CALCULATING BETA FOR A STOCK IN PYTHON

Single and Multiple Linear Regression, Implementations in Excel and Python, and How to Use Python for Finding a Stock's Beta using the CAPM and Fama-French Models

EMI STRATI

Table of Contents

1. Introduction

The purpose of this project is to acquaint finance majors with basic Python concepts and syntax. To do so, I have decided to demonstrate how we can perform a very easy statistical analysis, namely linear regression, to assess a stock's beta. At the end of this document, you should be acquainted with linear regression (theory and implementation), why we use it in finance, and how you can use it in Python to assess a stock's beta.

This guide is part of a bigger package that also includes a document on how to get started with Python and a PowerPoint that goes over basic concepts in Python coding. To make the most out of this guide, you should already have Python and an IDE installed. If you need help with that, refer to the other document titled "How to Set Up Python & IDE" where we go over the options for installing Python and why you should use an IDE.

I have tried my best to make this document as thorough as possible. This project can be easily understood by all audiences, even those with no prior experience in statistical analysis. If you are already knowledgeable about linear regression, feel free to skip to the part of this project that is most useful to you.

I need to state that you can perform linear regression in many easier ways, including the LINEST function in Excel. The purpose of this project is not to convince you to use Python for this process, but rather to give you an idea of the syntax and logic used in Python coding.

If any of the code in this document does not work for you, or if you come across a certain problem with Python that is not covered here, feel free to refer to the internet for a solution. Python is the most used open-source language, which means that you are likely not the first to face a specific issue. Stack Overflow is one of the richest Q&A websites where you will likely find a solution to your problem, or at least an alternative way of going about it.

With that out of the way, I hope you learn something new from this guide.

2. Linear Regression

2.1 What is Regression?

In statistics, regression analysis is a set of statistical methods for estimating the relationship between two (or more) variables in various problems. One of these variables is dependent – meaning it is affected by the other variables in the problem. The other variable(s) are independent, so they affect the dependent variable directly.

We use regression to identify the impact of certain features on a topic of interest. For example, we can study the effect that a certain type of advertisement has on a company's sales, or we can assess the relationship between the number of hours studied and the grades students received in a midterm.

There are different types of regression analysis, the most popular ones being:

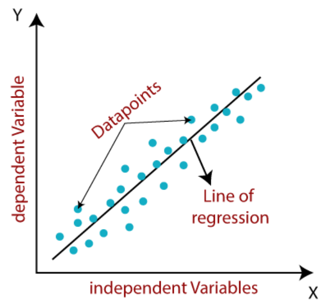
a. Simple Linear Regression – the most basic form of regression algorithms. Consists of a single parameter (or *feature*, or *independent variable*; these words can be used interchangeably) and one dependent variable. These two variables have a linear relationship. We denote simple linear regression by this equation: **$Y = \beta 0 + \beta 1X + \varepsilon$**. In finance, simple linear regression is most used when calculating factor risk in a Single Factor model, such as the Capital Asset Pricing Model (CAPM) (**Section 2.2** – Overview/Excel and **Section 4** - Python).

b. Multiple Linear Regression – Linear regression, but with two or more independent variables. The relationship between these variables and the dependent variable is once again linear, but since this method involves more variables, so we need to make more assumptions (one of them being the absence of multicollinearity). An example where we can use this method is when we want to find out how rainfall, temperature, and amount of fertilizer affect crop growth.

   In finance, this type of regression is used when we estimating the risk of multiple factors in a multifactor model (such as the Fama-French Model) (**Section 2.3** – Overview/Excel and **Section 5** - Python).

c. Logistic Regression – In this method, the dependent variable is binary: it can only take on the values of 0 and 1, or "Yes" and "No". There are many practical examples of logistic regression used in everyday life, for example, email spam detection. The spam detection algorithm takes notice of different email features, such as sender, typos, and unique word occurrences like "prize", "earn extra cash", "free gift", etc. These features are extracted to produce a feature vector that trains a logistic classifier. The email then receives a score, and if that score is higher than a certain threshold, the email gets marked as spam.

2.2 Simple Linear Regression



Simple linear regression is one of the most widely used regression algorithms. One of its biggest advantages is how simple it is to implement and interpret. As we mentioned above, it consists of only two variables: one independent (x) and one dependent (y). Linear regression seeks to find the linear relationship between these two variables, which means that the dependent variable (y) is changing according to the value of the independent (x) variable.

Linear regression estimates the coefficients of the linear equation. This form of analysis fits a straight line that minimizes the discrepancies between predicted and actual values. Mathematically, we represent the relationship between the two variables as:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon$$

Where:

Y = dependent variable
X = independent variable
β_0 = intercept of the line
β_1 = linear regression coefficient
ε = random error

Linear regression has 4 key assumptions:

1 – **Linearity**. The relationship between X and Y is linear.
2 – **Independence**. The residuals (or errors) are independent. Residuals are the differences between observed and predicted values of data. They are useful for assessing the quality of a model.
3 – **Homogeneity of variance**. The residuals have constant variance at every level of X.
4 – **Normality**. The residuals of the model are normally distributed.

If any of these assumptions are violated, the results of the linear regression are unreliable.

2.2.1 Simple Linear Regression in CAPM

In finance, simple linear regression is mostly used to calculate the risk of a factor in a single factor model. The most popular single factor model is the Capital Asset Pricing Model (CAPM), which is used to estimate the expected return of an investment based on its riskiness in respect to a benchmark (usually the S&P 500). The riskiness of a stock is expressed through its Beta (β), which captures its volatility relative to the market's volatility, known as *systematic* risk.

To understand beta, think of it this way: the market (benchmark used in regression) is given a value of 1. If a particular stock's beta is larger than 1, then the stock is more volatile than the overall market, and it will amplify its movements. If it is smaller than 1, then it is less volatile, meaning that it will ease market movements, and have less volatile price swings.

The formula for beta is as follows:

$$Beta\ Coefficient\ (\beta) = \frac{Covariance(R_e, R_m)}{Variance(R_m)}$$

Where:

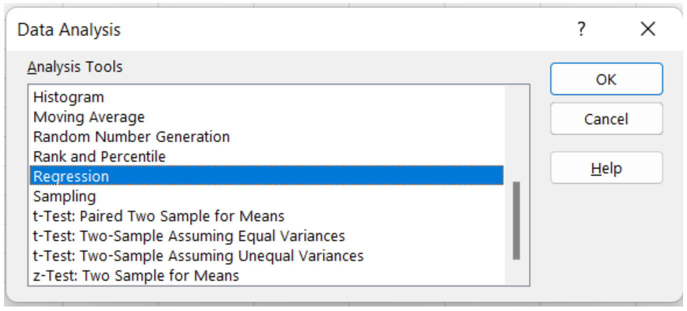
R_e = the return on an individual stock
R_m = the return on the overall market
Covariance = how changes in the return of the stock are related to changes in market returns
Variance = how far the market's data points spread out from their average value

Here are the steps on how to calculate the beta coefficient for a stock (check out the Excel sheet attached to this package):

1. Collect monthly data for a stock's price going back 5 years (60 months). The time frame choice is dependent on the analyst/investor. Some may choose 3 years, others may choose 1 year of weekly data.
   You can use FactSet, Yahoo! Finance, and many more websites for this step.
2. Collect monthly price data for the benchmark of choice.
   If your stock is included in the S&P 500, you should use that for the benchmark.
3. Calculate the return of the stock and benchmark for the price data collected.
4. Run a linear regression analysis. The X variable coefficient will be your stock's Beta.

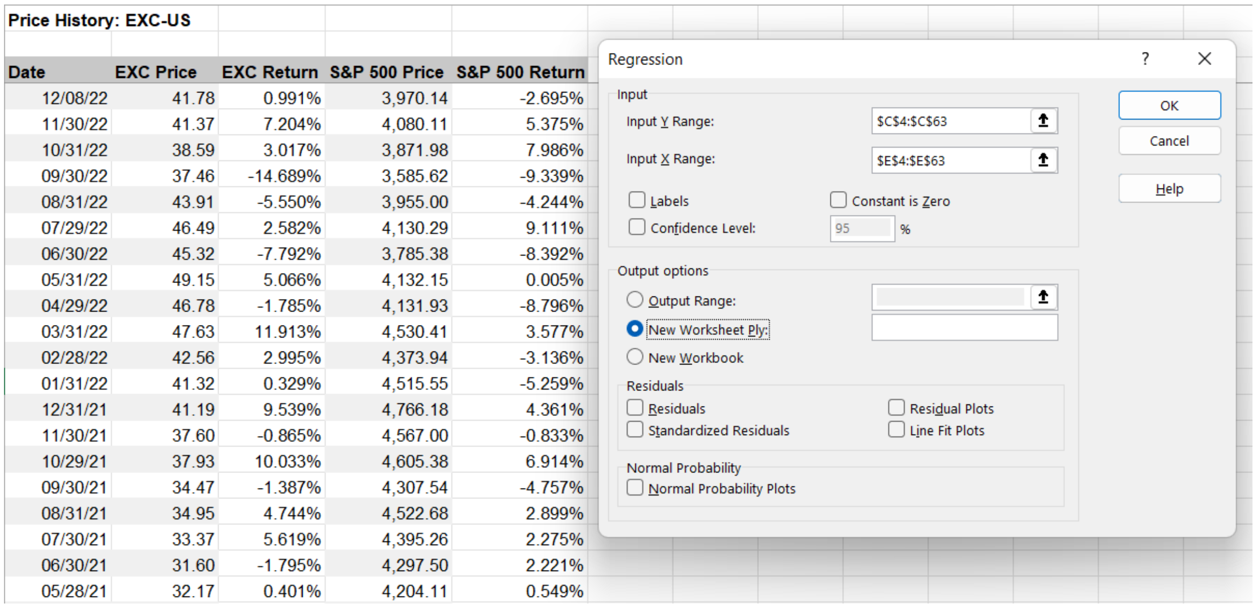
2.2.2 Excel Linear Regression Example

1. After procuring all your data, go to the Data tab.
2. Click on Data Analysis on the right side of the ribbon (if you do not have this option, you might need to add it through Options).
3. Scroll down on the list of different data analysis methods and click on "Regression".


Data Analysis
?
Analysis Tools
Histogram
Moving Average
Random Number Generation
Rank and Percentile
Regression
Sampling
t-Test: Paired Two Sample for Means
t-Test: Two-Sample Assuming Equal Variances
t-Test: Two-Sample Assuming Unequal Variances
z-Test: Two Sample for Means
OK
Cancel
Help

4. Select the **stock return data** as the Input Y Range and the **benchmark**

return data as the Input X Range. Customize output settings as desired and click on Ok.



Price History: EXC-US

Date	EXC Price	EXC Return	S&P 500 Price	S&P 500 Return
12/08/22	41.78	0.991%	3,970.14	-2.695%
11/30/22	41.37	7.204%	4,080.11	5.375%
10/31/22	38.59	3.017%	3,871.98	7.986%
09/30/22	37.46	-14.689%	3,585.62	-9.339%
08/31/22	43.91	-5.550%	3,955.00	-4.244%
07/29/22	46.49	2.582%	4,130.29	9.111%
06/30/22	45.32	-7.792%	3,785.38	-8.392%
05/31/22	49.15	5.066%	4,132.15	0.005%
04/29/22	46.78	-1.785%	4,131.93	-8.796%
03/31/22	47.63	11.913%	4,530.41	3.577%
02/28/22	42.56	2.995%	4,373.94	-3.136%
01/31/22	41.32	0.329%	4,515.55	-5.259%
12/31/21	41.19	9.539%	4,766.18	4.361%
11/30/21	37.60	-0.865%	4,567.00	-0.833%
10/29/21	37.93	10.033%	4,605.38	6.914%
09/30/21	34.47	-1.387%	4,307.54	-4.757%
08/31/21	34.95	4.744%	4,522.68	2.899%
07/30/21	33.37	5.619%	4,395.26	2.275%
06/30/21	31.60	-1.795%	4,297.50	2.221%
05/28/21	32.17	0.401%	4,204.11	0.549%

Regression

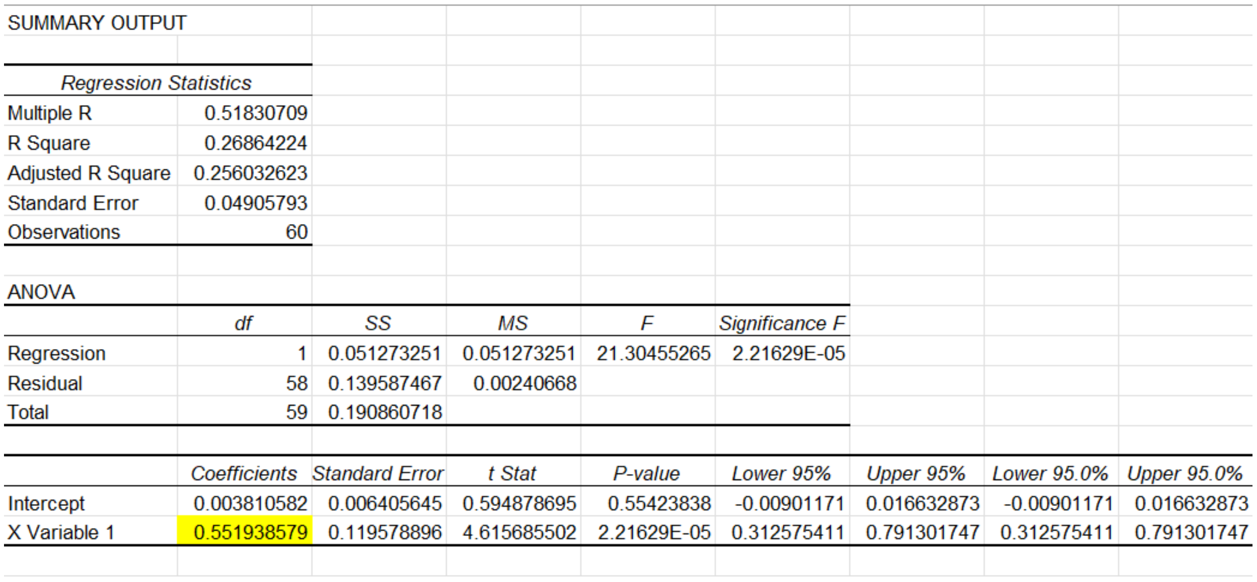
Input
Input Y Range: $C$4:$C$63
Input X Range: $E$4:$E$63
☐ Labels ☐ Constant is Zero
☐ Confidence Level: 95 %

Output options
○ Output Range:
● New Worksheet Ply:
○ New Workbook

Residuals
☐ Residuals ☐ Residual Plots
☐ Standardized Residuals ☐ Line Fit Plots

Normal Probability
☐ Normal Probability Plots

5. Look at the "X Variable 1" coefficient. That is your stock's beta.



SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.51830709
R Square	0.26864224
Adjusted R Square	0.256032623
Standard Error	0.04905793
Observations	60

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	1	0.051273251	0.051273251	21.30455265	2.21629E-05
Residual	58	0.139587467	0.00240668		
Total	59	0.190860718			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*Lower 95.0%*	*Upper 95.0%*
Intercept	0.003810582	0.006405645	0.594878695	0.55423838	-0.00901171	0.016632873	-0.00901171	0.016632873
X Variable 1	0.551938579	0.119578896	4.615685502	2.21629E-05	0.312575411	0.791301747	0.312575411	0.791301747

The company used in this example is Exelon, a utility company. Demand for utilities is inelastic, which makes them very resistant to economic cycles. Therefore, the beta for Exelon is 0.55 < 1.

If we were to measure the beta of a more volatile stock, such as NVIDIA, we would come up with a coefficient of about 1.7 (as of 12/9/22). NVIDIA is responsible for making graphical processing units (GPUs) and is currently suffering from low sales due to decreasing demand for chips. Its share price has been soaring as of late due to macroeconomic factors, such as inflation data being more favorable and the Fed deciding to slow the pace of interest rate hikes. NVIDIA's high beta is an indicator of the company's tendency to amplify market movements. The stock might be doing well now, but if the economy takes another downturn soon, it will *likely* perform worse than the overall market.

2.2.3 Other Uses of Linear Regression

Linear regression can be used anywhere where two or more pieces of data relate to each other. For example, you can use it to assess how GDP is impacted by changes in unemployment and inflation.

In addition, regression analysis is used in sales forecasting. We can use this model to determine how changes in our assumptions will impact revenue or expenses in the future.

2.3 Multiple Linear Regression

Multiple Linear Regression is like Simple Linear Regression, but with multiple independent (explanatory) variables.

Mathematically, the relationship between all the variables is expressed as:

$$y_i = \beta_0 + \beta_{i1}x_1 + \beta_{i2}x_2 + \cdots + \beta_{ip}x_p + \varepsilon$$

Where, for i = n:

y_i = dependent variable
x_i = independent (explanatory) variable
β_0 = y-intercept
β_{ip} = slope coefficient for each explanatory variable
ε = random error

Multiple Linear Regression has 5 key assumptions, 4 of which are the same as simple linear regression. The additional one is the assumption of **no multicollinearity**, meaning that the explanatory variables are **not** highly correlated with one another.

If any of these assumptions are violated, the results of the multiple linear regression are unreliable.

2.3.1 The Fama-French Model

The Fama-French Three Factor Model is an asset pricing model developed in 1992 that expands on the CAPM we saw in the previous section. In addition to the market risk factor, the Fama-French model also includes a firm's size and value as risk factors. These two additional factors are included in the model since it was observed that small-cap stocks tend to outperform large-cap stocks, and high book-to-market stocks (value stocks) tend to outperform low book-to-market stocks (growth stocks), suggesting that there is risk associated with them.

The index model used to estimate betas for the Fama-French model is as follows:

$$R_i = \alpha_i + \beta_{i,M} R_M + \beta_{i,SMB} SMB + \beta_{i,HML} HML + \varepsilon_i$$

Where:

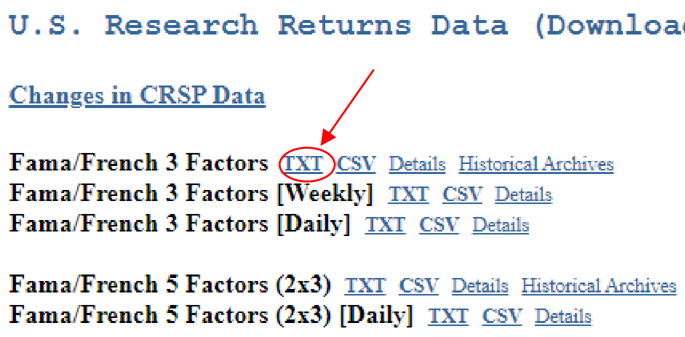
R_i = Return on the stock
a_i = intercept
RM = Market Risk Premium (market return – risk-free return)
SMB = Small Stock Risk Premium (SMB = Small Minus Big)
HML = Value Stock Risk Premium (HML = High Minus Low)
$\beta_{i,x}$ = Beta for each factor
ε_i = random error

2.3.2 Fama-French Multiple Linear Regression in Excel

To perform multiple linear regression in Excel, we use the same menu we used for simple linear regression. The only difference is that we have to select all predictor variables when inputting X range.

Here are the instructions on how to perform the Fama-French analysis in Excel:

1. Retrieve the monthly Fama-French 3 factor values. These can be found in the Kenneth R. French Data Library (https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html).



U.S. Research Returns Data (Downloadable Files)

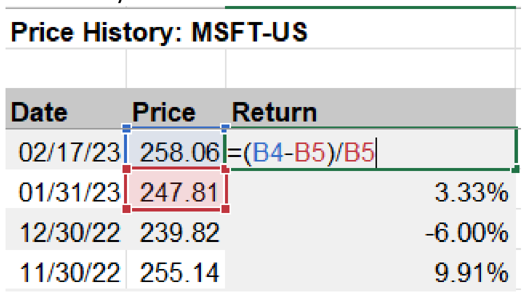
Changes in CRSP Data

Fama/French 3 Factors TXT CSV Details Historical Archives
Fama/French 3 Factors [Weekly] TXT CSV Details
Fama/French 3 Factors [Daily] TXT CSV Details

Fama/French 5 Factors (2x3) TXT CSV Details Historical Archives
Fama/French 5 Factors (2x3) [Daily] TXT CSV Details

2. After downloading the file, select and rearrange the monthly Fama-French data for the past 5 years from newest to oldest.

3. Find the monthly stock return data for the past 5 years (or other desired time frame). Download to Excel and find monthly percentage changes. You can use any website or platform for this, including Yahoo! Finance and FactSet.
(NOTE: In case the Fama-French dataset has not been updated to reflect the most recent months, only retrieve stock return data up to the latest provided month in the dataset).

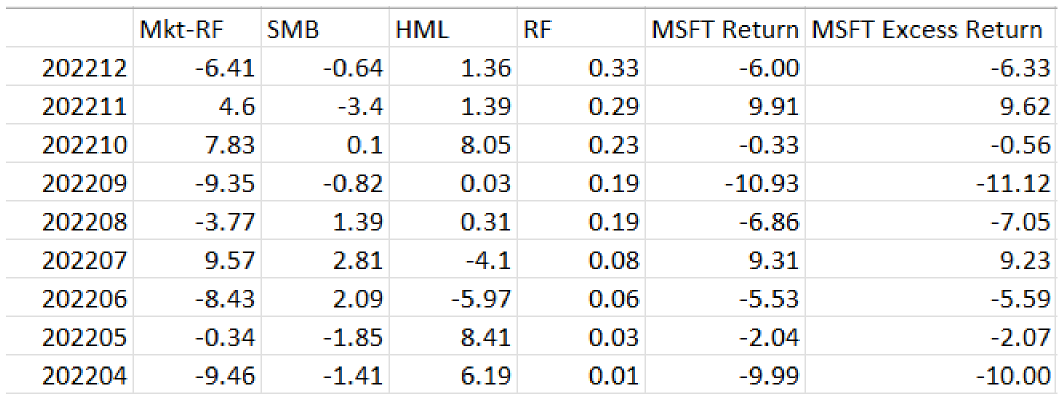


Price History: MSFT-US

Date	Price	Return
02/17/23	258.06	=(B4-B5)/B5
01/31/23	247.81	3.33%
12/30/22	239.82	-6.00%
11/30/22	255.14	9.91%

4. Add stock return data to the dataset, and convert it to the same units as the rest of the dataset (Fama-French data in the website above is expressed in percentages without the % sign, so you will need to either add a % to the FF data or remove it from your stock return data).

5. Calculate excess returns for our selected stock (Microsoft, in this case) by subtracting the risk-free rate from the stock's returns.

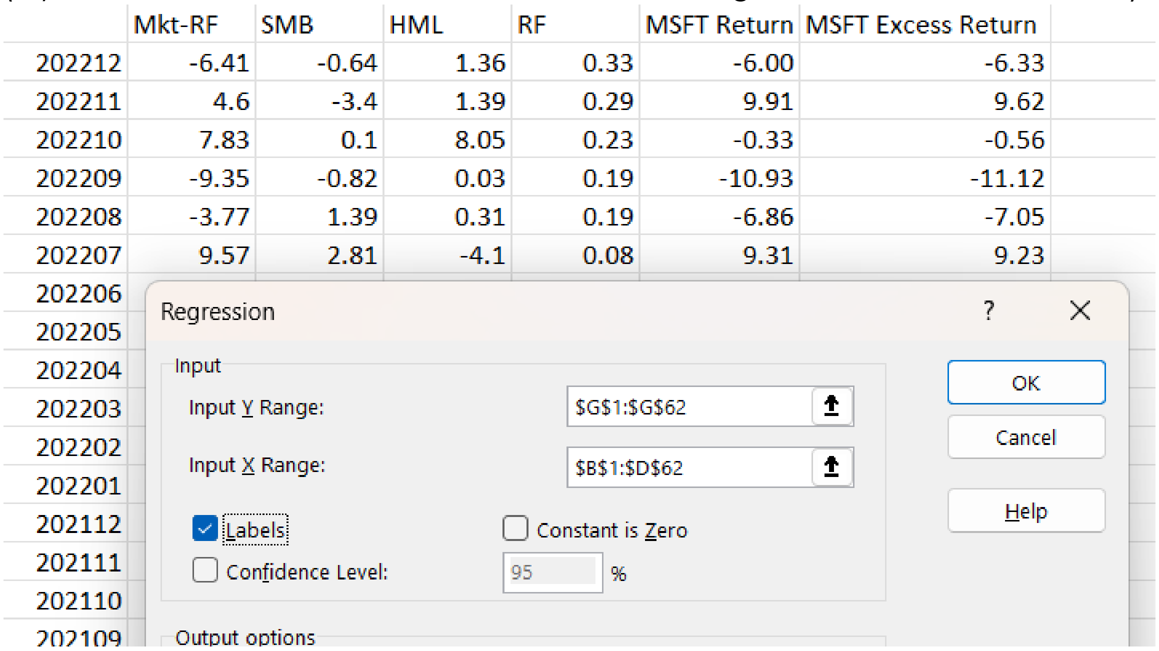


	Mkt-RF	SMB	HML	RF	MSFT Return	MSFT Excess Return
202212	-6.41	-0.64	1.36	0.33	-6.00	-6.33
202211	4.6	-3.4	1.39	0.29	9.91	9.62
202210	7.83	0.1	8.05	0.23	-0.33	-0.56
202209	-9.35	-0.82	0.03	0.19	-10.93	-11.12
202208	-3.77	1.39	0.31	0.19	-6.86	-7.05
202207	9.57	2.81	-4.1	0.08	9.31	9.23
202206	-8.43	2.09	-5.97	0.06	-5.53	-5.59
202205	-0.34	-1.85	8.41	0.03	-2.04	-2.07
202204	-9.46	-1.41	6.19	0.01	-9.99	-10.00

6. Go to the Data tab -> Data Analysis -> Regression.

7. Select the stock **excess** return data as your Y range, and the 3 factors (Mkt-Rf, SMB, and HML) as your X range.

(If you have also selected column labels, do not forget to check the Labels box).



	Mkt-RF	SMB	HML	RF	MSFT Return	MSFT Excess Return
202212	-6.41	-0.64	1.36	0.33	-6.00	-6.33
202211	4.6	-3.4	1.39	0.29	9.91	9.62
202210	7.83	0.1	8.05	0.23	-0.33	-0.56
202209	-9.35	-0.82	0.03	0.19	-10.93	-11.12
202208	-3.77	1.39	0.31	0.19	-6.86	-7.05
202207	9.57	2.81	-4.1	0.08	9.31	9.23
202206						
202205						
202204						
202203						
202202						
202201						
202112						
202111						
202110						
202109						

Regression

Input

Input Y Range: $G$1:$G$62

Input X Range: $B$1:$D$62

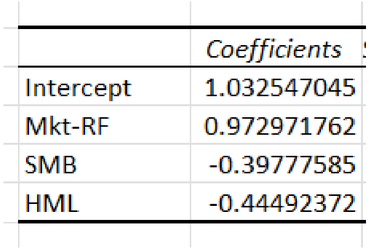
[✓] Labels

[ ] Constant is Zero

[ ] Confidence Level: 95 %

Output options

8. Select your desired output range and press OK.



	Coefficients
Intercept	1.032547045
Mkt-RF	0.972971762
SMB	-0.39777585
HML	-0.44492372

9. Analyze summary output and calculate expected returns.

The summary output for our regression has SMB and HML < 0.

SMB < 0 indicates a large-cap stock, meanwhile HML < 0 indicates a growth stock (and vice-versa).

The intercept (alpha) is quite large. A non-zero alpha implies historical mispricing, with respect to the FF model.

Now that we have the betas for the 3 factors, we can estimate the expected return for MSFT.

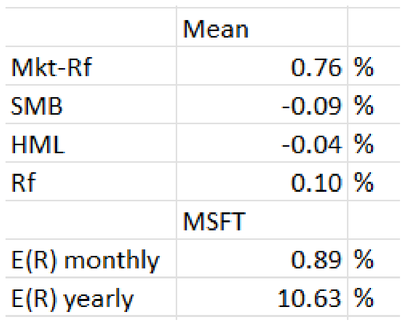
The asset pricing model (different from the index model) is as follows:

$$E(R_i) = R_f + \beta_{i,M} E(R_M) + \beta_{i,SMB} E(SMB) + \beta_{i,HML} E(HML)$$

To estimate MSFT's expected return, we need to:

1. Find a mean value for each of the 3 factors, as well as the risk-free rate.
2. Substitute the mean values and betas in the formula above.
3. Multiply the result by 12, to get the yearly expected return.

According to this model, the yearly expected return for MSFT is 10.63%.



	Mean	
Mkt-Rf	0.76	%
SMB	-0.09	%
HML	-0.04	%
Rf	0.10	%
	MSFT	
E(R) monthly	0.89	%
E(R) yearly	10.63	%

3. Linear Regression in Python

3.1 Python Packages for Linear Regression

To implement linear regression in Python, we'll need to apply the proper packages and the correct functions from each of them.

In this Project Package, you can find a PowerPoint that goes over crucial terms and concepts that we use in Python programming. However, I will provide some definitions in this document as well.

In Python (and many other programming tools, such as R), a **package** is a folder that contains modules and other folders, which themselves contain more modules and folders. When we use our own technological devices, we tend to categorize our data (apps, photos, other files) in folders and subfolders, depending on specific criteria, so we can manage and organize them more efficiently. Similarly, Python packages help build a well-organized hierarchy of data and functions, which makes them easier to access.

In short, a Python package is a folder that contains various modules as files. Python **modules** may contain several classes, functions, variables, etc. Below you will find the packages necessary for implementing linear regression in Python.

3.1.1 NumPy

NumPy is one of the fundamental Python packages for scientific analysis. It offers comprehensive mathematical functions, random number generators, linear algebra routines, and more.

Here is a very useful NumPy user guide that explains important features of the package: NumPy user guide.

3.1.2 Scikit-learn

The package **scikit-learn** is a widely used Python library for machine learning, which builds on top of NumPy (and other packages). It provides means for preprocessing data, reducing dimensionality, **implementing regression**, classifying, clustering, and more.

To learn more about linear models and how this package works, use the following website: scikit-learn Linear Models.

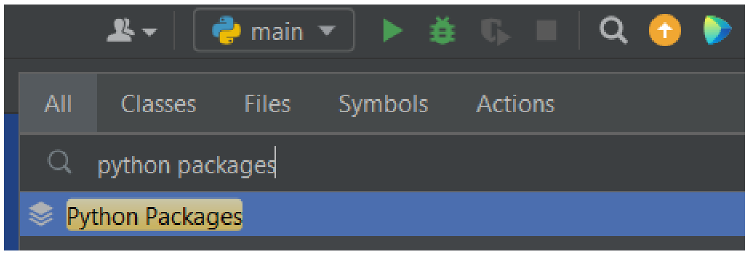
3.1.3 How to Install Packages in Python

The only prerequisite for installing **NumPy** and **scikit-learn** is Python itself. In this file, we will go over how to install these packages using the IDE *PyCharm*. You can download PyCharm here, or you can choose another IDE.

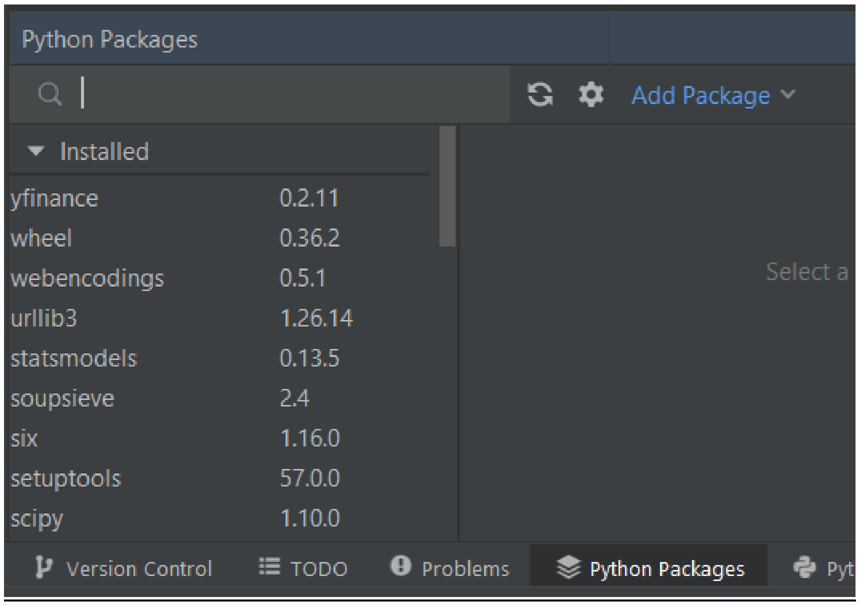
Alternative methods for installing packages in Python include using ***pip*** (the package installer), ***Conda*** (a "language-agnostic" package manager), etc.

Here is how to install the above packages using PyCharm:

1. Open PyCharm.
2. Click on the looking glass, type "Python packages", and select the following.



3. The following menu will pop up at the bottom of the page with all your current packages. Look up each package you need by name, then click Install.



yfinance	0.2.11
wheel	0.36.2
webencodings	0.5.1
urllib3	1.26.14
statsmodels	0.13.5
soupsieve	2.4
six	1.16.0
setuptools	57.0.0
scipy	1.10.0

3.2 Easy Linear Regression Example in Python

3.2.1 Overview of Steps

Now that you have all of the necessary packages installed, we are going to go over how to implement a simple linear regression.

There are 5 basic steps to this process:

1. **Import the packages and classes that you need.** You already have the packages. Now you need the classes.
2. **Provide data to work with and perform necessary transformations.**
3. **Create a regression model and fit it with the data.**
4. **Check the results of model fitting to decide if the model is satisfactory.**
5. **Apply model for predictions.**

Usually, these are the steps you need to take for most regression implementations in **machine learning**. When it comes to using linear regression for **stock beta analysis**, we do not need to check model fit, or apply it for predictions. *If you are only interested in learning how to use Python and/or linear regression for assessing a stock's beta, check the last part of this document.*

3.2.2 Implementation

Step 1: Import packages and classes.

The first step is to import the packages you installed above, as well as the class *LinearRegression* from *sklearn.linear_model.*

You might be wondering where *sklearn.linear_model* came from. This is a **module** that is found within the *scikit-learn* package you installed, that contains all the code we need to use for linear model analysis. The point of installing these packages is to have access to all the modules within them, such as the one above.

```
import numpy as np
from sklearn.linear_model import LinearRegression
```

Step 2: Provide data

The second step is to provide the data to work with. Let's assume you are trying to find the linear relationship between the number of dollars spent on advertising in a month and the number of sales made by a popular online store. You have data for the previous 7 months. When visualized in a table, your dataset looks like this:

Monthly Sales (in thousands of dollars)	Online Advertising (in thousands of dollars)
368	1.7
348	1.5
665	2.8
954	5
331	1.3
556	2.2
376	1.3

NOTE: This dataset is used to illustrate how to *implement* the regression model. Normally, you would not use this few data instances. The rule of thumb is 10 data points for each independent variable.

To translate this table to code, you will have to create two **arrays** of the same length. Arrays are a fundamental data structure in programming languages, which serve as containers that can hold multiple items at the same time. Arrays can be called through the package **numpy** by adding **.array** at the end of it.

```
advertising = np.array([1.7, 1.5, 2.8, 5, 1.3, 2.2, 1.3]).reshape((-1,1))
sales = np.array([368, 348, 665, 954, 331, 556, 376])
```

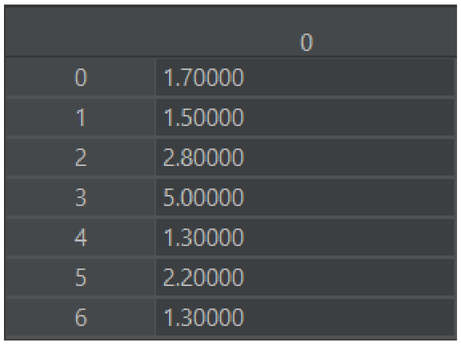
We put advertising first, instead of sales, because advertising is our X (independent) variable. Through advertising, we will be able to make predictions about sales.

As you can see, in the code above, we used the phrase "np.array". The shortcut "np" is what we named the package **numpy** after importing it. This is something we do to save time, but it is not necessary for writing code. We could have also just imported **numpy** without renaming it and written our code like this:

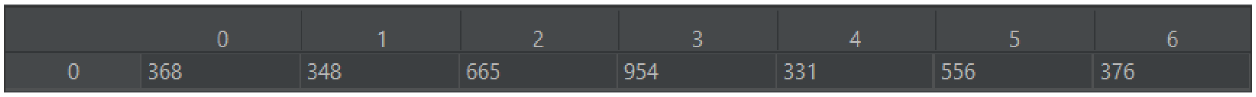
```
advertising = numpy.array([368, 348, 665, 954, 331, 556, 376])
```

Something else that might have caught your attention is that we use the function **.reshape((-1,1))** at the end of the first array. We need to call this function because we have to turn this array into the column of a table that has as many rows as there are data points. If we don't call this function, our model will not understand that the two arrays (*sales* and *advertising*) are a **series of observed values for a single data instance**. So, we want to make our two arrays resemble a table to implement the linear regression model.

Now, our arrays look like this:



	0
0	1.70000
1	1.50000
2	2.80000
3	5.00000
4	1.30000
5	2.20000
6	1.30000



	0	1	2	3	4	5	6
0	368	348	665	954	331	556	376

Step 3: Create a model and fit it

The next step is to create our linear regression model and fit it using our data.

The first thing we do is create an instance of the class *LinearRegression* to represent our regression model. Use the statement below to create the variable named *model* as an instance of *LinearRegression*.

```
model = LinearRegression()
```

Within the brackets, you have the option to provide several parameters for the linear regression model. I will not go into details on these parameters, but they are the following: *fit_intercept*, *normalize*, *copy_X*, and *n_jobs*. Your model, as you have written it, uses the default values for these parameters.

To start using the model, you need to call **.fit()**:

```
model.fit(advertising, sales)
```

What **.fit()** does is calculate the optimal values for the intercept and coefficient of the linear regression equation. In machine learning, this is called **fitting the model**.

So, in summary, the last two lines of your model look like this:

```
model = LinearRegression()
model.fit(advertising, sales)
```

If you want, you can replace them by this one line:

```
model = LinearRegression().fit(advertising, sales)
```

Step 4: Get results.

Now that your model is fitted, you can get the results of your algorithm. The information we will extract for this particular code is the **Coefficient of Determination (R^2)**. This is a statistical measure that tells us how much of the variance in the dependent variable (sales) is explained by the independent variable (advertising). In short, it tells us how well our model predicts the number of sales experienced by the stores. It ranges from 0 to 1 – the higher, the better.

```
r2 = model.score(advertising, sales)
print(f"coefficient of determination: {r2}")
```

When we run this code, it will print out the ***R^2*** value. In our case, this value is:

```
>>> coefficient of determination: 0.9629280956745225
```

This value is almost 1, suggesting that our model is really good at explaining the variance in sales depending on advertising amount. Since we did not preprocess our original data, this may suggest that we got lucky in terms of data quality.

Next, we want to find the intercept and coefficient for our model.

```
print(f"intercept: {model.intercept_}")
print(f"slope: {model.coef_}")
```

The code above yields the following results:

```
>>> intercept: 128.2752169197397

>>> slope: [170.89072668]
```

Step 5: Predict Response

Now that we have our model, we can apply it on the dataset to see if it predicts our data correctly. To do this, we need to call the **.predict()** function.

```
y_pred = model.predict(advertising)
print(f"predicted response:\n{y_pred}")
```

```
>>> predicted response:

>>> [418.78945228 384.61130694 606.76925163 982.72885033 350.43316161
504.23481562 350.43316161]
```

Here is a table representation of money spend on advertisement, actual sales, and predicted sales:

Online Advertisement ($k)	Actual Sales	Predicted Sales
1.7	368	419
1.5	348	385
2.8	665	607
5	954	983
1.3	331	350
2.2	556	504
1.3	376	350

3.2.3 Interpretation of Results

In part **2.2 What is Linear Regression?** we went over the linear regression formula. Once again, this type of analysis is used to assess a mathematical representation of the linear relationship between two variables. As mentioned above, the equation for linear regression is:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon$$

In the code above, we were able to assess the intercept (β_0) and slope (β_1) of the line. We can replace these values in the equation above, to find the equation for sales as advertisement spending changes.

$$Sales = 128.275 + 170.891 * advertisement$$

We can interpret the intercept of the model as the number of sales when advertisement spending is 0. Therefore, we can assess that even without advertising, our stores are able to make 128,275 sales.

The slope of the line tells us that for every additional dollar unit (in our case 1 unit = $1k) spent in online advertisement, the company increases its sales by 170.891 *1000 = 170,891 units.

As you can see from the table with predicted values above, our linear model was able to make predictions that were *close* to the actual values of sales made by the online store.

3.3 Python Code

Here is the text version of the simple linear regression implemented above.

```
import numpy as np
from sklearn.linear_model import LinearRegression

#build arrays
advertising = np.array([1.7, 1.5, 2.8, 5, 1.3, 2.2, 1.3]).reshape((-1, 1))
sales = np.array([368, 348, 665, 954, 331, 556, 376])
```

```

#define model
model = LinearRegression()
model.fit(advertising, sales)
# you can rewrite these two lines as:
# model = LinearRegression().fit(sales, advertising)

#find coefficient of determination
r2 = model.score(advertising, sales)
print(f"coefficient of determination: {r2}")

#find model intercept and slope
print(f"intercept: {model.intercept_}")
print(f"slope: {model.coef_}")

#make predictions
y_pred = model.predict(advertising)
print(f"predicted response:\n{y_pred}")
```

4. Using Linear Regression for CAPM in Python

4.1 Package for stock price retrieval

When we use Excel to perform linear regression for beta analysis, we need to procure the price data for both the stock and its benchmark on our own. Luckily, in Python, we have a package called **yFinance** that does the hard work for us. This package offers us a way to download market data directly from Yahoo! Finance.

To install the **yFinance** package, we need to do the same thing we did for **numpy** and **scikit-learn**, which is:

1. Open PyCharm.
2. Look up "Python packages" and click on it.
3. Look up each individual "yfinance" in the menu that pops up and click install.

4.2 The Data

4.2.1 Retrieving the data

First, we import all packages:

```python
import yfinance as yf
import numpy as np
from sklearn.linear_model import LinearRegression
```

Next, we create a list of tickers, where the first one is the stock and the second one is the benchmark. For the stock, I will be using NVDA. You can use any stock you like, even international ones, for as long as you use the exact ticker they have on Y! Finance.

For the benchmark, I will be using the S&P 500 index, which in Y! Finance has the ticker ^GSPC. You can also use SPY, which is the ETF tracking the S&P 500.

```python
tickers = [“NVDA”, “^GSPC”]
```

Once you have selected your stock and benchmark, the next thing you need to do is create a dataframe of historical prices. For this, we use the **yf.download()** function.

Here are some things to keep in mind when writing your code:

1. For this regression we are using monthly data for the past 5 years. To make this easy for us, the **yfinance** package offers us the option to specify *period* and *interval* for our data.
2. For period, we will use "5y".
3. For interval, we will use "1mo".

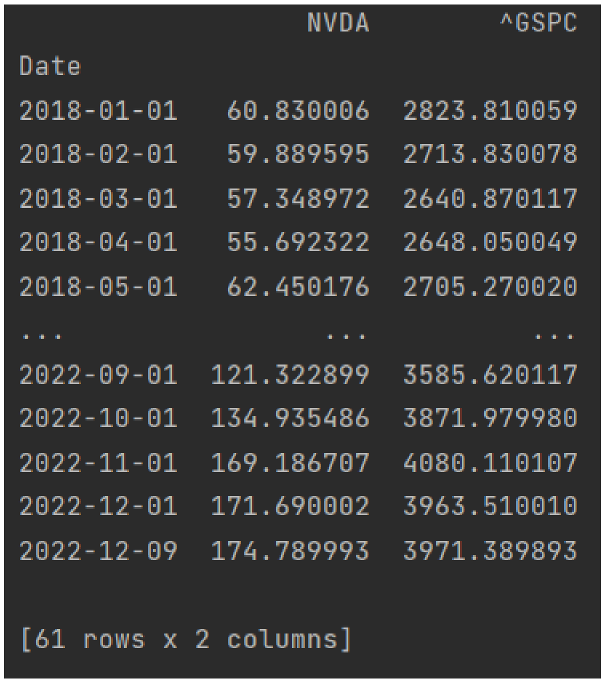
4. If you want to retrieve different price data, change the values of these two parameters. Alternatively, you can specify the dates desired (click here to learn more).
5. You should use adjusted closing prices, so it is not recommended to change the last argument in the code (adjusted closing prices are preferred because they are already adjusted for stock splits and dividends, which increases our beta's accuracy).

```
price_data = yf.download(tickers, period=”5y”, interval=”1mo” )[‘Adj Close’]
```

Next, we should view the downloaded data to make sure it is clean and therefore useful.

```
print(price_data)
```

The result should look something like this:



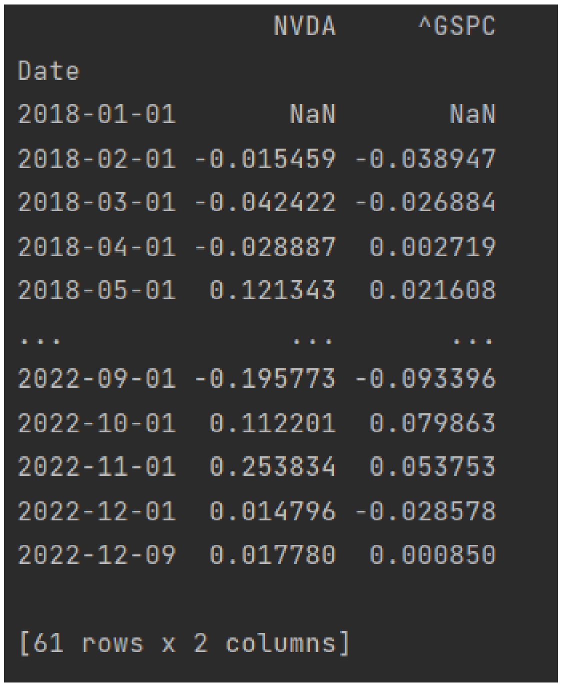
```
                  NVDA        ^GSPC
Date
2018-01-01   60.830006  2823.810059
2018-02-01   59.889595  2713.830078
2018-03-01   57.348972  2640.870117
2018-04-01   55.692322  2648.050049
2018-05-01   62.450176  2705.270020
...                ...          ...
2022-09-01  121.322899  3585.620117
2022-10-01  134.935486  3871.979980
2022-11-01  169.186707  4080.110107
2022-12-01  171.690002  3963.510010
2022-12-09  174.789993  3971.389893

[61 rows x 2 columns]
```

4.2.2 Standardizing and Cleaning the Data

For beta regression analysis we use changes in prices, not prices, therefore we need to transform the data we gathered into measures of fluctuation. To do so, we use the **.pct_change()** function.

```
price_change = price_data.pct_change()
print(price_change)
```

Now, the dataset looks like this:

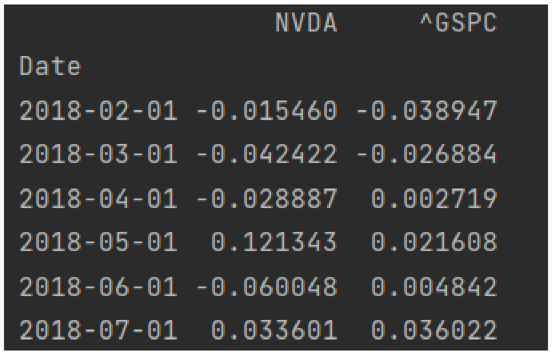


```
                   NVDA     ^GSPC
Date
2018-01-01          NaN       NaN
2018-02-01 -0.015459 -0.038947
2018-03-01 -0.042422 -0.026884
2018-04-01 -0.028887  0.002719
2018-05-01  0.121343  0.021608
...              ...       ...
2022-09-01 -0.195773 -0.093396
2022-10-01  0.112201  0.079863
2022-11-01  0.253834  0.053753
2022-12-01  0.014796 -0.028578
2022-12-09  0.017780  0.000850

[61 rows x 2 columns]
```

As you can see, the first data instance in the data frame above is “NaN”, since we cannot calculate a price change without having an older data point to reference. We cannot use a null value for our regression, as it may cause an error, so what we need to do is remove this instance (data cleaning). Being the first row, its index number is 0 (not 1).

```
final_data = price_change.drop(price_change.index[0])
print(final_data)
```

The top of our cleaned-up dataset now looks like the one below. We can now use it to perform beta regression analysis.



```
                   NVDA     ^GSPC
Date
2018-02-01 -0.015460 -0.038947
2018-03-01 -0.042422 -0.026884
2018-04-01 -0.028887  0.002719
2018-05-01  0.121343  0.021608
2018-06-01 -0.060048  0.004842
2018-07-01  0.033601  0.036022
```

4.3 Model Implementation

As you may recall, linear regression requires two inputs, x and y, where x is the independent (market) variable and y is the dependent (stock) variable. X is the S&P 500's return and Y is NVDA's return. It has to be this way and not the other way around because we are trying to assess the stock's volatility *compared to the overall market*.

In Python, we are going to approach this problem the same way we approached the company's example in part 4.2. First, we will create two arrays, benchmark (x) and stock (y) and we will reshape the first array.

```
benchmark = np.array(final_data[“^GSPC”]).reshape((-1,1))
stock = np.array(final_data[“NVDA”])
```

Let's break down the code by component:

a. *benchmark* is the array's name
b. *np* is what we renamed the **numpy** package to make our coding more efficient
c. *.array()* is the function we call from the **numpy** package for the creation of an array
d. *final_data[]* is the name we gave our dataset after we were done with data transformation and clean-up. We need to reference this dataframe because we are trying to extract stock information found within it
e. “*^GSPC*” is the Yahoo! Finance ticker for the S&P 500 benchmark, for which we have saved and transformed price data
f. *.reshape((-1,1))* is the function we use to reshape the benchmark array into a “column”. By giving the array structural awareness, the linear regression function will be able to understand that each NVDA data point corresponds to a S&P 500 data point.

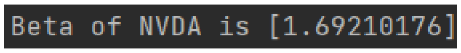
Next, we define and fit the linear regression model:

```
model = LinearRegression().fit(benchmark, stock)
```

Now that we have our model, we retrieve the Beta coefficient of the NVDA stock:

```
print(f”Beta of {tickers[0]} is”, model.coef_)
```

This is the output:



```
Beta of NVDA is [1.69210176]
```

It turns out that the beta for NVDA is almost 1.7. This is very close to Yahoo Finance's 1.75 value. The slight discrepancy between their value and the one we got might be due to both time period and interval used, but it is not a cause of concern.

4.4 Python Code

Here is a copy of the Python code, with notes:

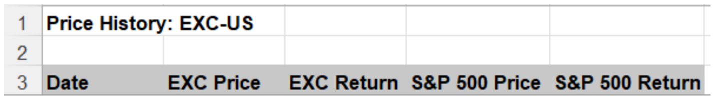
```python
import yfinance as yf
import numpy as np
from sklearn.linear_model import LinearRegression

# symbols = [stock, market]
tickers = ["NVDA", "^GSPC"]

# dataframe of historical stock prices
# since we are using monthly data for the past 5 years, we specify "5y" for period and
"1mo" for interval
# do not change the 'Adj close' parameter
price_data = yf.download(tickers, period="5y", interval="1mo" )['Adj Close']

# view historical data to confirm its accuracy
print(price_data)

# covert prices to percent change (%)
price_change = price_data.pct_change()
print(price_change)

# delete first row containing null value
final_data = price_change.drop(price_change.index[0])
print(final_data)

# create arrays for x (benchmark) and y (stock) variables for regression
benchmark = np.array(final_data["^GSPC"]).reshape((-1,1))
stock = np.array(final_data["NVDA"])

# define and fit model
model = LinearRegression().fit(benchmark, stock)

# print out stock's beta
print(f"Beta of {tickers[0]} is", model.coef_)
```

4.5 CAPM Using Data Uploaded from Excel

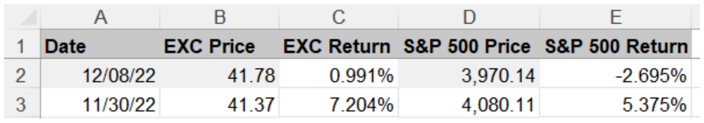
4.5.1 Implementation

If we already have the stock return data and their percentage changes, but we want to perform the linear regression in Python, here is how to proceed:

We first clean up the Excel file, so that the sheet contains only the column names and data.



1	Price History: EXC-US				
2					
3	Date	EXC Price	EXC Return	S&P 500 Price	S&P 500 Return



	A	B	C	D	E
1	Date	EXC Price	EXC Return	S&P 500 Price	S&P 500 Return
2	12/08/22	41.78	0.991%	3,970.14	-2.695%
3	11/30/22	41.37	7.204%	4,080.11	5.375%

Since we are already in Excel, we can also get rid of the columns we are not going to use (date & price data columns), however this is not necessary as we will need to call on the return columns in Python anyways.

Import the necessary packages:

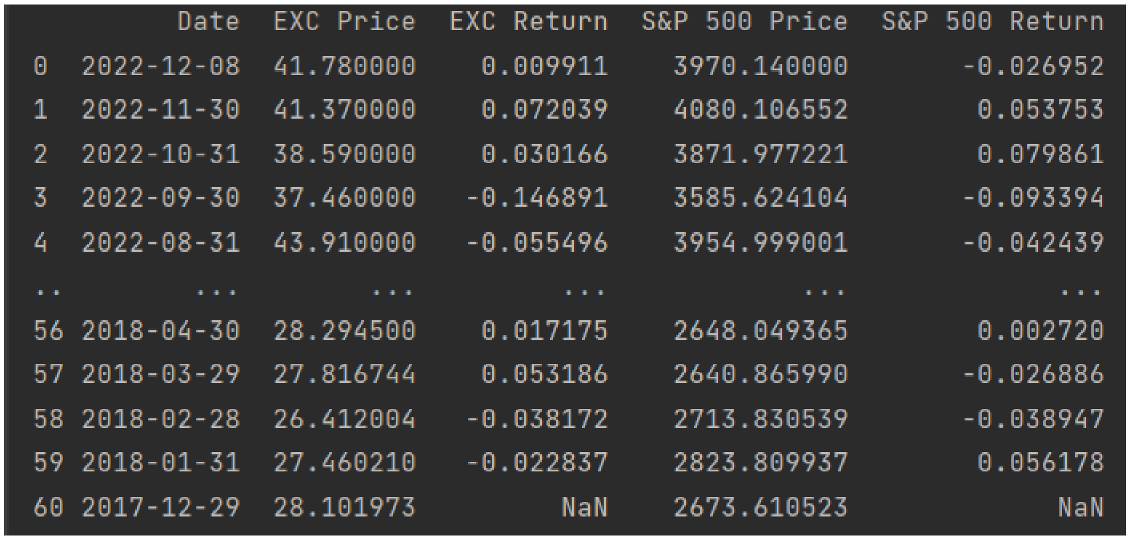
```python
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
```

Read the dataframe using pandas (if errors arise, please check Section 4.5.3). In the ***read_excel()*** function, your IDE should ideally allow you to find the Excel file without you having to specify the exact file path.

NOTE: if your data is in a csv file, replace the ***read_excel()*** function with the ***read_csv()*** function, and replace the file extension to ***csv*** as well.

```python
data = pd.read_excel('Exelon Return Data.xlsx')
df = pd.DataFrame(data)
# print out the dataset
print(df)
```

Once you print out the dataset, it should look like the following:



```
          Date  EXC Price  EXC Return  S&P 500 Price  S&P 500 Return
0   2022-12-08  41.780000    0.009911    3970.140000       -0.026952
1   2022-11-30  41.370000    0.072039    4080.106552        0.053753
2   2022-10-31  38.590000    0.030166    3871.977221        0.079861
3   2022-09-30  37.460000   -0.146891    3585.624104       -0.093394
4   2022-08-31  43.910000   -0.055496    3954.999001       -0.042439
..         ...        ...         ...            ...             ...
56  2018-04-30  28.294500    0.017175    2648.049365        0.002720
57  2018-03-29  27.816744    0.053186    2640.865990       -0.026886
58  2018-02-28  26.412004   -0.038172    2713.830539       -0.038947
59  2018-01-31  27.460210   -0.022837    2823.809937        0.056178
60  2017-12-29  28.101973         NaN    2673.610523             NaN
```

Now you can reference the columns without needing the Excel file.

Next, initialize the model features:

```python
8.  x = df.loc[:, "S&P 500 Return"].dropna()
9.  benchmark = x.values.reshape(-1,1)
10. y = df.loc[:, "EXC Return"].dropna()
11. stock = y.values
```

In line 8, we specified the correct column for our independent variable using the ***loc[]*** function which allows calling on rows and columns using their label. In the same line of code we also removed the missing value at the end of the column.

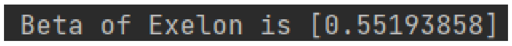
In line 9, we reshaped the x data using the ***reshape(-1,1)*** function. Recall from the previous sections that we need this function to give the array structural awareness. We then renamed the array as "benchmark".

In lines 10-11, we repeat the same steps above but for the depend variable y. We call on the right column ("*EXC Return*"), we drop missing values, and assign the y values to a new variable named "stock". We do not need to reshape the y column.

Finally, we fit the model, and print out the results:

```python
model = LinearRegression().fit(benchmark, stock)
print("Beta of Exelon is", model.coef_)
```

The result of the linear regression will look as follows:



```
Beta of Exelon is [0.55193858]
```

This is the same coefficient we got in Section 2.2.2, where we performed linear regression on the Excel file directly.

4.5.2 Python Code

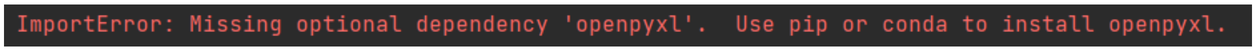
This is the Python code, with notes:

```python
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression

# read the data from the Excel file
data = pd.read_excel('Exelon Return Data.xlsx')
df = pd.DataFrame(data)
# print out the dataset
print(df)

# initialize features
x = df.loc[:, "S&P 500 Return"].dropna()
benchmark = x.values.reshape(-1,1)
y = df.loc[:, "EXC Return"].dropna()
stock = y.values

# Create a model
```

```
model = LinearRegression().fit(benchmark, stock)

# print out stock's beta
print("Beta of Exelon is", model.coef_)
```

4.5.3 Potential Error

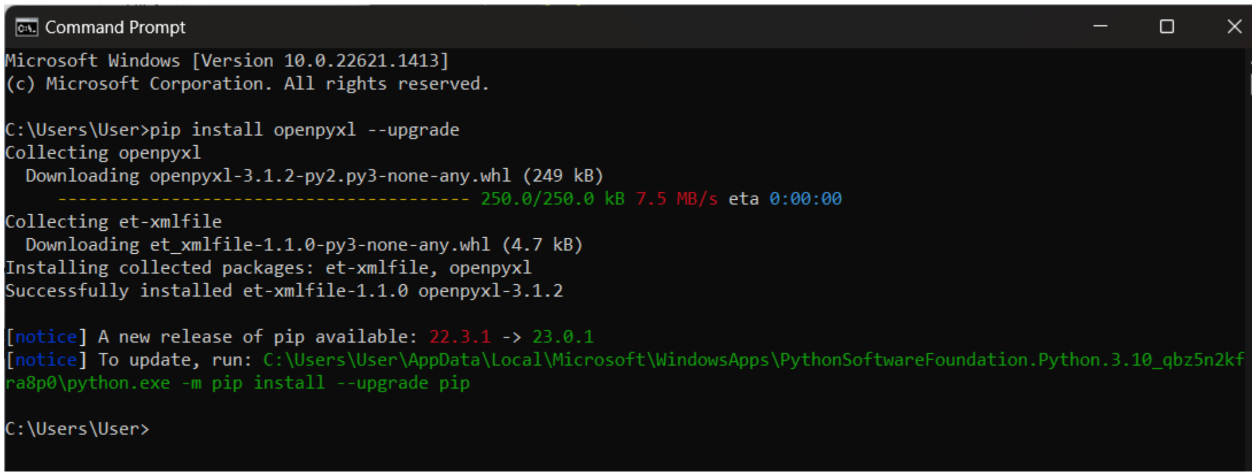
If you wrote the code down correctly, using the right file path, but you get the following error, follow these instructions:



```
ImportError: Missing optional dependency 'openpyxl'.  Use pip or conda to install openpyxl.
```

Look up "pip" or "Command Prompt" on your device.

Open the application, and write the following: **pip install openpyxl –upgrade** and hit *Enter*. If that does not work, try writing **pip install pandas –upgrade**.



```
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\Users\User>pip install openpyxl --upgrade
Collecting openpyxl
  Downloading openpyxl-3.1.2-py2.py3-none-any.whl (249 kB)
     ---------------------------------------- 250.0/250.0 kB 7.5 MB/s eta 0:00:00
Collecting et-xmlfile
  Downloading et_xmlfile-1.1.0-py3-none-any.whl (4.7 kB)
Installing collected packages: et-xmlfile, openpyxl
Successfully installed et-xmlfile-1.1.0 openpyxl-3.1.2

[notice] A new release of pip available: 22.3.1 -> 23.0.1
[notice] To update, run: C:\Users\User\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.10_qbz5n2kf
ra8p0\python.exe -m pip install --upgrade pip

C:\Users\User>
```

If that does not work either, simply install the "openpyxl" package from settings, as shown in Section 3.1.3).

5. Using Linear Regression for the Fama-French Model in Python

5.1 Required Packages

To calculate the risk of the 3 Fama-French factors for a particular stock, we need the following packages:
a. yFinance – retrieves stock return data from Yahoo! Finance, the same package we used for CAPM beta regression;
b. statsmodels – conducts statistical tests like multiple linear regression, without needing to reshape arrays. You can read more about this package here.
c. getFamaFrenchFactors – automatically retrieves data from the Kenneth R. French library (which we did manually in the Excel example). More information about this package can be found here.

5.2 The Data

The first step we need to take is to install the required packages.

```
import yfinance as yf
import statsmodels.api as sm
import getFamaFrenchFactors as ff
```

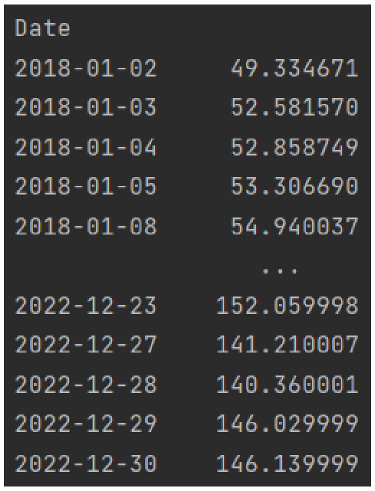
For this regression, we need historical stock prices and the Fama-French factor data. But, before we retrieve this data, we need to specify the stock and time range we want to use. (The yFinance package also gives us the option to specify the time period and interval of our choosing, without us needing to write down the dates, however since the Fama-French dataset has not been updated to reflect the last couple of months, I had to use the past 5 years up to the latest update).

```
ticker = 'nvda'
start = '2017-12-31'
end = '2022-12-31'
```

Next, we download the historical stock prices from Yahoo! Finance, using the download function and specifying the ticker, start, and end dates as the function arguments. We can specifically call the Adj Close column data by writing down its name within square brackets.

```
stock_data = yf.download(ticker, start, end)[‘Adj Close’]
```

The stock data, as extracted from Yahoo! Finance, is daily, therefore we will need to resample it.

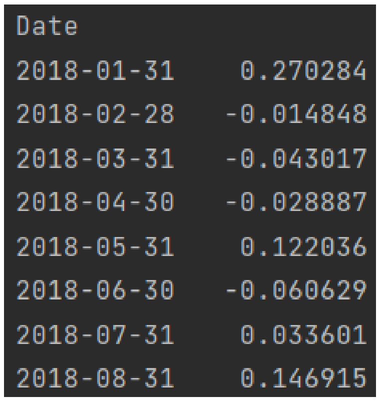


```
Date
2018-01-02     49.334671
2018-01-03     52.581570
2018-01-04     52.858749
2018-01-05     53.306690
2018-01-08     54.940037
                 ...
2022-12-23    152.059998
2022-12-27    141.210007
2022-12-28    140.360001
2022-12-29    146.029999
2022-12-30    146.139999
```

To resample, we use the resample() function, and specify "M" (monthly) for the argument. The next arguments, *last()* and *pct_change()*, call on the resampled dataset and convert the values to percentage changes. Lastly, the *dropna()* function removes missing values (the first value of the dataset, since there are no prior available values to calculate a percentage change).

```
stock_returns = stock_data.resample('M').last().pct_change().dropna()
stock_returns.name = "Month_Return"
print(stock_returns)
```

The new stock_returns dataset looks like this:

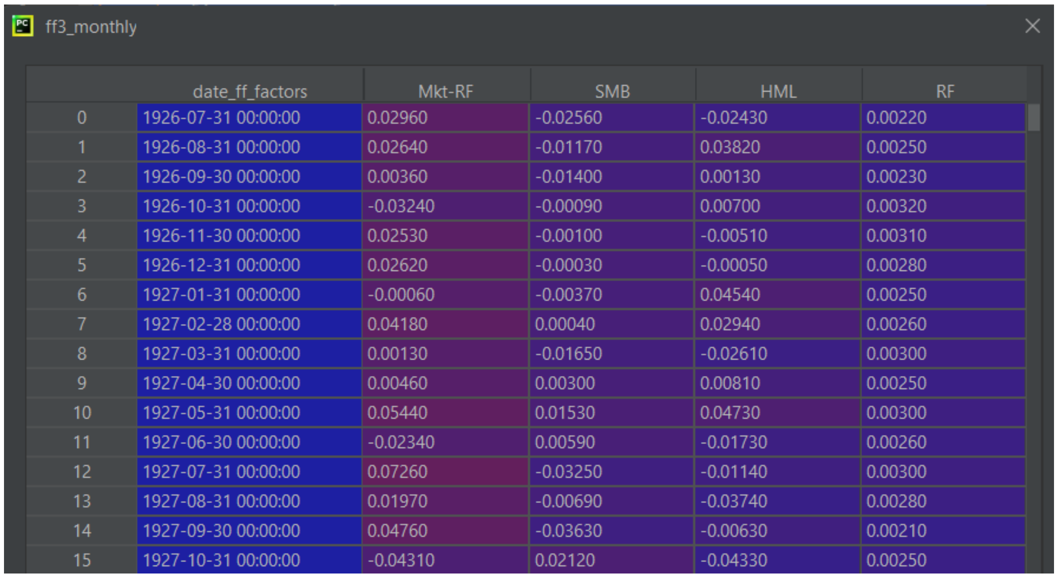


```
Date
2018-01-31     0.270284
2018-02-28    -0.014848
2018-03-31    -0.043017
2018-04-30    -0.028887
2018-05-31     0.122036
2018-06-30    -0.060629
2018-07-31     0.033601
2018-08-31     0.146915
```

Next, we retrieve the values for the Fama French factors using the appropriate package. We want to use monthly data, so we specify "frequency='m'".

```
ff3m = ff.famaFrench3Factor(frequency = 'm')
```

The ff3_monthly dataset looks like this:

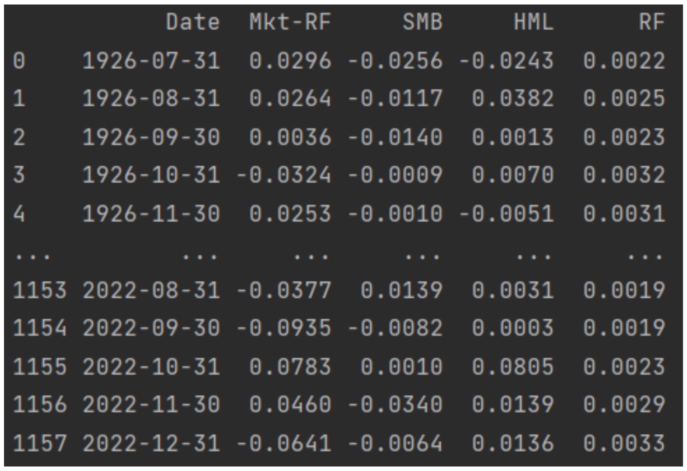


	date_ff_factors	Mkt-RF	SMB	HML	RF
0	1926-07-31 00:00:00	0.02960	-0.02560	-0.02430	0.00220
1	1926-08-31 00:00:00	0.02640	-0.01170	0.03820	0.00250
2	1926-09-30 00:00:00	0.00360	-0.01400	0.00130	0.00230
3	1926-10-31 00:00:00	-0.03240	-0.00090	0.00700	0.00320
4	1926-11-30 00:00:00	0.02530	-0.00100	-0.00510	0.00310
5	1926-12-31 00:00:00	0.02620	-0.00030	-0.00050	0.00280
6	1927-01-31 00:00:00	-0.00060	-0.00370	0.04540	0.00250
7	1927-02-28 00:00:00	0.04180	0.00040	0.02940	0.00260
8	1927-03-31 00:00:00	0.00130	-0.01650	-0.02610	0.00300
9	1927-04-30 00:00:00	0.00460	0.00300	0.00810	0.00250
10	1927-05-31 00:00:00	0.05440	0.01530	0.04730	0.00300
11	1927-06-30 00:00:00	-0.02340	0.00590	-0.01730	0.00260
12	1927-07-31 00:00:00	0.07260	-0.03250	-0.01140	0.00300
13	1927-08-31 00:00:00	0.01970	-0.00690	-0.03740	0.00280
14	1927-09-30 00:00:00	0.04760	-0.03630	-0.00630	0.00210
15	1927-10-31 00:00:00	-0.04310	0.02120	-0.04330	0.00250

As you can see, the first column of the dataset is named "date_ff_factors". We can rename this column as "Date" for practicality, since we will be using it later on. The *inplace* argument, when set to "True", modifies the data in place and updates the dataframe.

```
12. ff3m.rename(columns={"date_ff_factors": "Date"}, inplace=True)
13. print(ff3m)
```

The dataset now looks like the following:

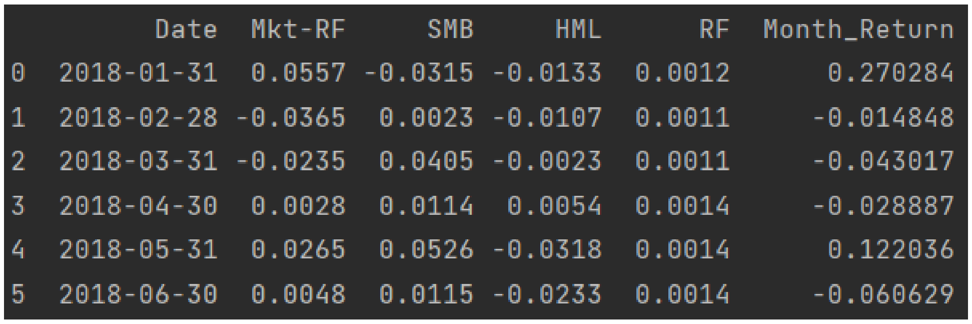


```
           Date  Mkt-RF     SMB     HML      RF
0    1926-07-31  0.0296 -0.0256 -0.0243  0.0022
1    1926-08-31  0.0264 -0.0117  0.0382  0.0025
2    1926-09-30  0.0036 -0.0140  0.0013  0.0023
3    1926-10-31 -0.0324 -0.0009  0.0070  0.0032
4    1926-11-30  0.0253 -0.0010 -0.0051  0.0031
...         ...     ...     ...     ...     ...
1153 2022-08-31 -0.0377  0.0139  0.0031  0.0019
1154 2022-09-30 -0.0935 -0.0082  0.0003  0.0019
1155 2022-10-31  0.0783  0.0010  0.0805  0.0023
1156 2022-11-30  0.0460 -0.0340  0.0139  0.0029
1157 2022-12-31 -0.0641 -0.0064  0.0136  0.0033
```

The final thing we need to do before implementing the model is merging the two datasets (stock_returns and ff3m) together.

```
14. ff_data = ff3m.merge(stock_returns, on='Date')
```

The final dataset looks like this:



```
         Date  Mkt-RF     SMB     HML      RF  Month_Return
0  2018-01-31  0.0557 -0.0315 -0.0133  0.0012      0.270284
1  2018-02-28 -0.0365  0.0023 -0.0107  0.0011     -0.014848
2  2018-03-31 -0.0235  0.0405 -0.0023  0.0011     -0.043017
3  2018-04-30  0.0028  0.0114  0.0054  0.0014     -0.028887
4  2018-05-31  0.0265  0.0526 -0.0318  0.0014      0.122036
5  2018-06-30  0.0048  0.0115 -0.0233  0.0014     -0.060629
```

5.3 Model Implementation

To calculate the betas for the 3 factors, the first thing we need to do is separate the independent variables from the dependent variable.

The independent variables (the Mkt-Rf, SMB, and HML columns) will be put in a separate list.

```
X = ff_data[['Mkt-RF', 'SMB', 'HML']]
```

The dependent variable of the relationship will be the excess returns of the selected stock. To calculate excess returns, we will subtract the risk-free rate from the percentage return of the stock.

```
y = ff_data['Month_Return'] - ff_data['RF']
```

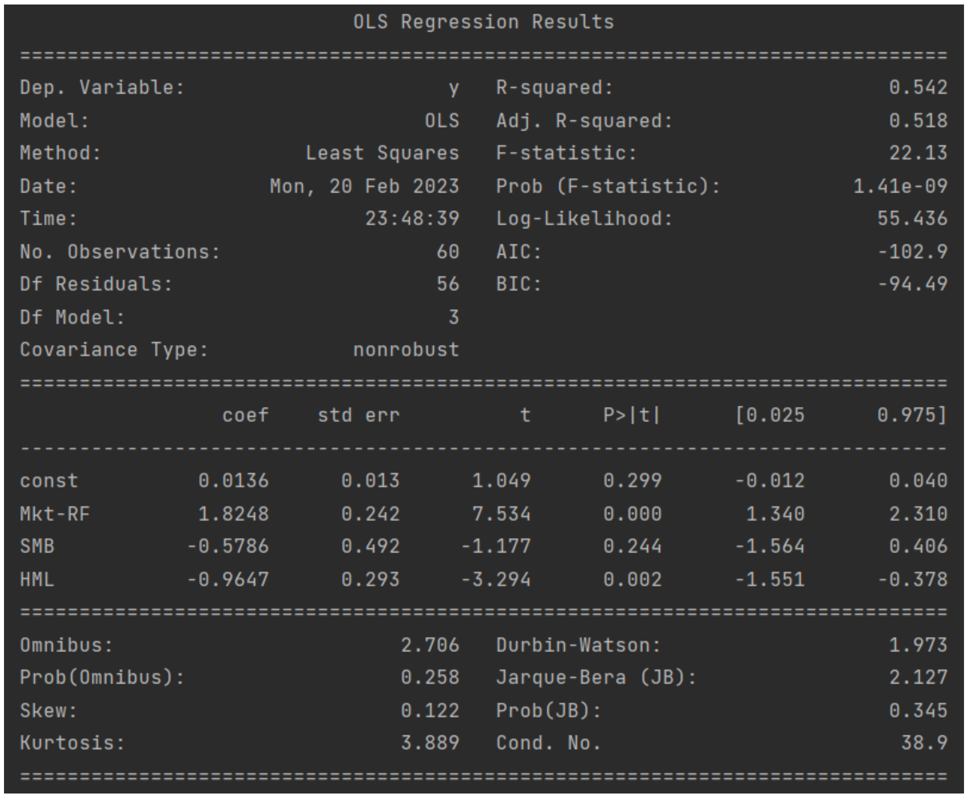
Next, we use the add_constant() function to add a constant to X. This line is required by the statsmodels package because without it, it will try to fit a line through the origin, and our regression will not have an intercept. An intercept is not included by default and should be added by the user.

Afterwards, we fit the model using the *OLS()* function of the statsmodels package, and print the summary of the regression results.

(OLS stands for ordinary least squares which is the conventional technique for estimating coefficients of linear regression equations).

```
X = sm.add_constant(X)
ff_model = sm.OLS(y, X).fit()
print(ff_model.summary())
```

The results are as follows:



```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.542
Model:                            OLS   Adj. R-squared:                  0.518
Method:                 Least Squares   F-statistic:                     22.13
Date:                Mon, 20 Feb 2023   Prob (F-statistic):           1.41e-09
Time:                        23:48:39   Log-Likelihood:                 55.436
No. Observations:                  60   AIC:                            -102.9
Df Residuals:                      56   BIC:                            -94.49
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0136      0.013      1.049      0.299      -0.012       0.040
Mkt-RF         1.8248      0.242      7.534      0.000       1.340       2.310
SMB           -0.5786      0.492     -1.177      0.244      -1.564       0.406
HML           -0.9647      0.293     -3.294      0.002      -1.551      -0.378
==============================================================================
Omnibus:                        2.706   Durbin-Watson:                   1.973
Prob(Omnibus):                  0.258   Jarque-Bera (JB):                2.127
Skew:                           0.122   Prob(JB):                        0.345
Kurtosis:                       3.889   Cond. No.                         38.9
==============================================================================
```

As you can see, the betas for the market, SMB, and HML factors are 1.82, -0.58, and -0.96, respectively. This indicates that NVDA is a company with a lot of systematic risk, it is a large company, and a growth company.

5.4 Estimating Expected Returns

As a reminder, to estimate a stock's expected yearly return according to the Fama-French asset pricing model, we use the following formula:

$$E(R_i) = R_f + \beta_{i,M} E(R_M) + \beta_{i,SMB} E(SMB) + \beta_{i,HML} E(HML)$$

To implement this formula, we first save beta values to 3 variables.

```
intercept, b1, b2, b3 = ff_model.params
```

Then, we calculate the mean values of the risk-free rate, market premium, size premium, and value premium.

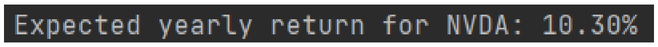
```
rf = ff_data['RF'].mean()
market_premium = ff3m['Mkt-RF'].mean()
size_premium = ff3m['SMB'].mean()
value_premium = ff3m['HML'].mean()
```

Afterwards, we write down the formula. Since the data we are using is monthly, we will need to multiply the number we get by 12.

```python
expected_monthly_return = rf + b1 * market_premium + b2 * size_premium + b3 * value_premium
expected_yearly_return = expected_monthly_return * 12
```

Finally, we print out the result, and format it to our liking.

```python
print("Expected yearly return for " + ticker.upper() + ": " + str("{:.2%}".format(expected_yearly_return)))
```



```
Expected yearly return for NVDA: 10.30%
```

5.5 Python Code

Here is a copy of the Python code, with notes:

```python
import yfinance as yf
import statsmodels.api as sm
import getFamaFrenchFactors as ff

# extracting the data

ticker = 'nvda'
start = '2017-12-01'
end = '2022-12-31'

# extract the data from Y!finance and convert it to % returns
stock_data = yf.download(ticker, start, end)["Adj Close"]
stock_returns = stock_data.resample('M').last().pct_change().dropna()
stock_returns.name = "Month_Return"
print(stock_returns)

# extract the Fama-French factor data and rename date column
ff3m = ff.famaFrench3Factor(frequency='m')
ff3m.rename(columns={"date_ff_factors": "Date"}, inplace=True)
print(ff3m)

# merge stock return and fama french factor datasets together
ff_data = ff3m.merge(stock_returns, on='Date')
print(ff_data)

# linear regression
# list of independent variables
X = ff_data[['Mkt-RF', 'SMB', 'HML']]
# dependent variable = excess returns = returns (%) - risk-free rate change
y = ff_data['Month_Return'] - ff_data['RF']
# adding constant to add the intercept to the model
# intercept is not included by default and should be added by the user
X = sm.add_constant(X)
# fitting the model
ff_model = sm.OLS(y, X).fit()
print(ff_model.summary())

```

```
# calculating expected returns
# specify the model parameters needed for the formula
intercept, b1, b2, b3 = ff_model.params
# finding the average of the factors, and rf rate
rf = ff_data['RF'].mean()
market_premium = ff3m['Mkt-RF'].mean()
size_premium = ff3m['SMB'].mean()
value_premium = ff3m['HML'].mean()
# monthly expected return
expected_monthly_return = rf + b1 * market_premium + b2 * size_premium + b3 *
value_premium
# yearly expected return
expected_yearly_return = expected_monthly_return * 12
print("Expected yearly return for " + ticker.upper() + ": " +
str("{:.2%}".format(expected_yearly_return)))
```

Works Cited

"Linear Regression in Machine Learning", *javaTpoint*, https://www.javatpoint.com/linear-regression-in-machine-learning