22147118 Kazuki Li

Summary of the Python code 2~10

Chapter 2

UKHP.py

1. The library pandas and numpy are imported and given the names pd and np respectively. By locating the Excel file, the data can be read on the python. Data.head() gives the first 5 data on the Excel data, which is ordered by date. The number of data to be shown can be changed by adding the number of data in () of the code data.head().

```
In [15]: import pandas as pd
         import numpy as np

         data = pd.read_excel('/Users/kazukili/Desktop/Python/Excel/UKHP.xlsx', index_col=0)
         data.head()
```

Out[15]:

Month	Average House Price
1991-01-01	53051.721106
1991-02-01	53496.798746
1991-03-01	52892.861606
1991-04-01	53677.435270
1991-05-01	54385.726747

2. The data extracted from the Excel is then organized and displayed into mean, standard deviation, skewness, and kurtosis. In a constantly increasing house price over the last 27 years, the mean gives a fairly accurate average value of the data in comparison to other measures of central tendency. However, in this context where the housing price is never stagnant, the mean doesn't really represent the typical value. It is also expected for the standard deviation to be high as the data values are constantly increasing in over 300 data points. The skewness of -0.11 indicates a slightly negatively skewed distribution and the data is more concentrated on the right. There are more high-priced housing and the mean is not close to mode; thus, the growth of the housing price must have increased. As for kurtosis, the coefficient of excess kurtosis of -1.59 indicates lighter tails and a smaller peak. While the growth rate of housing prices isn't constant, it seems to be rather stable as there are low chance of outliers occurring and data are spread closely to the mean. However, it somewhat conflicts with the high standard deviation and coefficient of excess kurtosis of -1.59 should be a relatively small excess kurtosis.

```
In [25]: print(data['Average House Price'].mean())
         print(data['Average House Price'].std())
         print(data['Average House Price'].skew())
         print(data['Average House Price'].kurtosis())

         data.describe()
```

```
124660.48446512519
56387.16566469951
-0.11014547214986718
-1.591926785005042
```

Out[25]:

	Average House Price
count	327.000000
mean	124660.484465
std	56387.165665
min	49601.664241
25%	61654.141609
50%	150946.108249
75%	169239.278727
max	211755.925562

3. This coding calculates the log difference in each house price by its previous pricing. With only the log difference it is difficult to determine the exact rate of increase or decrease in the house price, but it still shows whether it increased or decreased and if the growth rate increased. Log difference data is then described with basic statistical features including data count, mean, standard deviation, minimum, maximum, and the values for 25, 50, and 75% percentile.

```
In [41]: def LogDiff(x):
             x_diff = 100*np.log(x/x.shift(1))
             return x_diff
```

```
In [27]: data['dhp'] = LogDiff(data['Average House Price'])
         data.head()
```

Out[27]:

Month	Average House Price	dhp
1991-01-01	53051.721106	NaN
1991-02-01	53496.798746	0.835451
1991-03-01	52892.861606	-1.135343
1991-04-01	53677.435270	1.472432
1991-05-01	54385.726747	1.310903

```
In [28]: data.describe()
```

Out[28]:

	Average House Price	dhp
count	327.000000	326.000000
mean	124660.484465	0.424402
std	56387.165665	1.114586
min	49601.664241	-3.464027
25%	61654.141609	-0.256025
50%	150946.108249	0.447332
75%	169239.278727	1.145618
max	211755.925562	3.731686

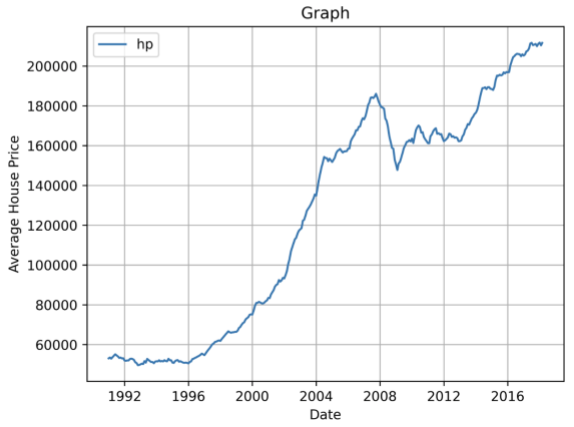
4. The log difference column is labeled as dhp, possible to reference this data for later use. With the average house price labeled as hp, the data is graphed with the date on the x-axis and the pricing on the y-axis. While remaining stagnant until 1996, house prices experienced exponential growth, which left the house price to remain and continue its growth at a high price level. Given that the mean is 124660, it is evident that the standard deviation is high as there is no concentration of data around the mean. Slight negative skewness can also be explained with the clump of data with high pricing from 2004 onward.

```
In [29]: data1 = pd.DataFrame({'dhp':LogDiff(data['Average House Price'])})
         data1 = data1.dropna()
         data1.head()
```

Out[29]:

Month	dhp
1991-02-01	0.835451
1991-03-01	-1.135343
1991-04-01	1.472432
1991-05-01	1.310903
1991-06-01	1.318181

```
In [30]: import matplotlib.pyplot as plt

         plt.figure(1, dpi=600)
         plt.plot(data['Average House Price'], label='hp')

         plt.xlabel('Date')
         plt.ylabel('Average House Price')
         plt.title('Graph')
         plt.grid(True)

         plt.legend()
         plt.show()
```



5. The following coding saves the Python coding on this notebook to a file ‘UKHP.pickle’.

```
In [34]: data.to_excel('/Users/kazukili/Desktop/Python/Excel/UKHP_workfile.xlsx')
```

```
In [37]: import pickle

         with open('/Users/kazukili/Desktop/UKHP.pickle', 'wb') as handle:
             pickle.dump(data, handle)
```

```
In [38]: with open('/Users/kazukili/Desktop/UKHP.pickle', 'rb') as handle:
             data = pickle.load(handle)
```

fund_managers.py

1. In addition to Risky Ricky’s mean return being 1% higher than that of Safe Steve, the skewness of Risky Ricky is also larger. Risky Ricky is more positively skewed, indicating most of the returns to be concentrated on the left side with lower returns. However, Risky Ricky is more volatile as the standard deviation of 20 is extremely high in comparison to Safe Steve. High kurtosis also indicates the scatter of data as they are not concentrated around the center. Based on the ranks of both investments, both of them are close to normal distribution. Both rank of their mean is

```
In [7]: print(data['Risky Ricky'].mean(), data['Safe Steve'].mean())
        print(data['Risky Ricky'].std(), data['Safe Steve'].std())
        print(data['Risky Ricky'].skew(), data['Safe Steve'].skew())
        print(data['Risky Ricky'].kurtosis(), data['Safe Steve'].kurtosis())
```

```
5.6923076923076925 4.6923076923076925
20.36084729484759 6.612924408366485
1.1154523149102482 0.29450652269478483
2.1042142637505923 1.4590345079578664
```

```
In [8]: data.describe()
```

Out[8]:

	Year	Risky Ricky	Safe Steve	Ricky Ranks	Steve Ranks
count	13.00000	13.000000	13.000000	13.00000	13.000000
mean	2011.00000	5.692308	4.692308	7.00000	6.846154
std	3.89444	20.360847	6.612924	3.89444	3.782551
min	2005.00000	-23.000000	-8.000000	1.00000	1.000000
25%	2008.00000	-7.000000	2.000000	4.00000	4.000000
50%	2011.00000	4.000000	4.000000	7.00000	7.000000
75%	2014.00000	14.000000	7.000000	10.00000	10.000000
max	2017.00000	56.000000	19.000000	13.00000	13.000000

close to the median. The ranks for each percentile are evenly distributed, which doesn't mean that they are normally distributed, but they should be bell-shaped.

2. The correlation between the two investments is graphed and the correlation value between Risky Ricky and Safe Steve is given to be 0.87. Both of them move closely in the same direction.

3. IRR is derived based on the cash flow of -107, 5, …, 105 over the 5-year span. The IRR is given to be 0.0345 or 3.45%. As IRR is the rate for the NPV of the future cash flows to equal to 0, it is also the annual return over the 5 years.

```
In [9]: data.corr()
Out[9]:
```

	Year	Risky Ricky	Safe Steve	Ricky Ranks	Steve Ranks
Year	1.000000	0.142928	0.385059	0.021978	-0.197996
Risky Ricky	0.142928	1.000000	0.870048	-0.934285	-0.766741
Safe Steve	0.385059	0.870048	1.000000	-0.796004	-0.944866
Ricky Ranks	0.021978	-0.934285	-0.796004	1.000000	0.763700
Steve Ranks	-0.197996	-0.766741	-0.944866	0.763700	1.000000

```
In [10]: import numpy_financial as npf

cashflow = pd.Series(index=[0,1,2,3,4,5], name='Cashflow', data=[-107,5,5,5,5,105])
print(cashflow)

npf.irr(cashflow)

0   -107
1      5
2      5
3      5
4      5
5    105
Name: Cashflow, dtype: int64
Out[10]: 0.034517484085994976
```

Data_management.py

1. Data and numbers in Python are labeled into separate names, which can be referenced in coding to extract or use them. In this case, the number 1 is labeled 'a' and when 'a' is asked to be typed, the result gives 'int' as an integer for 1.

2. Similarly, numbers that are not integers are given as 'float'

3. For data given in True or False as the result, they are displayed as 'bool'. It is the abbreviation for Boolean data, which has only 2 possible values.

4. For data such as dates are classified into string data; thus, they are given as str.

5. As each name can only be labeled to one data, the data assigned to a name will be overwritten if another data is assigned to a used name.

6. It is invalid to include space in the naming of the data, underscore should be used instead. Also, the names shouldn't start with a number, and upper and lower case are considered different.

7. There is a list of keywords in Python that are used as commands and are not recognized for naming purposes.

8. Some command requires indentation and ignoring it will cause an error.

9. Mathematical formulas can be written directly, which can be calculated when printed.

```
In [1]: a = 1

In [2]: type(a)
Out[2]: int

In [3]: a = 1.2
        type(a)
Out[3]: float

In [4]: b = True
        c = False

In [5]: type(b)
Out[5]: bool

In [6]: type(c)
Out[6]: bool

In [7]: date = '31/12/2016'
        type(date)
Out[7]: str

In [3]: my value = 1
          Cell In[3], line 1
            my value = 1
                  ^
SyntaxError: invalid syntax

In [15]: myvalue = 1

In [4]: my_vlaue = 1

In [6]: 3value = 1
          Cell In[6], line 1
            3value = 1
             ^
SyntaxError: invalid decimal literal
```

10. The function can be inserted along with the variables. In the given example, ** is used for the exponent, and as mentioned previously name is used to substitute the variable x instead of directly typing the integer in.

```
In [7]: value = 1
        Value = 2
        print(value)
        print(Value)

1
2
```

```
In [11]: add_two_num = 1 + 2
         subtrat_two_num = 10 - 2
         multiply_two_num = 17*2
         divide_two_num = 24/6

         print(add_two_num)
         print(subtrat_two_num)
         print(multiply_two_num)
         print(divide_two_num)

3
8
34
4.0
```

```
In [12]: my_data = 12
         def func(x):
             y = x**2
             return y

         func(my_data)

Out[12]: 144
```

```
In [8]: import keyword
        keyword.kwlist

Out[8]: ['False',
         'None',
         'True',
         'and',
         'as',
         'assert',
         'async',
         'await',
         'break',
         'class',
         'continue',
         'def',
         'del',
         'elif',
         'else',
         'except',
         'finally',
         'for',
         'from',
         'global',
         'if',
         'import',
         'in',
         'is',
         'lambda',
         'nonlocal',
         'not',
         'or',
         'pass',
         'raise',
         'return',
         'try',
         'while',
         'with',
         'yield']
```

```
In [9]: def func():
            return print('hello world')
        func()

hello world
```

```
In [10]: def func():
         return print('hello world')
         func()

  Cell In[10], line 2
    return print('hello world')
    ^
IndentationError: expected an indented block after function definition on line 1
```

Chapter 3

Simple_linear_regression.py

1. The time series data of an index's price and its index future's price is given throughout the years in 1-month intervals. The actual price of the index is named spot with the index futures labeled as futures. Then, the OLS regressor is used to determine the a and b estimates of the linear regression where the spot is the dependent variable and futures for the independent variable. The resulting coefficients are given in the table with 'coef,' where the coefficient of intercept is a (the y-intercept) and the coefficient of futures is b (futures is the independent variable). Spot = -2.8378 + 1.0016futures + u, where random error u is normally distributed with a mean of 0. It should also be noted that the R-squared value for this regression is 1, which fits and explains the data perfectly. The result also gives a t-score and p-value that can be used for assessing the significance of the OLS regression result. Although the significant level and its corresponding t-statistics value are not given, since the t-score also represents the distance between the estimate and the hypothesized value, we can make estimations on

```
In [2]: import pandas as pd
        import numpy as np
        import statsmodels.formula.api as smf

        data = pd.read_excel('/Users/kazukili/Desktop/Python/Excel/SandPhedge.xlsx', index_col=0)

        data.head()

Out[2]:
```

Date	Spot	Futures
1997-09-01	947.280029	954.50
1997-10-01	914.619995	924.00
1997-11-01	955.400024	955.00
1997-12-01	970.429993	979.25
1998-01-01	980.280029	987.75

```
In [3]: formula = 'Spot ~ Futures'
        results = smf.ols(formula, data).fit()
        print(results.summary())

                            OLS Regression Results
==============================================================================
Dep. Variable:                   Spot   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 1.005e+06
Date:                Sat, 16 Sep 2023   Prob (F-statistic):               0.00
Time:                        22:49:58   Log-Likelihood:                -826.86
No. Observations:                 247   AIC:                             1658.
Df Residuals:                     245   BIC:                             1665.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -2.8378      1.489     -1.906      0.058      -5.771       0.095
Futures        1.0016      0.001   1002.331      0.000       1.000       1.004
==============================================================================
Omnibus:                      245.415   Durbin-Watson:                   1.326
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            10091.673
Skew:                          -3.814   Prob(JB):                         0.00
Kurtosis:                      33.371   Cond. No.                     5.05e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.05e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

the significance. A T-score of -1.906 for α is relatively small; thus, the null hypothesis of α=0 might be retained. In comparison t-score of 1002.231 for β is extremely high and the null hypothesis of β=0 is likely rejected. The same can be done for p-value, which is displayed as $p > |t|$. The significant level can be chosen to be compared with the p-value directly. α have a p-value just above the common significance level of 5%; thus, the null hypothesis is retained. β have a p-value smaller than 0.000; thus, the null hypothesis is rejected with a significance level of possibly 1%.

2. As the previous regression is based on the prices, another regression is calculated based on the return which is free of unit and clearer in showing the level of increase or decrease. Log return is first calculated from the original data file, which is then used for OLS regression. β in this regression model can also be represented as optimal hedge ratio as β in this case shows the sensitivity of changes in index future return to the index return. However, this index and its index future aren't really an optimal option for hedging as the slope of 0.9751 indicates that both investments grow in high correlation with each other.

```
In [4]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_spot' : LogDiff(data['Spot']),
                             'ret_future':LogDiff(data['Futures'])})
        data.head()
```

Out[4]:

Date	ret_spot	ret_future
1997-10-01	-3.508608	-3.247557
1997-11-01	4.362145	3.299927
1997-12-01	1.560914	2.507563
1998-01-01	1.009901	0.864266
1998-02-01	6.807837	6.159189

```
In [ ]: data.describe()
```

```
In [6]: formula = 'ret_spot ~ ret_future'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

OLS Regression Results

Dep. Variable:	ret_spot	R-squared:	0.989
Model:	OLS	Adj. R-squared:	0.989
Method:	Least Squares	F-statistic:	2.147e+04
Date:	Sat, 16 Sep 2023	Prob (F-statistic):	7.54e-240
Time:	23:39:58	Log-Likelihood:	-157.16
No. Observations:	246	AIC:	318.3
Df Residuals:	244	BIC:	325.3
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.0131	0.029	0.444	0.658	-0.045	0.071
ret_future	0.9751	0.007	146.543	0.000	0.962	0.988

Omnibus:	48.818	Durbin-Watson:	2.969
Prob(Omnibus):	0.000	Jarque-Bera (JB):	671.062
Skew:	-0.016	Prob(JB):	1.91e-146
Kurtosis:	11.091	Cond. No.	4.45

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Chapter 4

Hedging_revisited.py

1. The following code performs an f-test based on the index pricing from the previous chapter. The null hypothesis is the coefficient of futures, or β in the regression, to be 1. The result shows an f-score of 2.58, which can be compared with the f-statistics of the significance level chosen. In this result, the degrees of freedom are given, where df_denom is the degree of freedom for the denominator and df_num is for the numerator. With this information, an f-table of 0.05 significance level can be used to locate the f-statistics, but Python is used in this example as the degree of freedom for the denominator was large. The result was 3.88 in this example. As the f-score is smaller than the 0.05 significance level, it is not significant enough to reject the null hypothesis. The P-value can be compared directly with the chosen significance level and a p-value of 0.1 is larger than most significance levels; thus, the null hypothesis is retained.

```
In [2]: formula = 'Spot ~ Futures'
        hypotheses = 'Futures = 1'
        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=2.5846385570647095, p=0.10919294686206271, df_denom=245, df_num=1>
```

```
In [6]: import scipy.stats as stats

        alpha = 0.05  # Significance level
        df_denom = 245  # Degrees of freedom in the denominator
        df_num = 1  # Degrees of freedom in the numerator

        # Calculate the critical F-statistic
        critical_f_statistic = stats.f.ppf(1 - alpha, df_num, df_denom)

        print("Critical F-Statistic:", critical_f_statistic)
```

```
Critical F-Statistic: 3.879694140626173
```

2. The same test is performed but on the estimate of log returns regression. The f-statistics for the critical value are calculated again based on the new degrees of freedom, but a significance level of 0.01

```
In [4]: formula = 'ret_spot ~ ret_future'
        hypotheses = 'ret_future = 1'

        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=14.02980960966968, p=0.00022456631728787958, df_denom=244, df_num=1>
```

is used instead, as the p-value is smaller than 0.01 and the f-score was large. The F-score of 14.03 proved significant when compared to the critical f-statistic of 6.74. Both results reject the null hypothesis of β=1.

```
In [8]: import scipy.stats as stats

alpha = 0.01  # Significance level
df_denom = 244  # Degrees of freedom in the denominator
df_num = 1  # Degrees of freedom in the numerator

# Calculate the critical F-statistic
critical_f_statistic = stats.f.ppf(1 - alpha, df_num, df_denom)

print("Critical F-Statistic:", critical_f_statistic)

Critical F-Statistic: 6.739882263278692
```

Chapter 5

CAPM.py

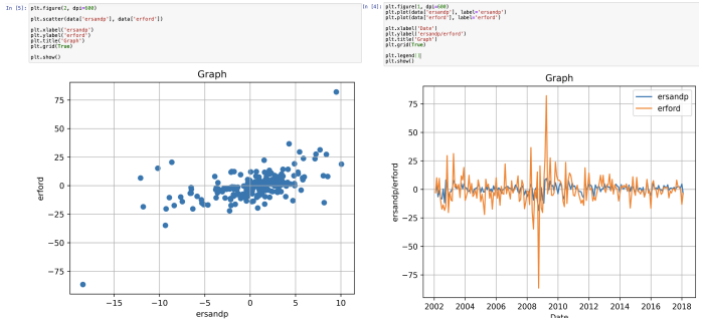
1. Excel sheet capm.xlsx consists of the prices of 5 different market investments and the return of risk-free investment over the years. The data is loaded into the Python code. The following code calculates the prices of the S&P500 index and Ford to their respective log returns, which is then used to find its difference with the risk-free-investment, US treasury bill. In the case of the S&P500, as it is an index that includes various US stocks, its difference with risk-free investment can be considered a market risk premium as the S&P500 can represent the market performance well.

```
In [3]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_sandp' : LogDiff(data['SANDP']),
                             'ret_ford' : LogDiff(data['FORD']),
                             'USTB3M' : data['USTB3M']/12,
                             'ersandp' : LogDiff(data['SANDP']) - data['USTB3M']/12,
                             'erford' : LogDiff(data['FORD']) - data['USTB3M']/12})
        data.head()
```

Out[3]:

Date	ret_sandp	ret_ford	USTB3M	ersandp	erford
2002-01-01	NaN	NaN	0.140000	NaN	NaN
2002-02-01	-2.098486	-2.783480	0.146667	-2.245153	-2.930147
2002-03-01	3.608011	10.273611	0.152500	3.455511	10.121111
2002-04-01	-6.338466	-3.016541	0.145833	-6.484299	-3.162375
2002-05-01	-0.912297	9.814706	0.146667	-1.058964	9.668039

2. The difference with risk-free investment for the two investments is then generated into a graph and scatter plot. We could see a weak positive correlation between the two in the scatter plot with some scattering and outliers. The graph also shows a level of overlap in the two investments, but despite following the same trend of growth and decline, Ford shows a much larger rate of change. Ford is more sensitive to market changes than S&P500.



3. The coefficient of the simple linear regression is obtained with the OLS regressor, where the excess return of Ford is the dependent variable and the excess return for S&P500 is the independent variable. In this case, the excess return for S&P500 can also be written as the return of market – risk-free rate, the same as the calculation for 'ersandp' in the first code. It provides a better representation of the CAPM

```
In [7]: formula = 'erford ~ ersandp'
        results = smf.ols(formula, data).fit()
        print(results.summary())

                            OLS Regression Results
==============================================================================
Dep. Variable:                 erford   R-squared:                       0.337
Model:                            OLS   Adj. R-squared:                  0.334
Method:                 Least Squares   F-statistic:                     97.26
Date:                Sun, 17 Sep 2023   Prob (F-statistic):           8.36e-19
Time:                        22:48:00   Log-Likelihood:                -735.26
No. Observations:                 193   AIC:                             1475.
Df Residuals:                     191   BIC:                             1481.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.9560      0.793     -1.205      0.230      -2.520       0.608
ersandp        1.8898      0.192      9.862      0.000       1.512       2.268
==============================================================================
Omnibus:                       71.412   Durbin-Watson:                   2.518
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              677.387
Skew:                           1.079   Prob(JB):                     8.08e-148
Kurtosis:                      11.921   Cond. No.                         4.16
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

formula. The resulting coefficient has -0.9560 for α, which is debatable as α is the risk-free rate in CAPM and it's unlikely for risk-free investment to have negative return. The t-score of -1.205 and p-value of 0.230 are both not enough to reject the null hypothesis of α=0. Therefore, the α=0.9560 is not significant, and by interpreting α as 0, the resulting expected return of Ford is only reliant on its relative risk to the market and market risk premium. Ford offers a return that is around its risk-adjusted expected return, its performance is close to the market performance. The β value resulted in 1.8898, with strong significance to reject the null hypothesis as its t-score is high and the p-value is extremely small. It is expected for Ford to have βa value larger than 1 as the graphical depiction of Ford showed its strong volatility. Also, it indicates Ford is sensitive to market movements, which is again proved in the graph where Ford moved in conjunction with the S&P500, but with a larger rate of change to prove its sensitivity.

4. Finally, the F-test is performed again with the null hypothesis being β=1. Along with the Python coding to calculate critical F-statistic at a 1% significance level with the given degrees of freedom. The f-score is large enough and the p-value is small enough, with a significance level of 0.01, to reject the null hypothesis. Estimated β is significant.

```
In [8]: # F-test: hypothesis testing
        formula = 'erford ~ ersandp'
        hypotheses = 'ersandp = 1'

        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)

<F test: F=21.560441202492132, p=6.365321036035383e-06, df_denom=191, df_num=1>
```

```
In [9]: import scipy.stats as stats

        alpha = 0.01  # Significance level
        df_denom = 191  # Degrees of freedom in the denominator
        df_num = 1  # Degrees of freedom in the numerator

        # Calculate the critical F-statistic
        critical_f_statistic = stats.f.ppf(1 - alpha, df_num, df_denom)

        print("Critical F-Statistic:", critical_f_statistic)

Critical F-Statistic: 6.769437739013415
```

Chapter 6

Multi-hypothesis_CAPM.py

1. The following codes retract the results from the previous chapter, which was saved as capm.pickle file. The f-test this time is to see if both α and β is equal to 1, which can be seen on the coding of hypotheses = 'erford = intercept = 1'. With the f-statistic of 1% critical value being 4.72, f-score of 12.94, and p-value of approximately 0 strongly rejects the null hypothesis that both estimates to be 1. The result is expected as β was already proven not to be equal to 1 in the previous chapter.

```
In [9]: import pandas as pd
        import numpy as np
        import pickle

        data = pd.read_excel('/Users/kazukili/Desktop/Python/Excel/capm2.xlsx', index_col=0)

        def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_sandp' : LogDiff(data['SANDP']),
                             'ret_ford' : LogDiff(data['FORD']),
                             'USTB3M' : data['USTB3M']/12,
                             'ersandp' : LogDiff(data['SANDP']) - data['USTB3M']/12,
                             'erford' : LogDiff(data['FORD']) - data['USTB3M']/12})

        with open('capm.pickle', 'wb') as handle:
            pickle.dump(data, handle)
```

```
In [10]: with open('capm.pickle', 'rb') as handle:
             data = pickle.load(handle)
```

```
In [11]: import statsmodels.formula.api as smf
         # F-test: multiple hypothesis tests
         formula = 'erford ~ ersandp'
         hypotheses = 'ersandp = Intercept = 1'

         results = smf.ols(formula, data).fit()
         f_test = results.f_test(hypotheses)
         print(f_test)

<F test: F=12.94374019574073, p=5.349256152606565e-06, df_denom=191, df_num=2>
```

```
In [12]: import scipy.stats as stats

         alpha = 0.01  # Significance level
         df_denom = 191  # Degrees of freedom in the denominator
         df_num = 2  # Degrees of freedom in the numerator

         # Calculate the critical F-statistic
         critical_f_statistic = stats.f.ppf(1 - alpha, df_num, df_denom)

         print("Critical F-Statistic:", critical_f_statistic)

Critical F-Statistic: 4.718011183428308
```

Chapter 7

APT.py

1. The following codes are used to form the APT formula of Microsoft's expected return based on the following risk factors and economic indicators. S&P500 for the market risk, CPI for consumer purchasing power, IPI to show the price changes in the manufacturing and industrial sector, M1 supply for the supply of liquid forms of money, and consumer credit series to show their interaction with creditors. US treasury bills' returns over different maturing dates are also included, which is apparently used for credit spread series. In this example, the credit spread series is calculated by the difference in the yields of AAA bonds to BAA bonds. The government-issued low-risk US treasury bills should be the AAA bonds in this case, but it's unclear to me if the BAA bonds used in this example are included in the Excel data.

```
In [1]: import pandas as pd
        import numpy as np
        import statsmodels.formula.api as smf

        data = pd.read_excel('/Users/kazukili/Desktop/Python/Excel/macro.xlsx', index_col=0)
        data.head()
```

Out[1]:

Date	MICROSOFT	SANDP	CPI	INDPRO	M1SUPPLY	CCREDIT	BMINUSA	USTB3M	USTB10Y
1986-03-01	0.095486	238.899994	108.8	56.5414	624.3	606.7990	1.50	6.76	7.78
1986-04-01	0.111979	235.520004	108.6	56.5654	647.0	614.3669	1.40	6.24	7.30
1986-05-01	0.121528	247.350006	108.9	56.6850	645.7	621.9152	1.20	6.33	7.71
1986-06-01	0.106771	250.839996	109.5	56.4959	662.8	627.8910	1.21	6.40	7.80
1986-07-01	0.098958	236.119995	109.5	56.8096	673.4	633.6083	1.28	6.00	7.30

2. Since APT focuses on the effect of unexpected changes in these economic indicators, the value of the indicators at a specific time isn't worth much for APT. In this example, the consumers are expected to be naïve and expect the next period's value to remain the same as the current value. Therefore, the unexpected changes in this case will simply be the difference between each period as the consumer expects no change in the first place. The differences are calculated in the following code.

```
In [2]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'dspread' : data['BMINUSA'] -
                                         data['BMINUSA'].shift(1),
                             'dcredit' : data['CCREDIT'] - \
                                         data['CCREDIT'].shift(1),
                             'dprod' : data['INDPRO'] - \
                                       data['INDPRO'].shift(1),
                             'rmsoft' : LogDiff(data['MICROSOFT']),
                             'rsandp' : LogDiff(data['SANDP']),
                             'dmoney' : data['M1SUPPLY'] - \
                                        data['M1SUPPLY'].shift(1),
                             'inflation' : LogDiff(data['CPI']),
                             'term' : data['USTB10Y'] - data['USTB3M'],
                             'dinflation' : LogDiff(data['CPI']) - \
                                            LogDiff(data['CPI']).shift(1),
                             'mustb3m' : data['USTB3M']/12,
                             'rterm' : (data['USTB10Y'] - data['USTB3M']) - \
                                       (data['USTB10Y'] - data['USTB3M']).shift(1),
                             'ermsoft' : LogDiff(data['MICROSOFT']) - \
                                         data['USTB3M']/12,
                             'ersandp' : LogDiff(data['SANDP']) - \
                                         data['USTB3M']/12})
        data.head()
```

Out[2]:

Date	dspread	dcredit	dprod	rmsoft	rsandp	dmoney	inflation	term	dinflation	mustb3m	rterm	ermsoft	ersandp
1986-03-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.02	NaN	0.563333	NaN	NaN	NaN
1986-04-01	-0.10	7.5679	0.0240	15.833171	-1.424818	22.7	-0.183993	1.06	NaN	0.520000	0.04	15.413171	-1.944818
1986-05-01	-0.20	7.5483	0.1196	8.183334	4.900851	-1.3	0.275862	1.38	0.459655	0.527500	0.32	7.655834	4.373351
1986-06-01	0.01	5.9758	-0.1891	-12.945833	1.401091	17.1	0.549452	1.40	0.273590	0.533333	0.02	-13.479167	0.867757
1986-07-01	0.07	5.7173	0.3137	-7.598084	-6.047514	10.6	0.000000	1.30	-0.549452	0.500000	-0.10	-8.098084	-6.547514

3. The APT formula is then set up with the expected return of Microsoft equal to the sum of all the risk factors. The OLS regression derives estimates of the coefficient for all the factors in the formula. By just looking at the coefficient, the Expected return of Microsoft is most sensitive to 'rterm', which should be the change in yield spread of US treasury bonds of different maturing dates. Most of the coefficients have t-values and p-values that may not prove their significance. However, it still rejects the null hypothesis of all coefficients being 0 as some coefficients have extremely low p values close to 0. The f-statistic is also high.

```
In [4]: formula = 'ermsoft ~ ersandp + dprod + dcredit +                    dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

OLS Regression Results

Dep. Variable:	ermsoft	R-squared:	0.345
Model:	OLS	Adj. R-squared:	0.333
Method:	Least Squares	F-statistic:	28.24
Date:	Mon, 18 Sep 2023	Prob (F-statistic):	3.52e-31
Time:	14:28:18	Log-Likelihood:	-1328.3
No. Observations:	383	AIC:	2673.
Df Residuals:	375	BIC:	2704.
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.3260	0.475	2.789	0.006	0.391	2.261
ersandp	1.2808	0.094	13.574	0.000	1.095	1.466
dprod	-0.3030	0.737	-0.411	0.681	-1.752	1.146
dcredit	-0.0254	0.027	-0.934	0.351	-0.079	0.028
dinflation	2.1947	1.264	1.736	0.083	-0.291	4.681
dmoney	-0.0069	0.016	-0.441	0.659	-0.037	0.024
dspread	2.2601	4.140	0.546	0.585	-5.881	10.401
rterm	4.7331	1.716	2.758	0.006	1.359	8.107

Omnibus:	21.147	Durbin-Watson:	2.097
Prob(Omnibus):	0.000	Jarque-Bera (JB):	63.505
Skew:	-0.006	Prob(JB):	1.62e-14
Kurtosis:	4.995	Cond. No.	293.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

4. This time the f-test is done on the null hypothesis stating that dprod, dcredit, dmoney, and dspread equals 0. The result shows an f-score lower than the f-statistic for a 5% significance level and p-value of 0.8. Botfaills to reject the null hypothesis.

```
In [5]: hypotheses = 'dprod = dcredit = dmoney = dspread = 0'

        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=0.4138785595228427, p=0.7986453783393681, df_denom=375, df_num=4>
```

```
In [6]: import scipy.stats as stats

        alpha = 0.05  # Significance level
        df_denom = 375  # Degrees of freedom in the denominator
        df_num = 4  # Degrees of freedom in the numerator

        # Calculate the critical F-statistic
        critical_f_statistic = stats.f.ppf(1 - alpha, df_num, df_denom)

        print("Critical F-Statistic:", critical_f_statistic)
```

```
Critical F-Statistic: 2.395744983869647
```

Step-wise_reg.py

1. Since the previous regression has various factors that pose no significance; therefore, step-wise regression is used to rebuild the expected return of Microsoft by selecting only the factors that matter. Uni-directional forward method is used where it's a forward step-wise regression that adds variables to the regression based on their significance. The P-value is used as the criteria and with a lower p-value indicating higher significance, the lowest p-value is selected first. The algorithm for this step-wise regression stops when the p-value reaches 0.2, which is done in the coding by removing factors that have a p-value larger than 0.2 from the step-wise regression candidates.

2. Due to the errors that resulted in my Python, my step-wise regression included 'mustb3m' and 'inflation', which aren't part of the original factors included in the APT regression model. Instead, the result should be only ersandp, rterm and dinflation

```
In [8]: res = forward_selected(data,'ermsoft','ersandp')
        print(res.model.formula)
ermsoft ~ ersandp + rterm + mustb3m + dinflation + inflation
```

3. The result is proven to be wrong due to the errors in the following code, which shows variables with p-values higher than the pre-specified 0.2. While the result is incorrect, the adjusted r-squared value, which can be obtained from a separate code, of 0.343 indicates that with the regression built only the selected variables can explain 34.3% of the results in the dependent variable.

```
In [9]: print(res.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                ermsoft   R-squared:                       0.351
Model:                            OLS   Adj. R-squared:                  0.343
Method:                 Least Squares   F-statistic:                     40.83
Date:                Mon, 18 Sep 2023   Prob (F-statistic):           1.55e-33
Time:                        15:36:45   Log-Likelihood:                -1326.5
No. Observations:                 383   AIC:                             2665.
Df Residuals:                     377   BIC:                             2689.
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.0963      0.670     -0.144      0.886      -1.414       1.222
ersandp        1.2709      0.092     13.855      0.000       1.091       1.451
rterm          4.8564      1.716      2.831      0.005       1.483       8.230
mustb3m        4.3107      1.925      2.240      0.026       0.526       8.095
dinflation     2.3406      1.418      1.650      0.100      -0.448       5.129
inflation     -0.2901      1.517     -0.191      0.848      -3.273       2.693
==============================================================================
Omnibus:                       26.016   Durbin-Watson:                   2.113
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               83.347
Skew:                          -0.167   Prob(JB):                     7.97e-19
Kurtosis:                       5.261   Cond. No.                         23.3
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

```
In [11]: print(res.rsquared_adj)
0.342683870204411105
```

Chapter 8

CAPM_quantreg.py.py

1. The following code retracts the CAPM formula estimated for Ford with S&P500 as the predictor variable. Quantile regression is formed based on the Ford's expected variable at 50% quantile.

2. The estimates for different quantiles can be obtained efficiently, by replacing the value of quantile q from a single estimate, like 0.5 in the first code, to x, which is defined in the code as 0.1 to 1.0 with a 0.1 interval.

```
In [1]: import pandas as pd
        import numpy as np
        import statsmodels.formula.api as smf
        import pickle
        import matplotlib.pyplot as plt

        with open('capm.pickle', 'rb') as handle:
            data = pickle.load(handle)
        data = data.dropna()
```

```
In [2]: # regression
        # quantile(50)
        res = smf.quantreg('erford ~ ersandp', data).fit(q=0.5)
        print(res.summary())
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                erford   Pseudo R-squared:               0.1724
Model:                       QuantReg   Bandwidth:                       5.340
Method:                 Least Squares   Sparsity:                       16.78
Date:                Mon, 18 Sep 2023   No. Observations:                 193
Time:                        15:52:03   Df Residuals:                     191
                                        Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -1.4896      0.606     -2.457      0.015      -2.685      -0.294
ersandp        1.4384      0.146      9.822      0.000       1.150       1.727
==============================================================================
```

3. As the results are quite long and need to scroll through to view the results for each percentile, the key results of coefficient of intercept, represented as α, and coefficient of ersandp β as well as the lower and upper bounds of 95% confidence interval are put into a single table. The code also includes the fitted values of y across percentiles. While it is expected for alpha to be lower for lower quantiles, as the y-value is naturally smaller in lower quantiles, the beta happens to be much higher only at the 10% quantile. It implies that the excess return of Ford is lower, it's much more sensitive to the excess return of S&P500. It implies that Ford is vulnerable to extreme events in the market.

```
In [3]: # Simultaneous-quantile regression
        # 10 20 30 40 50 60 70 80 90, ten quantiles
        quantiles = np.arange(0.10, 1.00, 0.10)

        for x in quantiles:
            print('-----------------------------------------------------------')
            print('{0:0.01f} quantile'.format(x))
            res = smf.quantreg('erford ~ ersandp', data).fit(q=x)
            print(res.summary())
```

```
                                                 Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -3.3479      0.641     -5.223      0.000      -4.612      -2.084
ersandp        1.5005      0.155      9.669      0.000       1.194       1.807
==============================================================================

-----------------------------------------------------------
0.5 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:               0.1724
Model:                       QuantReg   Bandwidth:                       5.340
Method:                 Least Squares   Sparsity:                        16.78
Date:                Mon, 18 Sep 2023   No. Observations:                  193
Time:                        16:16:32   Df Residuals:                      191
                                        Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -1.4896      0.686     -2.457      0.015      -2.685      -0.294
```

```
In [4]: def model_paras(data, quantiles):
            '''
            Parameters:
            ------------
                data: pandas DataFrame with dependent and independent variables
                quantiles: quantile number

            Returns:
            ------------
                quantreg_res: pandas DataFrame with model parameters for each
                              quantile regression specification
                y_hat: pandas DataFrame with all the fitted value of y
            '''
            parameters = []
            y_pred = {}
            for q in quantiles:
                res = smf.quantreg('erford ~ ersandp', data).fit(q=q)
                # obtain regression's parameters
                alpha = res.params['Intercept']
                beta = res.params['ersandp']
                lb_pval = res.conf_int().loc['ersandp'][0]
                ub_pval = res.conf_int().loc['ersandp'][1]
                # obtain the fitted value of y
                y_pred[q] = res.fittedvalues
                # save results to lists
                parameters.append((q,alpha,beta,lb_pval,ub_pval))

            quantreg_res = pd.DataFrame(parameters, columns=['q', 'alpha',
            y_hat = pd.DataFrame(y_pred)
            return quantreg_res, y_hat
```

```
In [5]: quantreg_paras, y_hats = model_paras(data, quantiles)
```

```
In [6]: print(quantreg_paras)
```

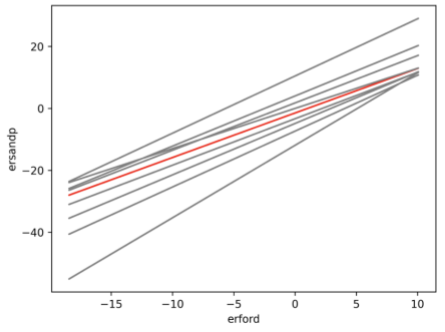
```
    q      alpha      beta        lb        ub
0  0.1 -11.929063  2.340422  1.653477  3.027366
1  0.2  -7.349600  1.804414  1.437253  2.171575
2  0.3  -4.878332  1.659638  1.338293  1.980982
3  0.4  -3.347896  1.500533  1.194442  1.806624
4  0.5  -1.489629  1.438449  1.149566  1.727332
5  0.6  -0.018844  1.296706  1.001296  1.592115
6  0.7   1.756783  1.528341  1.169294  1.887389
7  0.8   4.001336  1.619863  1.124772  2.114954
8  0.9  10.456266  1.847333  0.469290  3.225376
```

```
In [7]: y_hats.head()
```

Out[7]:

Date	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	
2002-02-01	-17.183667	-11.400785	-8.604472	-6.716822	-4.719166	-2.930147	-1.674577	0.364496	6
2002-03-01	-3.841712	-1.114428	0.856563	1.837213	3.480946	4.461936	7.037983	9.598789	16
2002-04-01	-27.105056	-19.049959	-15.639919	-13.077801	-10.816960	-8.427072	-8.153439	-6.502338	-1
2002-05-01	-14.407484	-9.260408	-6.635828	-4.936906	-3.012894	-1.392009	0.138325	2.285960	8
2002-06-01	-29.869803	-21.181518	-17.600453	-14.850385	-12.516204	-9.958874	-9.958874	-8.415887	-3

```
In [10]: plt.figure(1, dpi=600)
         for i in quantreg_paras.q:
             x = data['ersandp']
             y = y_hats[i]
             if i == 0.50:
                 plt.plot(x,y,color='red')
             else:
                 plt.plot(x,y,color='grey')

         plt.ylabel('ersandp')
         plt.xlabel('erford')
         plt.show()
```



4. The quantile regression results can also be graphically depicted. It is coded so that the 0.5 quantile regression model is in red and the rest as gray. The results are just as described numerically in the table from the previous code. The intercept for the percentiles is ordered in ascension from 0.1 to 0.9. 0.1 percentile having the highest beta/slope.

Chapter 9

PCA.py

1. Excel data with the monthly return for bonds of different maturing dates over the years are given. The data is imported into the jupyter notebook. The following codes are used to calculate the eigenvectors for the data, which is already in the form of returns on 'm' rows and the different bonds on 'n' columns as specified in the code. This m-by-n matrix M is then standardized with the M = code, where it's reduced by the mean so the data are centered around 0 and divided by the standard deviation to cancel out the unit. The calculation for the covariance matrix, the eigenvalue of that matrix then the calculation of eigenvector for each corresponding eigenvalue are all done with the Python command. The resulting table shows the coefficients of each principal component, which are the eigenvectors calculated, for each bond. The contribution of each variable in determining each PC can be observed in the table.

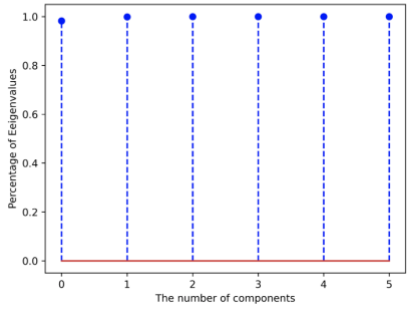
```
In [5]: def princomp(data):
            '''
            Computing eigenvalues and eigenvectors of covariance matrix

            Parameters:
            ----------
            data: the m-by-n DataFrame which corresponds m rows of observations
                  and n columns of variables.

            Returns:
            ----------
            coeff: a n-by-n matrix (eigenvectors) where it contains all coefficients
                   of principal component for each variable.

            latent: a vector of the eigenvalues
            '''

            M = data.apply(lambda x: (x-mean(x))/std(x) ) # normalize

            # Note: cov inputs require a column vector;
            # That is each row of m represents a variable,
            # and each column is a single observation of all those variables
            # To tackle this, simply add argument rowvar=True
            # or transpose M matrix
            [latent,coeff] = linalg.eig(cov(M.transpose()))
            # attention: latent (eigenvalues) is not always sorted
            latent = sort(latent)[::-1]

            # convert arrays to DataFrame or Series
            coeff = pd.DataFrame(coeff.T, columns=data.columns)
            latent = pd.Series(latent, name='Eigenvalue')

            return coeff, latent

        coeff, latent = princomp(data)

        coeff
```

Out[5]:

	GS3	GS1	GS10	GS5	GS3M	GS6M
0	0.411254	0.410250	0.403091	0.409037	0.407122	0.408684
1	-0.120079	0.264723	-0.641679	-0.366564	0.463403	0.393242
2	0.544686	0.340138	-0.508784	0.211559	-0.527801	-0.073690
3	-0.290093	0.549916	0.346741	-0.412648	-0.485797	0.294843
4	0.505527	-0.530655	0.111503	-0.418840	-0.165136	0.497712
5	-0.424330	-0.248837	-0.185049	0.554849	-0.278686	0.581599

2. In the following code, the cumulative sum of eigenvalues is divided by the sum of the eigenvalues, which is also the variance as the data is already standardized. This gives the fraction of the total variance explained by the principal components. The result shows the first PC captures 97% of the variations in the data, which rises to almost 100% after the second PC is added. This is visually represented in the graph, where the eigenvalue is already extremely high in the first PC. As the first two PCs already capture the majority of the total variation, it is possible to use only the first two PCs and graph the data onto a 2-dimensional graph with the two PCs on the axis. The command for dot product is also apparently available in Python and graphing can be done with importing the required libraries.

```
In [6]: perc_lat = cumsum(latent)/sum(latent)
        print(perc_lat)

0    0.982335
1    0.998853
2    0.999718
3    0.999952
4    0.999985
5    1.000000
Name: Eigenvalue, dtype: float64

In [7]: plt.figure(1, dpi=600)
        plt.stem(range(len(perc_lat)),perc_lat,'--b')
        plt.ylabel('Percentage of Eeigenvalues')
        plt.xlabel('The number of components')
        plt.show()
```



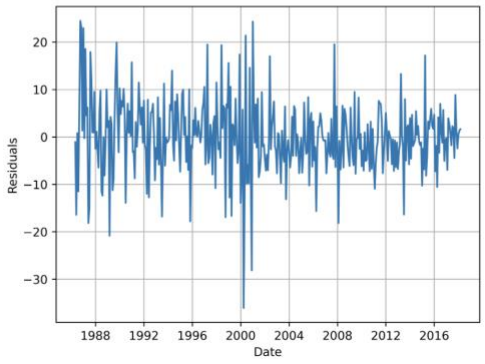
Chapter 10

Heteroskedaticity_testing.py

1. The data from macro.pickle, which is the APT regression of Microsoft's excess return, is retracted. As the file does not contain the actual OLS regression results, the regression model is reconstructed in the code. While the OLS regression results in Python include information like standard errors that show the standard deviations of the residuals, the specific values for them are not given. To test for heteroskedasticity for the model's residuals, the actual values of residuals are obtained with results.resid, which can be graphed with the following codes. However, it is not enough to claim if the residuals are heteroskedastic or not without tests.

```
In [2]: import pickle
        import statsmodels.formula.api as smf
        import statsmodels.stats.api as sms
        import matplotlib.pyplot as plt
        from statsmodels.compat import lzip

        with open('macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

        formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
```

```
In [3]: plt.figure(1, dpi=600)
        plt.plot(results.resid)
        plt.xlabel('Date')
        plt.ylabel('Residuals')
        plt.grid(True)
        plt.show()
```



2. The following codes test the null hypothesis being the absence of heteroskedasticity, which is written as residual variance to be independent of the independent variable in the regression model; thus, change in it will not affect the residuals. The results are given with Lagrange multiplier statistic and f-score with their respective p-values. Although the significance value is not chosen, the p-value of 0.87 for both results retains the null hypothesis that there is no heteroskedasticity in the residuals. The tests done on this regression model should be reliable.

```
In [4]: # breusch-pagan heteroskedasticity test
        name = ['Lagrange multiplier statistic', 'p-value',
                'f-value', 'f p-value']
        test = sms.het_breuschpagan(results.resid, results.model.exog)
        lzip(name, test)

Out[4]: [('Lagrange multiplier statistic', 3.16066015042023),
         ('p-value', 0.8697526743786463),
         ('f-value', 0.44577025527121483),
         ('f p-value', 0.8729003404188317)]
```

White_modified_standard_error.py

1. Despite there being no heteroskedasticity detected in the model, improving the regression model with the white modified standard error for the coefficients can still be helpful as the ATP regression model contains more coefficients than usual. As expected, the results did not show significant changes in standard error in comparison to the original regression results (which are in Chapter 7). However, the effect of heteroskedasticity-robust standard errors can be seen in the improved t-score, being larger, and p-value, being smaller.

```
In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit +                    dinflation + dmon
        results = smf.ols(formula, data).fit(cov_type='HC1')
        print(results.summary())
```

OLS Regression Results

Dep. Variable:	ermsoft	R-squared:	0.345
Model:	OLS	Adj. R-squared:	0.333
Method:	Least Squares	F-statistic:	29.89
Date:	Tue, 19 Sep 2023	Prob (F-statistic):	9.27e-33
Time:	12:12:51	Log-Likelihood:	-1328.3
No. Observations:	383	AIC:	2673.
Df Residuals:	375	BIC:	2704.
Df Model:	7		
Covariance Type:	HC1		

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	1.3260	0.459	2.888	0.004	0.426	2.226
ersandp	1.2808	0.093	13.778	0.000	1.099	1.463
dprod	-0.3030	0.635	-0.478	0.633	-1.547	0.941
dcredit	-0.0254	0.021	-1.219	0.223	-0.066	0.015
dinflation	2.1947	1.307	1.679	0.093	-0.367	4.756
dmoney	-0.0069	0.011	-0.630	0.528	-0.028	0.014
dspread	2.2601	3.428	0.659	0.510	-4.458	8.978
rterm	4.7331	1.727	2.741	0.006	1.349	8.117

Omnibus:	21.147	Durbin-Watson:	2.097
Prob(Omnibus):	0.000	Jarque-Bera (JB):	63.505
Skew:	-0.006	Prob(JB):	1.62e-14
Kurtosis:	4.995	Cond. No.	293.

Notes:
[1] Standard Errors are heteroscedasticity robust (HC1)

Newey_west_standard_error.py.py

1. In this code, the Newey-West procedure is used to detect heteroskedasticity and to derive standard errors robust to autocorrelation in residuals. Similarly to the code for white modified standard error, the Newey-West procedure can be commanded by the code HAC after cov-type=. The lag length is substituted in the next line of code, which determines the existence of possible autocorrelation up to 6 months back.

```
In [2]: import statsmodels.formula.api as smf
        import pickle

        with open('macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [3]: formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit(cov_type='HAC',
                                             cov_kwds={'maxlags':6,'use_correction':True})
        print(results.summary())
```

OLS Regression Results

Dep. Variable:	ermsoft	R-squared:	0.345
Model:	OLS	Adj. R-squared:	0.333
Method:	Least Squares	F-statistic:	24.93
Date:	Tue, 19 Sep 2023	Prob (F-statistic):	6.73e-28
Time:	12:28:22	Log-Likelihood:	-1328.3
No. Observations:	383	AIC:	2673.
Df Residuals:	375	BIC:	2704.
Df Model:	7		
Covariance Type:	HAC		

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	1.3260	0.503	2.638	0.008	0.341	2.311
ersandp	1.2808	0.100	12.822	0.000	1.085	1.477
dprod	-0.3030	0.522	-0.581	0.561	-1.325	0.719
dcredit	-0.0254	0.022	-1.135	0.256	-0.069	0.018
dinflation	2.1947	1.314	1.671	0.095	-0.380	4.769
dmoney	-0.0069	0.011	-0.628	0.530	-0.028	0.015
dspread	2.2601	2.842	0.795	0.426	-3.310	7.830
rterm	4.7331	1.759	2.692	0.007	1.286	8.180

Omnibus:	21.147	Durbin-Watson:	2.097
Prob(Omnibus):	0.000	Jarque-Bera (JB):	63.505
Skew:	-0.006	Prob(JB):	1.62e-14
Kurtosis:	4.995	Cond. No.	293.

Notes:
[1] Standard Errors are heteroscedasticity and autocorrelation robust (HAC) using 6 la
gs and with small sample correction

Autocorrelation.py.py

1. Based on the data from macro.pickle again, the autocorrelation for the residuals is tested with Durbin and Watson, which is restricted to first-order autocorrelation. The result of 2.10 indicates a negative correlation between the residual and its immediately previous one; thus, if the residual is larger or smaller than the average value, the residual at the next immediately succeeding point moves toward the mean.

```
In [2]: # durbin_watson
        formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()

        residuals = results.resid
        sms.durbin_watson(residuals)

Out[2]: 2.0973940504299917
```

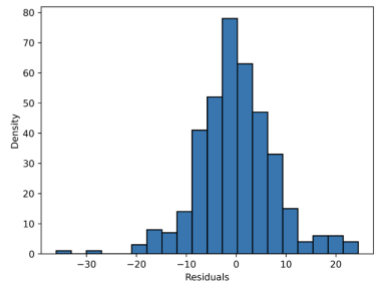
2. Another test is done for testing higher-order autocorrelation through the Breusch-Godfrey test. Just like the Durbin and Watson test, the Breusch-Godfrey test is also implemented as a command in Python. The lag length is set at 10 and the resulting p-values, of around 0.9, indicate no signs or negligible autocorrelation in 10 months period. It seems that in the short run, there is a sign of reversion to the mean, but in the residuals of the 10-month space, there is no sign of autocorrelation.

```
In [4]: name = ['Lagrange multiplier statistic', 'p-value',
                'f-value', 'f p-value']
        results1 = sms.acorr_breusch_godfrey(results, 10)
        lzip(name, results1)

Out[4]: [('Lagrange multiplier statistic', 4.766591436782221),
         ('p-value', 0.9062145800242334),
         ('f-value', 0.45998207324796836),
         ('f p-value', 0.915017998157289)]
```

Non-normality.py

1. The macro.pickle is loaded in this code to determine whether the residuals are normally distributed or not. As normal distributions are bell-shaped, the histogram of the residuals is coded. The histogram is bell-shaped but with few outliers at the negative values of residuals, which might indicate a negative skew. The cluster of high-valued residuals may lead to leptokurtic distribution.

```
In [3]: formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
results = smf.ols(formula, data).fit()
residuals = results.resid

plt.figure(1, dpi=600)
plt.hist(residuals,20,edgecolor='black',linewidth=1.2)
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()
```



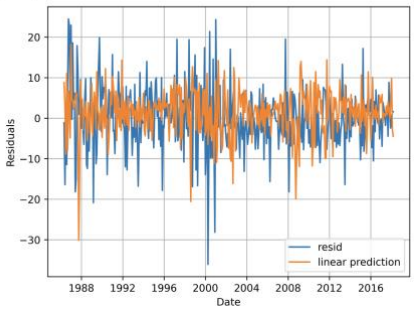
2. More accurate tests for the normality of the residuals are tested with the Jarque-Bera test, which calculates a test statistic based on the skewness and kurtosis of the data. The p-value of the Jarque-Bera statistics is then calculated to be used for comparison with the chosen level of significance. The result strongly denies the null hypothesis of it being a normal distribution, but it is expected from the histogram that correctly displayed the possibility of negative skewness and positive kurtosis.

```
In [4]: name = ['Jarque-Bera', 'Chi^2 two-tail prob.', 'Skew', 'Kurtosis']
test = sms.jarque_bera(residuals)
lzip(name, test)

Out[4]: [('Jarque-Bera', 63.50547267158968),
 ('Chi^2 two-tail prob.', 1.6216675409916346e-14),
 ('Skew', -0.005612677479835745),
 ('Kurtosis', 4.99482560590665)]
```

Dummy_variable.py.py

1. The comparison between the residuals and the fitted variables, of the Microsoft excess return in the macro.pickle, can be visually graphed. 2 obvious outliers of the negative residuals are located around 2000 and 2001, which is likely the two negative outliers found in the histogram in the previous code.

```
In [3]: plt.figure(1,dpi=600)
plt.plot(residuals, label='resid')
plt.plot(y_fitted, label='linear prediction')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.grid(True)
plt.legend()
plt.show()
```



2. The specific values of those 2 residuals as well as their dates can be found located with the following code, which lists the two smallest residuals.

```
In [4]: residuals.nsmallest(2)

Out[4]: Date
2000-04-01   -36.075347
2000-12-01   -28.143156
dtype: float64
```

3. The effects of these outliers are mitigated by removing, by setting them as 0, them with a dummy variable of binary formats that sets the residual to 0 when the dummy variable equals 1 and leaves the residual unchanged when the dummy variable equals 0. Two separate dummy variable is created for the two outliers, which become 1 at the respective dates. Both dummy variables are proven to be significant with their low p-value and a slight increase in the adjusted R-squared value indicates the improved fit of the regression model to the data, which is just from fitting to the two outliers.

```
In [5]: data['APR00DUM'] = np.where(data.index == '2000-4-1', 1, 0)
data['DEC00DUM'] = np.where(data.index == '2000-12-1', 1, 0)

# regression
formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm +
results = smf.ols(formula, data).fit()
print(results.summary())
```

OLS Regression Results

Dep. Variable:	ermsoft	R-squared:	0.406
Model:	OLS	Adj. R-squared:	0.392
Method:	Least Squares	F-statistic:	28.33
Date:	Tue, 19 Sep 2023	Prob (F-statistic):	2.22e-37
Time:	16:42:41	Log-Likelihood:	-1309.7
No. Observations:	383	AIC:	2639.
Df Residuals:	373	BIC:	2679.
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.4198	0.454	3.125	0.002	0.526	2.313
ersandp	1.2539	0.090	13.897	0.000	1.076	1.431
dprod	-0.3211	0.705	-0.456	0.649	-1.707	1.065
dcredit	-0.0157	0.026	-0.603	0.547	-0.067	0.035
dinflation	1.4421	1.215	1.187	0.236	-0.946	3.830
dmoney	-0.0057	0.015	-0.383	0.702	-0.035	0.024
dspread	1.8693	3.955	0.473	0.637	-5.908	9.647
rterm	4.2642	1.641	2.599	0.010	1.038	7.490
APR00DUM	-37.0288	7.576	-4.888	0.000	-51.926	-22.132
DEC00DUM	-28.7300	7.546	-3.807	0.000	-43.569	-13.891

Omnibus:	20.084	Durbin-Watson:	2.088
Prob(Omnibus):	0.000	Jarque-Bera (JB):	28.877
Skew:	0.404	Prob(JB):	5.36e-07
Kurtosis:	4.076	Cond. No.	560.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Multicollinearity.py.py

1. The multicollinearity of the ATP regression model, from the macro.pickle, can be checked by looking for correlation amongst the independent variables in the correlation matrix. Excluding the leading diagonal, which should be 1 as it compares the same variable, there doesn't seem to be any high correlation among the independent variables

```
In [2]: data = data[['dprod','dcredit','dinflation','dmoney','dspread','rterm']]
        data.corr()
```

Out[2]:

	dprod	dcredit	dinflation	dmoney	dspread	rterm
dprod	1.000000	0.094273	-0.143551	-0.052514	-0.052756	-0.043751
dcredit	0.094273	1.000000	-0.024604	0.150165	0.062818	-0.004029
dinflation	-0.143551	-0.024604	1.000000	-0.093571	-0.227100	0.041606
dmoney	-0.052514	0.150165	-0.093571	1.000000	0.170699	0.003801
dspread	-0.052756	0.062818	-0.227100	0.170699	1.000000	-0.017622
rterm	-0.043751	-0.004029	0.041606	0.003801	-0.017622	1.000000

Reset_ramsey.py.py

1. The APT regression model, from the macro.pickle, is tested with RESET to see if the regression model is missing any more non-linear variables. With the degrees of freedom set at 4, the RESET test in this example considers the addition of up to fourth-order terms. The resulting regression model did not have a high enough p-value of the f-score to reject the null hypothesis of the original regression model not needing any more variables. The regression with potentially omitted variables also has a smaller f-score compared to the original model, indicating that the addition of the new variable caused the model to not fit as well as the original model. The original APT regression model can be said to have included all the necessary risk factors.

```
In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()

        reset_ramsey(results,degree=4)
```

```
Out[2]: <class 'statsmodels.stats.contrast.ContrastResults'>
        <F test: F=0.9937673047317103, p=0.3957441239174355, df_denom=372, df_num=3>
```

Chow_test.py.py

1. Known-structural break test, Chow test is performed on the regression model from the macro.pickle. The break point is set at 1996/01/01 and the residual sum of squares, degrees of freedom, and the number of observations used for f-score calculation is calculated manually in the coding. The result of the f-statistic is 1.99. The degrees of freedom in this example can be calculated with k_total, which is the total number of parameters for the single regression model from the entire data set as the DF for the numerator. The sum of data points used in the separate regression model minus 2 k_total for the denominator's DF. The degrees of freedom are then used to get the 10% f-critical value with Python. 10% might be somewhat high for the critical value, but it allows for the rejection of the null hypothesis that the parameters are indeed stable across two subsamples.

```
In [3]: # split samples
        data1 = data['1996-01-01']
        data2 = data['1996-01-01':]

        # get rss of whole sample
        RSS_total, N_total, K_total = get_rss(data)
        # get rss of the first part of sample
        RSS_1, N_1, K_1 = get_rss(data1)
        # get rss of the second part of sample
        RSS_2, N_2, K_2 = get_rss(data2)

        nominator = (RSS_total - (RSS_1 + RSS_2)) / K_total
        denominator = (RSS_1 + RSS_2) / (N_1 + N_2 - 2*K_total)

        result = nominator/denominator
```

```
In [4]: result
```

```
Out[4]: 1.9894610789616265
```

```
In [5]: import scipy.stats as stats

        # Define the significance level
        alpha = 0.10

        # Define the degrees of freedom
        df_num = 7
        df_denom = 370

        # Calculate the F-statistic for a given significance level
        # The inverse survival function (isf) is used to find the critical value
        f_critical = stats.f.isf(alpha, df_num, df_denom)

        print(f"F-critical value at {alpha*100}% significance level:", f_critical)
```

```
F-critical value at 10.0% significance level: 1.7330569340076045
```

2. The CUSUM test is then performed to identify for structural break without any prior knowledge of the break point. The resulting test statistic is proven to be significant at a 5% critical value and the p-value is also significant at this critical value. Therefore, the null hypothesis of no structural break point is rejected, which doesn't conflict with the Chow test.

```
In [15]: import statsmodels.api as sm
         from statsmodels.compat import lzip
         import statsmodels.stats.diagnostic as sms

         # Fit the OLS regression model
         formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'
         results = smf.ols(formula, data).fit()

         # Get the residuals using the OLSInfluence class
         influence = results.get_influence()
         resid = influence.resid

         # Perform the CUSUM test on the residuals
         name = ['test statistic', 'pval', 'crit']
         test = sms.breaks_cusumolsresid(resid)
         result_summary = lzip(name, test)
         print(result_summary)

         [('test statistic', 1.5494823024826427), ('pval', 0.01643003551635983), ('crit', [(1, 1.63), (5, 1.36), (10,
         1.22)])]
```

3. Another test that is used as an unknown structural break test, is the recursive estimation. The initial sample size is set at 11 and 373 iterations were performed, which means a total of 374 adjustments of the parameters including the first addition of new data that is not included in the iteration. The constantly adjusted parameters through time are put into a graph along with the standard error added or subtracted from the actual parameter beta estimated. The graph shows 3 lines for the value of beta and upper and lower bands of the margin of error. The result shows instability at the start with the regression model based on only 11 data samples, but it can be explained by the lack of data, and each data has a large impact on the regression model. As more data are added, the beta parameter becomes stable.

```
In [16]: def recursive_reg(variable, i, interval):
             '''
             Parameters:
             -----------
                 variable: the string literals of a variable name in regression
                           formula.
                 i: the serial number of regression.
                 interval: the number of consective data points in initial sample

             Returns:
             -----------
                 coeff: the coefficient estimation of the variable
                 se: the standard errors of the variable
             '''
             formula = 'ermsoft ~ ersandp + dprod + dcredit +                    dinflation +
             results = smf.ols(formula, data.iloc[:i+interval]).fit()
             coeff = results.params[variable]
             se = results.bse[variable]

             return coeff, se
```

```
In [17]: parameters = []
         for i in range(373):
             coeff, se = recursive_reg('ersandp', i, 11)

             parameters.append((coeff,se))

         parameters = pd.DataFrame(parameters, columns=['coeff','se'],

         parameters['ersandp + 2*se'] = parameters['coeff'] + 2*parameters['se']
         parameters['ersandp - 2*se'] = parameters['coeff'] - 2*parameters['se']
```

```
In [19]: plt.figure(1, dpi=600)
         plt.plot(parameters['coeff'], label=r'$\beta_{ersandp}$')
         plt.plot(parameters['ersandp + 2*se'], label=r'$\beta_{ersandp} + 2*SE$',        lines
         plt.plot(parameters['ersandp - 2*se'], label=r'$\beta_{ersandp} - 2*SE$',        lines

         plt.xlabel('Date')
         plt.grid(True)
         plt.legend()
         plt.show()
```