Get rf
                sim_data = pd.merge(left=sim_data, right=monthly_rf[['Month',
'Rf']].drop_duplicates(), left_index=True, right_on='Month')
                sim_data = sim_data.set_index('Month')

                # Compute Sharp ratio
                tmp_rf_mean = stats.mstats.gmean(list(1+sim_data['Rf']))-1

                tmp_active_mean = stats.mstats.gmean(list(1+sim_data['Ret_active']))-1
                tmp_active_std = stats.gstd(list(1+sim_data['Ret_active']))-1
                tmp_sharpe_active = (tmp_active_mean - tmp_rf_mean) / tmp_active_std

                tmp_index_mean = stats.mstats.gmean(list(1+sim_data['Ret_index']))-1
                tmp_index_std = stats.gstd(list(1+sim_data['Ret_index']))-1
                tmp_sharpe_index = (tmp_index_mean - tmp_rf_mean) / tmp_index_std

                diff_sharpe_ratio = np.append(diff_sharpe_ratio, tmp_sharpe_active-
tmp_sharpe_index)

                # Conduct t-test for active > index
                temp_t_stat = stats.ttest_rel(sim_data['Ret_active'],
sim_data['Ret_index'])[0]
                t_stats_active_best = np.append(t_stats_active_best, temp_t_stat)
                t_stats_index_best = np.append(t_stats_index_best, -temp_t_stat)

            # Collect data for chart
            monthly_overview_sim2.loc[month]['p_active_best'] = np.mean(active_best_month*1)
            monthly_overview_sim2.loc[month]['Avg_Diff_Ret'] = np.mean(diff_ret_month)

            # Print status
            print("Month completed:  ", start_month)


        # Compute fraction of t-stats that reject active > index
            # We reject if the test stat is greater than the critical value
            # 't_rejections_active_best' means rejection in favor of active (alternative
hypothesis is that active is best)
        significance_level = 0.05
        critical_value = stats.t.ppf(1-significance_level, holding_period-1)
        t_rejections_active_best =
sum(1*(t_stats_active_best>critical_value))/len(t_stats_active_best)
        t_rejections_index_best =
sum(1*(t_stats_index_best>critical_value))/len(t_stats_index_best)

        # Compute and append results to output table
        figureS02 = figureS02.append({ \
                    'Holding period' : holding_period,\
                    'p_active_preferred' : (1*(diff_ret>0)).mean()*100 ,\
                    'Avg_diff_ret' : (( (1+diff_ret.mean())**(1/(holding_period/12)) ) -1)
*100 ,\
                    'Median_diff_ret' : (( (1+np.median(diff_ret))**(1/(holding_period/12)) )
-1) *100 ,\
                    'Std_diff_ret' : diff_ret.std()*100/math.sqrt(holding_period/12) ,\
                    'Sharp_ratio_diff' : diff_sharpe_ratio.mean() ,\
                    '%_reject_alt_hyp_active_best' : t_rejections_active_best ,\
                    '%_reject_alt_hyp_index_best' : t_rejections_index_best ,\
                    }, ignore_index=True)

        # Print status
        print("\n\n Holding period completed:  ", holding_period)

    ## Note: cum_ret is blocked in the return. Not in use due to no SD and t-test
    ##       the way we initially though. cum_ret codelines are blocked above, but exists and
works.
    return figureS02, sim_data_all, monthly_overview_sim2 #, cum_ret

def plot_simulation02(sim_data_all, monthly_overview_sim2, min_inv, N_sims, holding_period,
exportMode):

    '''' Step 3: Plot results '''
    hold_per_chart = holding_period

    lw = 0.05

    fig, ax1 = plt.subplots()

    # Collect min and max return (to optimize the plot)
    min_ret = 0
    max_ret = 0

    # Get data for the holding period to be plotted
    sim_data_chart = sim_data_all[hold_per_chart]

    # Loop through each starting month
    for month in sim_data_chart.keys():

        # Loop through each simulation
        for i in range(0, N_sims):

            # Get simulation data
            temp_data = sim_data_chart[month][i]

            # Add simulation data to chart
            plt.plot(temp_data.index, temp_data['CumRet_Diff']*100, linewidth=lw)

            # Update max and min
            min_ret = min(min_ret,min(temp_data['CumRet_Diff']))
            max_ret = max(max_ret,max(temp_data['CumRet_Diff']))

    # Add mean difference
    temp_overview = monthly_overview_sim2[monthly_overview_sim2['p_active_best']!='nan']
    plt1 = ax1.plot(temp_overview.index, temp_overview['Avg_Diff_Ret']*100, linewidth = 2,
color='k', label='Return difference')

    # Add p of active > index
    ax2 = ax1.twinx()
    plt2 = ax2.plot(temp_overview.index, temp_overview['p_active_best']*100, linewidth = 2,
color='k', linestyle='dotted', label='Probability of active best')

    # Define x ticks
    x_ticks = [pd.to_datetime(np.int64(19900101), format='%Y%m%d'),
pd.to_datetime(np.int64(19950101), format='%Y%m%d'),\
               pd.to_datetime(np.int64(20000101), format='%Y%m%d'),
pd.to_datetime(np.int64(20050101), format='%Y%m%d'),\
               pd.to_datetime(np.int64(20100101), format='%Y%m%d'),
pd.to_datetime(np.int64(20150101), format='%Y%m%d'),\
               pd.to_datetime(np.int64(20200101), format='%Y%m%d')]

    # Define y ticks
    def round_down(num, divisor):
        return num - (num%divisor)

    def round_base(x, base=5):
        return base * round(x/base)
    y_min = round_base(round_down(min_ret,1/5),1/5)*100
    y_max = round_base(max_ret,1/5)*100
    y_ticks = np.linspace(y_min, y_max, 5) #int((y_max-y_min)/(10)+1))

    # Format x axex
    ax1.set_xticks(x_ticks)
    ax1.set_xlim([pd.to_datetime(np.int64(19880101),
format='%Y%m%d'),pd.to_datetime(np.int64(20220101), format='%Y%m%d')])
    ax1.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))

    ax2.set_yticks([0, 25, 50, 75, 100])

    # Format y-axis
    ax1.set_ylabel('Difference in cumulative returns (%)')
    ax2.set_ylabel('Probability (%)', rotation=270)
    ax1.set_ylim(y_min+0.05, max_ret+0.05)
    ax1.set_yticks(y_ticks)

    # Include legend
    plts = plt1+plt2
    ax1.legend(plts, [l.get_label() for l in plts], loc=0, prop=garamond)

    # Add line at y=0
    ax1.axhline(y=0.0, color='k', linestyle='-', linewidth=1)
    ax2.axhline(y=50, color='grey', linestyle='-', linewidth=0.5)

    # Set Garamond font
    msc.chart_garamond(plt,ax1)
    msc.chart_garamond(plt,ax2)

    # Export figure
    if exportMode:
        plt.savefig('04_simulations/simulation_B_' + str(min_inv) + '_Nsim' + str(N_sims) +
'_holdper' + str(holding_period) + '.eps', format='eps')

    plt.close()

#%% Perform simulations

exportMode = False
plot_output = True

# Initiate variables
N_sims = 100
np.random.seed(2504)
holding_periods = [12, 24, 36, 48, 60]
min_invs = ['Small', 'Medium', 'Large', 'All']

## Period for plot only
period_starts = [1991]
period_ends =   [2019]
##

sim02 = {}
plot_hold_per = holding_periods[-1]

# Keep track of progress
counter = 0
no_sims = len(period_starts) * len(min_invs)

# Loop through periods
for period_start, period_end in zip(period_starts, period_ends):

    # Loop through minimum investment
    for min_inv in min_invs:
        name = str(period_start) + '-' + str(period_end) + '_' + str(min_inv)
        sim02[name] = simulation02(data_fundmonths, period_start, period_end, min_inv, N_sims,
holding_periods, exportMode)
        counter = counter + 1
        print("\n\n\nSimulation " + str(counter) + " done (of " + str(no_sims) + ")!\n\n\n")

''' Export results '''

# Store in one output table
colNames = ["Sample period",\
            "Minimum investment", \
            "Holding period", \
            "p_active_preferred", \
            "Avg_diff_ret", \
            "Median_diff_ret", \
            "Std_diff_ret"]
figureS02 = pd.DataFrame(columns=colNames)

# Loop through periods and append tables to output figure
for period_start, period_end in zip(period_starts, period_ends):

    # Loop through minimum investment
    for min_inv in min_invs:
        name = str(period_start) + '-' + str(period_end) + '_' + str(min_inv)
        temp = sim02[name][0]
        temp['Minimum investment'] = min_inv
        temp['Sample period'] = str(period_start) + '-' + str(period_end)

        figureS02 = figureS02.append(temp)

# Export table
if exportMode:
    figureS02.to_excel('04_simulations/simulation_B_Nsims' + str(N_sims) + '.xlsx')


#%% Plot result

if plot_output:
    for plot_hold_per in holding_periods:
        plot_name = str(1991) + '-' + str(2019) + '_' + str('All')
        plot_simulation02(sim02[plot_name][1], sim02[plot_name][2], 'All', N_sims,
plot_hold_per, exportMode)
```

04_traditional_v4.1.py

```
#%% Regression method

def msc_regress(output_table, data_fm_in, fund_type, model_name, weight, year_start, year_end,
min_inv, return_full_all=False):

    data_fm = data_fm_in.copy()

    # Define models
    if      model_name == 'CAPM':
        X_var_names = ['Rm-Rf']
    elif    model_name == 'FF3':
        X_var_names = ['Rm-Rf', 'SMB', 'HML']
    elif    model_name == 'Carhart':
        X_var_names = ['Rm-Rf', 'SMB', 'HML', 'PR1YR']
    elif    model_name == 'FF5':
        X_var_names = ['Rm-Rf', 'SMB', 'HML', 'RMW', 'CMA']
        data_fm.dropna(subset=['RMW', 'CMA'], how='all', inplace=True)
    else:
        print('Model is not specified!')
        return None

    # Define period
    first_month = pd.to_datetime(str(year_start) + "0101", format='%Y%m%d')
    last_month = pd.to_datetime(str(year_end) + "1231", format='%Y%m%d')
    period_name = str(year_start) + "-" + str(year_end)

    ''' Step 1: Prepare data for fund type, period and min_inv '''

    # Filter out fund type and min_inv (the latter only for active)
        # Here we also incorporate the min_inv columns
    if fund_type == 'Active':
        data_fm = data_fm[ (data_fm['Index fund']==False) ]

        data_fm = data_fm[data_fm['Return_' + min_inv].isna()==False] # Remove nan's for
min_inv group
        data_fm['Return'] = data_fm['Return_' + min_inv]
        data_fm['NAV'] = data_fm['NAV_' + min_inv]

    elif fund_type == 'Index':
        data_fm = data_fm[ (data_fm['Index fund']==True) ]
        data_fm['Return'] = data_fm['Return_All']
        data_fm['NAV'] = data_fm['NAV_All']

    # Only allow EW for Index
    if (fund_type == 'Index') and (weight == 'TNAVW'):
        weight = 'EW'
        print('TNAV-W and Index chosen. Only EW allowed for index. Model uses EW!')

    # Get period data
    data_fm = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <= last_month)]

    # Filter out funds without NAV
    if fund_type == 'Active' and weight == 'TNAVW':
        data_fm = data_fm[ (data_fm['NAV']>=0) ]

    if len(data_fm['Return'])==0:
        print("No data suits the filters. Model not estimated!")
        return output_table, pd.DataFrame()

    # Create excess return column
    data_fm['Ri-Rf'] = data_fm['Return'] - data_fm['Rf']

    ''' Step 2: Filter out funds with less than 24 months of data '''

    funds = {}
    unique_funds = msc.unique(data_fm['Unique fund name'])

    for fund in unique_funds:
        temp_count = sum(data_fm['Unique fund name']==fund)
        if temp_count >= 24:
            funds[fund] = sum(data_fm['Unique fund name']==fund)

    unique_funds = list(funds.keys())
```

```
if len(unique_funds)==0:
    print("No fund has more than 24 returns. Model not estimated!")
    return output_table, pd.DataFrame()

''' Step 3: Regression: First step '''
reg_outputs = pd.DataFrame()

reg_output_df = pd.DataFrame()

temp_count_returns = 0

# Loop through funds and estimate one model per fund
for fund in unique_funds:

    # Get data for fund
    temp_data_fm = data_fm[data_fm['Unique fund name']==fund].copy()
    temp_data_fm = temp_data_fm.reset_index()
    temp_data_fm = temp_data_fm.sort_values(by=['Month'], ascending = True)

    # Get data and estimate model
    Y = (temp_data_fm['Ri-Rf'])*100
    X = temp_data_fm[X_var_names]*100

    X = sm.add_constant(X)
    model = sm.OLS(Y,X).fit(cov_type='HC3')

    ''' Store model output '''

    # Store model
    temp_output = pd.DataFrame(model.params, columns=['Coeff'])
    temp_tvalues = model.tvalues

    temp_tvalues['Rm-Rf'] = (model.params['Rm-Rf']-1)/model.HC3_se['Rm-Rf'] # H0=1 for MKT
    temp_output['t stat'] = temp_tvalues
    temp_output.loc['R2'] = [model.rsquared, 'nan']
    temp_output.loc['adj R2'] = [model.rsquared_adj, 'nan']
    temp_output.loc['N returns'] = [model.nobs, 'nan']
    temp_count_returns = temp_count_returns + model.nobs

    # Store weigts (to be used in Step 4)
    if weight == 'EW':
        temp_output.loc['Weight'] = [1/len(unique_funds), 'nan']
    elif weight == 'TNAVW':
        temp_output.loc['Weight'] = [sum(temp_data_fm['NAV']), 'nan']

    # Append to main output
    reg_outputs = reg_outputs.append(temp_output, ignore_index=False)

    ''' Create time-series of the output '''

    # Get fund data (df to be modified and appended back)
    temp_reg_output_df = temp_data_fm.copy()
    temp_reg_output_df = temp_reg_output_df.sort_values(by=['Month'], ascending = True)

    # Shift NAV columns one month and remove first month (so we use start of month NAV)
    if weight == 'TNAVW':
        temp_reg_output_df['NAV'] = temp_reg_output_df['NAV'].shift(1)
        temp_reg_output_df = temp_reg_output_df.iloc[1:][:]

    # Loop through coefficients and add as columns
    for x_var in ["const"] + X_var_names:
       temp_reg_output_df["est_" + x_var] = model.params.loc[x_var]

    temp_reg_output_df["est_AR"] = model.resid + model.params.loc['const']
        # Note: model.params is estimated on the data multiplid by 100
        # Hence, the alpha is 100x too high. But since est_AR is independent of
        # alpha (it is added and subtracted when we compute it here),
        # we do not have to adjust by dividing by 100.

    # Append to main dataframe
    reg_output_df = reg_output_df.append(temp_reg_output_df, ignore_index=False)

# Rename NAV column to reflect that it is now start of month NAV
if weight == 'TNAVW':
    reg_output_df = reg_output_df.rename({'NAV':'NAV_t-1'})

''' Step 3.1: Create time-series '''
```

```
    # Get names of estimated variables
    est_X_var_names = ["const"] + X_var_names + ["AR"]
    est_X_var_names = ["est_" + est_var for est_var in est_X_var_names]

    # Get unique months
    temp_months = msc.unique(reg_output_df['Month'])
    temp_months.sort()
    coeffs_time_series = pd.DataFrame(index=temp_months, columns=est_X_var_names)

    # Loop through months and estimate VW coeffs
    for month in temp_months:

        # Get data for the month
        temp_data_month = reg_output_df[reg_output_df['Month']==month]

        # Set weights
        if weight == 'TNAVW':
            temp_weights = temp_data_month['NAV'] / sum(temp_data_month['NAV'])
        elif weight == 'EW':
            temp_weights = 1 / len(temp_data_month['NAV'])

        # Compute weighted coeffs and add to dataframe
        temp_data_coeffs = temp_data_month[est_X_var_names]
        temp_coeffs = np.multiply(temp_weights, np.transpose(temp_data_coeffs)).sum(axis=1)
        coeffs_time_series.loc[month] = temp_coeffs

    ''' Step 4: Compute weighted regression outputs '''

    # Create output dataframe
    reg_output_cols = ['Coeff', 't stat_FM', 'p val_FM', 't stat_avg', 'p val_avg', 't
stat_Carhart', 'p val_Carhart']
    reg_output = pd.DataFrame(index=["AR"] + list(temp_output.index), columns=reg_output_cols)

    output_values = list(temp_output.index)

    # For EW
    if weight == "EW":

        # Loop through output coeffs
        for o_value in output_values:
            reg_output.loc[o_value]['Coeff'] = reg_outputs.loc[o_value]['Coeff'].mean()
#sum(np.multiply(reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True), weights_S4))

            # Skip certain coeffs for t-stat and p-val
            if o_value not in ['R2', 'adj R2', 'N returns', 'Weight']:

                # Average t stat
                reg_output.loc[o_value]['t stat_avg'] = reg_outputs.loc[o_value]['t
stat'].mean() #sum(np.multiply(reg_outputs.loc[o_value]['t stat'].reset_index(drop=True),
weights_S4))
                reg_output.loc[o_value]['p val_avg'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_avg']),
len(reg_outputs.loc[o_value]['Coeff'])-1)

                # Test versus 0 for all coeff's except MKT factor
                temp_coeff = reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True)
                if o_value == 'Rm-Rf':
                    H_0 = 1
                else:
                    H_0 = 0

                # Fama-Macbeth inspired t-stat
                reg_output.loc[o_value]['t stat_FM'] = stats.ttest_1samp(temp_coeff, H_0)[0]
                reg_output.loc[o_value]['p val_FM'] = stats.ttest_1samp(temp_coeff, H_0)[1]

        # Add AR with Carhart inspired t-test
        reg_output.loc['AR']['Coeff'] = coeffs_time_series['est_AR'].mean()
        reg_output.loc['AR']['t stat_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[0]
        reg_output.loc['AR']['p val_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[1]

    # For TNAVW
    if weight == "TNAVW":
        from statsmodels.stats.weightstats import DescrStatsW
        # Get standardizeed weights
        temp_weights = reg_outputs.loc['Weight']['Coeff'] /
sum(reg_outputs.loc['Weight']['Coeff'])
        temp_weights = np.array(temp_weights)

        # Loop through output coeffs
        for o_value in output_values:
            reg_output.loc[o_value]['Coeff'] =
sum(np.multiply(np.array(reg_outputs.loc[o_value]['Coeff']), temp_weights))

            # Skip certain coeffs for t-stat and p-val
            if o_value not in ['R2', 'adj R2', 'N returns', 'Weight']:

                # Average t stat
                reg_output.loc[o_value]['t stat_avg'] =
sum(np.multiply(np.array(reg_outputs.loc[o_value]['t stat']), temp_weights))
                reg_output.loc[o_value]['p val_avg'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_avg']),
len(reg_outputs.loc[o_value]['Coeff'])-1)

                # Test versus 0 for all coeff's except MKT factor
                temp_coeff = reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True)
                if o_value == 'Rm-Rf':
                    H_0 = 1
                else:
                    H_0 = 0

                # Fama-Macbeth inspired t-stat
                temp_coeff_weighted_stats = DescrStatsW(np.array(temp_coeff),
weights=temp_weights)
                temp_N = len(temp_coeff)
                temp_mean = temp_coeff_weighted_stats.mean
                temp_std = temp_coeff_weighted_stats.std * (temp_N/(temp_N-1))**(1/2)    #
Adjust to correct sample size adjustment

                reg_output.loc[o_value]['t stat_FM'] = (temp_mean - H_0) /
(temp_std/(temp_N**(1/2)))
                reg_output.loc[o_value]['p val_FM'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_FM']), temp_N-1)

        # Add AR with Carhart inspired t-test (AR's are already weighted, hence we compute it
similarly as EW)
        reg_output.loc['AR']['Coeff'] = coeffs_time_series['est_AR'].mean()
        reg_output.loc['AR']['t stat_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[0]
        reg_output.loc['AR']['p val_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[1]

    # Add number of funds
    reg_output.loc['N funds'] = [len(unique_funds), 'nan', 'nan', 'nan', 'nan', 'nan', 'nan']

    # Update N returns
    reg_output.loc['N returns'] = [temp_count_returns, 'nan', 'nan', 'nan', 'nan', 'nan',
'nan']

    ''' Step 5: Add descriptives '''
    reg_output.transpose()
    reg_output['Period'] = period_name
    reg_output['min_inv'] = min_inv
    reg_output['Weight'] = weight
    reg_output['Funds'] = fund_type
    reg_output['Model'] = model_name

    ''' Step 6: Merge into main table '''
    output_table = output_table.append(reg_output, ignore_index=False)

    if return_full_all==True:
        return output_table, coeffs_time_series, reg_output_df

    return output_table, coeffs_time_series

#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import numpy as np
import MScThesis_v4 as msc
import statsmodels.api as sm
import matplotlib.pyplot as plt
import scipy.stats as stats

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['YearMonth'] = [str(x)[:-2] for x in data_fundmonths['Month']]
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Import Garamond
garamond = msc.import_garamond()

# Create columns for regressions
data_fundmonths['Rm-Rf'] = data_fundmonths['Rm'] - data_fundmonths['Rf']

#%% Create CMA and RMW

# Import 5-factor for Europe
KF_eur5f = pd.read_csv("01_uncleaned_data/04 Europe_5_Factors.csv", sep=',', skiprows=lambda
x: x in [0, 2])
KF_eur5f = KF_eur5f.iloc[0:354]
temp_cols = list('eur5f_' + KF_eur5f.columns)
temp_cols[0] = 'YearMonth'
KF_eur5f.columns = temp_cols
KF_eur5f['YearMonth'] = KF_eur5f['YearMonth'].str.replace(' ', '')
KF_eur5f[list(KF_eur5f.columns)[1:]] =
KF_eur5f[list(KF_eur5f.columns)[1:]].apply(pd.to_numeric, errors='coerce', axis=1)/100

# Import Bernt Ødegaard factors
factors_raw = pd.read_excel("01_uncleaned_data/04 Factor Data Norwegian Equities_v2.xlsx")
factors_raw['YearMonth'] = factors_raw.apply(lambda x: str(x['date'])[:-2], axis = 1)
factors_raw['YearMonth'] = [str(x)[:-2] for x in factors_raw['date']]

# Merge datasets
KF_eur5f = pd.merge(left=KF_eur5f, right=data_fundmonths[['YearMonth', 'Rm-
Rf']].drop_duplicates(), on='YearMonth', how='left')
KF_eur5f = pd.merge(KF_eur5f, factors_raw, on='YearMonth', how='left')

# Regress RMW on the other three factors to create our RMW
Y = KF_eur5f['eur5f_RMW']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['RMW'] = model.resid + model.params['const']

# Regress CMA on the other three factors to create our CMA
Y = KF_eur5f['eur5f_CMA']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['CMA'] = model.resid + model.params['const']

# Merge CMA and RMW into data_fundmonths
data_fundmonths = pd.merge(data_fundmonths, KF_eur5f[['YearMonth', 'RMW', 'CMA']],
on='YearMonth', how='left')

#%% Figure T01

# Create summary table
output_figure01 = pd.DataFrame()
```

```
# Set model settings
model_names = ['CAPM', 'FF3', 'Carhart', 'FF5']
# model_names = ['FF5']
weights = ['EW', 'TNAVW']
fund_types = ['Active']

# Set sample settings
min_invs = ['All']
year_starts = [1981, 1991, 2006, 1981]
year_ends   = [1990, 2005, 2019, 2019]

# Define counter and print message
count_models = len(year_starts) * len(min_invs) * len(weights) * len(fund_types) *
len(model_names)
counter = 1
print("\nStarting estimation of", count_models, "models!\n")

# Loop to estimate models
for year_start, year_end in zip(year_starts, year_ends):

    for min_inv in min_invs:

        for weight in weights:

            for fund_type in fund_types:

                for model_name in model_names:
                    model_id = str(year_start) + "-" + str(year_end) + "_" + str(min_inv) +
"_" + weight + "_" + model_name
                    output_figure01, __ = msc_regress(output_figure01, data_fundmonths,
fund_type, model_name, weight, year_start, year_end, min_inv)
                    print("Model", counter, "of", count_models, "done!")
                    counter = counter + 1

# Export output
if exportMode:
    output_figure01.to_excel('04_traditional/figure01_data.xlsx')

#%% Figure T02

# Create summary table
output_figure02 = pd.DataFrame()

# Redefine variables
min_invs = ['Small', 'Medium', 'Large', 'All']

year_starts = [1981, 1991, 2006, 1991, 1981]
year_ends   = [1990, 2005, 2019, 2019, 2019]

coeff_time_series_active = {}

# Define counter and print message
count_models = len(year_starts) * len(min_invs) * len(weights) * len(fund_types) *
len(model_names)
counter = 1
print("\nStarting estimation of", count_models, "models!\n")

# Loop to estimate models
for year_start, year_end in zip(year_starts, year_ends):

    for min_inv in min_invs:

        for weight in weights:

            for fund_type in fund_types:

                for model_name in model_names:
                    model_id = str(year_start) + "-" + str(year_end) + "_" + str(min_inv) +
"_" + weight + "_" + model_name
                    output_figure02, coeff_time_series_active[model_id] =
msc_regress(output_figure02, data_fundmonths, fund_type, model_name, weight, year_start,
year_end, min_inv)
                    print("Model", counter, "of", count_models, "done!")
                    counter = counter + 1

# Export output
```

```
if exportMode:
    output_figure02.to_excel('04_traditional/figure02_data.xlsx')


#%% Figure T03

# Create summary table
output_figure03 = pd.DataFrame()

# Set model settings
weights = ['EW']
fund_types = ['Index']

# Set sample settings
min_invs = ['All']
year_starts = [1991, 2006, 1991]
year_ends   = [2005, 2019, 2019]

coeff_time_series_index = {}

# Define counter and print message
count_models = len(year_starts) * len(min_invs) * len(weights) * len(fund_types) *
len(model_names)
counter = 1
print("\nStarting estimation of", count_models, "models!\n")

# Loop to estimate models
for year_start, year_end in zip(year_starts, year_ends):

    for min_inv in min_invs:

        for weight in weights:

            for fund_type in fund_types:

                for model_name in model_names:
                    model_id = str(year_start) + "-" + str(year_end) + "_" + str(min_inv) +
"_" + weight + "_" + model_name
                    output_figure03, coeff_time_series_index[model_id] =
msc_regress(output_figure03, data_fundmonths, fund_type, model_name, weight, year_start,
year_end, min_inv)
                    print("Model", counter, "of", count_models, "done!")
                    counter = counter + 1

# Export output
if exportMode:
    output_figure03.to_excel('04_traditional/figure03_data.xlsx')

#%% Figure T04

# Set model settings
weights = ['EW', 'TNAVW']

# Set sample settings
min_invs = ['Small', 'Medium', 'Large', 'All']
year_starts = [1991, 2006, 1991]
year_ends   = [2005, 2019, 2019]

# Create dataset
colNames = ["Period", \
            "Min_inv", \
            "Weight" , \
            "Model", \
            "a_active", \
            "a_index", \
            "a_p", \
            "AR_active", \
            "AR_index", \
            "AR_p"]

output_figure04 = pd.DataFrame(columns=colNames)

# Loop to estimate p values
for year_start, year_end in zip(year_starts, year_ends):

    for min_inv in min_invs:
        for weight in weights:

                for model_name in model_names:

                    model_id_active = str(year_start) + "-" + str(year_end) + "_" +
str(min_inv) + "_" + weight + "_" + model_name
                    model_id_index = str(year_start) + "-" + str(year_end) + "_" + str("All")
+ "_" + "EW" + "_" + model_name

                    # Test alpha returns
                    tmp_active_a = coeff_time_series_active[model_id_active]['est_const']
                    tmp_index_a = coeff_time_series_index[model_id_index]['est_const']
                    if weight == 'TNAVW':
                        tmp_index_a = tmp_index_a[1:]
                    a_t, a_p = stats.ttest_rel(tmp_active_a, tmp_index_a)

                    # Test abnormal returns
                    tmp_active_AR = coeff_time_series_active[model_id_active]['est_AR']
                    tmp_index_AR = coeff_time_series_index[model_id_index]['est_AR']
                    if weight == 'TNAVW':
                        tmp_index_AR = tmp_index_AR[1:]

                    # Store result
                    AR_t, AR_p = stats.ttest_rel(tmp_active_AR, tmp_index_AR)
                    output_figure04 = output_figure04.append({ \
                        'Period'    : str(year_start) + "-" + str(year_end),\
                        'Model'     : model_name,\
                        'Min_inv'   : min_inv,\
                        'Weight'    : weight,\
                        'a_active'  : tmp_active_a.mean(),\
                        'a_index'   : tmp_index_a.mean(),\
                        'a_p'       : a_p,\
                        'AR_active' : tmp_active_AR.mean(),\
                        'AR_index'  : tmp_index_AR.mean(),\
                        'AR_p'      : AR_p,\
                        }, ignore_index=True)

# Export output
if exportMode:
    output_figure04.to_excel('04_traditional/figure04_data.xlsx')

#%% Figure 05

year_starts = [1991, 2006]
year_ends   = [2005, 2019]

weight = 'EW'
min_inv = 'All'

for model_name in model_names:
    F05_reg_output_index = {}
    F05_reg_output_active = {}


    # Loop to estimate models
    for year_start, year_end in zip(year_starts, year_ends):

        model_id = str(year_start) + "-" + str(year_end) + "_" + str(min_inv) + "_" + weight +
"_" + model_name

        # Index
        fund_type = 'Index'
        __, __, F05_reg_output_index[model_id] = msc_regress(output_figure03, data_fundmonths,
fund_type, model_name, weight, year_start, year_end, min_inv, return_full_all=True)

        # Active
        fund_type = 'Active'
        __, __, F05_reg_output_active[model_id] = msc_regress(output_figure03,
data_fundmonths, fund_type, model_name, weight, year_start, year_end, min_inv,
return_full_all=True)

    # Concatenate data
    F05_reg_output_full = pd.DataFrame()

    for df in F05_reg_output_index.keys():
        F05_reg_output_full = F05_reg_output_full.append(F05_reg_output_index[df])
    for df in F05_reg_output_active.keys():
        F05_reg_output_full = F05_reg_output_full.append(F05_reg_output_active[df])

    # Get sum dfs
    F05_reg_output_full_active = F05_reg_output_full[F05_reg_output_full['Index fund']==False]
    F05_reg_output_full_index = F05_reg_output_full[F05_reg_output_full['Index fund']==True]

    # Compute averages
    temp_months = msc.unique(F05_reg_output_full['Month'])
    temp_months.sort()
    F05_monthly_average = pd.DataFrame(index=temp_months, columns=['Active', 'Index'])

    for month in F05_monthly_average.index:
        F05_monthly_average.loc[month]['Active'] =
F05_reg_output_full_active[F05_reg_output_full_active['Month']==month]['est_AR'].mean()
        F05_monthly_average.loc[month]['Index'] =
F05_reg_output_full_index[F05_reg_output_full_index['Month']==month]['est_AR'].mean()


    ''' Create chart '''
    # import matplotlib.ticker as mticker
    fig = plt.figure()

    # Scatter plot of average AR
    plt.scatter(F05_monthly_average.index, F05_monthly_average['Active']/100, s=0.5, c='b',
label='Active funds')
    plt.scatter(F05_monthly_average.index, F05_monthly_average['Index']/100, s=0.5, c='r',
label = 'Index funds')

    # Add line on y=0
    plt.axhline(y=0, color='black', linestyle='-', linewidth=0.5)

    # Format chart
    ax = plt.axes()
    ax.set_ylim(-.07,.07)
    vals = ax.get_yticks()
    ax.set_yticklabels(['{:,.1%}'.format(x) for x in vals])

    plt.legend(prop=garamond)
    plt.ylabel('Abnormal return')
    msc.chart_garamond(plt, ax)

    # Save chart
    if exportMode:
        plt.savefig('04_traditional/figure05_' + model_name + '.eps', format='eps')

    plt.close()
```

05_OtherMktRet_v2.0.py

```
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import os as os
import sys
import numpy as np
import MScThesis_v4 as msc
import statsmodels.api as sm
import scipy.stats as stats

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

#%% Import new alternative market data

# Global variables
file_name =  os.path.basename(sys.argv[0])  # Store name of this file
file_path =  os.path.realpath(file_name)    # Store filepath of this file
file_path =  file_path.strip(file_name)

# Import file
imp_name = file_path + "/05_OtherMktRet/02 market_portfolios_monthly.xlsx"
mkt_returns = pd.read_excel(imp_name)

# Merge files
data_fundmonths = pd.merge(data_fundmonths, mkt_returns, how='left', left_on='Month',
right_on='date')
del data_fundmonths['date']

# Simple fix for share class problem (this file only uses the 'All' group)
data_fundmonths['Return'] = data_fundmonths['Return_All']
data_fundmonths['NAV'] = data_fundmonths['NAV_All']

#%% Import OSEBX data

# Import file
imp_name = file_path + "/05_OtherMktRet/02 OSEBX.xlsx"
mkt_OSEBX = pd.read_excel(imp_name, sheet_name='Python')

# Merge files
data_fundmonths = pd.merge(data_fundmonths, mkt_OSEBX, how='left', left_on='Month',
right_on='Date')
del data_fundmonths['Date']

# Convert months to month format (not to be done in the start due to the merging on dates)
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Create column for excess returns
data_fundmonths['Rm-Rf'] = data_fundmonths['Rm'] - data_fundmonths['Rf']
data_fundmonths['Ri-Rf'] = data_fundmonths['Return'] - data_fundmonths['Rf']


#%% Regression method (adjusted to allow for custom market return)
def msc_regress(output_table, data_fm_in, fund_type, model_name, weight, year_start, year_end,
min_inv, mkt_ret, return_full_all=False):

    data_fm = data_fm_in.copy()
```

```python
# Clear nan's
data_fm.dropna(subset=[mkt_ret], how='all', inplace=True)


# Define models
if      model_name == 'CAPM':
    X_var_names = [mkt_ret]
elif    model_name == 'FF3':
    X_var_names = [mkt_ret, 'SMB', 'HML']
elif    model_name == 'Carhart':
    X_var_names = [mkt_ret, 'SMB', 'HML', 'PR1YR']
else:
    print('Model is not specified!')
    return output_table, pd.DataFrame()

# Define period
first_month = pd.to_datetime(str(year_start) + "0101", format='%Y%m%d')
last_month = pd.to_datetime(str(year_end) + "1231", format='%Y%m%d')
period_name = str(year_start) + "-" + str(year_end)

''' Step 1: Prepare data for fund type, period and min_inv '''

# Filter out fund type and min_inv (the latter only for active)
if fund_type == 'Active':
    data_fm = data_fm[ (data_fm['Index fund']==False) ]
    if min_inv != 'All':
        data_fm = data_fm[ (data_fm['Min investment']<=min_inv) ]
elif fund_type == 'Index':
    data_fm = data_fm[ (data_fm['Index fund']==True) ]

# Get period data
data_fm = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <= last_month)]

# Filter out funds without NAV
if weight == 'TNAVW':
    data_fm = data_fm[ (data_fm['NAV']>=0) ]

if len(data_fm['Return'])==0:
    print("No data suits the filters. Model not estimated!")
    return output_table, pd.DataFrame()

''' Step 2: Filter out funds with less than 24 months of data '''

funds = {}
unique_funds = msc.unique(data_fm['Unique fund name'])

for fund in unique_funds:
    temp_count = sum(data_fm['Unique fund name']==fund)
    if temp_count >= 24:
        funds[fund] = sum(data_fm['Unique fund name']==fund)

unique_funds = list(funds.keys())

''' Step 3: Regression: First step '''
reg_outputs = pd.DataFrame()

reg_output_df = pd.DataFrame()

temp_count_returns = 0

# Loop through funds and estimate one model per fund
for fund in unique_funds:

    # Get data for fund
    temp_data_fm = data_fm[data_fm['Unique fund name']==fund].copy()
    temp_data_fm = temp_data_fm.reset_index()
    temp_data_fm = temp_data_fm.sort_values(by=['Month'], ascending = True)

    # Get data and estimate model
    Y = (temp_data_fm['Ri-Rf'])*100
    X = temp_data_fm[X_var_names]*100

    X = sm.add_constant(X)
    model = sm.OLS(Y,X).fit(cov_type='HC3')

    ''' Store model output '''
```

```
        # Store model
        temp_output = pd.DataFrame(model.params, columns=['Coeff'])
        temp_tvalues = model.tvalues
        if model_name=='FF5':
            temp_tvalues['FF5_Rm-rf'] = (model.params['FF5_Rm-rf']-1)/model.HC3_se['FF5_Rm-
rf']
        else:
            temp_tvalues[mkt_ret] = (model.params[mkt_ret]-1)/model.HC3_se[mkt_ret] # H0=1 for
MKT
        temp_output['t stat'] = temp_tvalues
        temp_output.loc['R2'] = [model.rsquared, 'nan']
        temp_output.loc['adj R2'] = [model.rsquared_adj, 'nan']
        temp_output.loc['N returns'] = [model.nobs, 'nan']
        temp_count_returns = temp_count_returns + model.nobs

        # Store weigts (to be used in Step 4)
        if weight == 'EW':
            temp_output.loc['Weight'] = [1/len(unique_funds), 'nan']
        elif weight == 'TNAVW':
            temp_output.loc['Weight'] = [sum(temp_data_fm['NAV']), 'nan']

        # Append to main output
        reg_outputs = reg_outputs.append(temp_output, ignore_index=False)

        ''' Create time-series of the output '''

        # Get fund data (df to be modified and appended back)
        temp_reg_output_df = temp_data_fm.copy()
        temp_reg_output_df = temp_reg_output_df.sort_values(by=['Month'], ascending = True)

        # Shift NAV columns one month and remove first month (so we use start of month NAV)
        if weight == 'TNAVW':
            temp_reg_output_df['NAV'] = temp_reg_output_df['NAV'].shift(1)
            temp_reg_output_df = temp_reg_output_df.iloc[1:][:]

        # Loop through coefficients and add as columns
        for x_var in ["const"] + X_var_names:
           temp_reg_output_df["est_" + x_var] = model.params.loc[x_var]

        temp_reg_output_df["est_AR"] = model.resid + model.params.loc['const']
            # Note: model.params is estimated on the data multiplid by 100
            # Hence, the alpha is 100x too high. But since est_AR is independent of
            # alpha (it is added and subtracted when we compute it here),
            # we do not have to adjust by dividing by 100.

        # Append to main dataframe
        reg_output_df = reg_output_df.append(temp_reg_output_df, ignore_index=False)

    # Rename NAV column to reflect that it is now start of month NAV
    if weight == 'TNAVW':
        reg_output_df = reg_output_df.rename({'NAV':'NAV_t-1'})

    ''' Step 3.1: Create time-series '''

    # Get names of estimated variables
    est_X_var_names = ["const"] + X_var_names + ["AR"]
    est_X_var_names = ["est_" + est_var for est_var in est_X_var_names]

    # Get unique months
    temp_months = msc.unique(reg_output_df['Month'])
    temp_months.sort()
    coeffs_time_series = pd.DataFrame(index=temp_months, columns=est_X_var_names)

    # Loop through months and estimate VW coeffs
    for month in temp_months:

        # Get data for the month
        temp_data_month = reg_output_df[reg_output_df['Month']==month]

        # Set weights
        if weight == 'TNAVW':
            temp_weights = temp_data_month['NAV'] / sum(temp_data_month['NAV'])
        elif weight == 'EW':
            temp_weights = 1 / len(temp_data_month['NAV'])

        # Compute weighted coeffs and add to dataframe
        temp_data_coeffs = temp_data_month[est_X_var_names]
        temp_coeffs = np.multiply(temp_weights, np.transpose(temp_data_coeffs)).sum(axis=1)
        coeffs_time_series.loc[month] = temp_coeffs

    ''' Step 4: Compute weighted regression outputs '''

    # Create output dataframe
    reg_output_cols = ['Coeff', 't stat_FM', 'p val_FM', 't stat_avg', 'p val_avg', 't
stat_Carhart', 'p val_Carhart']
    reg_output = pd.DataFrame(index=["AR"] + list(temp_output.index), columns=reg_output_cols)

    output_values = list(temp_output.index)

    # For EW
    if weight == "EW":

        # Loop through output coeffs
        for o_value in output_values:
            reg_output.loc[o_value]['Coeff'] = reg_outputs.loc[o_value]['Coeff'].mean()
#sum(np.multiply(reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True), weights_S4))

            # Skip certain coeffs for t-stat and p-val
            if o_value not in ['R2', 'adj R2', 'N returns', 'Weight']:

                # Average t stat
                reg_output.loc[o_value]['t stat_avg'] = reg_outputs.loc[o_value]['t
stat'].mean() #sum(np.multiply(reg_outputs.loc[o_value]['t stat'].reset_index(drop=True),
weights_S4))
                reg_output.loc[o_value]['p val_avg'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_avg']),
len(reg_outputs.loc[o_value]['Coeff'])-1)

                # Test versus 0 for all coeff's except MKT factor
                temp_coeff = reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True)
                if o_value == mkt_ret:
                    H_0 = 1
                else:
                    H_0 = 0

                # Fama-Macbeth inspired t-stat
                reg_output.loc[o_value]['t stat_FM'] = stats.ttest_1samp(temp_coeff, H_0)[0]
                reg_output.loc[o_value]['p val_FM'] = stats.ttest_1samp(temp_coeff, H_0)[1]

        # Add AR with Carhart inspired t-test
        reg_output.loc['AR']['Coeff'] = coeffs_time_series['est_AR'].mean()
        reg_output.loc['AR']['t stat_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[0]
        reg_output.loc['AR']['p val_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[1]

    # For TNAVW
    if weight == "TNAVW":
        from statsmodels.stats.weightstats import DescrStatsW

        # Get standardizeed weights
        temp_weights = reg_outputs.loc['Weight']['Coeff'] /
sum(reg_outputs.loc['Weight']['Coeff'])
        temp_weights = np.array(temp_weights)

        # Loop through output coeffs
        for o_value in output_values:
            reg_output.loc[o_value]['Coeff'] =
sum(np.multiply(np.array(reg_outputs.loc[o_value]['Coeff']), temp_weights))

            # Skip certain coeffs for t-stat and p-val
            if o_value not in ['R2', 'adj R2', 'N returns', 'Weight']:

                # Average t stat
                reg_output.loc[o_value]['t stat_avg'] =
sum(np.multiply(np.array(reg_outputs.loc[o_value]['t stat']), temp_weights))
                reg_output.loc[o_value]['p val_avg'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_avg']),
len(reg_outputs.loc[o_value]['Coeff'])-1)

                # Test versus 0 for all coeff's except MKT factor
                temp_coeff = reg_outputs.loc[o_value]['Coeff'].reset_index(drop=True)
                if o_value == mkt_ret:
                    H_0 = 1
                else:
                    H_0 = 0

                # Fama-Macbeth inspired t-stat
                temp_coeff_weighted_stats = DescrStatsW(np.array(temp_coeff),
weights=temp_weights)
                temp_N = len(temp_coeff)
                temp_mean = temp_coeff_weighted_stats.mean
                temp_std = temp_coeff_weighted_stats.std * (temp_N/(temp_N-1))**(1/2)    #
Adjust to correct sample size adjustment

                reg_output.loc[o_value]['t stat_FM'] = (temp_mean - H_0) /
(temp_std/(temp_N**(1/2)))
                reg_output.loc[o_value]['p val_FM'] =
2*stats.t.sf(np.abs(reg_output.loc[o_value]['t stat_FM']), temp_N-1)

        # Add AR with Carhart inspired t-test (AR's are already weighted, hence we compute it
similarly as EW)
        reg_output.loc['AR']['Coeff'] = coeffs_time_series['est_AR'].mean()
        reg_output.loc['AR']['t stat_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[0]
        reg_output.loc['AR']['p val_Carhart'] =
stats.ttest_1samp(coeffs_time_series['est_AR'], 0)[1]

    # Add number of funds
    reg_output.loc['N funds'] = [len(unique_funds), 'nan', 'nan', 'nan', 'nan', 'nan', 'nan']

    # Update N returns
    reg_output.loc['N returns'] = [temp_count_returns, 'nan', 'nan', 'nan', 'nan', 'nan',
'nan']

    ''' Step 5: Add descriptives '''
    reg_output.transpose()
    reg_output['Period'] = period_name
    reg_output['min_inv'] = min_inv
    reg_output['Weight'] = weight
    reg_output['Funds'] = fund_type
    reg_output['Model'] = model_name

    ''' Step 5.1: Clean output '''
    if model_name == 'FF5':
        reg_output.index = [x.strip("FF5_") for x in reg_output.index]

    ''' Step 6: Merge into main table '''
    output_table = output_table.append(reg_output, ignore_index=False)

    if return_full_all==True:
        return output_table, coeffs_time_series, reg_output_df

    return output_table, coeffs_time_series


#%%

# Set variables
reg_mkt_ret = ['Rm', 'EW', 'VW', 'Allshare', 'OBX', 'OSEBX']
reg_periods = [[1991, 2019], [1991, 2005], [2006, 2019]]

# Create new columns in data fundmonths (where rf is subtracted)
var_names = []
for mkt_ret in reg_mkt_ret:

    data_fundmonths[mkt_ret + '-rf'] = data_fundmonths[mkt_ret] - data_fundmonths['Rf']
    var_names.append(mkt_ret + '-rf')

# Create periods dfs
[periods_names, periods_dfs, __, __] = msc.create_period_dfs(data_fundmonths, reg_periods)


#%% Figure 02 (and data for Figure 03)
figure02_index = pd.DataFrame()
figure02_active = pd.DataFrame()

coeff_time_series_active = {}
coeff_time_series_index = {}

model_names = ['CAPM', 'FF3', 'Carhart']
```

```
reg_periods = [[1991, 2019]]
[periods_names, __, __, __] = msc.create_period_dfs(data_fundmonths, reg_periods)


# Define counter and print message
count_models = (len(model_names) * len(periods_names) * len(var_names))*2
counter = 2
print("\nStarting estimation of", count_models, "models!\n")

# Loop to estimate models
for period in periods_names:
    period_data = periods_dfs[period]

    # Loop through market returns
    for mkt_ret in var_names:

        for model in model_names:
            temp_name = period + '_' + mkt_ret + '_' + model

            # Estimate models
            figure02_index, coeff_time_series_index[temp_name] = msc_regress(figure02_index,
period_data, 'Index', model, 'EW', 1991, 2019, 'All', mkt_ret, return_full_all=False)
            figure02_active, coeff_time_series_active[temp_name] =
msc_regress(figure02_active, period_data, 'Active', model, 'EW', 1991, 2019, 'All', mkt_ret,
return_full_all=False)

            print("Model", counter, "of", count_models, "done!")
            counter = counter + 2
if exportMode:
    figure02_index.to_excel('05_OtherMktRet/figure02_data.xlsx')
#%% Figure 03

figure03 = pd.DataFrame()

# Loop to estimate p values
for period in periods_names:

    # Loop through market returns
    for mkt_ret in var_names:

        for model in model_names:
                model_id_active = period + '_' + mkt_ret + '_' + model
                model_id_index = period + '_' + mkt_ret + '_' + model

                # Test alpha returns
                tmp_active_a = coeff_time_series_active[model_id_active]['est_const']
                tmp_index_a = coeff_time_series_index[model_id_index]['est_const']
                a_t, a_p = stats.ttest_rel(tmp_active_a, tmp_index_a)

                # Test abnormal returns
                tmp_active_AR = coeff_time_series_active[model_id_active]['est_AR']
                tmp_index_AR = coeff_time_series_index[model_id_index]['est_AR']

                # Store result
                AR_t, AR_p = stats.ttest_rel(tmp_active_AR, tmp_index_AR)
                figure03 = figure03.append({ \
                    'Period'     : period,\
                    'Model'      : model,\
                    'Market return' : mkt_ret,\
                    'a_active'   : tmp_active_a.mean(),\
                    'a_index'    : tmp_index_a.mean(),\
                    'a_p'        : a_p,\
                    'AR_active'  : tmp_active_AR.mean(),\
                    'AR_index'   : tmp_index_AR.mean(),\
                    'AR_p'       : AR_p,\
                    }, ignore_index=True)

# Export output
if exportMode:
    figure03.to_excel('05_OtherMktRet/figure03_data.xlsx')
```

05_SSD_test_v1.1.py

```
#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Stochastic dominance

def create_CDF(returns, z):
    CDF=[]
    for value in z:
        CDF.append(sum((returns<=value)*1)/len(returns))
    return CDF

def create_CDF2(returns, z):
    CDF = [sum((returns<=z_val)*1)/len(returns) for z_val in z]
    return CDF

def supremum(x):
    return max(x)

def SD_test(G, F, bins_each, print_status, n_sim_p):

    # Get PDFs of each distribution
    [t_n_g, t_bins_g] = np.histogram(G, bins_each, normed=True)
    [t_n_f, t_bins_f] = np.histogram(F, bins_each, normed=True)

    # Collect bins
    t1 = pd.DataFrame()
    t1['bins'] = list(t_bins_g) + list(t_bins_f)
    t1.sort_values(by=['bins'], inplace=True)

    # Re-calculate PDFs with common bins
    [t_n_g, t_bins_g] = np.histogram(G, t1['bins'], normed=True)
    [t_n_f, t_bins_f] = np.histogram(F, t1['bins'], normed=True)

    ''' STEP 2: Compute the test statistics '''

    # Fix uneven lengths on bins and counts
    t_n_g = np.append(t_n_g,0)
    t_n_f = np.append(t_n_f,0)

    # Collect bins and PDFs
    t3 = pd.DataFrame()
    t3['bins'] = list(t_bins_g) + list(t_bins_f)

    t3['G_pdf'] = list(t_n_g) + [None] * len(t_n_f)
    t3['F_pdf'] = [None] * len(t_n_g) + list(t_n_f)

    t3 = t3.fillna(0)                              # Fill 0 observations on NaN's
    t3.sort_values(by=['bins'], inplace=True)    # Sort table asceending on bins

    # Compute and collect CDFs
    t3['G_cdf'] = create_CDF2(G, t3['bins'])
    t3['F_cdf'] = create_CDF2(F, t3['bins'])
    t3['diff_cdf'] = t3['G_cdf'] - t3['F_cdf']

    # Compute N and M
    N = len(F)
    M = len(G)

    # Compute and collect S functions
    t3['G_S'] = integrate.cumtrapz(t3['G_cdf'], t3['bins'], initial=0)
    t3['F_S'] = integrate.cumtrapz(t3['F_cdf'], t3['bins'], initial=0)
    t3['diff_S'] = t3['G_S'] - t3['F_S']

    # Compute test statistics for FSD and SSD
    S_hat_FSD = (N*M/(N+M))**(1/2) * supremum(t3['diff_cdf'])
    S_hat_SSD = (N*M/(N+M))**(1/2) * supremum(t3['diff_S'])
    ''' STEP 3 and 4: Simulate p-value n_sim_p times '''

    # Create lists for the binary result of the inequalities
    sim_reject_FSD = []
    sim_reject_SSD = []

    # Loop to estimate the p-value n_sim_p times
    for j in range(n_sim_p):

        # Start j-timer
        if j<3:
            timer1_start = time.time()

        # Generate N(0,1) random variable
        U = np.random.normal(0,1,N)

        # Compute the Brownian Bridge Process; Formula (4); Barret and Donald (2003) p. 79
        B = []

        # timer2_start = time.time()
        for tmp_z, tmp_F_hat in zip(t3['bins'], t3['F_cdf']):
            tmp_1 = 1*(F<=tmp_z) - tmp_F_hat
            B.append(1/math.sqrt(N)*sum( np.multiply(tmp_1, U)))

        # Check the inequalities
        sim_reject_FSD.append( 1*(supremum(B) > S_hat_FSD) )               # Formula (5);
Barret and Donald (2003) p. 79
        tmp_integral = integrate.cumtrapz(B, t3['bins'], initial=0)
        sim_reject_SSD.append( 1*(supremum(tmp_integral) > S_hat_SSD) )   # Formula (5);
Barret and Donald (2003) p. 79

        # Print progress after first j
        if j<1 and print_status:
            print("\nj-iteration no. " + str(j) + " done!")

            # Check timer and print
            timer1_elapsed = datetime.timedelta(seconds=(time.time()-timer1_start))   # Check
timer
            print("Time elapsed on j-loop: ", timer1_elapsed)
            print("Estimated time remaining of this test: ", timer1_elapsed*(n_sim_p-j)-
timer1_elapsed)

    # Estimate and store p-value
    p_FSD = sum(sim_reject_FSD)/len(sim_reject_FSD)
    p_SSD = sum(sim_reject_SSD)/len(sim_reject_SSD)

    return p_FSD, p_SSD

#%% Import and prepare

# Import packages
import pandas as pd
import math
import numpy as np
import MScThesis_v4 as msc

from scipy import integrate
import time
import datetime

# Global settings
exportMode = True   # If this is set to true, the codes generate output-files
                     # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')

# Import Garamond
garamond = msc.import_garamond()

#%% Quickfix to allow for simulating the subsamples

# Define the period
```

```
'''NOTE: Must change to correct period before the code is runned'''
#reg_periods = [[1991, 2005]]
reg_periods = [[2006, 2019]]

# Create periods dfs
[periods_names, periods_dfs, __, __] = msc.create_period_dfs(data_fundmonths, reg_periods)

data_fundmonths = periods_dfs[periods_names[0]]

#%%

print("Starting simulation for period: " + periods_names[0])

# Simulation settings
''' Update before main run'''
np.random.seed(2504)
bins_each = 750  # Target: 750
n_sim_p = 500    # Target: 500

reg_min_inv = ['Small', 'Medium', 'Large', 'All']

# Get index fund data (do not filter on minimum investment due to sample size)
data_index = data_fundmonths[data_fundmonths['Index fund']==True]['Return_All']

# Create output table
colnames = ['Minimum investment', 'Index FSD Active', 'Active FSD Index', 'Index SSD Active',
'Active SSD Index']
figure01 = pd.DataFrame(columns=colnames)

counter = 1

# Loop to estimate models
for min_inv in reg_min_inv:

    # Get data for active funds min inv group (after share class problem fix)
    data_active = data_fundmonths[ (data_fundmonths['Index fund']==False) ].copy()
    data_active = data_active[data_active['Return_' + min_inv].isna()==False] # Remove nan's
for min_inv group
    data_active = data_active['Return_' + min_inv]

    # bins_each = 500 #int(max(len(data_active), len(data_index))/2)

    # Run tests
    print("\nStarting test " + str(counter) + " of " + str(len(reg_min_inv)*2))
    counter += 1
    [p_FSD_1, p_SSD_1] = SD_test(data_index, data_active, bins_each, True, n_sim_p)
    print("\nStarting test " + str(counter) + " of " + str(len(reg_min_inv)*2))
    counter += 1
    [p_FSD_2, p_SSD_2] = SD_test(data_active, data_index, bins_each, True, n_sim_p)

    # Store results
    figure01 = figure01.append({ \
        'Minimum investment' : min_inv,\
        'Index FSD Active' : p_FSD_1,\
        'Active FSD Index' : p_FSD_2,\
        'Index SSD Active' : p_SSD_1,\
        'Active SSD Index' : p_SSD_2,\
        }, ignore_index=True)   # t stats

# Store output
if exportMode:
    figure01.to_excel('05_SSD_test/figure01_binseach' + str(bins_each) + '_nsimp' +
str(n_sim_p) + '_' + periods_names[0] + '.xlsx')
```

06_SSD_Crane&Crotty_v0.2.py

```
#%% Regression method

def msc_regress(output_table, data_fm_in, fund_type, model_name, weight, year_start, year_end,
min_inv, return_full_all=False):

    data_fm = data_fm_in.copy()

    # Define models
    if      model_name == 'CAPM':
        X_var_names = ['Rm-Rf']
    elif    model_name == 'FF3':
        X_var_names = ['Rm-Rf', 'SMB', 'HML']
    elif    model_name == 'Carhart':
        X_var_names = ['Rm-Rf', 'SMB', 'HML', 'PR1YR']
    elif    model_name == 'FF5':
        X_var_names = ['Rm-Rf', 'SMB', 'HML', 'RMW', 'CMA']
        data_fm.dropna(subset=['RMW', 'CMA'], how='all', inplace=True)
    else:
        print('Model is not specified!')
        return None

    # Define period
    first_month = pd.to_datetime(str(year_start) + "0101", format='%Y%m%d')
    last_month = pd.to_datetime(str(year_end) + "1231", format='%Y%m%d')
    period_name = str(year_start) + "-" + str(year_end)

    ''' Step 1: Prepare data for fund type, period and min_inv '''

    # Filter out fund type and min_inv (the latter only for active)
        # Here we also incorporate the min_inv columns
    if fund_type == 'Active':
        data_fm = data_fm[ (data_fm['Index fund']==False) ]

        data_fm = data_fm[data_fm['Return_' + min_inv].isna()==False] # Remove nan's for
min_inv group
        data_fm['Return'] = data_fm['Return_' + min_inv]
        data_fm['NAV'] = data_fm['NAV_' + min_inv]

    elif fund_type == 'Index':
        data_fm = data_fm[ (data_fm['Index fund']==True) ]
        data_fm['Return'] = data_fm['Return_All']
        data_fm['NAV'] = data_fm['NAV_All']

    # Only allow EW for Index
    if (fund_type == 'Index') and (weight == 'TNAVW'):
        weight = 'EW'
        print('TNAV-W and Index chosen. Only EW allowed for index. Model uses EW!')

    # Get period data
    data_fm = data_fm[(data_fm['Month'] >= first_month) & (data_fm['Month'] <= last_month)]

    # Filter out funds without NAV
    if fund_type == 'Active' and weight == 'TNAVW':
        data_fm = data_fm[ (data_fm['NAV']>=0) ]

    if len(data_fm['Return'])==0:
        print("No data suits the filters. Model not estimated!")
        return output_table, pd.DataFrame()

    # Create excess return column
    data_fm['Ri-Rf'] = data_fm['Return'] - data_fm['Rf']

    ''' Step 2: Filter out funds with less than 24 months of data '''

    funds = {}
    unique_funds = msc.unique(data_fm['Unique fund name'])

    for fund in unique_funds:
        temp_count = sum(data_fm['Unique fund name']==fund)
        if temp_count >= 24:
            funds[fund] = sum(data_fm['Unique fund name']==fund)

    unique_funds = list(funds.keys())
    if len(unique_funds)==0:
        print("No fund has more than 24 returns. Model not estimated!")
        return output_table, pd.DataFrame()

    ''' Step 3: Regression: First step '''
    reg_outputs = pd.DataFrame()

    reg_output_df = pd.DataFrame()

    temp_count_returns = 0

    # Loop through funds and estimate one model per fund
    for fund in unique_funds:

        # Get data for fund
        temp_data_fm = data_fm[data_fm['Unique fund name']==fund].copy()
        temp_data_fm = temp_data_fm.reset_index()
        temp_data_fm = temp_data_fm.sort_values(by=['Month'], ascending = True)

        # Get data and estimate model
        Y = (temp_data_fm['Ri-Rf'])*100
        X = temp_data_fm[X_var_names]*100

        X = sm.add_constant(X)
        model = sm.OLS(Y,X).fit(cov_type='HC3')

        ''' Store model output '''

        # Store model
        temp_output = pd.DataFrame(model.params, columns=['Coeff'])
        temp_tvalues = model.tvalues
        temp_tvalues['Rm-Rf'] = (model.params['Rm-Rf']-1)/model.HC3_se['Rm-Rf'] # H0=1 for MKT
        temp_output['t stat'] = temp_tvalues
        temp_output.loc['R2'] = [model.rsquared, 'nan']
        temp_output.loc['adj R2'] = [model.rsquared_adj, 'nan']
        temp_output.loc['N returns'] = [model.nobs, 'nan']
        temp_count_returns = temp_count_returns + model.nobs

        # Store weigts (to be used in Step 4)
        if weight == 'EW':
            temp_output.loc['Weight'] = [1/len(unique_funds), 'nan']
        elif weight == 'TNAVW':
            temp_output.loc['Weight'] = [sum(temp_data_fm['NAV']), 'nan']

        # Append to main output
        reg_outputs = reg_outputs.append(temp_output, ignore_index=False)


    return reg_outputs


#%% Self-created functions

# Clear chosen variables
def clear_variables(variables):
    for var in variables:
        try:
            del globals()[var]
        except:
            pass

#%% Import and prepare

# Import packages
import pandas as pd
import MScThesis_v4 as msc
import statsmodels.api as sm

# Global settings
exportMode = False  # If this is set to true, the codes generate output-files
                    # (which may overwrite existing files!)

# Import data
[data_log, data_overview, data_fundmonths] = msc.import_data('Last') # 'Last' or 'vX.Y'

# Convert months to month format
data_fundmonths['YearMonth'] = [str(x)[:-2] for x in data_fundmonths['Month']]
```

```
data_fundmonths['Month'] = pd.to_datetime(data_fundmonths['Month'].astype(str),
format='%Y%m%d')


# Import Garamond
garamond = msc.import_garamond()

# Create columns for regressions
data_fundmonths['Rm-Rf'] = data_fundmonths['Rm'] - data_fundmonths['Rf']

#%% Create CMA and RMW

# Import 5-factor for Europe
KF_eur5f = pd.read_csv("01_uncleaned_data/04 Europe_5_Factors.csv", sep=',', skiprows=lambda
x: x in [0, 2])
KF_eur5f = KF_eur5f.iloc[0:354]
temp_cols = list('eur5f_' + KF_eur5f.columns)
temp_cols[0] = 'YearMonth'
KF_eur5f.columns = temp_cols
KF_eur5f['YearMonth'] = KF_eur5f['YearMonth'].str.replace(' ', '')
KF_eur5f[list(KF_eur5f.columns)[1:]] =
KF_eur5f[list(KF_eur5f.columns)[1:]].apply(pd.to_numeric, errors='coerce', axis=1)/100

# Import Bernt Ødegaard factors
factors_raw = pd.read_excel("01_uncleaned_data/04 Factor Data Norwegian Equities_v2.xlsx")
factors_raw['YearMonth'] = factors_raw.apply(lambda x: str(x['date'])[:-2], axis = 1)
factors_raw['YearMonth'] = [str(x)[:-2] for x in factors_raw['date']]

# Merge datasets
KF_eur5f = pd.merge(left=KF_eur5f, right=data_fundmonths[['YearMonth', 'Rm-
Rf']].drop_duplicates(), on='YearMonth', how='left')
KF_eur5f = pd.merge(KF_eur5f, factors_raw, on='YearMonth', how='left')

# Regress RMW on the other three factors to create our RMW
Y = KF_eur5f['eur5f_RMW']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['RMW'] = model.resid + model.params['const']

# Regress CMA on the other three factors to create our CMA
Y = KF_eur5f['eur5f_CMA']
X = KF_eur5f[['Rm-Rf', 'SMB', 'HML']]
X = sm.add_constant(X)
model = sm.OLS(Y,X).fit(cov_type='HC3')
KF_eur5f['CMA'] = model.resid + model.params['const']

# Merge CMA and RMW into data_fundmonths
data_fundmonths = pd.merge(data_fundmonths, KF_eur5f[['YearMonth', 'RMW', 'CMA']],
on='YearMonth', how='left')


#%% Figure T01

# Create summary table
SKIP = pd.DataFrame()

output = {}

# Set model settings
model_names = ['CAPM', 'FF3', 'Carhart', 'FF5']
# model_names = ['FF5']
weights = ['EW']
fund_types = ['Active', 'Index']

# Set sample settings
min_invs = ['All']
year_starts = [1991, 2006]
year_ends   = [2005, 2019]

# Define counter and print message
count_models = len(year_starts) * len(min_invs) * len(weights) * len(fund_types) *
len(model_names)
counter = 1
print("\nStarting estimation of", count_models, "models!\n")

# Loop to estimate models
```

```
for year_start, year_end in zip(year_starts, year_ends):

    for min_inv in min_invs:

        for weight in weights:

            for fund_type in fund_types:

                for model_name in model_names:
                    model_id = fund_type + "_" + str(year_start) + "-" + str(year_end) + "_" +
str(min_inv) + "_" + weight + "_" + model_name
                    # output_figure01, __ = msc_regress(output_figure01, data_fundmonths,
fund_type, model_name, weight, year_start, year_end, min_inv)
                    temp_output = msc_regress(SKIP, data_fundmonths, fund_type, model_name,
weight, year_start, year_end, min_inv)

                    output[model_id] = temp_output.loc['const']

                    print("Model", counter, "of", count_models, "done!")
                    counter = counter + 1

# Export output
for o_table in output.keys():
    output[o_table].to_excel('06_SSD_Crane&Crotty/' + o_table + '.xlsx')
```

m05_SD_matlab_v3_1.m

Note: The code is based on the Matlab code published by Whang (2019) as discussed in Appendix G of our thesis.

```
% Choose period
period = '2006-2019';
method = "Multiplier"; % 'Recentered' or 'Multiplier'

if period == "2006-2019"
    data_fundmonths = readtable("m05_SD_sanity/Data fundmonths/2006-
2019_data_fundmonths_v1.1.xlsx", 'ReadVariableNames',true, 'ReadRowNames', true);
elseif period == "1991-2005"
    data_fundmonths = readtable("m05_SD_sanity/Data fundmonths/1991-
2005_data_fundmonths_v1.1.xlsx", 'ReadVariableNames',true, 'ReadRowNames', true);
end

% Create dummy variables
data_fundmonths.small = (data_fundmonths.Return_Small > -100);
data_fundmonths.medium = (data_fundmonths.Return_Medium > -100);
data_fundmonths.large = (data_fundmonths.Return_Large > -100);
data_fundmonths.all = (data_fundmonths.Return_All > -100);

min_inv_groups = ["Small", "Medium", "Large", "All"];

% Output table
I_FSD_A = [];
A_FSD_I = [];

I_SSD_A = [];
A_SSD_I = [];

n_active = [];
n_index = [];

method_used = cell(1,0);


%%

% Get data for index funds
data_index = data_fundmonths(data_fundmonths.IndexFund == 1,:);
data_index.Return = data_index.Return_All;

count = 0;
for i = 1:length(min_inv_groups)
       group = min_inv_groups(i);

       % Get correct data for active funds
       if group == "Small"
           temp_data = data_fundmonths(data_fundmonths.small == 1,:);
           temp_data.Return = temp_data.Return_Small;
       elseif group == "Medium"
           temp_data = data_fundmonths(data_fundmonths.medium == 1,:);
           temp_data.Return = temp_data.Return_Medium;
       elseif group == "Large"
           temp_data = data_fundmonths(data_fundmonths.large == 1,:);
           temp_data.Return = temp_data.Return_Large;
       elseif group == "All"
           temp_data = data_fundmonths;
           temp_data.Return = temp_data.Return_All;
       end

       % Get data for active funds
        data_active = temp_data(temp_data.IndexFund == 0,:);


        % Get returns
        ret_active = data_active.Return;
        ret_index = data_index.Return;

        % Store output
        I_FSD_A(end+1) = SD_test(ret_index, ret_active, 1, method);
        A_FSD_I(end+1) = SD_test(ret_active, ret_index, 1, method);
        disp("FSD DONE");

        I_SSD_A(end+1) = SD_test(ret_index, ret_active, 2, method);
        A_SSD_I(end+1) = SD_test(ret_active, ret_index, 2, method);
        disp("FSD DONE");

        n_index(end+1) = height(data_index);
        n_active(end+1) = height(data_active);

        method_used{end+1} = method;

        % Increase counter
        count = count + 1;

end

% Merge lists to output table
output = array2table(zeros(count,0));
output.group = min_inv_groups';
output.I_FSD_A = I_FSD_A';
output.A_FSD_I = A_FSD_I';

output.I_SSD_A = I_SSD_A';
output.A_SSD_I = A_SSD_I';

output.n_active = n_active';
output.n_index = n_index';

output.method = method_used';

% Store output
filename = strcat('m05_SD_sanity/results_', period , '_', method , '.xlsx')
writetable(output,filename)
%%


function f = SD_test(sample1, sample2, SDorder, bmethod)


    %--------------------------- Input --------------------------%
    B=500;
    ngrid=100;
    %------------------------ Subroutine ------------------------%
    % Measure the size of data
    N1=size(sample1,1);
    N2=size(sample2,1);
    % Construct a Support
    pooled=sort([sample1; sample2]);
    if ngrid==0
    grid=pooled;
    else
    grid=linspace(min(pooled), max(pooled), ngrid);
    end
    % Compute the Test Statistic
    operator=@(X,z)(X<= z).*(z-X).^(SDorder-1)/factorial(SDorder-1);
    rawcdf1=mean(bsxfun(operator,sample1,reshape(grid, [1,1,ngrid])),1);
    rawcdf2=mean(bsxfun(operator,sample2,reshape(grid, [1,1,ngrid])),1);
    cdf1=squeeze(rawcdf1); % ECDF
    cdf2=squeeze(rawcdf2); % ECDF
    stat=sqrt(N1*N2/(N1+N2))*max(cdf1-cdf2); % test statistic
    % Multiplier method
    if strcmp(bmethod,"Multiplier")==1
    temp1=bsxfun(operator,sample1,reshape(grid,[1,1,ngrid]))-repmat(rawcdf1, [N1,1,1]);
    temp2=bsxfun(operator,sample2,reshape(grid,[1,1,ngrid]))-repmat(rawcdf2, [N2,1,1]);
    bcdf1=sqrt(N1)*mean(repmat(temp1,[1,B,1]).*randn(N1,B,ngrid), 1);
    bcdf2=sqrt(N2)*mean(repmat(temp2,[1,B,1]).*randn(N2,B,ngrid), 1);
    lambda=N2/(N1+N2);
    bksstat=max(sqrt(lambda)*bcdf1 - sqrt(1-lambda)*bcdf2,[],3);
    % Recentered bootstrap
    elseif strcmp(bmethod,"Recentered")==1
    index1=randi(N1,N1,B);
    index2=randi(N2,N2,B);
    bsample1=sample1(index1); % bootstrap sample
    bsample2=sample2(index2); % bootstrap sample
    bcdf1=mean(bsxfun(operator,bsample1,reshape(grid,[1,1,ngrid])),1) -
repmat(rawcdf1,[1,B,1]);
```

```
    bcdf2=mean(bsxfun(operator,bsample2,reshape(grid,[1,1,ngrid])),1) -
repmat(rawcdf2,[1,B,1]);
    bksstat=sqrt(N1*N2/(N1+N2))*max(bcdf1 - bcdf2,[],3);
    % Pooled sample bootstrap
    elseif strcmp(bmethod,"Pooled")==1
    index=randi(N2+N1,B);
    index1=index(1:N1,:);
    index2=index(N1+1:N2+N1,:);
    bsample1=pooled(index1); % bootstrap sample
    bsample2=pooled(index2); % bootstrap sample
    bcdf1=mean(bsxfun(operator,bsample1,reshape(grid,[1,1,ngrid])),1);
    bcdf2=mean(bsxfun(operator,bsample2,reshape(grid,[1,1,ngrid])),1);
    bksstat=sqrt(N1*N2/(N1+N2))*max(bcdf1 - bcdf2,[],3);
    end
    %---------------------------- Output ----------------------------%
    f = sum(bksstat>stat)/B

end
```

m06_SSD_CraneCrotty_v0_4.m

Note: The code is based on the Matlab code published by Whang (2019) as discussed in Appendix G of our thesis.

```
%% Define settings
folder = "06_SSD_Crane&Crotty/"; % Import folder

periods = ["1991-2005", "2006-2019"];

models = ["CAPM", "FF3", "Carhart", "FF5"];

test_type = "t-stat"; % Set to 'alpha' or 't-stat'

method = "Multiplier"; % Computation method; 'Recentered' or 'Multiplier'

%% Create variables for output table

% Output table
o_test_name = {};
o_period = {};
o_model_name = {};
o_n_active = [];
o_n_index = [];

I_FSD_A = [];
A_FSD_I = [];

I_SSD_A = [];
A_SSD_I = [];

%% Perform SD tests

counter = 0;
for i = 1:length(periods)

    % Get period name
    period = periods(i);

    for j = 1:length(models)

        % Get model name
        model = models(j);

        % Get active data
        fp_active = strcat(folder, "Active_", period, "_All_EW_", model, ".xlsx");
        data_active = readtable(fp_active, 'ReadVariableNames',true, 'ReadRowNames', true);

        % Get index data
        fp_index = strcat(folder, "Index_", period, "_All_EW_", model, ".xlsx");
        data_index = readtable(fp_index, 'ReadVariableNames',true, 'ReadRowNames', true);

        % Get alpha or t-stat data
        if test_type == "alpha"
            SD_data_active = data_active.Coeff;
            SD_data_index = data_index.Coeff;
        elseif test_type == "t-stat"
            SD_data_active = data_active.tStat;
            SD_data_index = data_index.tStat;
        end

        % Conduct SD-test
        I_FSD_A(end+1) = SD_test(SD_data_index, SD_data_active, 1, method);
        A_FSD_I(end+1) = SD_test(SD_data_active, SD_data_index, 1, method);
        disp("FSD DONE");

        I_SSD_A(end+1) = SD_test(SD_data_index, SD_data_active, 2, method);
        A_SSD_I(end+1) = SD_test(SD_data_active, SD_data_index, 2, method);
        disp("FSD DONE");

        % Store result
        o_test_name{end+1} = test_type;
        o_period{end+1} = period;
        o_model_name{end+1} = model;
        o_n_active(end+1) = height(data_active);;
        o_n_index(end+1) = height(data_index);;

        % Increase counter
        counter = counter + 1;

    end


end

%% Merge and export data

output = array2table(zeros(counter,0));

output.test_name = o_test_name';
output.period = o_period';
output.model_name = o_model_name';
output.n_active = o_n_active';
output.n_index = o_n_index';

output.I_FSD_A = I_FSD_A';
output.A_FSD_I = A_FSD_I';

output.I_SSD_A = I_SSD_A';
output.A_SSD_I = A_SSD_I';

% Store output
fp_output = strcat(folder, test_type , '_results.xlsx')
writetable(output,fp_output)


%% Function for stochastic dominance test

function f = SD_test(sample1, sample2, SDorder, bmethod)


    %--------------------------- Input --------------------------%
    B=500;
    ngrid=100;
    %------------------------ Subroutine ------------------------%
    % Measure the size of data
    N1=size(sample1,1);
    N2=size(sample2,1);
    % Construct a Support
    pooled=sort([sample1; sample2]);
    if ngrid==0
    grid=pooled;
    else
    grid=linspace(min(pooled), max(pooled), ngrid);
    end
    % Compute the Test Statistic
    operator=@(X,z)(X<= z).*(z-X).^(SDorder-1)/factorial(SDorder-1);
    rawcdf1=mean(bsxfun(operator,sample1,reshape(grid, [1,1,ngrid])),1);
    rawcdf2=mean(bsxfun(operator,sample2,reshape(grid, [1,1,ngrid])),1);
    cdf1=squeeze(rawcdf1); % ECDF
    cdf2=squeeze(rawcdf2); % ECDF
    stat=sqrt(N1*N2/(N1+N2))*max(cdf1-cdf2); % test statistic
    % Multiplier method
    if strcmp(bmethod,"Multiplier")==1
    temp1=bsxfun(operator,sample1,reshape(grid,[1,1,ngrid]))-repmat(rawcdf1, [N1,1,1]);
    temp2=bsxfun(operator,sample2,reshape(grid,[1,1,ngrid]))-repmat(rawcdf2, [N2,1,1]);
    bcdf1=sqrt(N1)*mean(repmat(temp1,[1,B,1]).*randn(N1,B,ngrid), 1);
    bcdf2=sqrt(N2)*mean(repmat(temp2,[1,B,1]).*randn(N2,B,ngrid), 1);
    lambda=N2/(N1+N2);
    bksstat=max(sqrt(lambda)*bcdf1 - sqrt(1-lambda)*bcdf2,[],3);
    % Recentered bootstrap
    elseif strcmp(bmethod,"Recentered")==1
    index1=randi(N1,N1,B);
    index2=randi(N2,N2,B);
    bsample1=sample1(index1); % bootstrap sample
    bsample2=sample2(index2); % bootstrap sample
    bcdf1=mean(bsxfun(operator,bsample1,reshape(grid,[1,1,ngrid])),1) -
repmat(rawcdf1,[1,B,1]);
    bcdf2=mean(bsxfun(operator,bsample2,reshape(grid,[1,1,ngrid])),1) -
repmat(rawcdf2,[1,B,1]);
    bksstat=sqrt(N1*N2/(N1+N2))*max(bcdf1 - bcdf2,[],3);
    % Pooled sample bootstrap
    elseif strcmp(bmethod,"Pooled")==1
    index=randi(N2+N1,B);
    index1=index(1:N1,:);
    index2=index(N1+1:N2+N1,:);
    bsample1=pooled(index1); % bootstrap sample
    bsample2=pooled(index2); % bootstrap sample
    bcdf1=mean(bsxfun(operator,bsample1,reshape(grid,[1,1,ngrid])),1);
    bcdf2=mean(bsxfun(operator,bsample2,reshape(grid,[1,1,ngrid])),1);
    bksstat=sqrt(N1*N2/(N1+N2))*max(bcdf1 - bcdf2,[],3);
    end
    %---------------------------- Output ----------------------------%
    f = sum(bksstat>stat)/B

end
```

MScThesis_v4.py

```python
#%% Overview

# 1.0: Various functions and methods
# 2.0: Fil import/export functions and methods
# 3.0: Data manipulation functions and methods

#%% Import packages

import os as os
import sys
import pandas as pd

#%% 1.0: Various functions and methods

# Extract unique values from list
def unique(list1):
    """Returns all unique items in a list"""
    return list(set(list1))

# Extract non-unique values from list
def non_unique(list1):
    list1=list(list1)
    non_uniques = []
    for item in list1:
        if list1.count(item)>1:
            non_uniques.append(item)
    non_uniques = unique(non_uniques)
    return non_uniques

# Print and return a list with items that are not in both lists
def compare_lists(list1, list2,printList=True):
    """Compares list2 to list1 and returns any items in list1 that are not in list2"""
    difference = []             # List with differences
    for item in list1:
        if not item in list2:   # Store differences in list
            difference.append(item)
            if printList: # Print the items, if printList is not False
                print(item)
    return difference

# Get properties of Garamond font
def import_garamond():
    from matplotlib import font_manager as fm, rcParams

    # Define current filepath
    file_name =  os.path.basename(sys.argv[0])  # Store name of this file
    file_path =  os.path.realpath(file_name)    # Store filepath of this file
    file_path =  file_path.strip(file_name)

    # Get Garamont ttf-file (needs to be stored locally)
    fpath = os.path.join(rcParams["datapath"], file_path.strip(file_name)+ "/00 Fonts 
etc/garamond regular.ttf")
    prop = fm.FontProperties(fname=fpath)

    return prop

# Set chart font to garamond (except legend)
def chart_garamond(plt, ax):
    garamond = import_garamond()
    ax.xaxis.get_label().set_fontproperties(garamond)
    ax.yaxis.get_label().set_fontproperties(garamond)
    ax.title.set_fontproperties(garamond)
    for label in (ax.get_xticklabels() + ax.get_yticklabels()):
        label.set_fontproperties(garamond)

#%% 2.0: Fil import/export functions and methods

# Import datasets
def import_data(imp_version):
    """
    Import data_log, data_overview, and data_fundmonths
    Suggested usage: [data_log, data_overview, data_fundmonths] = msc.import_data('Last')
    """
    # Set filepaths
    file_name =  os.path.basename(sys.argv[0])  # Store name of this file
    file_path =  os.path.realpath(file_name)    # Store filepath of this file
    file_path =  file_path.strip(file_name)
    imp_path  =  file_path + "/01_cleaning_output"

    # Import log file
    data_log = pd.read_excel(imp_path + "/data_log.xlsx", index_col=0)

    # Set file version
    if imp_version == 'Last':
        version_no_imp = data_log.loc[0]['Value/Filename']
    if imp_version != 'Last':
        version_no_imp = imp_version

    # Import dfs
    imp_name_dfs = imp_path + "/data_overview_" + version_no_imp + ".xlsx"
    data_overview = pd.read_excel(imp_name_dfs, index_col=0)

    # Import fundmonths
    imp_name_fundmonths = imp_path + "/data_fundmonths_" + version_no_imp + ".xlsx"
    data_fundmonths = pd.read_excel(imp_name_fundmonths, index_col=0)

    return data_log, data_overview, data_fundmonths

# Export datasets
def export_data(next_version, data_overview, data_fundmonths, data_fundmonths_discarded):

    # Set file paths
    file_name =  os.path.basename(sys.argv[0])  # Store name of this file
    file_path =  os.path.realpath(file_name)    # Store filepath of this file
    file_path =  file_path.strip(file_name)
    out_path  =  file_path + "/01_cleaning_output"

    # Set version number
    data_log = pd.read_excel(out_path + "/data_log.xlsx")
    del data_log['Unnamed: 0']
    version_no_curr = data_log.loc[0]['Value/Filename']
    if next_version == 'Overwrite':
        version_no_new = version_no_curr
    if next_version == 'Version':
        version_no_new = version_no_curr[0] + str (int(version_no_curr[1])+1) + '.0'
    if next_version == 'Minor change':
        version_no_new = version_no_curr[0:3] + str (int(version_no_curr[3])+1)

    # Export files
    data_overview.to_excel(out_path + "/data_overview_"+ version_no_new + ".xlsx")
    data_fundmonths.to_excel(out_path + "/data_fundmonths_"+ version_no_new + ".xlsx")
    if isinstance(data_fundmonths_discarded, pd.DataFrame):
        data_fundmonths_discarded.to_excel(out_path + "/data_fundmonths_discarded_"+
version_no_new + ".xlsx")

    # Update log
    data_log.loc[0]['Value/Filename'] = version_no_new
    data_log.to_excel(out_path + "/data_log.xlsx")

#%% 3.0: Data manipulation functions and methods

# Create data_dfs
def create_dfs_OLD(data_overview, data_fundmonths, min_no_returns):
    '''THIS ONE IS UPDATED WITH THE METHOD BELOW; TO BE DELETED'''
    """
    Creates a dictionary with dataframes for each fund.

    Parameters
    ----------
    data_overview : pd.dataframe
        DESCRIPTION.
    data_fundmonths : pd.dataframe
        DESCRIPTION.
    min_no_returns : int
        Least number of returns required to be included in dfs

    Returns
    -------
    data_dfs : dictionary
        DESCRIPTION.
    """

    # Create dfs
    data_dfs = {}

    # Populate return data for OSE funds
    for index in data_overview.index: # For all funds

        short_tag = data_overview.loc[index,'Short tag']
        include = data_overview.loc[index,'Include']
        n_returns = data_overview.loc[index,'number_of_returns']

        # If OBI large data
        if short_tag == "OBI_large_Ticker" and include == True and n_returns>min_no_returns-1:
            temp_ticker = data_overview.loc[index,'OSE_Ticker']
            temp_unique_fund_name = data_overview.loc[index,'Unique fund name']
            data_dfs[temp_unique_fund_name] =
data_fundmonths[data_fundmonths['OSE_Ticker']==temp_ticker].copy()

        # If MS data
        if short_tag == "MS_ISIN" and include == True and n_returns>min_no_returns-1:
            temp_MS_secID = data_overview.loc[index,'MS_SecID']
            temp_unique_fund_name = data_overview.loc[index,'Unique fund name']
            data_dfs[temp_unique_fund_name] =
data_fundmonths[data_fundmonths['MS_SecID']==temp_MS_secID].copy()

    return data_dfs

# Create data_dfs
def create_dfs(data_overview, data_fundmonths, min_no_returns):
    """
    Creates a dictionary with dataframes for each fund.

    Parameters
    ----------
    data_overview : pd.dataframe
        DESCRIPTION.
    data_fundmonths : pd.dataframe
        DESCRIPTION.
    min_no_returns : int
        Least number of returns required to be included in dfs

    Returns
    -------
    data_dfs : dictionary
        DESCRIPTION.

    """

    # Create dfs
    data_dfs = {}

    if min_no_returns == 'skip':
        for index in data_overview.index: # For all funds

            # short_tag = data_overview.loc[index,'Short tag']
            include = data_overview.loc[index,'Include']

            # If fund should be included'
            if include == True:
                temp_unique_fund_name = data_overview.loc[index,'Unique fund name']
                data_dfs[temp_unique_fund_name] = data_fundmonths[data_fundmonths['Unique fund
name']==temp_unique_fund_name].copy()
        return data_dfs

    # Populate return data for OSE funds
    for index in data_overview.index: # For all funds

        # short_tag = data_overview.loc[index,'Short tag']
        include = data_overview.loc[index,'Include']
        n_returns = data_overview.loc[index,'number_of_returns']

        # If fund should be included'
        if include == True and n_returns>min_no_returns-1:
            temp_unique_fund_name = data_overview.loc[index,'Unique fund name']
            data_dfs[temp_unique_fund_name] = data_fundmonths[data_fundmonths['Unique fund
name']==temp_unique_fund_name].copy()
```

```
    return data_dfs

# Update data_fundmonths from data_dfs
def dfs_to_fundmonths(data_dfs):
    '''
    Create new data_fundmonths based on data_dfs

    Parameters
    ----------
    data_dfs : dictionary
        DESCRIPTION.

    Returns
    -------
    new_data_fundmonths : pd.dataframe
        DESCRIPTION.

    '''

    # Set column names to those of the first df in dfs
    new_data_fundmonths = pd.DataFrame(columns=data_dfs[list(data_dfs.keys())[0]].columns)

    # Append all dfs to new dataframe
    for unique_fund_name in data_dfs:
        new_data_fundmonths = new_data_fundmonths.append(data_dfs[unique_fund_name],
ignore_index=True)

    return new_data_fundmonths

# Add columns in data_overview that summarizes data_fundmonths
def summarize_fundmonths_OLD(data_overview, data_fundmonths):
    '''THIS ONE IS UPDATED WITH THE METHOD BELOW; TO BE DELETED'''
    # Create columns (or fill with nan if they exist)
    data_overview['number_of_returns'] = 'nan'
    data_overview['first_return'] = 'nan'
    data_overview['last_return'] = 'nan'
    data_overview['number_of_NAVs'] = 'nan'
    data_overview['missing_NAVs'] = 'nan'

    # Loop through data_overview
    for index in data_overview.index: # For all funds
         if data_overview.loc[index,'Short tag'] == "OBI_large_Ticker" and
data_overview.loc[index,'Include'] == True:
             temp_ticker = data_overview.loc[index, 'OSE_Ticker']
             data_overview.loc[index, 'number_of_returns'] =
data_fundmonths[data_fundmonths['OSE_Ticker']==temp_ticker].count()['Month']
             data_overview.loc[index, 'first_return'] =
data_fundmonths[data_fundmonths['OSE_Ticker']==temp_ticker]['Month'].min()
             data_overview.loc[index, 'last_return'] =
data_fundmonths[data_fundmonths['OSE_Ticker']==temp_ticker]['Month'].max()
             data_overview.loc[index, 'number_of_NAVs'] = 'nan' # We only have NAV on MS funds
             data_overview.loc[index, 'missing_NAVs'] = 'nan' # We only have NAV on MS funds


    # Populate return data for MS funds
    for index in data_overview.index: # For all funds
        if data_overview.loc[index,'Short tag'] == "MS_ISIN" and
data_overview.loc[index,'Include'] == True:
             temp_secID = data_overview.loc[index, 'MS_SecID']
             data_overview.loc[index, 'number_of_returns'] =
data_fundmonths[data_fundmonths['MS_SecID']==temp_secID].count()['Month']
             data_overview.loc[index, 'first_return'] =
data_fundmonths[data_fundmonths['MS_SecID']==temp_secID]['Month'].min()
             data_overview.loc[index, 'last_return'] =
data_fundmonths[data_fundmonths['MS_SecID']==temp_secID]['Month'].max()
             data_overview.loc[index, 'number_of_NAVs'] =
data_fundmonths[data_fundmonths['MS_SecID']==temp_secID]['NAV'].dropna().count()
             data_overview.loc[index, 'missing_NAVs'] = data_overview.loc[index,
'number_of_returns'] - data_overview.loc[index, 'number_of_NAVs']

    return data_overview

# Add columns in data_overview that summarizes data_fundmonths
def summarize_fundmonths(data_overview, data_fundmonths):

    # Create columns (or fill with nan if they exist)
    data_overview['number_of_returns'] = 'nan'
    data_overview['first_return'] = 'nan'
    data_overview['last_return'] = 'nan'
    data_overview['number_of_NAVs'] = 'nan'
    data_overview['missing_NAVs'] = 'nan'

    # Loop through data_overview
    for index in data_overview.index: # For all funds
        temp_unique_name = data_overview.loc[index, 'Unique fund name']
        data_overview.loc[index, 'number_of_returns'] =
data_fundmonths[data_fundmonths['Unique fund name']==temp_unique_name].count()['Month']
        data_overview.loc[index, 'first_return'] = data_fundmonths[data_fundmonths['Unique
fund name']==temp_unique_name]['Month'].min()
        data_overview.loc[index, 'last_return'] = data_fundmonths[data_fundmonths['Unique fund
name']==temp_unique_name]['Month'].max()

        # Fill NAV numbers for MS funds
        if data_overview.loc[index,'Short tag'] == "MS_ISIN" and
data_overview.loc[index,'Include'] == True:
            data_overview.loc[index, 'number_of_NAVs'] =
data_fundmonths[data_fundmonths['Unique fund name']==temp_unique_name]['NAV'].dropna().count()
            data_overview.loc[index, 'missing_NAVs'] = data_overview.loc[index,
'number_of_returns'] - data_overview.loc[index, 'number_of_NAVs']

    return data_overview

# Count missing values for variables of interest
def count_missing_values(data_fundmonths):
    data_missing_values = pd.DataFrame(columns=list(data_fundmonths.columns[7:16]))

    data_missing_values = data_missing_values.append(pd.Series(name='Full data'))
    data_missing_values = data_missing_values.append(pd.Series(name='Excluding 2019'))

    for col in data_missing_values.columns:
        data_missing_values.loc['Full data'][col] = data_fundmonths[col].isna().sum()
        data_missing_values.loc['Excluding 2019'][col] =
data_fundmonths.loc[data_fundmonths['Month']<20190101][col].isna().sum()

    return data_missing_values

# Remove and returns nan's from fundmonths
def fundmonths_remove_nan(data_fundmonths):
    data_fundmonths_new = data_fundmonths.dropna(subset = ['Return', 'Rm', 'Rf', 'SMB', 'HML',
'PR1YR', 'UMD'])
    index_discarded = compare_lists(data_fundmonths.index,data_fundmonths_new.index,False)
    data_fundmonths_discarded = data_fundmonths.loc[index_discarded]
    return data_fundmonths_new, data_fundmonths_discarded

# Create dataframes for the given periods
def create_period_dfs(data_fundmonths, periods):
    periods_dfs = {}
    periods_dfs_index = {}
    periods_dfs_active = {}
    periods_names = [] # Need this to keep the order (the dictionary changes the order)

    # Collect data
    for period in periods:

        # Get data
        period_start = pd.to_datetime(str(period[0]) + "01" + "01", format='%Y%m%d')
        period_end = pd.to_datetime(str(period[1]) + "12" + "31", format='%Y%m%d')
        period_data = data_fundmonths[ (data_fundmonths['Month'] >= period_start ) &
(data_fundmonths['Month'] <= period_end ) ]

        # Insert data to main dictionary
        period_name = str(period[0]) + "-" + str(period[1])
        periods_names.append(period_name)
        periods_dfs[period_name] = period_data

        # Insert data to sub dictionaries
        periods_dfs_index[period_name] = period_data[period_data['Index fund']==True]
        periods_dfs_active[period_name] = period_data[period_data['Index fund']==False]

    return periods_names, periods_dfs, periods_dfs_index, periods_dfs_active

# Create overview and dfs on a monthly level
def create_monthly_dfs(data_fundmonths):
```

```
# Create monthly overview
months = unique(data_fundmonths['Month'])
months.sort()
monthly = pd.DataFrame(index=months)

# Create monthly_dfs
monthly_dfs = {}
for month in monthly.index:
    monthly_dfs[month] = data_fundmonths[data_fundmonths['Month']==month].copy()

return monthly, monthly_dfs
```