Appendix

Data Analysis

This script implements all data processing, co-integration testing, calculations of buy and hold cumulative returns, and implementations of Bollinger Bands Statistical Arbitrage

```python
# This assignment implements the statistical analysis required for the
    705 co-integration/causality assignment

# Imports important python packages and data from data prcoecessing
import numpy as np # arithmetic operations
import pandas as pd # data analysis package
import csv as csv # read and write csvs
import random as rd # random functionality
import saspy as sas # Use saspy functionality in python
import matplotlib.pyplot as plt # Use MatLab functionality for plotting
import seaborn as sb # Imports seaborn library for use
import wrds as wrds# Wharton Research Data Services API
import pydatastream as pds # Thomas Reuters Datastream API
import yfinance as yf # Yahoo Finance API
import datetime as dt # Manipulate datetime values
import statsmodels.api as sm # Create Stats functionalities
# import johansen as jh # Ability to implement Johansen test to test
    for co-integration
import linearmodels as lp # Ability to use PooledOLS
from sklearn.linear_model import LinearRegression
from stargazer.stargazer import Stargazer
import finance_byu as fin # Python Package for Fama-MacBeth Regressions
from statsmodels.regression.rolling import RollingOLS # Use factor
    loadings
from stargazer.stargazer import Stargazer
import sympy as sy # convert latex code
import scipy as sc # Scipy packages
import tabulate as tb # Create tables in python
import itertools as it # Find combinations of lists

# This section contains useful links for mean reversion, pairs-trading
# https://letianzj.github.io/mean-reversion.html
# https://letianzj.github.io/cointegration-pairs-trading.html
# https://en.wikipedia.org/wiki/Cointegration#Engle%E2%80%93Granger_two
    -step_method

# Defines the pairs trading function for the first part of the
    assignment
# Requires cross-sectional, time-series data of stock/bond/forex
    returns
def pairs_trading(data, asset_1, asset_2):
    """[summary]

    Args:
        data ([type]): [description]
        asset_1 ([type]): [description]
        asset_2 ([type]): [description]

    Returns:
        [type]: [description]
```

```
    """
    # Start date to produce plots
    # Produces time-series plots overlaying one security with another, regression residuals
    # Add produces a one-dimensional array of residuals
    # Test both configurations
    # Initialise tstat

    # Determines suitable combinations of security pairs for pairs trading.
    # Firstly, mplements the Cointegrated Augmented Dicker-Fuller (CADF) test to determine optimal
    # Hedge ratio by linear regression against the two stocks and then tests for stationarity
    # of the residuals. CADR is also known as Engle-Granger Two-Step Method
    tstat_coint = np.inf
    # Set up defaults
    independant = [asset_1,asset_2]
    dependant = [asset_2, asset_1]
    for i in range(len(independant)):
        x = data[independant[i]].values.reshape(-1,1)
        y = data[dependant[i]].values
        lm_model = LinearRegression(copy_X=True, fit_intercept=True, normalize=False).fit(x, y) # fit() expects 2D array
        # print('pamameters: %.7f, %.7f' %(lm_model.intercept_, lm_model.coef_))
        yfit = lm_model.coef_ * data[independant[i]] + lm_model.intercept_
        res = data[dependant[i]] - yfit
        [tstat, pvalue, num_lags, num_obs, crit_values, icbest] = sm.tsa.stattools.adfuller(res, maxlag = 1)
        # print('tstat',tstat)
        if tstat < tstat_coint:
            # Update critical values
            tstat_coint = tstat
            pvalue_coint = pvalue
            num_lags_coint = num_lags
            num_obs_coint = num_obs
            crit_values_coint = crit_values
            icbest_coint = icbest
            hedge_ratio= lm_model.coef_
            inte_coint = lm_model.intercept_
            x_name = independant[i]
            y_name = dependant[i]
            data['res'] = res

    # Check if significant co-integration exists
    if tstat_coint < -2.86: # Critical Value
        status = 'Yes'
        # Plots the residuals and prices separately
        # Plots the prices
        data.plot(x='Date', y=[x_name, y_name], kind='line')
        plt.title('Time-series of Price: Independant:' + x_name + ', Dependant:' + y_name)
        plt.ylabel('Price')
        plt.xlabel('Date')
```

```python
        plt.savefig('results/regressions/'+ x_name + '-'+ y_name +'-regression.png')
        # Plots the residuals
        data.plot(x='Date', y='res', kind='line')
        plt.title('Time-series of Residuals: Independant:' + x_name + ',Dependant:' + y_name)
        plt.ylabel('Residual')
        plt.xlabel('Date')
        plt.savefig('results/residual/'+ x_name + '-'+ y_name +'-residuals.png')
        # Plots the residuals

        # Implements the Johansen test to mitigate accumulating errors in the two step process
        # Find the Hedge Ratio and Tests for Co-integration at the same time (Can be extended to
        # more than two stocks (Implement if there is time)!

        # Calculate log prices and returns for trading strategies (Check when add more stocks to the mix)
        # Prints proposed spreads
        data[x_name+'-log-price'] = np.log(data[x_name])
        data[y_name+'-log-price'] = np.log(data[y_name])
        # Standard
        data['trading-spread'] = data[y_name] - data[x_name]
        data['trading-spread-mean'] = data['trading-spread'].rolling(window=20).mean()
        data['trading-spread-std'] = data['trading-spread'].rolling(window=20).std()
        # Log (spread = log(x) - nlog(b))
        data['log-spread-price'] = data[y_name+'-log-price'] - hedge_ratio*data[x_name+'-log-price']
        data['log-spread-price-mean'] = data['log-spread-price'].rolling(window=20).mean()
        data['log-spread-price-std'] = data['log-spread-price'].rolling(window=20).std()
        # Calculates the Bollinger Bands
        # Sets scaler multipier
        scaler = 2
        # Calculates bands
        # Log Bands
        data['log-spread-price-upper'] = data['log-spread-price-mean'] + (scaler*data['log-spread-price-std'])
        data['log-spread-price-lower'] = data['log-spread-price-mean'] - (scaler*data['log-spread-price-std'])
        data['spread-price-upper'] = data['trading-spread-mean'] + (scaler*data['trading-spread-std'])
        data['spread-price-lower'] = data['trading-spread-mean'] - (scaler*data['trading-spread-std'])

        # Plot Log Price with Bollinger Bands
        data.plot(x='Date', y=['log-spread-price','log-spread-price-mean','log-spread-price-upper','log-spread-price-lower'], kind='line')
        plt.title('20 Moving Average Log Spread with Bollinger Bands:'+x_name+'-'+y_name)
        plt.ylabel('Spreads')
        plt.xlabel('Date')
```

```
        plt.legend(loc=2)
        plt.savefig('results/logspreads/' +x_name+'-'+y_name+'.png')

        # Plot Price Spread with Bollinger Bands
         # Plot Log Price with Bollinger Bands
        # data.plot(x='Date', y=['trading-spread','trading-spread-mean','spread-price-upper','spread-price-lower'], kind='line')
        # plt.title('20 Moving Average Spread with Bollinger Bands:'+x_name+'-'+y_name)
        # plt.ylabel('Spreads')
        # plt.xlabel('Date')
        # plt.legend(loc=2)
        # plt.savefig('charts/spread-' +x_name+'-'+y_name+'.png')

        # Implements Bollinger Trading Strategy (Assumes only trading on the spread, not accounting for transactions costs)
        # Initialise size of trade (Assumes order size of 1000, no current positions
        initial_capital = 1000
        money_at_risk_percentage = 0.01
        cents_at_risk = 0.10
        # Equation to determine order size
        order_size = initial_capital * money_at_risk_percentage / cents_at_risk
        capital = initial_capital # Time zero
        hr = hedge_ratio
        x_name_current_size = 0
        y_name_current_size = 0

        # Set lagged variables, positions and capital
        data['lagged-'+x_name] = data[x_name].shift(1)
        data['lagged-'+y_name] = data[y_name].shift(1)
        data['lagged-log-spread-price'] = data['log-spread-price'].shift(1)
        data['lagged-log-spread-price-mean'] = data['log-spread-price-mean'].shift(1)
        data[x_name+'size'] = 0
        data[y_name+'size'] = 0
        data[x_name+'order_size'] = 0
        data[y_name+'order_size'] = 0
        data['capital'] = 0
         # margin = revenue - costs
        data['margin'] = 0

        # Implements statistical arbitrage trading strategies
        for index, row in data.iterrows():
            # Hit Upper Band, Short the Spread
            if (data.at[index,'log-spread-price'] > data.at[index,'log-spread-price-upper']) and (x_name_current_size >= 0):
                capital = capital - int(hr*order_size)*data.at[index,x_name] + int(order_size)*data.at[index,y_name]
                # x_name
                data.at[index,x_name+'order_size'] = - int(hr*order_size) - x_name_current_size
                x_name_current_size = - int(hr*order_size)
                # y_name
                data.at[index,y_name+'order_size'] = order_size - y_name_current_size
```

```
            y_name_current_size = order_size
            # margin = revenue - costs
            data.at[index,'margin'] = -1*data.at[index,x_name+'order_size']*data.at[index,x_name] - 1*data.at[index,y_name+'order_size']*data.at[index,y_name]

        # Hit Lower Band, Long the Spread
        elif (data.at[index,'log-spread-price'] < data.at[index,'log-spread-price-lower']) and (x_name_current_size <= 0):
            capital = capital + int(hr*order_size)*data.at[index,x_name] - int(order_size)*data.at[index,y_name]
             # x_name
            data.at[index,x_name+'order_size'] = int(hr*order_size) - x_name_current_size
            x_name_current_size = int(hr*order_size)
            # y_name
            data.at[index,y_name+'order_size'] = - order_size - y_name_current_size
            y_name_current_size = - order_size
            # margin = revenue - costs
            data.at[index,'margin'] = -1*data.at[index,x_name+'order_size']*data.at[index,x_name] - 1*data.at[index,y_name+'order_size']*data.at[index,y_name]

        # Spread crosses from below average, flat long position
        elif (data.at[index,'log-spread-price'] > data.at[index,'log-spread-price-mean']) and (data.at[index,'lagged-log-spread-price'] < data.at[index,'lagged-log-spread-price-mean']) and (x_name_current_size > 0):
            capital = capital - int(x_name_current_size)*data.at[index,x_name] + int(y_name_current_size)*data.at[index,y_name]
             # x_name
            data.at[index,x_name+'order_size'] = -x_name_current_size
            x_name_current_size = 0
            # y_name
            data.at[index,y_name+'order_size'] = -y_name_current_size
            y_name_current_size = 0
            # margin = revenue - costs
            data.at[index,'margin'] = -1*data.at[index,x_name+'order_size']*data.at[index,x_name] - 1*data.at[index,y_name+'order_size']*data.at[index,y_name]

        # Spread crosses from above average, flat/cover short position
        elif (data.at[index,'log-spread-price'] < data.at[index,'log-spread-price-mean']) and (data.at[index,'lagged-log-spread-price'] > data.at[index,'lagged-log-spread-price-mean']) and (x_name_current_size < 0):
            capital = capital + int(x_name_current_size)*data.at[index,x_name] - int(y_name_current_size)*data.at[index,y_name]
            # x_name
            data.at[index,x_name+'order_size'] = -x_name_current_size
            x_name_current_size = 0
            # y_name
```

```
             data.at[index,y_name+'order_size'] = -y_name_current_size
             y_name_current_size = 0
             # margin = revenue - costs
             data.at[index,'margin'] = -1*data.at[index,x_name+'order_size']*data.at[index,x_name] - 1*data.at[index,y_name+'order_size']*data.at[index,y_name]


        # # Sets the capital level depending on the position
        # data.set_value(index, 'capital',capital)
        data.at[index, x_name+'size'] = x_name_current_size
        data.at[index, y_name+'size'] = y_name_current_size

    # Determine

    # Calculates the margin generated from holding all the realative positions
    # This is the price of the position multiplied by the position size held
    data['gain'] = data[x_name+'size']*data[x_name] + data[y_name+'size']*data[y_name]

    # Calculate cumulative-gains returns dataframe
    for index, row in data.iterrows():
        if index == 0:
            data.at[index,'accum-gain'] = data.at[index, 'gain']
        if index > 0:
             data.at[index,'accum-gain'] = data.at[index,'gain'] + data.at[index-1,'accum-gain']

    # Calculate statistical arbitrage cumulative return by dividing accumulative gain by initial cpatial

     # Plots the gains
    # data.plot(x='Date', y=['gain'], kind='line', figsize=(28,18))
    # plt.title('Gains on Trades-:'+x_name+'-'+y_name)
    # plt.ylabel('Gain', fontsize = 30)
    # plt.xlabel('Date', fontsize = 30)
    # plt.legend(loc=2, prop={'size': 30})
    # plt.xticks(size = 18)
    # plt.yticks(size = 18)
    # plt.savefig('charts/gain-' +x_name+'-'+y_name+'.png')

    # Plots the accumukated margin
    data.plot(x='Date', y=['accum-gain'], kind='line')
    plt.title('Accumulated Gain ($):'+x_name+'-'+y_name)
    plt.ylabel('Gain')
    plt.xlabel('Date')
    plt.savefig('results/gains/' +x_name+'-'+y_name+'.png')

    # Plots the order sizes
    # data.plot(x='Date', y=[x_name+'order_size',y_name+'order_size'], kind='line', figsize=(28,18))
    # plt.title('Order Sizes-:'+x_name+'-'+y_name, fontsize = 30)
    # plt.ylabel('Order Size', fontsize = 30)
    # plt.xlabel('Date', fontsize = 30)
    # plt.legend(loc=2, prop={'size': 30})
```

```
        # plt.xticks(size = 18)
        # plt.yticks(size = 18)
        # plt.savefig('charts/order-size-' +x_name+'-'+y_name+'.png')

        # Plots the Positions
        # data.plot(x='Date', y=[x_name+'size',y_name+'size'], kind='line', figsize=(28,18))
        # plt.title('Number of Positions-:'+x_name+'-'+y_name)
        # plt.ylabel('Positions', fontsize = 30)
        # plt.xlabel('Date', fontsize = 30)
        # plt.legend(loc=2, prop={'size': 30})
        # plt.xticks(size = 18)
        # plt.yticks(size = 18)
        # plt.savefig('charts/positions-' +x_name+'-'+y_name+'.png')

        # Calculate buy and hold return over forecast periods
        data = data.sort_values(by='Date')
        data['bhr'] = (data['trading-spread']/data['trading-spread'].shift(1)) - 1
        # Calculate Statistical Arbitrage Returns
        data['bbr'] = data['accum-gain']
        # data['bbr'] = (data['accum-gain']/data['accum-gain'].shift(1)) - 1
        data = data.dropna()
        data.reset_index(inplace = True, drop = True)

        # Calculates Buy and Hold Cuumulative Returns
        for index, row in data.iterrows():
            if index == 0:
                # Buy and Hold Strategy
                bhcr = data.at[index, 'bhr']
                data.at[index,'bhcr'] = bhcr
                # Statistical Arbitrage
                # bbcr = data.at[index, 'bbr']
                # data.at[index,'bbcr'] = bbcr
            if index > 0:
                # Buys and hold strategy
                bhcr = ((1+data.at[index,'bhr'])*(1+data.at[index-1,'bhcr']))-1
                data.at[index,'bhcr'] = bhcr
                # Statistical Arbitrage
                # bbcr = ((1+data.at[index,'bbr'])*(1+data.at[index-1,'bbcr']))-1
                # data.at[index,'bbcr'] = bbcr

        # Return last values cumulative return
        bhr = data.at[data.index[-1],'bhcr']
        # Return
        cg = data.at[data.index[-1],'accum-gain']

        # Calculate cumulative return from statistical arbitrage strategy
        # Plots the accumukated margin
        data.plot(x='Date', y=['bhcr'], kind='line')
        plt.title('Cumulative Buy & Hold Returns ($):'+x_name+'-'+y_name)
        plt.ylabel('Buy & Hold Returns')
        plt.xlabel('Date')
```

```
        plt.savefig('results/returns/buy-hold/'+x_name+'-'+y_name+'.png')

        data.plot(x='Date', y=['accum-gain'], kind='line')
        plt.title('Cumulative Returns ($):'+x_name+'-'+y_name)
        plt.ylabel('Buy and Hold Returns')
        plt.xlabel('Date')
        plt.savefig('results/returns/bollinger/'+x_name+'-'+y_name+'.png')

        # Calculate cumulative return from statistical arbitrage strategy
        # Plots the accumukated margin
        # data.plot(x='Date', y=['bhr','bbr'], kind='line', figsize=(28,18))
        # plt.title('Returns ($):'+x_name+'-'+y_name)
        # plt.ylabel('Returns', fontsize = 30)
        # plt.xlabel('Date', fontsize = 30)
        # plt.legend(loc=2, prop={'size': 30})
        # plt.xticks(size = 18)
        # plt.yticks(size = 18)
        # plt.savefig('charts/returns-' +x_name+'-'+y_name+'.png')

        # Calculates some average values for tranquil and crisis period
        # Tranquil period
        tran_start = '1/9/19'
        tran_end = '28/02/20'
        cris_start = '1/3/20'
        cris_end = '31/8/20'
        # Get tranquil dataframe
        tranquil = data[data["Date"] > tran_start]
        tranquil = tranquil[tranquil["Date"] <= tran_end]
        # Get crisis dataframe
        crisis = data[data["Date"] > cris_start]
        crisis = crisis[crisis["Date"] <= cris_end]

        # Find the averages for those periods
        tran_average_buy_hold = tranquil['bhr'].mean()
        tran_average_gain = tranquil['gain'].mean()
        cris_average_buy_hold = crisis['bhr'].mean()
        cris_average_gain = crisis['gain'].mean()


        # Returns variables from the function
        return x_name, y_name, tstat_coint, hedge_ratio, bhr, cg, order_size, status, tran_average_buy_hold, tran_average_gain, cris_average_buy_hold, cris_average_gain

        # Implements the trading strategy with both bands
    else:
        print("Co-integration between pairs does not exist")
        status = 'No'
        hedge_ratio = np.nan
        bhr = np.nan
        cg = np.nan
        order_size = np.nan
        tran_average_buy_hold = np.nan
        tran_average_gain = np.nan
```

```python
        cris_average_buy_hold = np.nan
        cris_average_gain = np.nan
        return x_name, y_name, tstat_coint, hedge_ratio, bhr, cg, order_size, status, tran_average_buy_hold, tran_average_gain, cris_average_buy_hold, cris_average_gain



# Defines the benchmarking function for the second part of the assignment (Placeholder)
def benchmarking(self):
    """[summary]

    Returns:
        [type]: [description]
    """
    return self
# Defines the financial co-integration and causality function (Placeholder)
def contagion_causality(self):
    """[summary]

    Returns:
        [type]: [description]
    """
    return self

# Downloads financial data from yahoo finance (ExxonMobil and Chevron to Test Co-Integration Example)
# This is to be replaced with the data outputs
# https://towardsdatascience.com/a-comprehensive-guide-to-downloading-stock-prices-in-python-2cd93ff821d4
# Test case setup
start_date = '2000-01-01'
end_date = '2019-12-31'
prices_1 = 'EWA'
prices_2 = 'EWC'

asset_1 = yf.download(prices_1, start = start_date, end = end_date, progress = False)
asset_1.to_csv('data/'+prices_1+'.csv')

asset_2 = yf.download(prices_2, start = start_date, end = end_date, progress = False)
asset_2.to_csv('data/'+prices_2+'.csv')

# Import data
asset_1_df = pd.read_csv('data/'+prices_1+'.csv')
asset_2_df =  pd.read_csv('data/'+prices_2+'.csv')

# Calculate the returns
asset_1_df[prices_1 +'-ret-(%)'] = (asset_1_df['Adj Close']/asset_1_df['Adj Close'].shift(1))-1
asset_1_df.rename(columns= {'Adj Close':prices_1}, inplace = True)
asset_1_df = asset_1_df.dropna(axis=0)

asset_2_df[prices_2 +'-ret-(%)'] = (asset_2_df['Adj Close']/asset_2_df['Adj Close'].shift(1))-1
```

```python
asset_2_df.rename(columns= {'Adj Close':prices_2}, inplace = True)
asset_2_df = asset_2_df.dropna(axis=0)

# Merge the data into one dataframe
data_df = pd.merge(asset_1_df[['Date',prices_1,prices_1 +'-ret-(%)']].copy(), asset_2_df[['Date',prices_2,prices_2 +'-ret-(%)']].copy(), how='left', left_on=['Date'], right_on = ['Date'])
data_df = data_df.dropna(axis=0)
data = data_df[['Date',prices_1,prices_2]].copy()

# Calls the pairs trading function
x_name, y_name, tstat_coint, hedge_ratio, bhr, cg, order_size, status, tran_average_buy_hold, tran_average_gain, cris_average_buy_hold, cris_average_gain = pairs_trading(data,prices_1,prices_2)
# Establishes
test_results_table = pd.DataFrame(columns=['Variable (x)', 'Variable (y)','tstat','Hedge Ratio', 'Buy & Hold Cumulative Return', 'Cumulative Gain (Bollinger Bands)', 'Order Size', 'Co-integration'])
# Creates new row to add to empty dataframe
new_row = {'Variable (x)':x_name, 'Variable (y)': y_name,'tstat': tstat_coint,'Hedge Ratio': hedge_ratio, 'Buy & Hold Cumulative Return': bhr, 'Cumulative Gain (Bollinger Bands)':cg, 'Order Size': order_size, 'Co-integration': status}
test_results_table = test_results_table .append(new_row, ignore_index = True)
# Rank via tStat (Indicates stength of mean reversion
test_results_table.sort_values(by = 'tstat')
test_results_table.to_excel('results/test_results_table.xlsx')

# Conduct pairs trading analysis for list of resources (Steel Stocks)
# Loads in data for pairs trading analysis
resources_data = pd.read_excel('data/data.xlsx')

# Get the pricing information for the data (List of names)
assets = list(resources_data.columns.values)
assets = assets[1:-1]

# Creates combinations for pairs analysis
pair_order_list = list(it.combinations(assets,2))

# Cleans data of values and re_index
resources_data = resources_data.dropna(axis = 0)
resources_data.reset_index(inplace = True, drop = True)

# Initialises final resources table
final_results_table = pd.DataFrame(columns=['Variable (x)', 'Variable (y)','tstat','Hedge Ratio', 'Buy & Hold Cumulative Return', 'Cumulative Gain (Bollinger Bands)', 'Order Size', 'Co-integration', 'tran_average_buy_hold', 'tran_average_gain', 'cris_average_buy_hold', 'cris_average_gain'])
for pair in pair_order_list:
    try:
        x_name, y_name, tstat_coint, hedge_ratio, bhr, cg, order_size, status, tran_average_buy_hold, tran_average_gain, cris_average_buy_hold, cris_average_gain = pairs_trading(resources_data,pair[0],pair[1])
        new_row = {'Variable (x)':x_name, 'Variable (y)': y_name,'tstat': tstat_coint,'Hedge Ratio': hedge_ratio, 'Buy & Hold Cumulative
```

```python
    Return': bhr, 'Cumulative Gain (Bollinger Bands)':cg, 'Order Size':
    order_size, 'Co-integration': status,'tran_average_buy_hold':
    tran_average_buy_hold, 'tran_average_gain':tran_average_gain, '
    cris_average_buy_hold':cris_average_buy_hold, 'cris_average_gain':
    cris_average_gain}
        final_results_table = final_results_table.append(new_row,
    ignore_index = True)
        print('Finished: ', pair)
    except:
        print('Error occurred')

# Rank via tStat (Indicates stength of mean reversion
final_results_table.sort_values(by = 'tstat')
final_results_table.to_excel('results/rank/final_results_table.xlsx')
```

Appendix

Data Processing

This script imports and updates all raw portfolio returns, Fama-French, BMG, Momentum, daily,and monthly data from assignment and Kenneth R. French sources.

```
# This completes the data processing for the BMG Empirical Assignment

# Imports important python packages
import numpy as np # arithmetic operations
import pandas as pd # data analysis package
import csv as csv # read and write csvs
import random as rd # random functionality
import saspy as sas # Use saspy functionality in python
import matplotlib.pyplot as plt # Use MatLab functionality for plotting
import seaborn as sb # Imports seaborn library for use
import wrds as wrds# Wharton Research Data Services API
import pydatastream as pds # Thomas Reuters Datastream API
import yfinance as yf # Yahoo Finance API
import datetime as dt # Manipulate datetime values
import statsmodels.api as sm # Create Stats functionalities
import sklearn as sl # ML functionality
from stargazer.stargazer import Stargazer
import finance_byu as fin # Python Package for Fama-MacBeth Regressions


# Creates dataframes to convert data
# Daily
pd_df = pd.read_excel('data.xlsx', sheet_name = 'portfolio_daily')
ffd_df = pd.read_excel('data.xlsx', sheet_name = 'fama_french_daily')
bmgd_df = pd.read_excel('data.xlsx', sheet_name = 'bmg_daily')
da_df = pd.read_excel('data.xlsx', sheet_name = 'daily_all')

# Monthly
pm_df = pd.read_excel('data.xlsx', sheet_name = 'portfolio_monthly')
ffm_df = pd.read_excel('data.xlsx', sheet_name = 'fama_french_monthly')
bmgm_df = pd.read_excel('data.xlsx', sheet_name = 'bmg_monthly')
ma_df = pd.read_excel('data.xlsx', sheet_name = 'monthly_all')

# Converts fama-french factors from percentages to fractions
# Daily
ffd_df['mktrf'] = ffd_df['mktrf']/100
ffd_df['smb'] = ffd_df['smb']/100
ffd_df['hml'] = ffd_df['hml']/100
ffd_df['rf'] = ffd_df['rf']/100
ffd_df['umd'] = ffd_df['umd']/100

# Monthly
ffm_df['rf'] = ffm_df['rf']/100

# Convert data columns from timestamps to datatime to enable matching
pd_df['date'] = pd.to_datetime(pd_df['date'], unit='s')
ffd_df['date'] = pd.to_datetime(ffd_df['date'], unit='s')
bmgd_df['date'] = pd.to_datetime(bmgd_df['date'], unit='s')
da_df['date'] = pd.to_datetime(da_df['date'], unit='s')

# Creates to dataframes
```

```
uda_df = da_df.copy()
uma_df = ma_df.copy()

# Daily adjustments and additions
# Updates umd and bmg prior to adding new additions
for index, row in uda_df.iterrows():
    date = uda_df.at[index,'date']
    try:
        # Gets the factors
        factors_df = ffd_df.loc[ffd_df['date']== date]
        # print(factors_df.head())
        bmg_df = bmgd_df.loc[bmgd_df['date']== date]
        # print(bmg_df.head())
        # Changes the value
        uda_df.at[index,'umd']= factors_df.iloc[0]['umd']
        uda_df.at[index,'bmg'] = bmg_df.iloc[0]['bmg']
    except:
        print("Error updating the umd and bmg foactor (2010 - 2016)")

# Add the portofilio returns data to the updated dataframes
# Sets sequence for portfolio returns
portfolios = list(range(1,31))
for index, row in pd_df.iterrows():
    # Set the time period imformation
    year = pd_df.at[index,'year']
    month = pd_df.at[index,'month']
    day = pd_df.at[index,'day']
    date = pd_df.at[index,'date']
    # Set the factor elements based on dates with index matching from
    dataframes
    # Locates the factors at the required date
    # Try statement to skip entries when portfolio, factor and bmg
    dates don't align.
    try:
        factors_df = ffd_df.loc[ffd_df['date']== date]
        bmg_df = bmgd_df.loc[bmgd_df['date']== date]
        mktrf = factors_df.iloc[0]['mktrf']
        smb = factors_df.iloc[0]['smb']
        hml = factors_df.iloc[0]['hml']
        rf = factors_df.iloc[0]['rf']
        umd = factors_df.iloc[0]['umd']
        bmg = bmg_df.iloc[0]['bmg']
        # Add the portfolio components
        for portfolio in portfolios:
            ret = pd_df.at[index,portfolio]
            # Creates dataframe to append
            d = {'ind': [portfolio] , 'ret': [ret],'year': [year],'
    month': [month],'day': [day],'date': [date],'mktrf': [mktrf],'smb':
    [smb],'hml': [hml],'rf': [rf],'umd': [umd],'bmg': [bmg]}
            row_df = pd.DataFrame(data=d)
            # Append to dataframe (use assignment)
            uda_df = uda_df.append(row_df,ignore_index= True)
    except:
        # Documents date omissions
        print("Warning - Error")
        line = date.strftime("%m/%d/%Y, %H:%M:%S")
        with open('omissions-daily.txt', 'a+') as f:
            f.seek(0)
```

```
            data = f.read(100)
            if len(data) > 0 :
                f.write("\n")
                # Append text at the end of file
                f.write(line)
# Create new csv file
uda_df.to_csv('updated_daily_all.csv')

# Monthly adjustments and additions
# Add the portofilio returns data to the updated dataframes
# Sets sequence for portfolio returns
portfolios = list(range(1,31))
for index, row in pm_df.iterrows():
    # Set the time period information
    year = pm_df.at[index,'year']
    month = pm_df.at[index,'month']
    day = pm_df.at[index,'day']
    date = pm_df.at[index,'date']
    eom = pm_df.at[index,'eom']
    # Set the factor elements based on dates with index matching from dataframes
    # Locates the factors at the required date
    # print("index: ",index)
    # print("date: ",date)
    # Try statement to skip entries when portfolio, factor and bmg dates don't align.
    try:
        factors_df = ffm_df.loc[ffm_df['eom']== eom]
        rf = factors_df.iloc[0]['rf']
        # Add the portfolio components
        for portfolio in portfolios:
            ret = pm_df.at[index,portfolio]
            # Creates dataframe to append
            d = {'year': [year],'month': [month],'day': [day],'date': [date], 'eom': [eom], 'ind': [portfolio] , 'ret': [ret], 'rf':[rf]}
            row_df = pd.DataFrame(data=d)
            # Append to dataframe (use assignment)
            uma_df = uma_df.append(row_df,ignore_index= True)
    except:
        # Documents date omissions
        print("Warning - Error")
        line = date.strftime("%m/%d/%Y, %H:%M:%S")
        with open('omissions-monthly.txt', 'a+') as f:
            f.seek(0)
            data = f.read(100)
            if len(data) > 0 :
                f.write("\n")
                # Append text at the end of file
                f.write(line)
# Create new csv file
uma_df.to_csv('updated_monthly_all.csv')
uma_df.to_excel('updated_monthly_all.xlsx')
```

Data Analysis

This script implements all data analysis performed in the assignment.

```python
# This completes the data analysis for the BMG Empirical Assignment
# Note: Data is processed using the finance-761-data-processing script

# Imports important python packages and data from data prcoccessing
import numpy as np # arithmetic operations
import pandas as pd # data analysis package
import csv as csv # read and write csvs
import random as rd # random functionality
import saspy as sas # Use saspy functionality in python
import matplotlib.pyplot as plt # Use MatLab functionality for plotting
import seaborn as sb # Imports seaborn library for use
import wrds as wrds# Wharton Research Data Services API
import pydatastream as pds # Thomas Reuters Datastream API
import yfinance as yf # Yahoo Finance API
import datetime as dt # Manipulate datetime values
import statsmodels.api as sm # Create Stats functionalities
import linearmodels as lp # Ability to use PooledOLS
import sklearn as sl # ML functionality
from stargazer.stargazer import Stargazer
import finance_byu as fin # Python Package for Fama-MacBeth Regressions
from statsmodels.regression.rolling import RollingOLS # Use factor loadings
from stargazer.stargazer import Stargazer
import sympy as sy # convert latex code
import scipy as sc # Scipy packages
import tabulate as tb # Create tables in python

# Establishes plotting setting for rolling regressions
sb.set_style('darkgrid')
pd.plotting.register_matplotlib_converters()

# Reads data csvs as dataframes
daily_all_df = pd.read_csv("updated_daily_all.csv")
monthly_all_df = pd.read_csv("updated_monthly_all.csv")


# Creates excess return variable for both daily and monthly
daily_all_df['eret'] = daily_all_df['ret']/100 - daily_all_df['rf'] # Check if this step is necessary
monthly_all_df['eret'] = monthly_all_df['ret']/100 - monthly_all_df['rf'] # Check if this step is necessary

# Creates month variables to both the daily and monthly sets
daily_all_df['m'] = daily_all_df['month'] + daily_all_df['year']*12
monthly_all_df['m'] = monthly_all_df['month'] + monthly_all_df['year']*12

daily_all_df.to_excel('excel/daily_all_df_excel_check.xlsx')
monthly_all_df.to_excel('excel/monthly_all_df_excel_check.xlsx')

# Correctly sorts the columns
daily_all_df.sort_values(by=['ind','year','month'],ascending=True, inplace=True)

# Creates a unique list of month values
```

```python
m_list = sorted(np.unique(daily_all_df['m']))
ind_list = sorted(np.unique(daily_all_df['ind']))

# Shifted for Stage 2 of the Fama MacBeth Regression
# monthly_all_df['eret'] = (monthly_all_df.sort_values(by=['m'], ascending=True)
#                          .groupby(['ind'])['eret'].shift(-1))

# Drop NaN Datas
monthly_all_df = monthly_all_df.dropna(axis=0, how = 'any')

# Start Fama-Macbeth Regressions

# This is Stage 1 of the Fama-Macbeth Regression - Estimate Factor Loadings (Crossed Checked)
# https://en.wikipedia.org/wiki/Fama%E2%80%93MacBeth_regression
# Create a new dataframe with every possible combination of month and index combinations available
factor_df = pd.DataFrame(columns=['ind','m', 'mktrf','smb','hml','umd','bmg'])
for i in ind_list:
    for j in m_list:
        # Loops over factor dataframe to get the desired values
        # Get slice of dataframe based on multiple columns
        index_df = daily_all_df[daily_all_df['ind'] == i]
        slice_df = index_df[index_df['m'] == j]
        # Perform the OLS Regressions on the sliced dataframne
        y = slice_df['eret']
        x = slice_df[['mktrf','smb','hml','umd','bmg']]
        x = sm.add_constant(x)
        model =sm.OLS(y,x).fit()
        # Save model parameters to column dataframe
        new_row = {'ind':i,'m':j, 'mktrf':model.params[1],'smb':model.params[2],'hml':model.params[3],'umd':model.params[4],'bmg':model.params[5]}
        # Append factor loading to the factors to the dataframe
        factor_df = factor_df.append(new_row, ignore_index = True)

# Produce average industry beta
average_ffb_df = pd.DataFrame(columns=['m','bmg'])
for m in m_list:
    cross_sectional_df = factor_df[factor_df['m'] == m]
    new_row = {'m':m,'bmg':cross_sectional_df['bmg'].mean()}
    print(new_row)
    average_ffb_df = average_ffb_df.append(new_row, ignore_index= True)


# PLot average ffb
average_ffb_df.plot(x='m', y=['bmg'], kind='line', figsize=(28,18))
plt.title('Average Industry BMG Time Series', fontsize = 30)
plt.ylabel('BMG', fontsize = 24)
plt.xlabel('Time (Date)', fontsize = 24)
plt.legend(loc=2, prop={'size': 18})
plt.xticks(size = 18)
plt.yticks(size = 18)
# Add caption to below plot python
plt.savefig('plots/bmg-time-series-premium.png')

```

```
raw_fama_macbeth_df = pd.merge(monthly_all_df, factor_df[['ind','m','mktrf','smb','hml','umd','bmg']].copy(),  how='right', left_on=['ind','m'], right_on = ['ind','m'])
raw_fama_macbeth_df.to_excel('excel/raw_fama_macbeth_df.xlsx')


# # Shift factors forward by one value (month)
factor_df['mktrf'] = (factor_df.sort_values(by=['m'], ascending=True)
                       .groupby(['ind'])['mktrf'].shift(1))
factor_df['smb'] = (factor_df.sort_values(by=['m'], ascending=True)
                       .groupby(['ind'])['smb'].shift(1))
factor_df['hml'] = (factor_df.sort_values(by=['m'], ascending=True)
                       .groupby(['ind'])['hml'].shift(1))
factor_df['umd'] = (factor_df.sort_values(by=['m'], ascending=True)
                       .groupby(['ind'])['umd'].shift(1))
factor_df['bmg'] = (factor_df.sort_values(by=['m'], ascending=True)
                       .groupby(['ind'])['bmg'].shift(1))

# Drop NaN Datas
factor_df = factor_df.dropna(axis=0, how = 'any')
# Merge the regression co-efficients with monthly dataset
fama_macbeth_df = pd.merge(monthly_all_df, factor_df[['ind','m','mktrf','smb','hml','umd','bmg']].copy(),  how='right', left_on=['ind','m'], right_on = ['ind','m'])
# Drops rows with NaN
fama_macbeth_df = fama_macbeth_df.dropna(axis=0, how = 'any')
# Saves Fama-Macbeth Regression Results to Excell
fama_macbeth_df.to_excel('excel/processed_fama_macbeth_df.xlsx')


time_series_df = pd.DataFrame(columns=['ind','value','alpha', 'mktrf','smb','hml','umd','bmg'])
# Work out the Fama-French Time series
for i in ind_list:
    # Gets section of dataframe for monthly date (i.e. days for ind x )
    cross_sectional_df = daily_all_df[daily_all_df['ind'] == i]
    # Performs regression using the cross section to get factor price
    y = cross_sectional_df['eret']
    x = cross_sectional_df[['mktrf','smb','hml','umd','bmg']]
    # print(cross_sectional_df.head(n=30))
    # input("Press Enter to continue...")
    x = sm.add_constant(x)
    model =RollingOLS(y,x).fit()
    model =sm.OLS(y,x).fit()
    # Save model parameters to column dataframe
    new_row_coef = {'ind':i,'value':'co-efficient','alpha': model.params[0],'mktrf':model.params[1],'smb':model.params[2],'hml':model.params[3],'umd':model.params[4],'bmg':model.params[5]}
    # Save model pvalues
    new_row_pvalue = {'ind':i,'value':'pvalue','alpha': model.pvalues[0],'mktrf':model.pvalues[1],'smb':model.pvalues[2],'hml':model.pvalues[3],'umd':model.pvalues[4],'bmg':model.pvalues[5]}
    # new_row_se = {'m':i,'alpha': model.std_errors[0],'mktrf':model.std_errors[1],'smb':model.std_errors[2],'hml':model.std_errors[3],'umd':model.std_errors[4],'bmg':model.std_errors[5]}
    # Append the new row to the dataframe
    time_series_df = time_series_df.append(new_row_coef, ignore_index = True)
```

```python
    time_series_df = time_series_df.append(new_row_pvalue, ignore_index = True)
    # factor_price_se_df = factor_price_se_df.append(new_row_se, ignore_index = True)
    # Append models to list
    stargazer = Stargazer([model])
    stargazer.custom_columns('FF Time-series-'+ str(i))
    expr = stargazer.render_latex()
    sy.preview(expr, viewer='file', filename='time-series/' + str(i)+'-regression.png')

# Prints the time-series (all months) accross industries
time_series_df.to_excel('excel/time-series-regression-table.xlsx')


# This is Stage 2 of the Fama-Macbeth Regression - Estimate Factor Prices from Monthly Data
# Create factor pricing dataframe
factor_price_df = pd.DataFrame(columns=['m', 'alpha', 'mktrf','smb','hml','umd','bmg'])
factor_price_se_df = pd.DataFrame(columns=['m', 'alpha', 'mktrf','smb','hml','umd','bmg'])

# Update the m list as excludes the first month
m_list = sorted(np.unique(fama_macbeth_df['m']))

# Run cross-sectional regression for each time period using monthly data
for i in m_list:
    # Gets section of dataframe for monthly date (i.e. ind 1-30 for month x )
    cross_sectional_df = fama_macbeth_df[fama_macbeth_df['m'] == i]
    # Performs regression using the cross section to get factor price
    y = cross_sectional_df['eret']
    x = cross_sectional_df[['mktrf','smb','hml','umd','bmg']]
    # print(cross_sectional_df.head(n=30))
    # input("Press Enter to continue...")
    x = sm.add_constant(x)
    # model = sm.OLS(endog = y,exog = x).fit()
    model =sm.OLS(y,x,).fit()
    # Save model parameters to column dataframe
    new_row_price = {'m':i,'alpha': model.params[0],'mktrf':model.params[1],'smb':model.params[2],'hml':model.params[3],'umd':model.params[4],'bmg':model.params[5]}
    # new_row_se = {'m':i,'alpha': model.std_errors[0],'mktrf':model.std_errors[1],'smb':model.std_errors[2],'hml':model.std_errors[3],'umd':model.std_errors[4],'bmg':model.std_errors[5]}
    # Append the new row to the dataframe
    factor_price_df = factor_price_df.append(new_row_price, ignore_index = True)
    # factor_price_se_df = factor_price_se_df.append(new_row_se, ignore_index = True)
    # Append models to list
    stargazer = Stargazer([model])
    expr = stargazer.render_latex()
    sy.preview(expr, viewer='file', filename='statistical-tables/' + str(i)+'-regression.png')

```

```python
# Converts Factor Prices to Excel
factor_price_df.to_excel('excel/ffb_stage_2_df.xlsx')

# Plot the dataframe
factor_price_df.plot(x='m', y=['bmg'], kind='line', figsize=(28,18))
plt.title('Average Industry BMG Risk Premium Time Series')
plt.ylabel('BMG Risk Premium', fontsize = 24)
plt.xlabel('Time (Date)', fontsize = 24)
plt.legend(loc=2, prop={'size': 18})
plt.xticks(size = 18)
plt.yticks(size = 18)
# Add caption to below plot python
plt.savefig('plots/bmg-risk-premium.png')

# This is Stage 3 of the Fama-Macbeth Regression - Estimate average factor pricing and error ()
# Calculate estimated factor prices across all time periods
factor_prices_average_dict = {'alpha': factor_price_df['alpha'].mean(),'mktrf':factor_price_df['mktrf'].mean(),'smb':factor_price_df['smb'].mean(),'hml':factor_price_df['hml'].mean(),'umd':factor_price_df['umd'].mean(),'bmg':factor_price_df['bmg'].mean()}
# factor_prices_se_average_dict = {'alpha': factor_price_se_df['alpha'].mean(),'mktrf':factor_price_se_df['mktrf'].mean(),'smb':factor_price_se_df['smb'].mean(),'hml':factor_price_se_df['hml'].mean(),'umd':factor_price_se_df['umd'].mean(),'bmg':factor_price_se_df['bmg'].mean()}

# Calculates unbiased standard error of the mean over requested axis
# https://www.geeksforgeeks.org/python-pandas-dataframe-sem/
factor_prices_sem_average_dict = {'alpha': factor_price_df['alpha'].sem(),'mktrf':factor_price_df['mktrf'].sem(),'smb':factor_price_df['smb'].sem(),'hml':factor_price_df['hml'].sem(),'umd':factor_price_df['umd'].sem(),'bmg':factor_price_df['bmg'].sem()}

# Print dictionaries to display factor prices and standard errors
alpha_mean = factor_price_df['alpha'].mean()
mktrf_mean = factor_price_df['mktrf'].mean()
smb_mean = factor_price_df['smb'].mean()
hml_mean = factor_price_df['hml'].mean()
umd_mean = factor_price_df['umd'].mean()
bmg_mean = factor_price_df['bmg'].mean()

# Performs one sample ttest on all variables
bmg_tstat,bmg_pvalue = sc.stats.ttest_1samp(a=factor_price_df['bmg'], popmean=factor_price_df['bmg'].mean())
mktrf_tstat,mktrf_pvalue = sc.stats.ttest_1samp(a=factor_price_df['mktrf'], popmean=factor_price_df['mktrf'].mean())
smb_tstat,smb_pvalue = sc.stats.ttest_1samp(a=factor_price_df['smb'], popmean=factor_price_df['smb'].mean())
hml_tstat,hml_pvalue = sc.stats.ttest_1samp(a=factor_price_df['hml'], popmean=factor_price_df['hml'].mean())
umd_tstat,umd_pvalue = sc.stats.ttest_1samp(a=factor_price_df['umd'], popmean=factor_price_df['umd'].mean())
alpha_tstat,alpha_pvalue = sc.stats.ttest_1samp(a=factor_price_df['alpha'], popmean=factor_price_df['alpha'].mean())

# Create dataframe
head = ['name', 'mean', 'tstat','pvalue']
```

```python
names = ['alpha', 'mktrf', 'smb','hml','umd','bmg']
means = [alpha_mean, mktrf_mean, smb_mean, hml_mean, umd_mean, bmg_mean]
tstats = [alpha_tstat, mktrf_tstat, smb_tstat, hml_tstat, umd_tstat, bmg_tstat]
pvalues = [alpha_pvalue, mktrf_pvalue, smb_pvalue, hml_pvalue, umd_pvalue, bmg_pvalue]

ffb_statistics = pd.DataFrame(columns=[head[0], head[1], head[2],head[3]])
# For loop to create
for i in range(len(names)):
    new_row = {head[0]:names[i],head[1]:means[i], head[2]:tstats[i], head[3]:pvalues[i]}
    ffb_statistics = ffb_statistics.append(new_row, ignore_index = True)
# Save to CSV
ffb_statistics.to_excel('excel/ffb_statistics.xlsx')

# Additional Analysis 1: Rolling Regression
# Implements Rolling Regressions for each industry (1-30) rolling through months in regressing
# https://www.statsmodels.org/dev/examples/notebooks/generated/rolling_ls.html
exog_vars = ['mktrf','smb','hml','umd','bmg']
for i in ind_list:
    cross_sectional_df = daily_all_df[daily_all_df['ind'] == i]
    # Create eret dataframe
    eret_df = cross_sectional_df[['date','eret']].copy()
    exog = sm.add_constant(cross_sectional_df[exog_vars])
    rols = RollingOLS(eret_df['eret'], exog, window=len(m_list))
    rres = rols.fit()
    fig = rres.plot_recursive_coefficient(variables=exog_vars, figsize=(14,18))
    path = "rolling-regressions/" + str(i) + "-rolling-regression.png"
    plt.savefig(path)

# Additional Analysis 2: Hedging Positions
# Implements Hedging Portfolio based on BMG rankings on the first date (Monthly)
# Imports S&P 500 Data
sp500_df = pd.read_excel('sp500.xlsx', sheet_name = 'sp500')

ranking_df = pd.DataFrame(columns=['ind','mean'])
# Sets bmg ranking from fama_macbeth
for i in ind_list:
    # This is an index
    index_rank_df = fama_macbeth_df[fama_macbeth_df['ind'] == i]
    new_row = {'ind':i, 'mean': index_rank_df['bmg'].mean()}
    ranking_df = ranking_df.append(new_row,ignore_index = True)
bmg_good_df = ranking_df.sort_values(by ='mean',ascending = True).head(5)
bmg_bad_df = ranking_df.sort_values(by = 'mean',ascending = True).tail(5)
# Create green list (Top 5, Green Stocks)
bmg_good = bmg_good_df['ind'].to_list()
# Create brown list (Bottom 5, Brown Stocks)
bmg_bad = bmg_bad_df['ind'].to_list()
```

```
# Create new dateframes with Python
good_returns = pd.DataFrame(columns=['m',str(bmg_good[0]),str(bmg_good
    [1]),str(bmg_good[2]),str(bmg_good[3]),str(bmg_good[4])])
bad_returns = pd.DataFrame(columns=['m',str(bmg_bad[0]),str(bmg_bad[1])
    ,str(bmg_bad[2]),str(bmg_bad[3]),str(bmg_bad[4])])

# Starts cumulative returns calculations
# Top 5 Green Stocks
for j in m_list:
    # Empty list to append to
    emp = []
    for i in bmg_good:
        #Gets the slicesd row
        index_df = fama_macbeth_df[fama_macbeth_df['ind'] == i]
        slice_df = index_df[index_df['m'] == j]
        idx = slice_df.loc[slice_df['ind'] == i].index
        emp.append(slice_df.at[idx[0],'ret'])
        # emp.append(slice_df.at[idx[0],'ret'] - slice_df.at[idx[0],'rf
    ']*100)
    # Append new row of returns data
    new_row = {'m':j,str(bmg_good[0]): emp[0],str(bmg_good[1]): emp[1],
     str(bmg_good[2]): emp[2], str(bmg_good[3]): emp[3],str(bmg_good[4])
    : emp[4]}
    good_returns = good_returns.append(new_row, ignore_index = True)

# Create equally weighting returns from the columns
good_returns['ret'] = ((good_returns[str(bmg_good[0])] + good_returns[
    str(bmg_good[1])] + good_returns[str(bmg_good[2])] + good_returns[
    str(bmg_good[3])] + good_returns[str(bmg_good[4])])/len(bmg_good))
    /100

# Calculate cumulative returns dataframe
for index, row in good_returns.iterrows():
    if index == 0:
        good_returns.at[index,'cr'] = 0
    if index > 0:
        good_returns.at[index,'cr'] = ((1+good_returns.at[index,'ret'])
    *(1+good_returns.at[index-1,'cr']))-1


good_returns.to_excel('excel/bmg_green.xlsx')
# Top 5 Brown Stocks
for j in m_list:
    # Empty list to append to
    emp = []
    for i in bmg_bad:
        #Gets the slicesd row
        index_df = fama_macbeth_df[fama_macbeth_df['ind'] == i]
        slice_df = index_df[index_df['m'] == j]
        idx = slice_df.loc[slice_df['ind'] == i].index
        emp.append(slice_df.at[idx[0],'ret'])
        # emp.append(slice_df.at[idx[0],'ret'] - slice_df.at[idx[0],'rf
    ']*100)
    # Append new row of returns data
    new_row = {'m':j,str(bmg_bad[0]): emp[0],str(bmg_bad[1]): emp[1],
    str(bmg_bad[2]): emp[2], str(bmg_bad[3]): emp[3],str(bmg_bad[4]):
    emp[4]}
    bad_returns = bad_returns.append(new_row, ignore_index = True)
```

```python

# Create equally weighting returns from the columns
bad_returns['ret'] = ((bad_returns[str(bmg_bad[0])] + bad_returns[str(bmg_bad[1])] + bad_returns[str(bmg_bad[2])] + bad_returns[str(bmg_bad[3])] + bad_returns[str(bmg_bad[4])])/len(bmg_bad))/100

# Calculate cumulative returns dataframe
for index, row in bad_returns.iterrows():
    if index == 0:
        bad_returns.at[index,'cr'] = 0
    if index > 0:
        bad_returns.at[index,'cr'] = ((1+bad_returns.at[index,'ret'])*(1+bad_returns.at[index-1,'cr']))-1

# Saves bad_returns to excel
bad_returns.to_excel('excel/bmg_brown.xlsx')
# Create the hedge cumulative returns
hedge_ret_df = good_returns['m'].copy()
hedge_ret_df = pd.merge(hedge_ret_df, good_returns[['m','ret']],  how='left', left_on=['m'], right_on = ['m'])
hedge_ret_df = hedge_ret_df.rename(columns = {'ret':'bmg_green_ret'})
hedge_ret_df = pd.merge(hedge_ret_df, bad_returns[['m','ret']],  how='left', left_on=['m'], right_on = ['m'])
hedge_ret_df = hedge_ret_df.rename(columns = {'ret':'bmg_brown_ret'})
hedge_ret_df['hedge_ret'] = hedge_ret_df['bmg_green_ret'] - hedge_ret_df['bmg_brown_ret']
for index, row in hedge_ret_df.iterrows():
    if index == 0:
        hedge_ret_df.at[index,'cr'] = 0
    if index > 0:
        hedge_ret_df.at[index,'cr'] = ((1+hedge_ret_df.at[index,'hedge_ret'])*(1+hedge_ret_df.at[index-1,'cr']))-1

# Plot the cumulartive returns
bad_returns.plot(x ='m', y='cr', kind = 'line')
plt.savefig('plots/bad_bmg_returns.png')

# Sets cumulative returns calculation
green_cr = good_returns[['m','cr']].copy()
green_cr = green_cr.rename(columns = {'cr':'bmg_green'})
brown_cr = bad_returns[['m','cr']].copy()
brown_cr = brown_cr.rename(columns = {'cr':'bmg_brown'})
sp500_cr = sp500_df[['m','cr']].copy()
sp500_cr = sp500_cr.rename(columns = {'cr':'sp500'})
hedge_cr = hedge_ret_df[['m','cr']].copy()
hedge_cr = hedge_cr.rename(columns = {'cr':'hedge'})

# hedge = pd.DataFrame(columns=['m','bmg_good','bmg_bad','sp500'])
hedge_df = good_returns['m'].copy()

# Merge dataframes
hedge_df = pd.merge(hedge_df, green_cr,  how='left', left_on=['m'], right_on = ['m'])
hedge_df = pd.merge(hedge_df, brown_cr,  how='left', left_on=['m'], right_on = ['m'])
hedge_df = pd.merge(hedge_df, sp500_cr,  how='left', left_on=['m'], right_on = ['m'])
```

```python
hedge_df = pd.merge(hedge_df, hedge_cr,  how='left', left_on=['m'], right_on = ['m'])
hedge_df.to_excel('excel/hedge_df.xlsx')

# Plot the dataframe
hedge_df.plot(x='m', y=['bmg_brown', 'bmg_green', 'hedge','sp500'], kind='line', figsize=(28,14))
plt.title('Hedging - BMG Green, BMG Brown, S&P 500', fontsize = 30)
plt.ylabel('Cumulative Return', fontsize = 24)
plt.xlabel('Time (m)', fontsize = 24)
plt.xticks(size = 18)
plt.yticks(size = 18)
plt.legend(loc=2, prop={'size': 18})
# Add caption to below plot python
plt.savefig('plots/hedging.png')

# Additional Analysis 3: Perform PooledOLS for event study
# Paris Agreement (24192) and Trump Election (24203)
# Sort the dataframe into the needed sections
pre_paris_df = daily_all_df[daily_all_df['m'] < 24192]
print(pre_paris_df.tail())
post_paris_df = daily_all_df[daily_all_df['m'] >= 24192]
print(post_paris_df.head())
pre_trump_df = daily_all_df[daily_all_df['m'] < 24203]
print(pre_trump_df.tail())
post_trump_df = daily_all_df[daily_all_df['m'] >= 24203]
print(post_trump_df.head())

# Reindex dataframes for PooledOLS
pre_paris_df = pre_paris_df.set_index(['m','ind'])
post_paris_df = post_paris_df.set_index(['m','ind'])
pre_trump_df = pre_trump_df.set_index(['m','ind'])
post_trump_df = post_trump_df.set_index(['m','ind'])

# Events
events = [pre_paris_df, post_paris_df, pre_trump_df, post_trump_df]
names = ['Pre-Paris-Agreement-(before-Dec-2015)', 'Post-Paris-Agreement-(Dec-2015-onwards)', 'Pre-Trump-Election-(before-Nov-2016)', 'Post-Trump-Election-(Nov-2016-onwards)']
name_count = 0
# Runs PooledOLS for
for ev in events:
    # Performs PooledOLS
    endo = ev['eret']
    exog = ev[['mktrf','smb','hml','umd','bmg']]
    exog = sm.add_constant(exog)
    model = lp.PooledOLS(endo, exog).fit(cov_type='clustered', cluster_entity=True)
    print(model, file=open("event-study/"+ names[name_count]+"pooledOLS.txt", "w"))
    name_count = name_count + 1
```