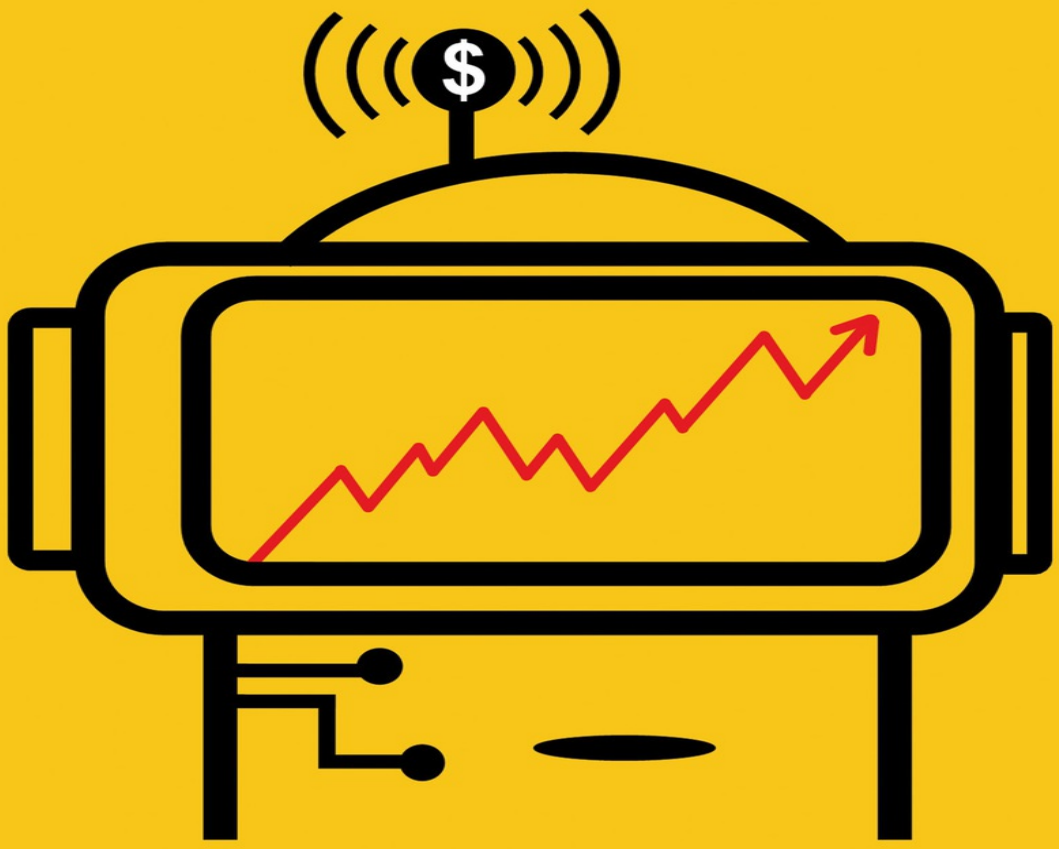
2nd Edition

BUILD YOUR OWN

AI INVESTOR

With Machine Learning and Python *Step by Step*



. . . even if you've never coded before

DAMON LEE

BUILD YOUR OWN

AI INVESTOR

With Machine Learning and Python, *Step by Step*

…Even if you've never coded before.

Damon Lee

Disclaimer

The author does not make any guarantee or other promise as to any results that are contained within or may be obtained from the content of this book. You should never make any investment decision without first consulting with your financial advisor and conducting your own research and due diligence. To the maximum extent permitted by law, the author disclaims any implied warranties of merchantability and liability in the event any information contained in this book proves to be inaccurate, incomplete or unreliable or results in any investment or other losses.

ISBN: 978-1-8381322-3-1

Table of Contents

Preface

About This Book

This book aims to teach almost anyone to be able to use freely available Python Machine Learning tools to build their own AI stock investor. The idea being that readers will finish with some knowledge about investing as well as a program of immediate use, made with their own hands, even if they never wrote code before.

Investing is a difficult activity fraught with uncertainty, compounded by questionable incentives, and occasionally involving unscrupulous characters. If a layperson can program their own “Robo-investor”, they can eliminate the last two issues and help minimise the first. At the very least they will have a better understanding of the strengths and limitations of their own creations.

The performance benchmark of the AI investor will be the S&P500 index. The performance of the investing AI made from the code shown in this book will be displayed publicly on the website *valueinvestingai.com*, starting from mid-March 2020. The AI will select stocks every March, as the annual reports are released around that time, and that data is required for stock selection. The stocks selected will be recorded on the Bitcoin blockchain, and further information can be found on the books website and on *ai-investor.net*.

Although trying to beat the S&P500 benchmark with instructions from a book sounds too good to be true, it has been achieved before. In fact, it has been done many times. It has been shown to be possible when approached from a value investing perspective, with long investing time horizons and acceptance of high volatility.

This guide is not intended to be an exhaustive course on Python, nor does it teach the intricacies of all Machine Learning algorithms. This guide is intended to be practical and hands-on, where the Machine Learning concepts are explored in intuitively useful code tutorials in a narrative manner toward the goal, building competency along the way. Furthermore, this guide aims to and to show where to look to delve deeper if desired. This is a guide for building something, what you do with what you build is your responsibility.

Everything is written in plain English where possible, with all the code to create the AI present in print. Readers who want to skip to the end and get the AI investor working immediately can do so. This book assumes some basic knowledge of mathematics, though nothing is present that can't be looked up on the internet and understood quickly.

This book is intended to be self-contained apart from using freely available code libraries and market data from *simfin.com*. Free data from *simfin.com* lags the present by 1 year, which is perfectly fine to complete all exercises with. To produce up-to-date stock selection with the investing AI, a SimFin+ subscription is required (€20 as of time of writing).

Because of the market data source, survivorship bias is present in the models, though is well mitigated by high backtest performance giving a margin of safety, as well as the use of Altman Z scores to account for default probability in stock selection. This issue may be eliminated by purchasing more detailed market data online should the reader wish to do so with little code change (though this is more expensive than *simfin.com*). In this book only common stocks listed in the USA are considered.

Who Should Read This Book

Besides people who just want to make the investing AI, there are others whom this book will be useful:

People learning to code with Python will find a book that holds your hand from Python installation and setup all the way to creating a working AI stock picker.

Those who have some programming experience, but want to understand Machine Learning algorithms will find tutorials to teach the concepts required, with a final hands-on application at the end that is useful.

Programmers with no investing knowledge will get an understanding of the stock market approached from a data perspective with good theoretical backing, in doing so understanding why there is more to a stock than a ticker symbol that goes up and down. It will provide perspective on how the investing problem should be approached, and how Machine Learning can be applied in a theoretically consistent manner.

Stock investors who want to know what Machine Learning and Python tools can do for them, will find that the coding knowledge will enable them to make far more interesting screens than what online services offer as well as gaining Machine Learning knowledge that can be used to inform investing decisions in as many ways as they have the creativity to imagine.

Book Roadmap

Chapter 1 is an introduction that contains the theoretical underpinning of our investing AI, explaining why it works, why it makes sense to correlate certain variables, and revealing why the correlations found are unlikely to be co-incidental fleeting things that may disappear tomorrow.

Chapters 2-3 gets into Python coding and basic Machine Learning concepts, with step by step tutorials designed to develop an intuitive understanding in the reader. The tutorials develop an understanding of the limitations of what can be used and why the Machine Learning algorithms work generally.

Chapter 4 draws on the previous coding and Machine Learning lessons to build the investing AI. All code is shown as a set of walkthrough instructions with explanations of what each line does and reasoning behind any design decisions.

Chapters 5-7 subjects our investing AIs to backtesting with historic stock price data to whittle down the number of potential prediction algorithms to get the best stock picker. The backtesting also allows us to see how the AI is likely to perform in reality as well as the impact that tweaks in the algorithm have on projected return and volatility. Chapter 7 contains the final investing AI stock picks for March 2021.

Miscellaneous

Python code executed in Jupyter Notebook cells is written with the below ‘Constant-width’ font, which is executable code. Constant-width font that is after a # is a code comment. Output is displayed immediately below with **In[]** or **Out[]**
cell markings to match the formatting in Python notebooks.

In[1]:

```
# This is a code comment
a = [1,2,3,4,5]
```

```
print(a) # Here some output is printed out
```

Out[1]:
[1, 2, 3, 4, 5]

Sometimes obvious or trivial code is left out. If displaying code that is part of a larger body, […] will be used.

> ## *Exercise 0*
>
> Throughout this book, there are small programming exercises to learn through hands-on activity. These are formatted like the text here. Typically this takes the form of a Jupyter Notebook download from the Github repository with instruction comments inside. Solutions are available there too.

Italics are used for *emphasis*, *book titles*, *websites* and *code* in body text. When code functions are presented in the text they will have brackets after them so you know to pass arguments between the brackets: *my_function()*.

All code is available from the following Github repository online through this link: *https://github.com/Damonlee92/Build_Your_Own_AI_Investor_2021*. If you are used to using Github, feel free to fork the code and show the community your personal AI Investor (provided you give original attribution).

Chapter 1 - Introduction

Investing in Stocks

What are stocks and why do they have value? Why do they keep changing in value? It's worth briefly going over the basics by answering these questions because our AI will be purchasing stocks for investment, so we need to know what it would be buying. A good book that fully covers these fundamentals is *Why Stock Go Up And Down* by William Pike and Patrick Gregory.

Stocks have value because they confer to the holder ownership in pieces of a company. Part ownership of a company sounds grand, although being a part-owner doesn't mean you directly run the place, you can't enter the office and tell people to do things if you have a bit of the company's stock.

The company is supervised for the shareholders by the board of directors. Essentially the board of directors work for you (the shareholder) in managing the business and hiring/firing important people. Anyone with a lot of shares can influence who goes on the board, sometimes large shareholders themselves have a seat on the board.

Once a year you get to vote on who goes on the board and sometimes vote on a meaningful company event like a big acquisition. Your vote is proportional to the number of shares you have, so chances are there are a lot of pension funds and the like with a bigger vote than individuals. A lot of ordinary people who hold stock don't bother voting for anything.

If you want to increase your stake in the company, you can buy some shares. The trading activity for shares is just an auction. Taking part is easy, most banks have a stockbroking platform, and some phone apps let you trade for free. There are sometimes mundane hoops to jump through: to own foreign stocks your stockbroker will usually need you to sign a form, for example.

Shareholders receive a dividend if the board of directors approve one, and if business profits accumulate over time and are used to expand the business, all else being unchanged, the company value will increase. After dividends and costs (such as employment etc.), the money either stays in the company bank account or gets used for something like new machinery or new product

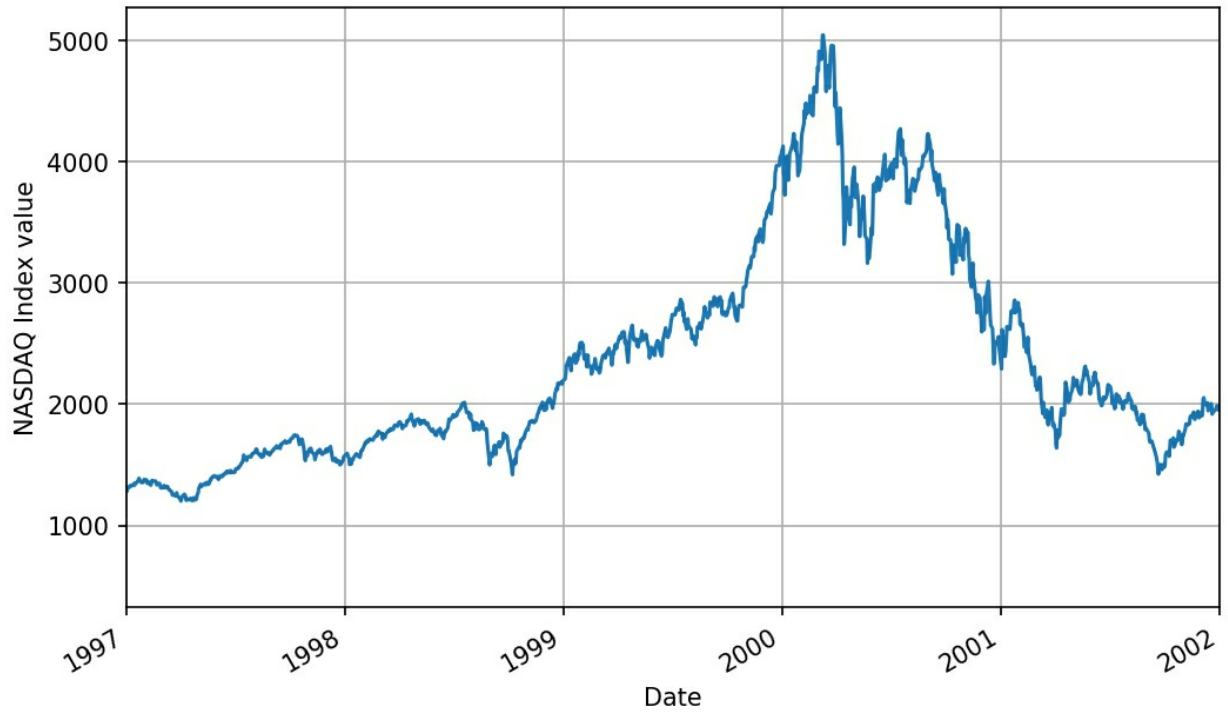
development. This kind of activity should increase the price of each share.

Note that it *should* increase the share price, it is not a certainty. Ultimately the price of a stock is simply what someone else is willing to pay for it. A reasonable person will say the stock should be valued based on the cash to be made by the physical underlying business over time, with some accounting for the money being received sooner or later, the certainty of it, and so on.

For most companies, the money being made (and thus the company value) doesn't change all that much over time. Reasonable people would say that the value of a company isn't going to change much from one month to the next.

Are people reasonable all of the time? No.

Things happen. When the world is pessimistic people might say the sky is falling. As I wrote the first edition in May of 2020 there were mass riots in the USA, a worldwide pandemic, a trade war between superpowers, and so on. The opposite is possible too, no doubt some reading this will be old enough to remember the dot com bubble, which burst 20 years ago. If you were born after that event, take a look at the wild swings in the NASDAQ index around that time:



Let us now invoke the eponymous Mr Market, an allegory introduced by

Benjamin Graham in his book *The Intelligent Investor* (perhaps the most famous book on investing). Imagine you run a business, Mr Market is a business partner who tells you what he thinks your business interest is worth every day, offering to sell some of his stake or buy more of your stake when he does so (He is an obliging man).

One day he might tell you he will pay many times over what you think your stake is worth, and another he might sell his stake for pennies. A prudent investor would gladly increase their stake when Mr Market is depressive and sells his stake too low, and decrease their stake when Mr Market is overly optimistic and wants to increase his stake at a high price.

To do this, the prudent investor must form their own opinion of what the value of the business is, independent of Mr Market (also independent of your neighbour, loud people on TV, your drinking friends, etc.) and acting when there is a margin of safety in case the valuation is a little wrong.

To come to a value for a business, that is, the value of the future cash flows from the business, discounted by the value of that money coming sooner or later (it's worth less if it comes in the future), an investor will need to consider several variables such as the quantity of debt, the stability of past earnings, the future risks, and so on. In *The Intelligent Investor,* Ben Graham gave a few formulaic investing guides for the public which considered these variables to try and value businesses. The underlying thinking behind his methods has been proven to work over time, notably by prominent followers like Warren Buffett and Seth Klarman.

Investing is about buying something that you can reasonably expect someone will pay more for in future. We can do this with a high probability of success if the item is purchased at considerably less than its intrinsic value. This is Value Investing.

Ben Graham pioneered this kind of investing technique in the Graham-Newman corporation (where a certain Mr Buffett worked), with employees screening stocks systematically with checklists by hand. If it sounds dull, it probably was (but lucrative).

Such strategies inevitably depend on other market participants eventually coming to your view about the value of a company. If you're worried that there is nothing tangible about stock prices, and that you are depending on

how others value the ephemeral “company ownership”, there is a lower bound to company value you can lean on: According to Seth Klarman in *Margin of Safety*, the ultimate tether to reality for company valuation is in liquidation. When the business folds the bondholders get priority to the company assets, anything left over belongs to the shareholders.

The simplest way to invest by taking advantage of this is to buy stocks that trade below this liquidation value, which is theoretically like buying dollar bills for 80 cents. To be sure of a return value investors advocate purchasing assets at a considerable margin of safety, and selling when there is no margin of safety, taking advantage of Mr Markets manic depressive quotes when they appear.

Finding stocks that meet the liquidation formula, valued below tangible book value, stopped being easy a long time ago. This shouldn’t be surprising given that this method of valuation is such an obvious one.

After market participants cotton on to this kind of valuation measure, investors need to develop a more nuanced (and less obvious) understanding of value to have an edge in the game. There have been many other formulaic investing guides written for the public that work without depending only on liquidation value, for example purchasing stocks with a low price relative to sales or earnings.

It is no accident that the investing formulas that work build on Ben Graham's methods, focusing on getting the most bang for your buck, whether it is more assets for your dollar (sometimes literally buying a dollar for 80 cents) or greater company efficiency per dollar (like return on equity). Screening for certain company metrics and ratios for “cheap” stocks for making investment decisions might be called formulaic value investing. There have been many books written on this subject, each giving slightly different formulas for finding value.

Formulaic Value Investing

Most formulaic value investing guides make a big point about human psychology, and for good reason. When the price of a stock goes up we feel validated, and we are likely to want to buy more. Conversely, when the price of a stock goes down we are quick to think “I’ve just lost thousands!”-a lot of

us probably see Mr Market when we look in the mirror.

The fluctuations are always going to be there because there are always going to be differences of opinion, and because the vagaries of human emotion are somewhat fundamental to our nature.

A common theme in formulaic value investing is to disregard volatility, which flies in the face of a large proportion of institutional investment thinking. A common axiom in the financial thinking is to equate price fluctuations with the risk of investment loss. Attempts to smooth portfolio performance abound. These fluctuations should in fact give the retail investor an advantage, as there is no institutional incentive to limit a portfolio by volatility.

The reason for such emphasis on volatility in the profession is that money managers are effectively selling a service, that is, their investment expertise in managing money. In the absence of detailed knowledge of how the manager invests money, a customer is far more likely to invest with a manager that has past performance as a straight line going up than one with large swings.

Sticking to a formula is no easy feat, but in a way that is to be expected. If everyone did it these strategies wouldn’t work. For a stock to be selling at some low ratio that appears in a value formula, it had to get there somehow, furthermore, it could easily fall further if crowd psychology takes over. Buying a stock because it is cheap only to see it get cheaper can be gut-wrenching, and there is always the chance that a value investing formula could be wrong for a particular stock.

The feelings an investor in these strategies will go through are understandable beyond just the price fluctuations too. They don’t truly know if the statistics are true, even if they come from an authoritative figure, and they don’t know if the formula that worked in the past was a fluke. They also don’t have a frame of reference for how important factors like Price/Earnings ratio are to a company, is 30 a high number? Is it low? Does it matter? Should I trust the investment guru if they say yes? Having little experience is a big psychological handicap.

Joel Greenblatt's formula, a formulaic value investing method detailed in *The Little Book That Beats the Market* (*www.magicformulainvesting.com*) gave a

30% compounded annual return. This is great information, but how many people would bet most of their wealth on it from being told this? Probably not that many. How many people would bet on it if they grabbed the data and checked it out for themselves? Probably more. And what if they had the skills to check the importance of the various formulas in predicting company performance? Probably more still.

So, what are these formulas? Well, they range from complicated factor models organisations like Blackrock and Vanguard use, down to simple formulas looking at accounting ratios like “Buy at a low Price to Sales ratio with no debt”.

Contrarian Investment strategies and *What Works on Wall Street* are two good examples of more recent formulaic investing books intended for the public that have been proven to work over time. These formulaic investing books give the reader formulas built from various accounting ratios, just like Ben Graham did. These methods are likely to continue to work in future given how scared most investors are of volatility.

There is even a whole book about just a single formula. *The Little Book That Beats the Market* advises buying stocks with a low Earnings Before Interest and Tax to Enterprise Value (EBIT/EV) ratio and a high EBIT/(Net Working Capital + Fixed Assets) ratio. Combining the two into a single score and simply buying the top-ranked scores.

In this book the approach we will take is a similar one, we will be taking these ratios and using them to pick stocks to try and beat the market. The twist here is that we will get the Machine Learning algorithms to do it for us, and furthermore, this book will give you the tools to call on the algorithms yourself and even analyse the stock market to come up with your own strategies and insight. Technology is the great equaliser.

Value Investing with Machine Learning

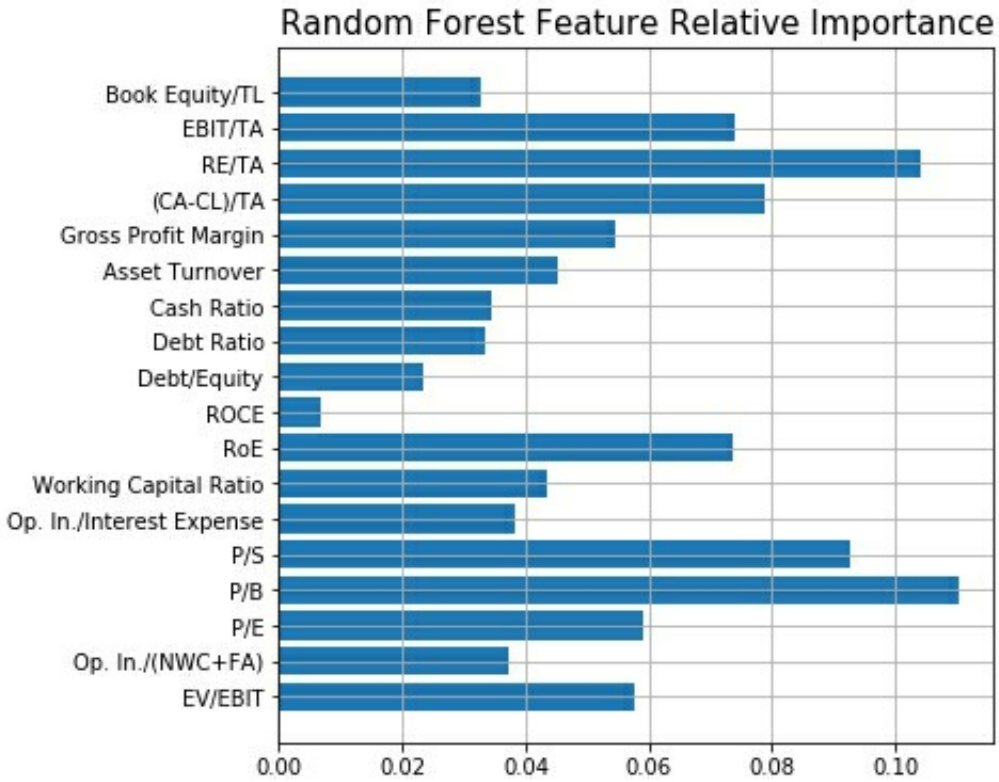
The activity of dealing with these company ratios lends itself readily to Machine Learning. In fact, Machine Learning is for the most part applied statistical analysis, and what are these formulaic investing studies if not statistical analyses to start with?

Machine Learning is ubiquitous in modern life for good or ill, from

recommending the next news story on Twitter to product recommendations on Amazon to running election campaigns. These algorithms are already used to drive vehicles, and no doubt many professional investors are using them to trade in financial markets as you read this.

Despite the wondrous capabilities of these algorithms, the base concepts behind their operation aren't that complicated. The value they confer to big corporations is derived from the vast amount of data they keep, not from the algorithms themselves. In fact, the source code for a lot of these tools is freely available, sometimes provided by the big corporations themselves: Facebook provides their PyTorch library, Google provides TensorFlow.

These tools have been honed over a long time, such that anyone could create a rudimentary AI in a weekend if they have the data to train it, doing anything from recognising photos of hot dogs to predicting house prices. In this book someone starting from zero can be taught to use Machine Learning algorithms to draw out useful information about the market, for example, the following chart is generated later on in this book after an algorithm called the 'Random Forest' discerns the apparent importance of various accounting ratios on stock performance. The AI is learning from these ratios the same way a stock analyst might.



We will be using a popular Machine Learning library to create our value investing AI called Scikit-learn. However, many early tutorials will be with raw code without the library to teach the concepts behind the algorithms before using the library to do the heavy lifting. The data we will use for our value investing AI to learn from is publicly available on the internet, collated from annual reports by *SimFin.com*.

This activity is similar to the statistical analysis done in formulaic investing books previously mentioned, except that the reader will do it themselves with code. Modern programming tools have advanced sufficiently far for anyone with a computer and internet connection to be able to apply Machine Learning algorithms easily. Anyone can use these tools to glean some insight from stock market data if they knew how to code and knew what to look for. This book hopes to provide the knowledge required to do this for the reader through the hands-on activity of building their own investing AI.

Machine Learning with Python

The Machine Learning algorithms are all well and good, but we will need to do some programming to use them. To someone who has never done any

coding before this might sound daunting, though there is no reason to worry because the programming language we will be using is Python.

Python has effectively become the Machine Learning *lingua franca*. The reasons for this, besides the first-mover advantage in some areas, read like a list of advantages over other languages:

Readability Counts

One of the maxims of Pythons design is that readability counts. This makes it easier to understand, easier to maintain, easy to learn. Python equivalent C++ code is typically half the size. There's no need to declare the type of variable you create or fiddle with memory address references and the like.

General Purpose

Python is general purpose. You can use it yourself to automate boring stuff on your computer like emails (*https://automatetheboringstuff.com/*), it's used heavily in big organisations in the new economy like Google and Netflix, it's used for scientific computing (your author has first-hand experience of this), it can even be used drive your car today (*https://comma.ai/*).

Quick Iteration

There is no need to compile your code because Python is an interpreted language. Just type it down, execute, and it runs. Life is easier with no need to wait a minute every time you make a code change. You can learn quickly when the cost of making a mistake is a mere second or two. Move fast and break things.

Fast Enough

There is some cost to the speed of code execution as there is no code compilation, but most of us aren't doing high-frequency trading or computing particle accelerator data. Besides, if you need speed you can always have a faster language do the important bits. The approach taken at Google early on was "Python where we can, C++ where we must". If it's good enough for Google, it's probably good enough for us.

Open Source

It is completely free.

Chapter 2 - Python Crash Course

Advanced users can skip this section, what's needed is Jupyter Notebook with Python 3.7 from an Anaconda set-up. We will use the default Anaconda environment and won't be bothering with setting up a separate environment or workspace because nothing advanced will be used here.

Background for Absolute Beginners

There are many ways to run Python code. You can just write it in a text file and execute it, you can use a development environment such (as Spyder) to get more interactivity, such as being able to see variables you have stored in memory, or you can use Python interactively from the command line.

We are going to use a Python 'Notebook' program, known as Jupyter Notebook to do our coding. Jupyter is used in this book is because it's used for a lot of real-world data science and it's a great environment to write code and iterate quickly, with data graphing capabilities right where you write the code, which helps with learning, furthermore it is quite user friendly.

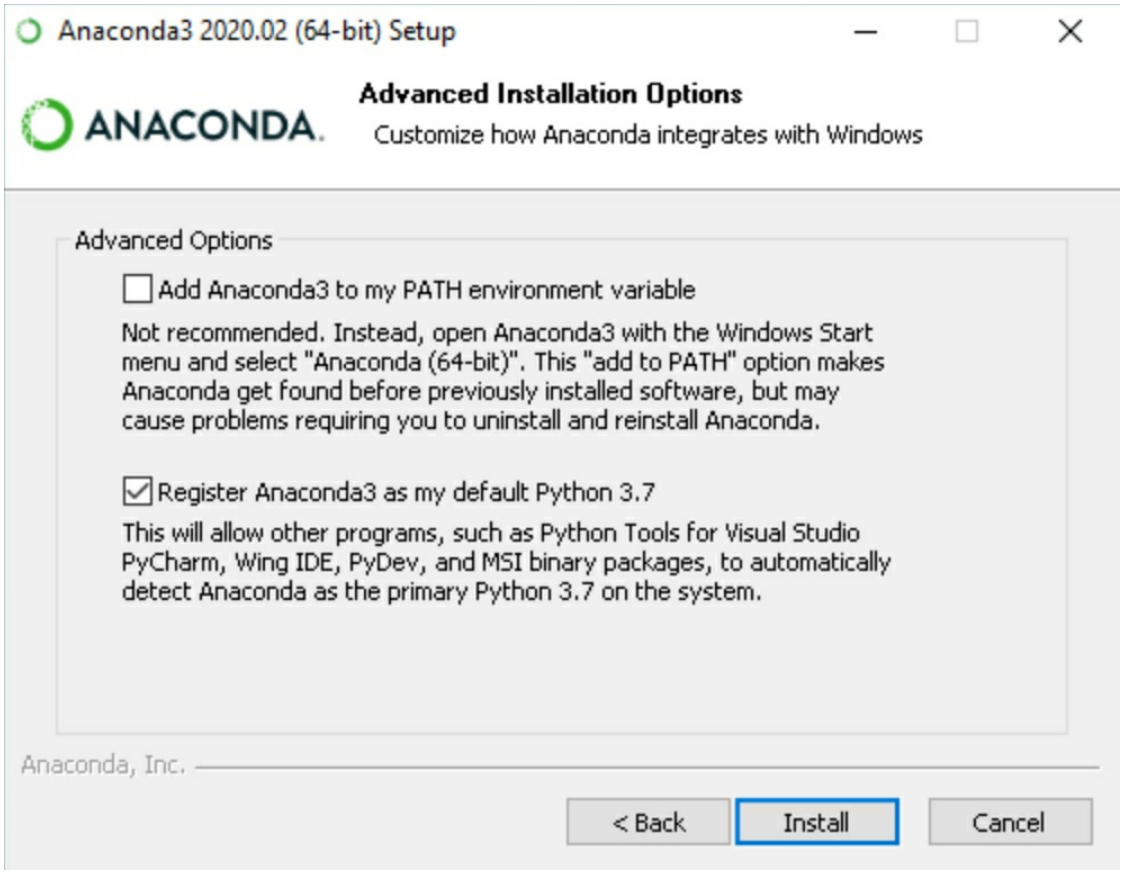
The official Python documentation is at *docs.python.org*, and is easy to look up functions and examples, so there's no need to memorise things.

Getting Python

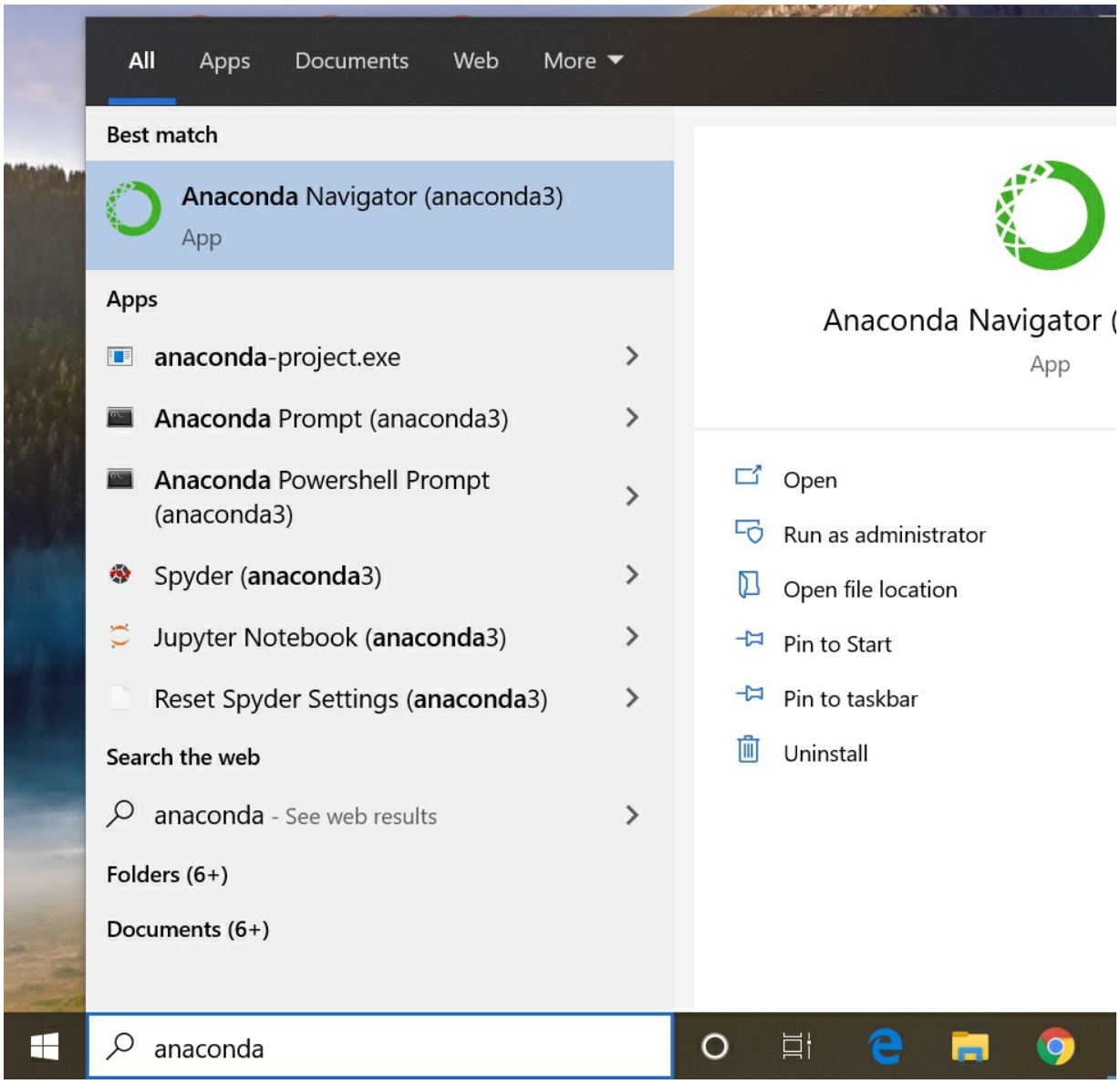
We will be using Python through the Anaconda distribution. This gives us Jupyter Notebook and all the commonly used scientific and Machine Learning packages in a single distribution, so we have everything we need and don't need to think about add-ons. For Windows users, there is a user interface which will make things more comfortable for some users (as opposed to doing everything from a system shell). The latest installation instructions can be found on the following website: *https://docs.anaconda.com/anaconda/install/* for Windows macOS and Linux.

A Windows installation is less straightforward, and as most absolute beginners will be using Windows, the important installation steps are provided here.

Go through the latest installation steps from the web address above, when reaching this window in the installation process leave the top box unchecked.



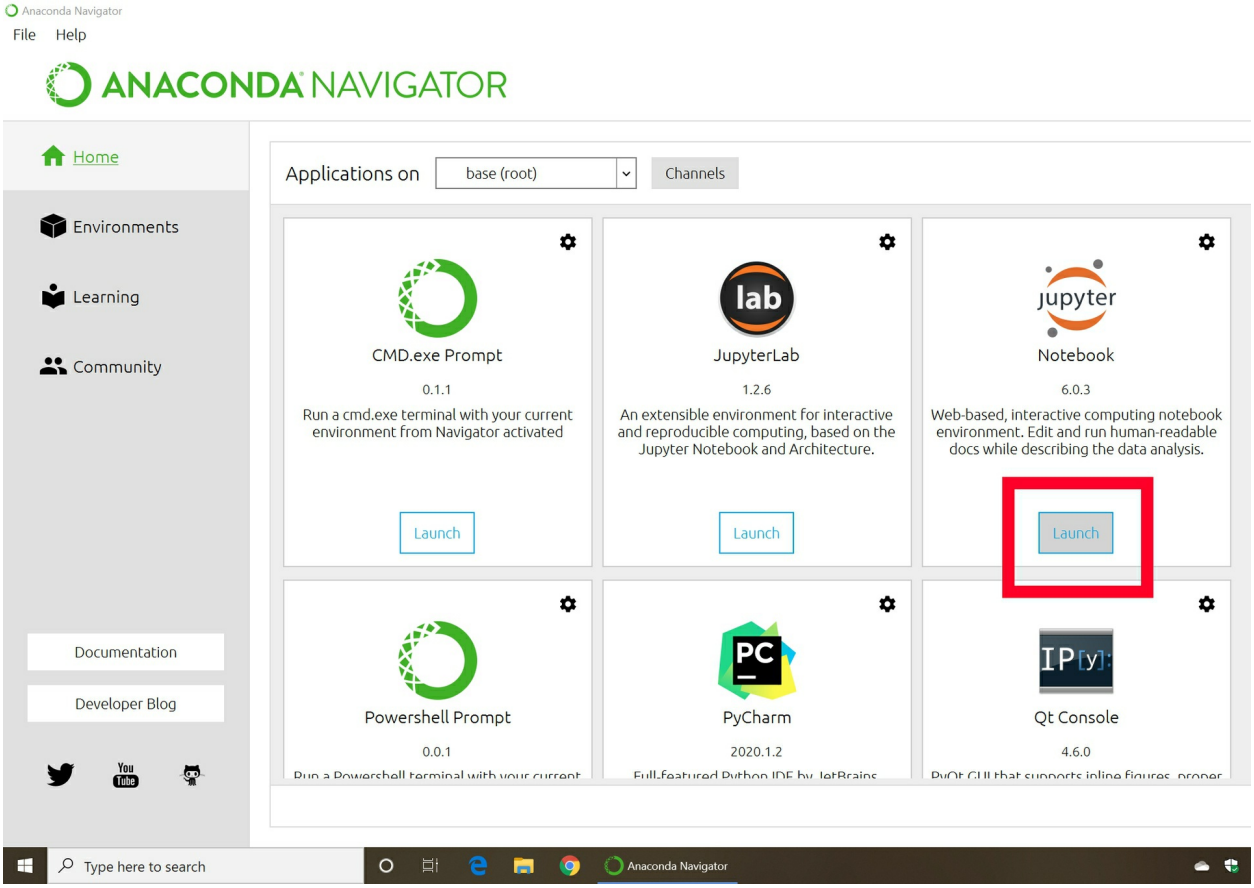
To launch Anaconda to begin writing a Python program, run the Anaconda Navigator application from the start menu or the search bar. When opening the application, some black command windows might pop in and out of existence, don’t be alarmed, these are just running start-up scripts.



When the Anaconda Navigator launches it will have its name in big green words on the top of the window, so you'll know when you get there. You can create and change Python environments from here and launch various related programs. In projects where you may want to depend on different modules, users create environments to eliminate variability in the modules (an updated module might change a function, breaking your code if you didn't set up a special environment).

All the Python coding that we will be doing will be from Jupyter Notebook, which is an interactive computing environment for Python coding that you interact with within your web browser. You can launch Jupyter Notebook

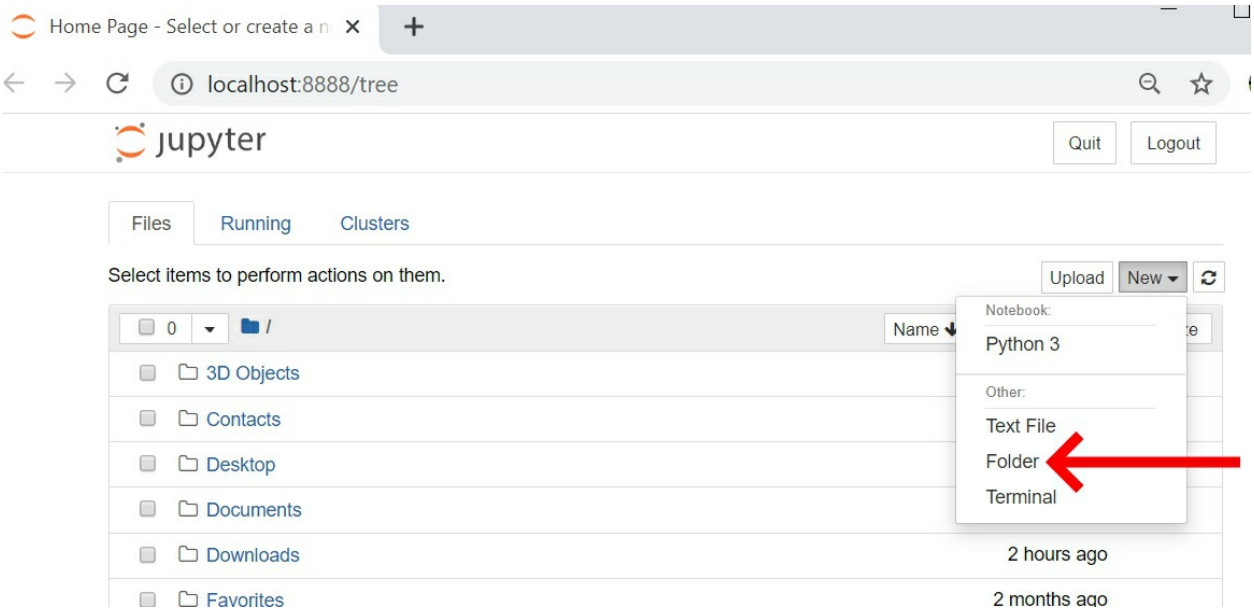
from the Anaconda Navigator:



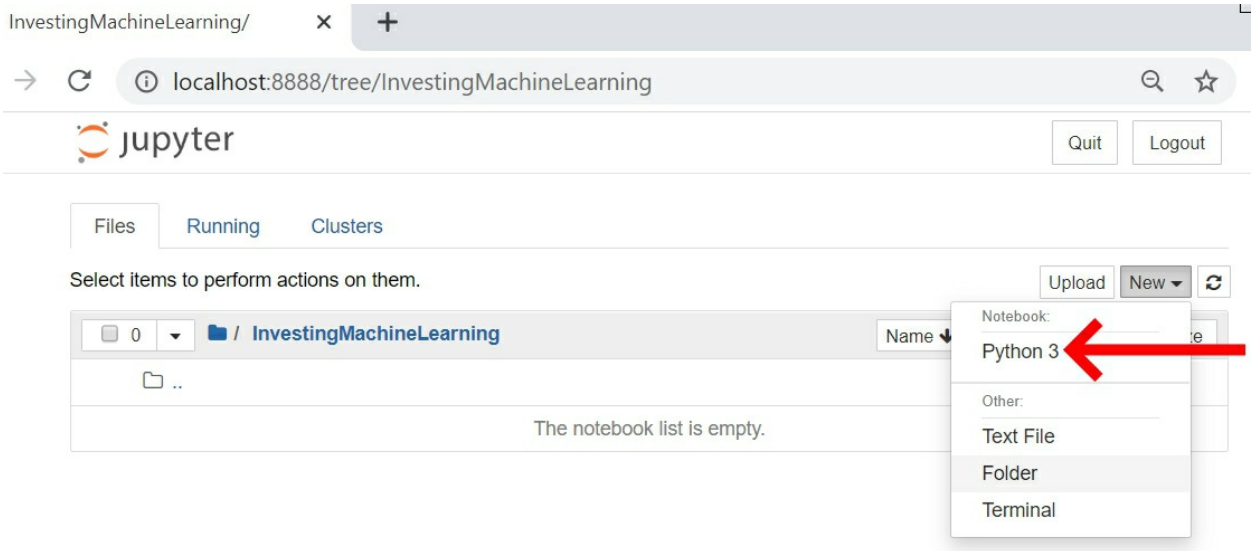
Once launched, your browser will open a locally hosted “web page”. This is Jupyter Notebook. Inside Jupyter you can browse folders as well as open/create Jupyter Notebooks to write/edit Python code.

Python Coding with Jupyter Notebook

Let’s write a Python program. The code in this book will place data and written Python code in the *C:\Users\<Your Username>* location. Create a new folder in this directory by selecting “New” and selecting “Folder”. Call the folder *InvestingMachineLearning* or whatever you would like your project to be called.



Enter the newly created folder and create your first Jupyter Notebook file by selecting “New”, then in the menu under the word “Notebook” select “Python 3”.

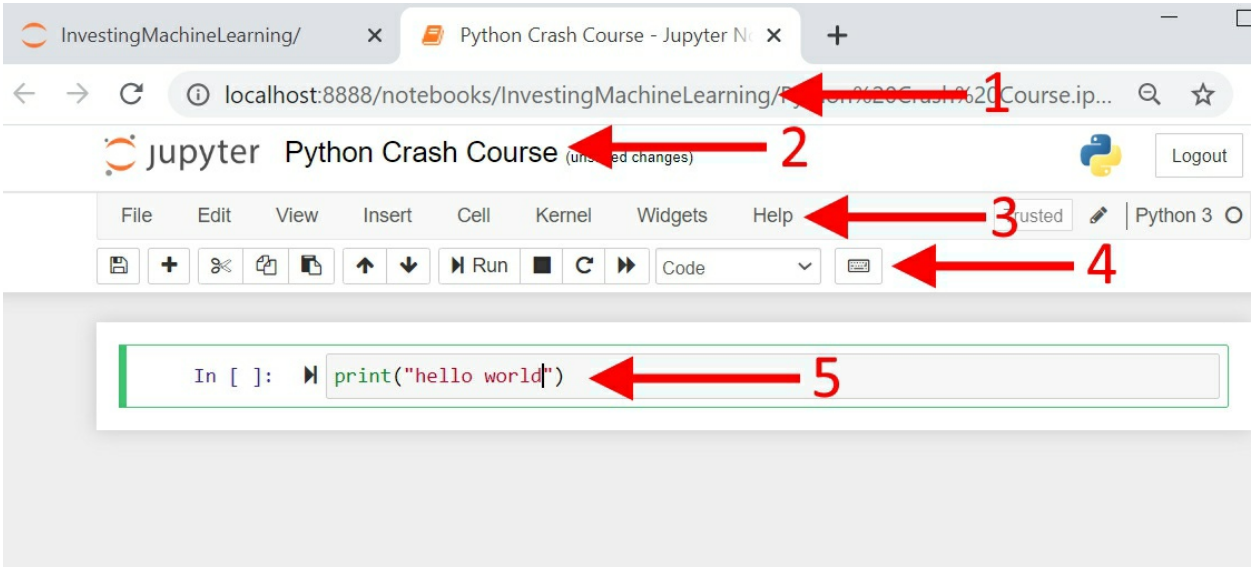


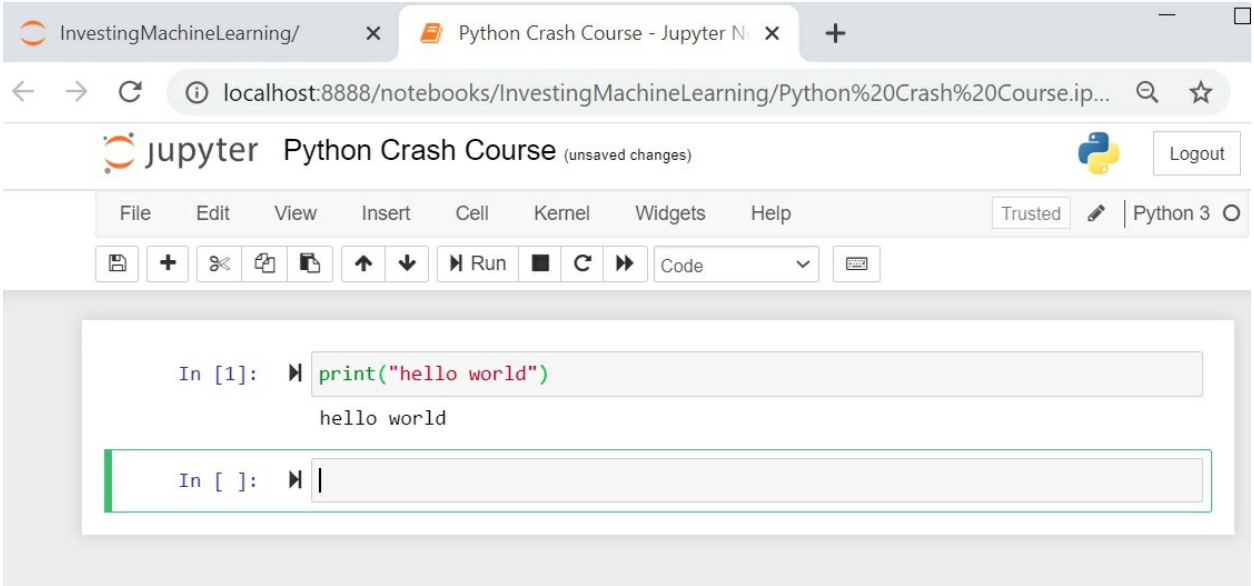
After this, your internet browser will open a new tab and put you into the Jupyter Notebook file. On your web page you may notice that the address in the browser starts with *localhost:8888/*, like that at point 1 in the following screenshot. Your computer is hosting a local webserver, and we interact with the Python interpreter through this web application.

At this point, the file is usually called *Untitled.ipynb* by default, the name of the notebook file we have open is at point 2. You can rename this by clicking

on the text, which will change the file name on your computer. Here the file has been renamed *Python Crash Course.pynb*.

In a Jupyter Notebook, all the things you type out are put into cells. These are the grey box you see with come code in at point 5. Just to check your notebook is working, copy in the "*Hello World*" command into the cell that you can see in the picture. To run a command in Jupyter hold down the Shift Key and press Enter whilst the cell is selected. You can also run the code in the cell with the buttons in the toolbar (4). In the secondary toolbar, you can also undo/redo text or cells, add new cells or delete cells, save the file and so on at point 3.



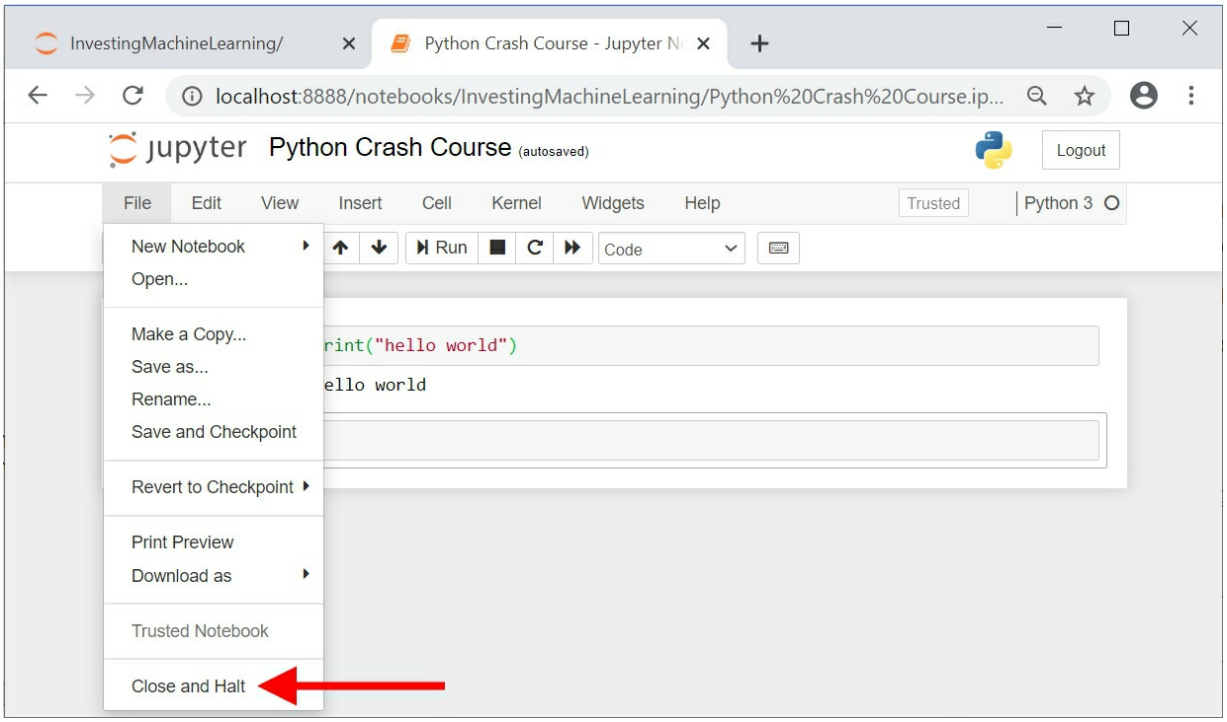


When code executes, any output will be displayed below your selected cell and a new cell will be created below the one you just executed, as you might want to write more code to do something else. To create new cells select the + button from the toolbar. To remove cells go through *Edit → Delete*, or press the *cut* button when you select the cell you want to remove (you can also paste that cell elsewhere if you do this).

Other Bits about Jupyter Notebook

You can check out the Jupyter Notebook user interface tour from the *Help* menu in the toolbar on the right.

To exit a notebook, click on File and select *Close and Halt* on the top left of the page. If you just close the tab, the Python Kernel will still be running in the background. This might not be what you want, for instance if you stored a lot of data in a variable, your random access memory (RAM) will still be used up. You can see which notebooks are open with Kernels running by looking at the main page and seeing which notebooks are highlighted in orange in the directory.



You can completely shut these notebooks down from the Jupyter homepage. If you've closed your Jupyter homepage without meaning to, you can get there by typing the *localhost:8888/* address into your web browser. To exit

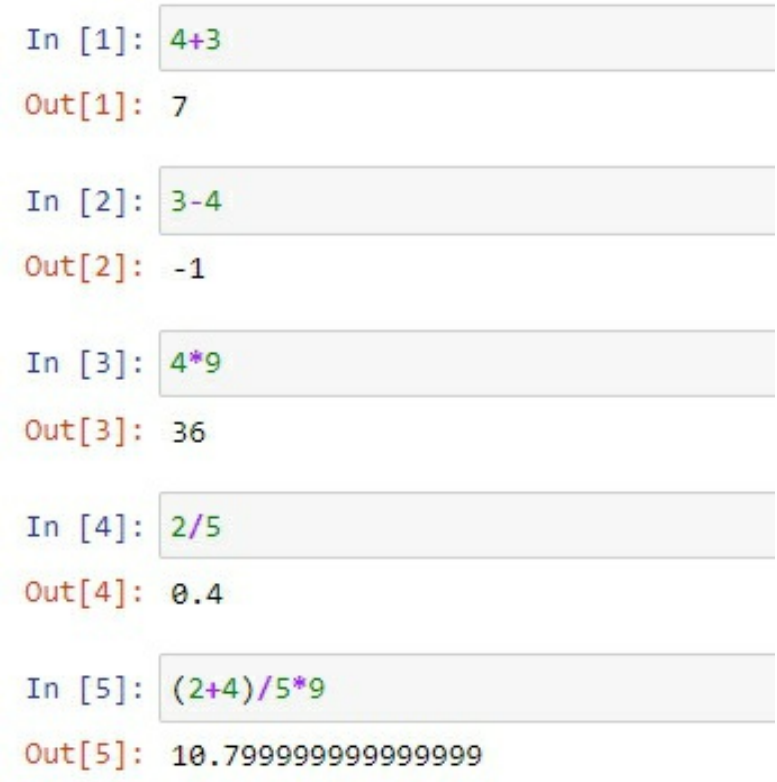
Jupyter properly, go to the main Jupyter page, close all running notebooks and click on Quit on the top right of the page (Don't forget to save your stuff).

Uninstalling is done the usual way for Windows programs, just search for "Anaconda". Some guides tell you to select the "Add to PATH" option in installation, which makes uninstalling a bit trickier.

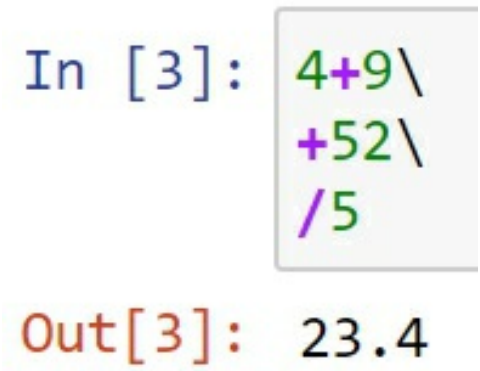
Basic Arithmetic

Now we begin with a quick overview of Python. If you're a stock market enthusiast with no coding experience who wants to run before walking, go straight to the *Pandas Library* section of this chapter.

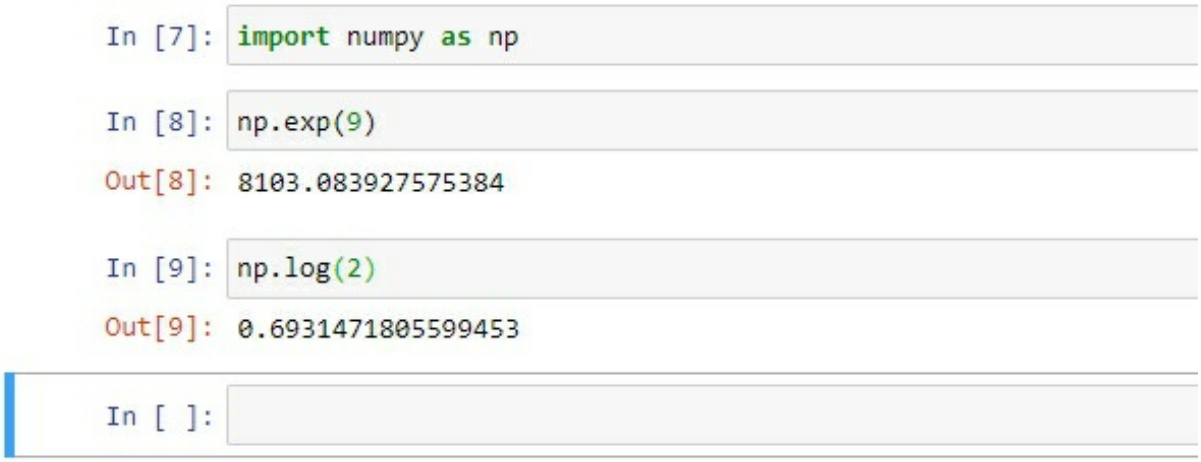
With Python, you can perform basic arithmetic (I'd be worried if you couldn't). In some Jupyter Notebook cells try typing in some basic arithmetic and executing the code, you'll see something similar to the screenshot. All this code is provided online if you wish to follow along with code execution without typing everything out (*https://github.com/Damonlee92/Build_Your_Own_AI_Investor_2021*).



You can continue any code on the next line with a backslash to signify a new line:



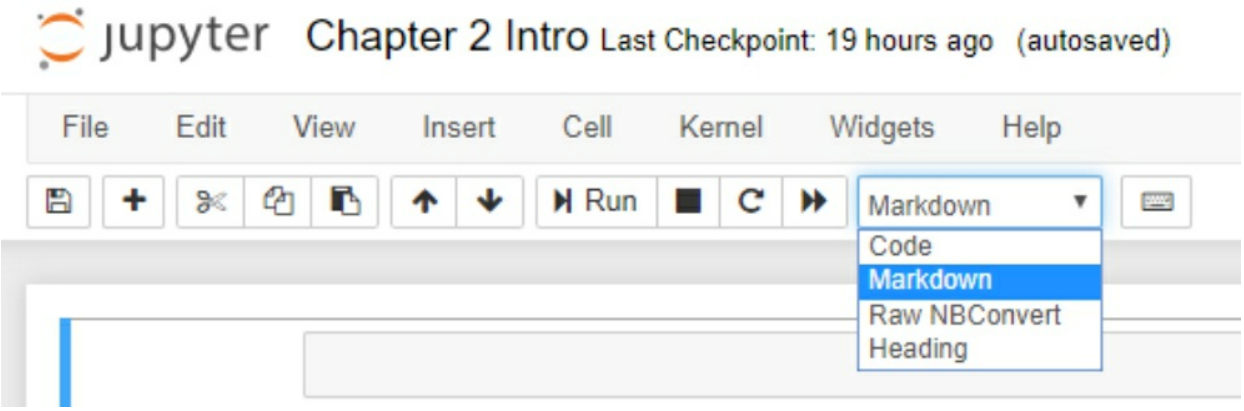
More complex mathematical functions are possible with the Numpy module (a very popular module), which is installed as part of your standard Anaconda environment. To use functions from a module, first import the module. For the Numpy module type in *Import numpy as np* and execute that line in a cell. After doing that, the functions from the module can be called by typing "*np.*" followed by the function from the module you are calling, for example, *np.exp(9)*. If you type *np.* and press tab, you will see a list of possible functions that that module allows you to use.



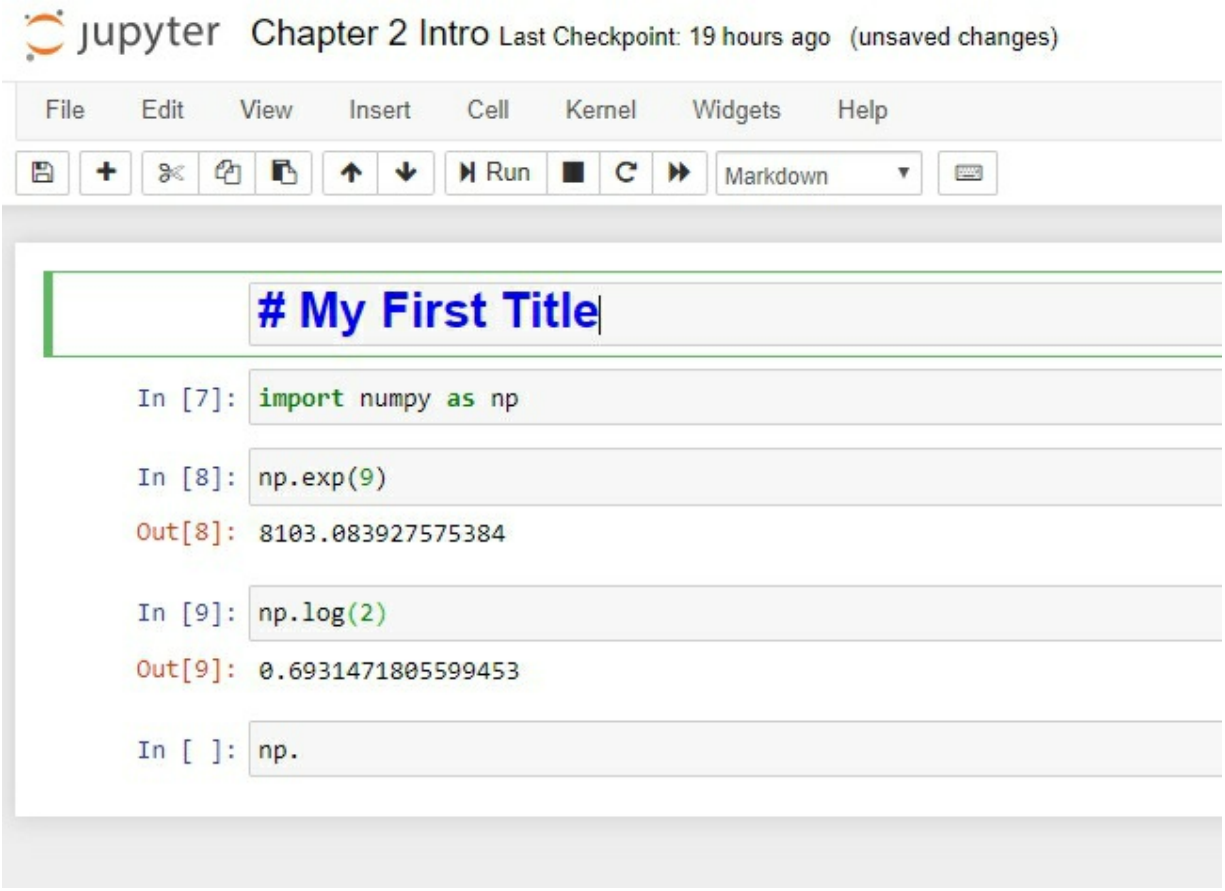
Code Comments, Titles and Text

Within a notebook, titles, comments, and headings may be added to help you organise your code and remind you of what your code does when you come back to it, or to help you convey information to others reading it. To add this

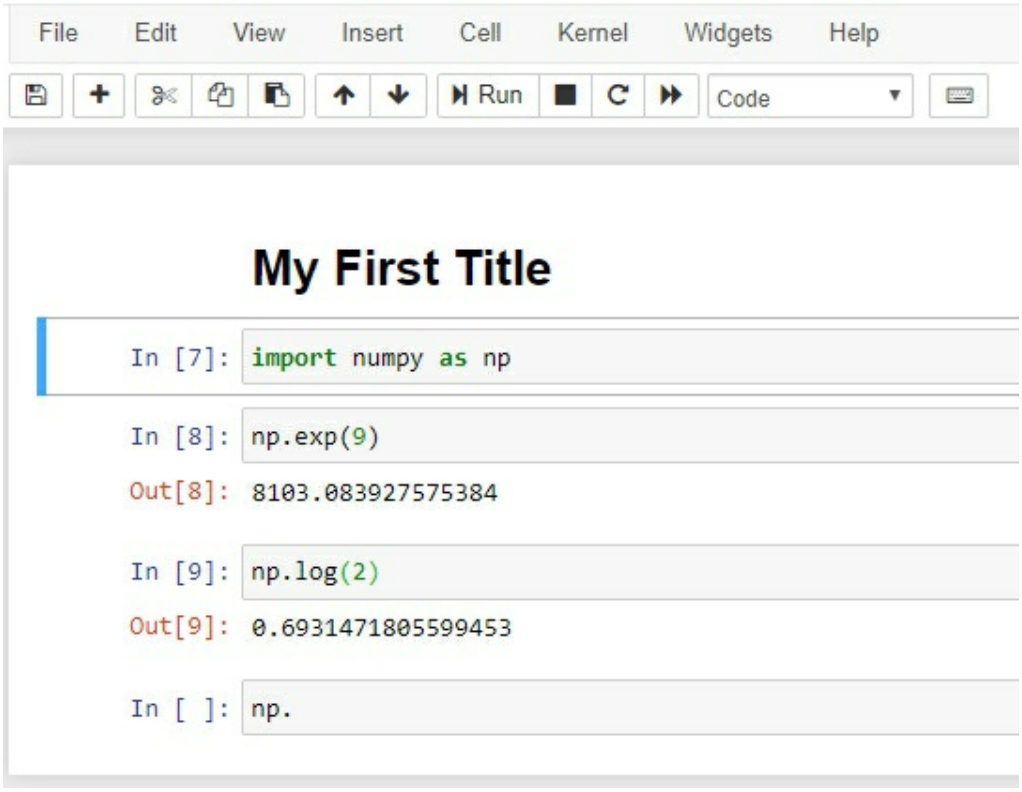
kind of lettering, select an empty cell and in the toolbar, select *Markdown*, which makes that cell a “Markdown” cell. Markdown cells don’t execute code, they just contain text, which you can format as you like.



Inside that Markdown cell type *# My Title*. The beginning hash # in the cell makes the cell a title cell, with the text after it becoming a title, which is quite large font. The text should appear blue to highlight this when typed out. The size of the title text may be made smaller (to be a sub-title) if more than one # is typed in.



Use Shift-Enter, as if executing code, to display the title. The default colour is black, though markdown cells do give you more text formatting options.



Any text lines not beginning with a # in a markdown cell will appear as normal text.

To make a Python code comment (which can be placed in a cell where this is code to be executed) just type # followed by any text you want to be plain text which will not execute.

Code comments can be done on multiple lines using triple quotes, for example:

In[1]:

```
'''
This code comment is between triple quotes.
None of this text will be executed as code.

Often this is used for explanations of functions,
or other bodies of text that are best placed
within the code.
'''


"""
The quotes can be single or double quotes.
They are interchangeable in Python,
however they must match to end the quote.
"""
```

Variable assignment

Variables can be assigned with =. When you assign a value to a variable there is no answer to be displayed below your cell this time around. Instead, the variable is now stored in computer memory and may be used later.

```
In [1]: import numpy as np # Here numpy is imported as np so I can just type

In [2]: x = np.exp(9) # make x a number.
        print('My value of a is: ', x) #The Print function may take more than

        My value of a is:  8103.083927575384

In [3]: y = np.log(x*2) # calculate a value for y
        print('My value for y is:', y, ' After computation with x') # Print f

        My value for y is: 9.693147180559945  After computation with x

In [4]: print(y)

        9.693147180559945

In [5]: print('Final answer of computation is:', y)

        Final answer of computation is: 9.693147180559945

In [ ]:
```

To display the value, the *print* function may be used, as seen before in the *Hello World* program written earlier. The *print* function can display multiple things if you separate the inputs with commas. To delete a variable from memory, type in *del* followed by the variable name.

```
In [4]: print(y)

        9.693147180559945

In [5]: print('Final answer of computation is:', y)

        Final answer of computation is: 9.693147180559945

In [6]: del y

In [7]: print(y)

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-7-d9183e048de3> in <module>
        ----> 1 print(y)

        NameError: name 'y' is not defined
```

Strings

Strings are a data type, just like numbers. Strings are just text which can be assigned to variables. You've been using strings in print functions already. Any text between the double quote " or single quote ' is a string. There are some things you can do with strings, you can concatenate them together with the + operator:

In[1]:

```
x = 'He loves you'
y = ' yeah yeah yeah'
print(x+y)
```

Out[1]:

He loves you yeah yeah yeah

You can have special characters in them like *tab* and *next line* with backslashes:

In[2]:

```
print('\tThis string starts with a tab.')
print('This string \nis split over\na few lines.')
```

Out[2]:

```
	This string starts with a tab.
This string
is split over
a few lines.
```

You can edit sections of strings on the fly with curly braces in place of text, and placing *.format* after the quotes with the text you want in place in brackets:

In[3]:

```
my_age=32
print('My age is: {}'.format( my_age - 2 ) )
```

Out[3]:

My age is: 30

Exercise 1 – Basic Python

Download the Exercise 1 Jupyter Notebook and follow the instructions laid out to get a feel for things. Try typing in some basic arithmetic and executing the code in the cell.

Import Numpy and use some Numpy mathematical functions, assign variables to values, and print some text out. If this is your first time, this is a good place to move fast and break things.

Python Lists

Lists in Python are simply containers that store things in order. They may be created with square brackets [ ]. Within the square brackets each item in the list is separated by a comma. Here is a list of numbers:

In[1]:

```
a = [1,2,3,4,5]
print(a)
```

Out[1]:
[1, 2, 3, 4, 5]

Lists in Python can contain a lot of different types of things. Types that exist include booleans (*True* or *False*, starting with a capital, which can be used in programming logic later), Strings, Integers (1, 2, 400, 321), floating-point numbers (1.23, 3.14, 5.5555), you get the idea. They can also contain never before seen objects that you can create. We will get to user-created objects later.

In[2]:

```
b = [1,2,3,4,6.4, True, False, 'Some Text']
b
```

Out[2]:
[1, 2, 3, 4, 6.4, True, False, 'Some Text']

With Python lists, using the + operator concatenates the lists together, just like for strings.

In[3]:

```
c = a+b
print(c)
```

Out[3]:
[1, 2, 3, 4, 5, 1, 2, 3, 4, 6.4, True, False, 'Some Text']

To access a single item from a list, use the square brackets [ ] immediately after the list variable, where the number within the brackets selects the item number from the list, starting from 0.

In[4]:

```
print('The second item in list c is:', c[1])
```

Out[4]:
The second item in list c is: 2

In[5]:

```
print('The tenth item in list c is:', c[9])
```

Out[5]:
The tenth item in list c is: 6.4

To select a range of items from a list, type in the first and last number of the selection you desire in the square brackets, separated by the colon :. For example, to see the first 10 items in our list *c*:

In[6]:

```
c[0:10]
```

Out[6]:
[1, 2, 3, 4, 5, 1, 2, 3, 4, 6.4]

Items in a list can be changed by referencing the list position and assigning it a new value. For example, here you can change the first item in the list.

In[7]:

```
c[0]=200
print(c)
```

Out[7]:
[200, 2, 3, 4, 5, 1, 2, 3, 4, 6.4, True, False, 'Some Text']

You can get the last item in the list by asking for the "*-1th*" element like so:

In[8]:

```
print('The last item in list c is:', c[-1])
```

Out[8]:
The last item in list c is: Some Text

The length of a list can be found by passing the list into the *len()* function.

In[9]:

```
my_length = len(c)
print('The length of my list is: ', my_length)
```

Out[9]:
The length of my list is: 13

Items can be removed from a list with the *.remove()* function. Here we have a list of stock tickers as strings, and we would like to remove one of them.

In[10]:

```
# Stocks list
stocks = ['TSLA', 'BRKA', 'RDSA', 'AMD', 'MMM']
print('Current stocks list: ', stocks)

# 'TSLA' is removed
stocks.remove('TSLA')

# Updated stocks list
print('Updated stocks list: ', stocks)
```

Out[10]:
Current stocks list:['TSLA', 'BRKA', 'RDSA', 'AMD', 'MMM']
Updated stocks list:['BRKA', 'RDSA', 'AMD', 'MMM']

And items can be added to a list with the *.append()* function.

In[11]:

```
#Add a new stock to end of list
stocks.append('SAVE')
print('Updated stocks list: ', stocks)
```

Out[11]:
Updated stocks list:['BRKA', 'RDSA', 'AMD', 'MMM', 'SAVE']

Tuples

A Tuple is defined as a list except that it is defined with normal brackets ( ) instead of square brackets [ ]. The difference is that tuples are immutable, you are not allowed to change items within them once they are defined. Use Tuples when you don’t want anyone changing your values.

```
my_tuple = (4, 5, 6, 7)
print('My tuple: ', my_tuple)
```

```
My tuple:  (4, 5, 6, 7)
```

```
my_tuple[0] = 5 # Won't work, Tuples are immutable.
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most re
<ipython-input-31-d49a0a200197> in <module>
----> 1 my_tuple[0] = 5 # Won't work, Tuples are immutable.

TypeError: 'tuple' object does not support item assignment
```

Sets

A *Set* is like a *List* except that it comes with curly braces { } instead of square brackets [ ]. It can be created by calling the function *set()* and passing a list as the argument. If you pass values into curly brackets directly you create a dictionary instead (more on that later). As in set theory, which you may remember in maths class, a set in Python contains a list of all unique values only.

In[1]:

```
print('My list: ',c) # take a look at the list
```

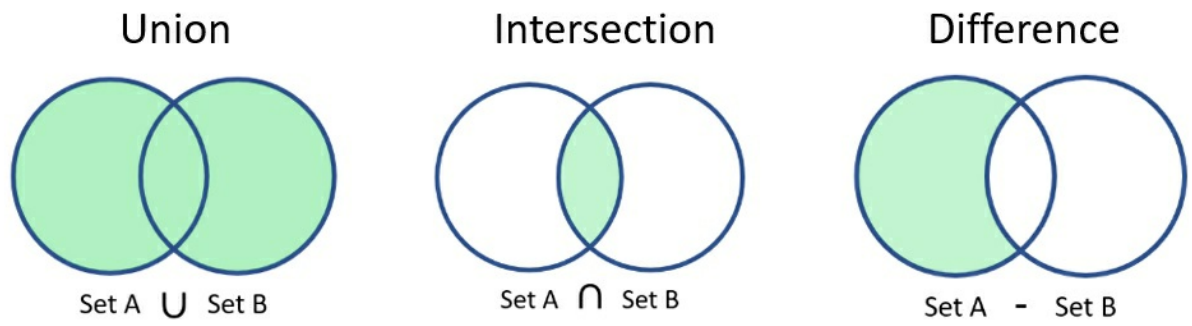
Out[1]:
My list: [200, 2, 3, 4, 5, 1, 2, 3, 4, 6.4, True, False, 'Some Text']

In[2]:

```
print('My set: ', set(c)) # make a set based on the list.
#Now only unique items appear
```

Out[2]:
My set: {False, 1, 2, 3, 4, 5, 6.4, 200, 'Some Text'}

The set theory operations are possible between sets in Python. We can try some of the operations that are graphically represented here:



If you haven't done these set theory operations before they should be obvious after using them in Python. The operations are called by typing a set variable and then adding the set operation we want to perform as a function, e.g. *A.union()*. For set *A*.

In[3]:

```
#Set A
A = {'Cow', 'Cat', 'Dog', 'Rabbit', 'Sheep'}

#Set B
B = {'Cat', 'Dog', 'Squirrel', 'Fox'}
#Calculating the Union of two sets
U = A.union(B)
print('The union of set A and set B is: ', U)
```

Out[3]:
The union of set A and set B is: {'Fox', 'Rabbit', 'Dog', 'Cat', 'Squirrel', 'Sheep', 'Cow'}

In[4]:

```
#Calculating the Intersection of two sets
n = A.intersection(B)
print('The intersection of set A and set B is: ', n)
```

Out[4]:
The intersection of set A and set B is: {'Dog', 'Cat'}

In[5]:

```
#Calculating the difference of two sets
d = A.difference(B)
print('The difference of set A and set B is: ', d)
```

Out[5]:
The difference of set A and set B is: {'Cow', 'Rabbit', 'Sheep'}

Sets can have items added or removed from them with *.add()* and *.remove()*, and you can turn sets into lists with *list()*.

Exercise 2 – Python Lists and Sets

If you've never coded before this may seem daunting. Don't worry, after writing real code to do something it'll fall into place, and not much needs to be memorised in reality. Download the Exercise 2 Jupyter Notebook and follow the steps to practice creating and manipulating lists and sets.

Custom Functions

So far only functions that are part of Python have been used. Often it is useful to make your own functions when you have operations that you carry out often and don't want to repeatedly write out. To create a function in python type in *def*, followed by a space, then the name of your new function, followed by brackets () and a semi-colon :. Any operations that are part of the function have to be indented by a tab and placed directly below where you defined the function:

In[1]:

```
# Here is a simple function. All it does is say hello.
def sayHello():
    print("Hello!")

#let's call the function.
sayHello()
```

Out[1]:
Hello!

If you want to pass an argument to your function (just like the exponential and logarithmic functions used earlier), add a variable between the brackets. You can pass as many variables to your function as you like. To do this, add them all between the brackets and separate them by commas. For the function to return something to us, we have to end the function with *return* followed by a value you want to be returned, which you may have calculated in your function code.

In[2]:

```
# You can pass values to functions (called arguments)
def functionWithArg(arg_1):
    print(arg_1)

# Let's call the function.
functionWithArg('Print stuff.')
```

Out[2]:
Print stuff.

In[3]:

```
# You can pass many arguments into your function,
# here 3 arguments are used.
def addThree(first, second, third):
    answer = first + second + third
    print('The sum result is:', answer)

    return answer

mySum = addThree(9,200,12)
```

Out[3]:
The sum result is: 221

In[4]:

```
print(mySum)
```

Out[4]:
221

It is a good idea to leave comments in the function, so you don't need to remember what your function does that you may have written awhile ago.

In[5]:

```
def multiplyThree(first, second, third):
    '''
    It is good practice to put code comments here
    To explain what the function does, for anyone who
    may use it in future.
    '''
    answer = first * second * third
    print('The multiplicative result is:', answer)

multiplyThree(4,3,2)
```

Out[5]:
The multiplicative result is: 24

Functions can accept many kinds of objects as arguments at the same time.

In[6]:

```
# You can also pass many different kinds of objects
# as arguments to functions
# Here we pass a list and a number into an argument.

def retFirstNStocks(stockList, n):
    return stockList[:n] # returns all items up to index n
```

```
friends_stock_list = ['F', 'AMD', 'GAW', 'TM17', 'DISH', 'INTC']
# Now we assign a new variable to what the function returns.
reduced_list = retFirstNStocks(friends_stock_list, 3)

#lets see what the returned variable is:
print('The object our function returned to us is:', reduced_list)
print('\n This object is a:', type(reduced_list))
```

Out[6]:
The object our function returned to us is: ['F', 'AMD', 'GAW']

This object is a: <class 'list'>

A function can return any kind of object you want, for instance, you can make it return a list, so it can return several things at once through the list if desired.

In[7]:

```
# A function can return many things in the 'return' line with a list.
def getDataFromList(stockList):
    '''
    This returns a list of:
    First 3 items in list
    first item
    last item
    number of items
    '''
    size = len(friends_stock_list)

    # remember, indexing starts at 0
    print('The first item is:', stockList[0])
    print('The last item is:', stockList[size-1])
    print('The number of items is:', size)

    return [stockList[:3], stockList[0], stockList[size-1], size]
```

Out[7]:
The first item is: F
The last item is: INTC
The number of items is: 6

[['F', 'AMD', 'GAW'], 'F', 'INTC', 6]

Arguments to functions can be given a default value when you create them, which may be useful if there are arguments you don’t want to specify often but which you want to leave open to change.

In[8]:

```
# Making the argument optional
def printMyNumber(num='I would like to have an argument please.'):
```

```
    print('My number is:', num)

printMyNumber(3.141) # With an argument
printMyNumber() # Without an argument, reverting to your default
```

Out[8]:
My number is: 3.141
My number is: I would like to have an argument please.

Logic and Loops

Basic Loops

If you want to run an operation repeatedly, you can use loops. A *for* loop will perform a loop for each value in a series you provide. For example, you can perform a loop for each item in a list. To create a *for* loop that loops over a list, type in *for X in List* where *X* is a new variable you want to use to refer to an individual item in the list, and the *List* is any Python list.

In[1]:

```
friends_stock_list = ['F', 'AMD', 'GAW', 'TM17', 'DISH', 'INTC']

#Here is a loop over friends_stock_list
for item in friends_stock_list:
    print(item)
```

Out[1]:
F
AMD
GAW
TM17
DISH
INTC

You can also loop over an increasing or decreasing range, for example a range going from 0 to 3 in a for loop can be created with *For i in range(0, 3):*.

In[2]:

```
#Here is a loop over a range beginning at 0 and ending at 3
for i in range(0,4):
    print(i, 'spam')
```

Out[2]:
0 spam
1 spam
2 spam
3 spam

Note that the range starts at 0 and ends at one before the arguments second number.

Often you can fill a list by using a loop with a single line with what Python calls list comprehensions as follows:

In[3]:

```
cube_numbers = [x**3 for x in range(10)]
print(cube_numbers)
```

Out[3]:
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

Booleans

A bool is an object which is either True or False. This is useful to us in creating logic.

In[1]:

```
# boolean
num = 99 # here we have a variable
num == 4 # the == operator will return a boolean here,
         # which is either True of False.
```

Out[1]:
False

In[2]:

```
# let's see what it returns when we as if it is 99.
num == 99
```

Out[2]:
True

booleans are also returned when using greater than (>) and less than (<) operators:

In[3]:

```
# Greater than and less than operators are also available to us.

print('Is num greater than 100?', (num > 100) )#(num>100)returns a bool
print('Is num Less than 100?', (num < 100) )#(num<100)returns a bool
```

Out[3]:
Is num greater than 100? False
Is num Less than 100? True

We can combine this logic with AND (&) and OR(|) operators to do some

boolean logic:

In[4]:

```
num2=100 # create a second number

bool1 = (num < 100)
print('boolean 1 is:',bool1)

bool2 = (num2 < 100)
print('boolean 2 is:',bool2)

bool3 = bool1 | bool2
print('\nboolean 3 (which is bool1 or bool2 being True) is:\n',bool3)
```

Out[4]:

```
boolean 1 is: True
boolean 2 is: False

boolean 3 (which is bool1 or bool2 being True) is:
True
```

Here is a table of commonly used operators that will return booleans.

Basic Operators	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&	And
|	Or

Logic in Loops

We can put logical operators in loops to do more complex operations. As a first example, lets use one of the boolean operators in a simple loop to see if a number in a list is greater than 6, printing the variables out so we can see

what is going on.

In[1]:

```
for i in [0,1,2,3,4,5,6,7,8,9,12,17,200]:
    boolIsGreaterThanSix = (i >= 6)
    print(i, boolIsGreaterThanSix)
```

Out[1]:

```
0 False
1 False
2 False
3 False
4 False
5 False
6 True
7 True
8 True
9 True
12 True
17 True
200 True
```

There are more complex operators available, for example the modulo operator returns the remainder of a division between the two numbers. From this remainder, we can use a boolean == operator to see if the remainder after division is equal to 0. The remainder from the modulo operator is equal to 0 then the number must be a multiple of 3.

In[2]:

```
#Here is a loop over a range, where a logic IF statement is present
# to only print numbers that are a multiple of 3
for i in range(0, 30):

    # The % is the modulo function,
    # it returns the remainder of a division.
    my_rem = (i % 3)

    if (my_rem == 0):
        print(i)
```

Out[2]:

```
0
3
6
9
12
15
18
21
```

24
27

We can combine logic with loops to output only numbers greater than a certain number (with the > operator) whilst being a multiple of 3:

In[3]:

```
#Multiple of 3 greater than 20
for i in range(0,40):

    # Firstly a Boolean,
    # is True if i is greater than 20.
    greaterThan20 = (i > 20)

    # Secondly a Number,
    # is the remainder of division.
    myRemainder = (i % 3)


    # Thirdly another Bool,
    # is True if i is multiple of 3.
    multOf3 = (myRemainder == 0)

    # If a multiple of 3 AND greater than 20.
    if (multOf3 & greaterThan20):
        print(i)
```

Out[3]:

21
24
27
30
33
36
39

We can separate bits of code out to custom functions. This is useful to do to remove clutter and keep code simple and easy to understand when looking at program flow.

In[4]:

```
def mult3AndGr20(number):
    '''
    Returns True if a multiple of 3 greater than 20.
    Otherwise returns False.
    '''
    # Firstly a Boolean,
    # is True if i is greater than 20.
    greaterThan20 = (i > 20)
```

```
    # Secondly a Number,
    # is the remainder of division.
    myRemainder = (i % 3)


    # Thirdly another Bool,
    # is True if i is multiple of 3.
    multOf3 = (myRemainder == 0)

    # If a multiple of 3 AND greater than 20.
    if (multOf3 & greaterThan20):
        return True
    else:
        return False


#Multiple of 3 greater than 20
for i in range(0,40):
    if(mult3AndGr20(i)):
        print(i)
```

Out[4]:

```
21
24
27
30
33
36
39
```

Exercise 3 – Logic and Loops

Download and try out the Exercise 3 Jupyter Notebook. You can imagine where these logic and looping tools can be used in useful ways. This exercise ends with a use for them in filtering a list of stocks for certain features.

Python Dictionaries

In Exercise 3, stock information was given to you as a group of independent lists. A better way to store information, where for example you may want each stock to have its own space, is with Python Dictionaries. We will not be using dictionaries much to build our AI Investor, however, they are a fundamental Python type for storing information, a bit like lists, that a user should be acquainted with.

In[1]:

```
# A python Dictionary is a kind of lookup table. It can store a series of any kind of object which is accessed with a 'key'.
```

```
dict1={} # This is how to create an empty dictionary.
# Here we fill our Python dictionary which stores data about one person.
dict1 = {
    'Name': 'Max',
    'Height': 1.9,
    'Test Scores':[50, 62, 78, 47],
    'Nationality':'German'
}
print(dict1)
```

Out[1]:
{'Name': 'Max', 'Height': 1.9, 'Test Scores': [50, 62, 78, 47], 'Nationality': 'German'}

Just like real dictionaries (you know, the ones printed on paper), there is a word that we may look up on the left, and on the right there is some information about it. Since you make your own dictionaries, you can put whatever you want on the right. The word on the left is the *key*, which you can use to look up information. It is created in a similar way to a set, with curly braces { }, but where each item is in fact two items, the key and the information you want to store under that *key*.

In[2]:

```
# The first item is the 'Key', the item after the colon :
# is the information to be accessed with the key.
dict1['Name'] # View the info behind item 'Name'
```

Out[2]:
'Max'

In[3]:

```
# You can view all the keys in your dictionary with the .keys() function
dict1.keys()
```

Out[3]:
dict_keys(['Name', 'Height', 'Test Scores', 'Nationality'])

In[4]:

```
# You can add items to Dictionaries by stating them:
dict1['Shoe Size'] = 7
print(dict1)
```

Out[4]:
{'Name': 'Max', 'Height': 1.9, 'Test Scores': [50, 62, 78, 47], 'Nationality': 'German', 'Shoe Size': 7}

In[5]:

```
# You can delete items with the del command
del dict1['Height']
print(dict1)
```

Out[5]:
{'Name': 'Max', 'Test Scores': [50, 62, 78, 47], 'Nationality': 'German', 'Shoe Size': 7}

Dictionaries give you a lot of flexibility in information storage. You can store whatever you want under each key, even another dictionary.

Exercise 4 – Python Dictionaries

Download and try out the Exercise 4 Jupyter Notebook where we will use dictionaries for storing and retrieving stock information. We don't be using them much so you can skip this bit if you want to get to building things faster.

NumPy

If you did the earlier exercises you will have imported Numpy and used it in a limited fashion. So what is Numpy? (pronounced Num-pie)

Numpy is a package for scientific computing in Python. A very large proportion of Python users use it (this explains why the module is included out-of-the-box). The bulk of its usefulness is in the array object that it provides, and the range of things it lets you do to arrays, although of course Numpy is useful for its simpler mathematical functions like exponential and logarithm too. We will go over the basics needed to understand the code here, full documentation is available on the official website *numpy.org*.

Arrays are essentially lists, except we might want a multi-dimensional one, and a way to perform matrix algebra on them, and for them to be really fast in case someone wants one to contain a million numbers to do a really hard sum. We'll use an array in the next exercise to demonstrate an applicable use in investment calculation.

In[1]:

```
import numpy as np # Import Numpy
# The library has mathematical functions like in the math library,
# for example:
print(np.log(9)) # Logarithm
print(np.exp(9)) # Exponential
print(np.sin(9)) # Sine
print(np.cos(9)) # Cosine
```

Out[1]:
2.1972245773362196
8103.083927575384

0.4121184852417566
-0.9111302618846769

Numpy arrays can be created in a variety of ways, and we can do things with them in a similar manner to Python lists. Obvious Numpy array functions such as *max*, *min* etc. can be called by stating the array and calling the function, like *.min()*.

In[2]:

```
# However the main use of Numpy is to do computing with matricies.
# To make a 1 dimensional array use np.array and pass a list to it:
# (It is like a list except all items in the array ARE ALWAYS
# the same type. e.g. strings, integers, floating point numbers)
a = np.array([1, 2, 3, 4, 5])
print(a)

# Can also make an array with range().
a = np.array(range(5,10))
print('\nAn array made with range() instead:', a)

# For evenly spaced numbers over a specified interval use linspace
a = np.linspace(1.2, 3.8, num=5)
print('\nAn array made with linspace() instead:', a)

# To see the type of object this is:
print('\nnumpy array type is: ', type(a))

# As with lists you can see individual items like so:
print('\nThe first item is:', a[0])
print('The second item is:', a[1])
print('The third item is:', a[2])

# You can change items in the array the same was as with lists:
a[0] = 9 # change first number to be 9.
print('\nThe first item is now:', a[0])

# As with lists, you can select a range of values with numpy:
a[0:3] = 99 # change the first 3 items to be 999
print('\nThe array is now:', a)

# we can find max, min, mean etc.
print('\nThe max of our array is:', a.max())
print('The min of our array is:', a.min())
print('The mean of our array is:', a.mean())
```

Out[2]:

[1 2 3 4 5]

An array made with range() instead: [5 6 7 8 9]

An array made with linspace() instead: [1.2 1.85 2.5 3.15 3.8]

numpy array type is: <class 'numpy.ndarray'>

The first item is: 1.2
The second item is: 1.8499999999999999
The third item is: 2.5

The first item is now: 9.0

The array is now: [99. 99. 99.   3.15 3.8]

The max of our array is: 99.0
The min of our array is: 3.1499999999999995
The mean of our array is: 60.79

Numpy arrays can be multidimensional, as you probably expected, being a scientific programming library. There are functions to reshape any matrix, transpose it, or return properties of the matrix:

In[3]:

```
# Numpy arrays can have more than one dimension,
# for example the following is a 2D matrix,
# created with a list of lists.
m = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(m)

# You can extract the size in any dimension from
# this array with .shape (Which returns a tuple)
print('\nThe matrix dimensions are:', m.shape)

# With multidimensional arrays you can see and
# change items within them with indexing, but you
# need as many indexes as there are dimensions to
# specify the item location.
# Remember indexing begins at 0!
print('\nItem in first row, third column is: ', m[0,2])
print('Item in second row, fifth column is: ', m[0,4])

# Arrays can be manipulated in various ways, for instance rehape:
rsm = m.reshape(5,2) # Change the shape to be 5 rows and 2 columns
print('\nReshaped matrix is: \n', rsm)

tm = m.transpose()
print('\nTransposed matrix is: \n', tm)

into_line = m.ravel()
print('\nUnraveled matrix is: \n', into_line)
```

Out[3]:

```
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]
```

The matrix dimensions are: (2, 5)

Item in first row, third column is: 3
Item in second row, fifth column is: 5

Reshaped matrix is:
[[1 2]
[3 4]
[5 6]
[7 8]
[9 10]]

Transposed matrix is:
[[1 6]
[2 7]
[3 8]
[4 9]
[5 10]]

Unraveled matrix is:
[1 2 3 4 5 6 7 8 9 10]

Numpy arrays can even have a large number of dimensions, you can make an 8-dimensional matrix if you really need that kind of thing. When creating an array, you can fill it with zeros (with *np.zeros*) or ones (*np.ones*) as well.

In[4]:

```
# You can also create numpy arrays with zeros, ones in them etc.
a_z = np.zeros((3,4)) # Create an array with just zeros in it, 3x4
print('\n', a_z)

a_o = np.ones((10,8)) # Create an array with just ones in it
print('\n', a_o)

a_pi = np.full((5,5), 3.14) # Create an array filled with one number
print('\n', a_pi)

# Remember these arrays are multidimensional,
# you can make a 3D array if you really need it...
d_mat = np.ones((3,3,3))
print('\n Here is a 3D matrix: \n', d_mat)
```

Out[4]:
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]

[[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]

```
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]]

[[3.14 3.14 3.14 3.14 3.14]
[3.14 3.14 3.14 3.14 3.14]
[3.14 3.14 3.14 3.14 3.14]
[3.14 3.14 3.14 3.14 3.14]
[3.14 3.14 3.14 3.14 3.14]]

Here is a 3D matrix:
[[[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]

 [[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]

 [[1. 1. 1.]
  [1. 1. 1.]
  [1. 1. 1.]]]
```

Indexing Numpy arrays is the same as for lists, and can even be done in multiple dimensions. Here is an example where we redefine the value of an entire section of a 2D array:

In[5]:

```
# Array indexing works just the same with 2D matrices,
# for instance with our array of ones:
a_o[2:6,2:]=9.87
print('Our large matrix of ones is now: \n', a_o)
```

Out[5]:

```
Our large matrix of ones is now:
[[1.   1.   1.   1.   1.   1.   1.   1.  ]
[1.   1.   1.   1.   1.   1.   1.   1.  ]
[1.   1.   9.87 9.87 9.87 9.87 9.87 9.87]
[1.   1.   9.87 9.87 9.87 9.87 9.87 9.87]
[1.   1.   9.87 9.87 9.87 9.87 9.87 9.87]
[1.   1.   9.87 9.87 9.87 9.87 9.87 9.87]
[1.   1.   1.   1.   1.   1.   1.   1.  ]
[1.   1.   1.   1.   1.   1.   1.   1.  ]
[1.   1.   1.   1.   1.   1.   1.   1.  ]
[1.   1.   1.   1.   1.   1.   1.   1.  ]]
```

The Numpy math functions shown earlier can be performed on a whole array just by passing that array as the argument to the function. Arrays may also have arithmetic done on them with other arrays of the same dimension, just like matrix mathematics.

In[6]:

```
# Arrays can be added to each other, divided by each other etc.
arr_1 = np.array([[1,2,3], [4,5,6], [7,8,9]])
arr_2 = np.array([[7,6,4], [12,5,1], [2,2,249]])

print('Array 1:\n', arr_1)
print('\nArray 2:\n', arr_2)

# Do some calculations on arrays
arr_3 = arr_2 + np.sin(arr_1) - arr_1/arr_2
print('\nArray 3:\n', arr_3)

# Remember arrays need to be the appropriate size.
```

Out[6]:

```
Array 1:
[[1 2 3]
[4 5 6]
[7 8 9]]

Array 2:
[[  7   6   4]
[ 12   5   1]
[  2   2 249]]

Array 3:
[[  7.69861384   6.57596409   3.39112001]
[ 10.90986417   3.04107573  -5.2794155 ]
[ -0.8430134   -1.01064175 249.37597391]]
```

Some concepts learned in the Logic and Loops exercise can be used on Numpy arrays.

In[7]:

```
# It is often useful to iterate over an array,
# lets make an array and iterate over it.
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
for x in a:
    print(x**2 + x)
```

Out[7]:

```
2
6
12
20
```

30
42
56
72

Data in arrays can be plotted simply with the *Matplotlib* library, which is an excellent library you can use to make graphs with your data in Python (Their official user guide web page is https://matplotlib.org/stable/users/index.html). Import this library for use just like with Numpy with:

In[8]:

```
#Import a plotting library for Python to plot some numpy arrays
from matplotlib import pyplot as plt
```

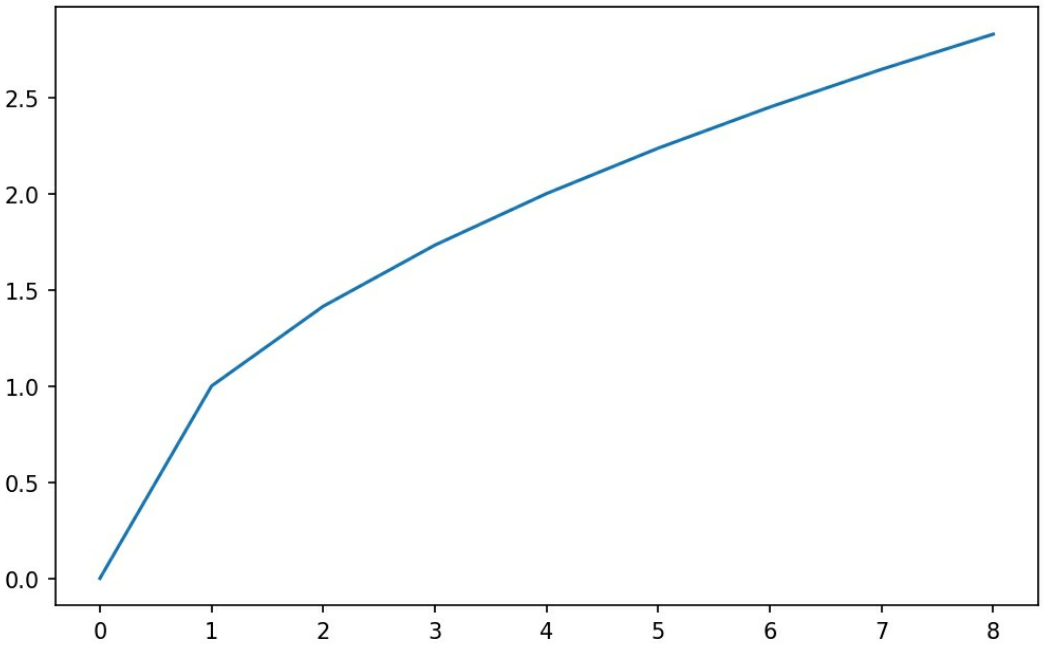
Here is an example creating an array of ascending numbers using *range()*, using Numpy to calculate the square root of all values, and plotting these in a graph. When plotting a graph with MatPlotLib, if you imported it as *plt* you would type in *plt.plot(x,y)* where *x* and *y* are the *x* and *y* values. Matplotlib is a large library with a multitude of functions you can use to make many kinds of graphs, with many formatting options, check them out at *https://matplotlib.org/*. Here is the default kind of plot that you can use, plotting a graph of the square root of *x* calculated using Numpy.

In[9]:

```
x = np.array(range(0,9))
y = np.sqrt(x) # run a numpy function on the entire array.
plt.plot(x,y)
print('\nOut array y has values: ', y)
```

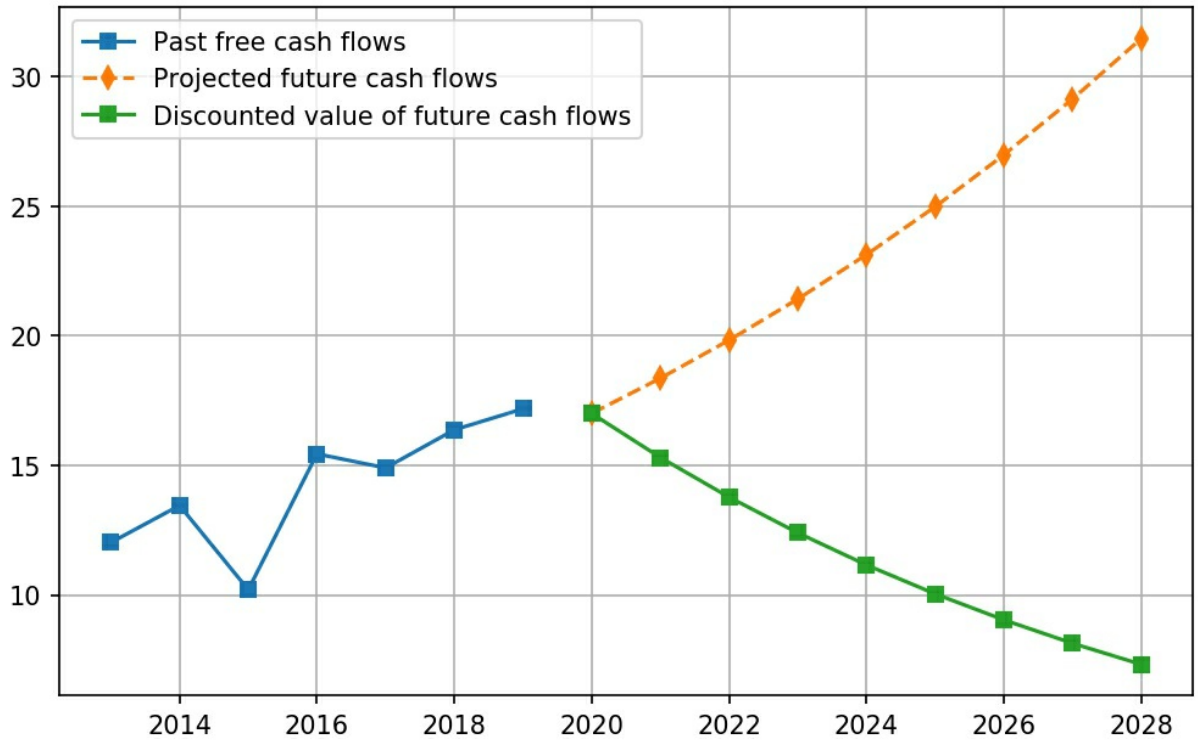
Out[9]:

Out array y has values: [0. 1. 1.41421356 1.73205081 2. 2.23606798 2.44948974 2.64575131 2.82842712]



Exercise 5 – Numpy Basics

Download the Exercise 5 Jupyter Notebook and follow the steps to use the tools we now have to do a discounted cash flow analysis of a company. Here is a graph from the exercise to whet your appetite.



Discounted Cash Flows

In case you've never come across a discounted cash flow calculation before, it is quite simple and is crucial to understanding what the value of any enterprise or investment is (even if you're just trading rocks).

Once you plot out a discounted cash flow diagram, the reasoning behind it is easier to understand, as you have already computed the equations with code. This is a good spot to explain the reasoning behind it, now that you know how to do a few things with matrices/lists, bear in mind it is always a theoretical exercise in predicting the future.

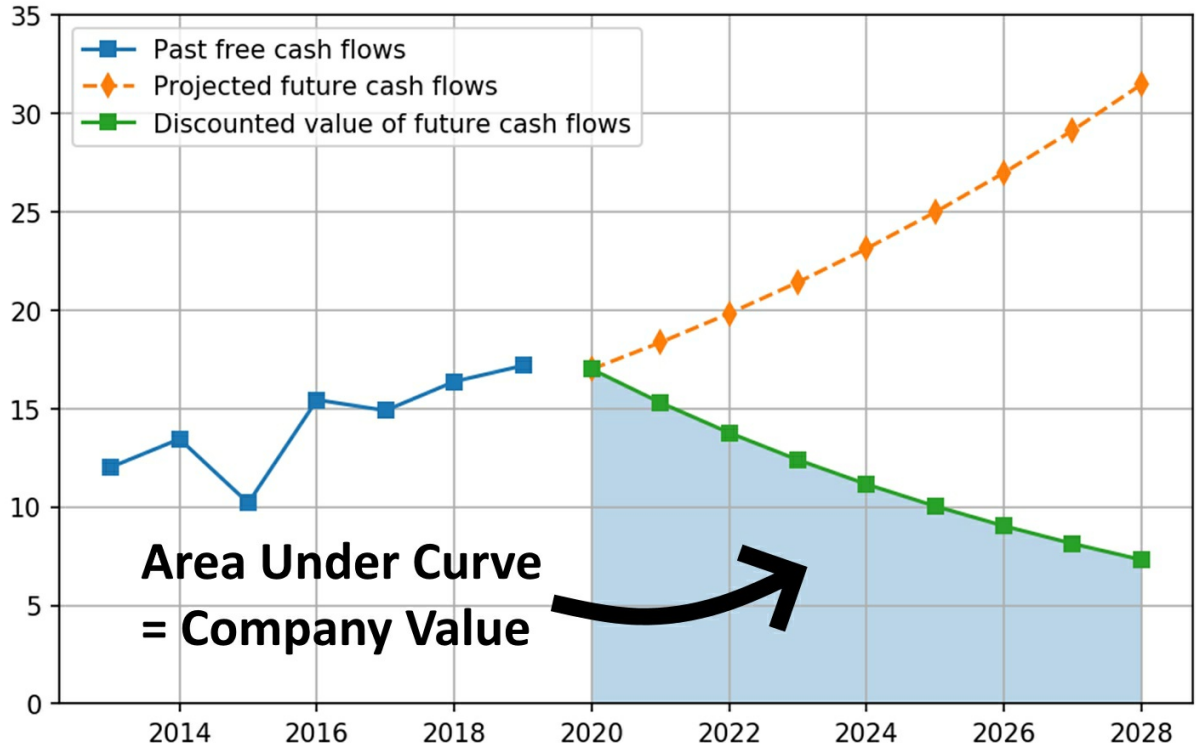
The value of any asset is related to the amount of cash it gets you in the future. If it were a house, the value of the rent being paid over time would be added up, if it were a something you saw on eBay that you wanted to flip, it would be the amount you think you would sell it for in future.

However the value of the investment is not the absolute value of this cash, obviously in the example of a house, the value might be infinite if this were the case (or the amount a buyer would get until they die). The value of money now is higher than the value of money in the future, because you can leave

the money you obtained now in a bank earning some interest. Your $100 might become $102 in a year, which is certainly greater than a $100 payment in that future. If you calculate this forward, your $500 rent income in 10 years might only be worth $300 to you now in the present.

Another complication is that future payments aren't certain. You may think you can get $200 for that collector's item you bought on eBay for $50, but there's a chance that it won't be a collector's item any more, or your dog might destroy it before it gets sold.

To accommodate these effects the value of the future cash flows is discounted. Every year the value of the payments (or single payment) is lessened a little bit more than the previous year. All up all the cash flows, which is basically the area under the curve, and you get the value of your asset.



Ultimately the value of something is what someone else is willing to pay you for it (but they are likely to pay the price a well-researched discounted cash flow calculation will provide). A good book demonstrating this technique as the main way to pick stocks is *The Five Rules For Successful Stock Picking* by Pat Dorsey.

The Pandas Library

So far, the data in this guide has been confined to information created within Jupyter Notebooks. To read and write data to your operating system and to analyse and manipulate data, we will be using the Pandas library. As a sneak peek at the investing AI creation chapter, we will learn how to use Pandas by reading in and manipulating/analysing real world stock data from the internet. Pandas stands for *Python Data Analysis Library*, where the words *Panel Data* are incorporated into the shortened name somehow. Their official documentation is here: (https://pandas.pydata.org/pandas-docs/stable/reference/index.html).

As with Numpy, you will need to import the module to start using its functions and objects. For Pandas this is commonly done with *import pandas as pd*, so we will use *pd.* when we want to call a Pandas function.

In Pandas, data is stored in table form in an object called a DataFrame. DataFrames are what Pandas are all about. DataFrames are just tables, though you can store a lot in them and do a lot with them. A DataFrame can be instantiated by passing a dictionary with data into the function *pd.DataFrame()*. The format for the dictionary has to have its keys as the DataFrame table headings, and a list of items under each key will be the data in the table:

In[1]:

```
#Import Pandas
import pandas as pd
#Creating a dictionary with some table data in it.
dict = {
    'Name':['Tom', 'Dick', 'Harry'],
    'Age':[24, 35, 29],
    'Shoe Size':[7,8,9]
}
#Creating a DataFrame with the data from the dictionary
df=pd.DataFrame(dict)
#The DataFrame can be viewed without the print() command.
df
```

Out[1]:

	Name	Age	Shoe Size
0	Tom	24	7
1	Dick	35	8
2	Harry	29	9

DataFrame data can be written to, and read from, files with the *DataFrame.to_csv('filename.csv')* and the *pd.read_csv('filename.csv')* functions. The file format *.csv* just means comma-separated value files, table data is stored there only, as opposed to spreadsheet files that can contain graphs and other data.

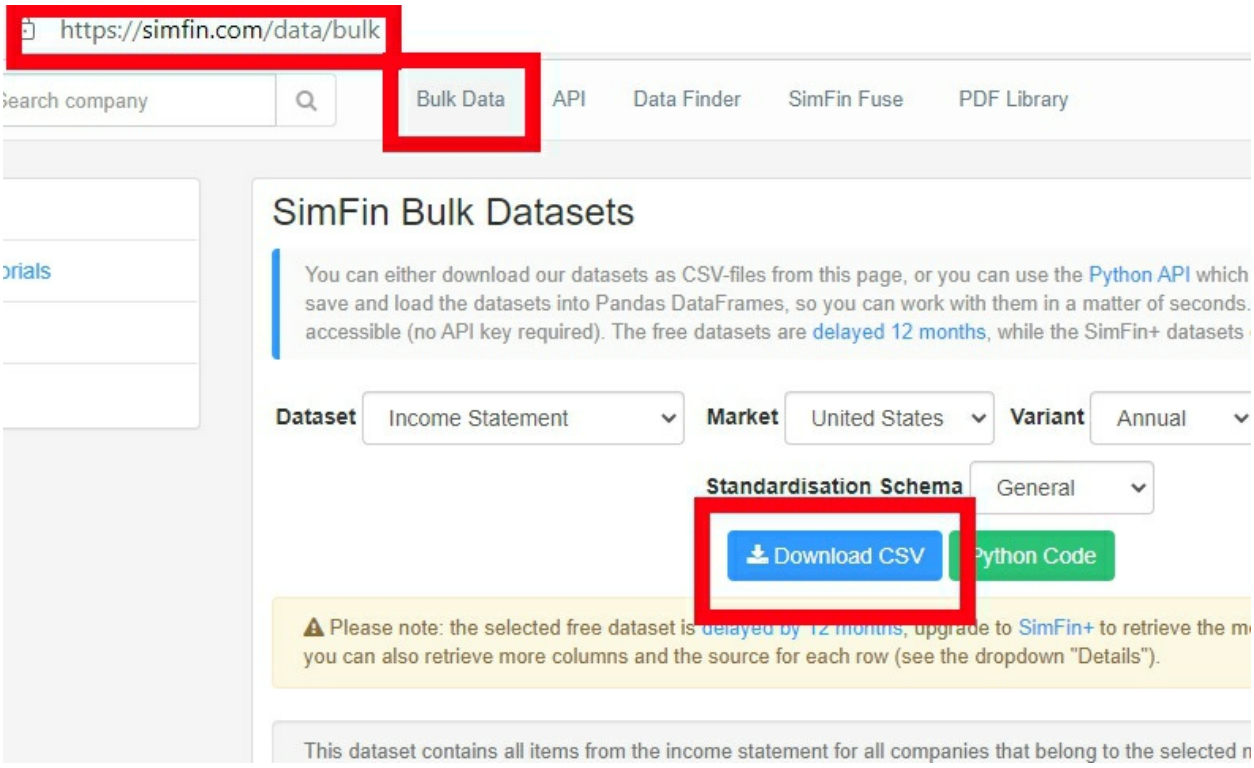
In[2]:

```
# Write the data from the DataFrame to a file
df.to_csv('people.csv')

# Now I want to read the data, lets create a new DataFrame. Remember to specify an index column!
my_data=pd.read_csv('people.csv', index_col=0)

# Have a look at the data
my_data
```

Out[2]:

	Name	Age	Shoe Size
0	Tom	24	7
1	Dick	35	8
2	Harry	29	9

So far there hasn't been much useful data to us in these tutorials. Let's change that and explore the things we can do with Pandas DataFrames. Bulk US and German stock data can be downloaded for free from the great people at *SimFin.com* (*https://simfin.com/data/bulk*). Download the Annual Income Statement data for US companies as a *.csv* file:



This file is roughly 3MB large. You can choose to store it wherever convenient, however, the directory location is needed in your code for pandas to know where to retrieve the file data with the *read_csv()* function.

It is worth looking at how the data is stored in any file you download first to see if there are any quirks. Spreadsheet software can open .csv files, though if the file is saved when open in spreadsheet software, it might mess with the file.

For the SimFin data, all values are separated with a semicolon ; instead of a comma, so to read the data correctly with Pandas the *read_csv()* function will require another argument to help it know how to parse the data. This argument is *delimiter=';'* after specifying the file name:

In[3]:

```
# Lets read some income statement data from SimFin (https://simfin.com/)
# Your download directory will differ.

Income_Data=pd.read_csv(\
'C:/Users/G50/Stock_Data/SimFin/us-income-annual/us-income-annual.csv',\
                        delimiter=';')
```

```
In [90]: # get the DataFrame shape (Might be useful for other computation)
         print('DataFrame shape is: ',Income_Data.shape)
         #Have a look at what our DataFrame looks like
         Income_Data

         DataFrame shape is:  (15125, 27)
```

Out[90]:

	Ticker	SimFinId	Currency	Fiscal Year	Fiscal Period	Report Date	Publish Date	Shares (Basic)	Shares (Diluted)	Revenue	...	Non-Operating Income (Loss)	Interest Expense, Net	Pretax Income (Loss), Adj.	
0	A	45846	USD	2008	FY	2008-10-31	2009-10-05	3.630000e+08	3.710000e+08	5.774000e+09	...	20000000.0	-10000000.0	815000000	
1	A	45846	USD	2009	FY	2009-10-31	2009-12-21	3.460000e+08	3.460000e+08	4.481000e+09	...	-40000000.0	-59000000.0	7000000	
2	A	45846	USD	2010	FY	2010-10-31	2010-12-20	3.470000e+08	3.530000e+08	5.444000e+09	...	-6000000.0	-76000000.0	560000000	1
3	A	45846	USD	2011	FY	2011-10-31	2011-12-16	3.470000e+08	3.550000e+08	6.615000e+09	...	-39000000.0	-72000000.0	1032000000	
4	A	45846	USD	2012	FY	2012-10-31	2012-12-20	3.480000e+08	3.530000e+08	6.858000e+09	...	-76000000.0	-92000000.0	1043000000	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
15120	low	186050	USD	2013	FY	2014-02-28	2014-03-31	1.059000e+09	1.061000e+09	5.341700e+10	...	-476000000.0	-476000000.0	3673000000	
15121	low	186050	USD	2014	FY	2015-02-28	2015-03-31	9.880000e+08	9.900000e+08	5.622300e+10	...	-516000000.0	-516000000.0	4276000000	
15122	low	186050	USD	2015	FY	2016-02-29	2016-03-29	9.270000e+08	9.290000e+08	5.907400e+10	...	-552000000.0	-552000000.0	4419000000	
15123	low	186050	USD	2016	FY	2017-02-28	2017-04-04	8.800000e+08	8.810000e+08	6.501700e+10	...	-645000000.0	-645000000.0	5201000000	
15124	low	186050	USD	2017	FY	2018-02-28	2018-04-02	8.390000e+08	8.400000e+08	6.861900e+10	...	-633000000.0	-633000000.0	5953000000	-4

15125 rows × 27 columns

You will notice there is quite a lot of data in that 3MB. 15,125 rows and 27 columns. Not all the columns are visible as there is not enough space on your screen, to get around this Jupyter just puts a '…' for the bulk of the DataFrame rows. You can see what all the headings are by calling the *.keys()* DataFrame function. In Pandas the column headings are called keys, and the columns themselves are called a series.

In[4]:

```
# The headings of columns are called keys.
# The columns themselves are called series.
# To see what all the columns are in our DataFrame,
# use the .keys() function.

print(Income_Data.keys())
```

Out[4]:

Index(['Ticker', 'SimFinId', 'Currency', 'Fiscal Year', 'Fiscal Period', 'Report Date', 'Publish Date', 'Shares (Basic)', 'Shares (Diluted)', 'Revenue', 'Cost of Revenue', 'Gross Profit', 'Operating Expenses', 'Selling, General & Administrative', 'Research & Development', 'Depreciation & Amortization', 'Operating Income (Loss)', 'Non-Operating Income (Loss)', 'Interest Expense, Net', 'Pretax Income (Loss), Adj.', 'Abnormal Gains (Losses)', 'Pretax Income (Loss)', 'Income Tax (Expense) Benefit, Net', 'Income (Loss) from Continuing Operations', 'Net Extraordinary Gains (Losses)', 'Net Income', 'Net Income (Common)'], dtype='object')

To view a single column series, use square brackets after stating the DataFrame variable and specify the key. For instance, here we want to see the column that has all the stock tickers from the SimFin information:

In[5]:

```
# To see series data for a single key:
Income_Data['Ticker']
```

Out[5]:

```
0        A
1        A
2        A
3        A
4        A
        ...
15120    low
15121    low
15122    low
15123    low
15124    low
Name: Ticker, Length: 15125, dtype: object
```

Obtaining series data like this yields a Pandas series object. You can convert this type to a Numpy array use the *.values* command. This might be useful if you wanted to calculate discounted cash flows or similar calculation from DataFrame data.

In[6]:

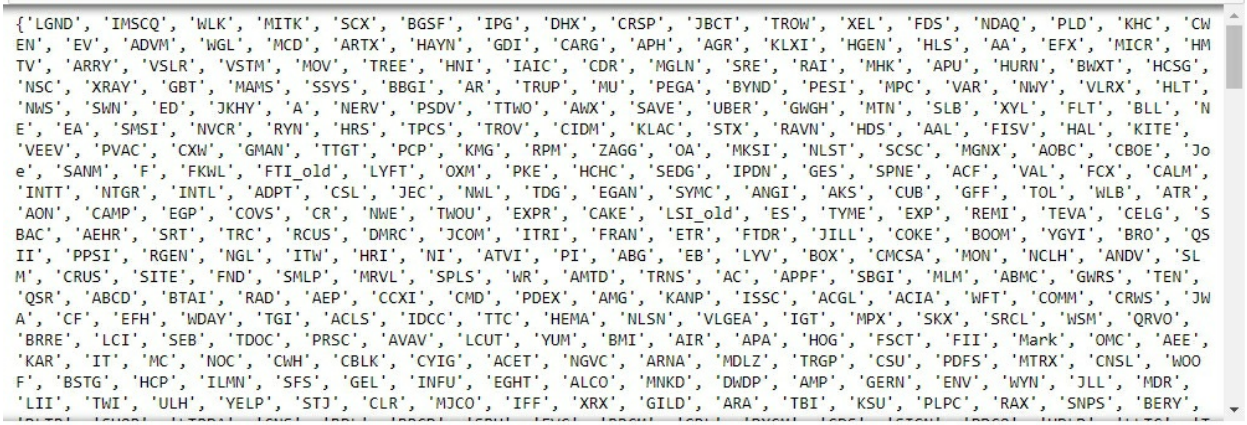
```
print('\nThe resulting type of object from our series selection is a:\n'      , type(Income_Data['Report Date']) )
print('\nusing .values after our selection turns this into a:\n',
      type(Income_Data['Report Date'].values) )
```

Out[6]:

```
The resulting type of object from our series selection is a:
<class 'pandas.core.series.Series'>

using .values after our selection turns this into a:
<class 'numpy.ndarray'>
```

Notice that in this dataset there are many listings for each stock ticker, as several fiscal years are present. To get a list of unique values, so we can see all stocks present, we can use our knowledge of sets we learned earlier, remember that a set only contains one of each instance.

```
# To get a list of tickers that exist, we could use the set operations we learned earlier.
my_set = set(Income_Data['Ticker'].values)
print(my_set)
```



```
{'LGND', 'IMSCQ', 'WLK', 'MITK', 'SCX', 'BGSF', 'IPG', 'DHX', 'CRSP', 'JBCT', 'TROW', 'XEL', 'FDS', 'NDAQ', 'PLD', 'KHC', 'CW
EN', 'EV', 'ADVM', 'WGL', 'MCD', 'ARTX', 'HAYN', 'GDI', 'CARG', 'APH', 'AGR', 'KLXI', 'HGEN', 'HLS', 'AA', 'EFX', 'MICR', 'HM
TV', 'ARRY', 'VSLR', 'VSTM', 'MOV', 'TREE', 'HNI', 'IAIC', 'CDR', 'MGLN', 'SRE', 'RAI', 'MHK', 'APU', 'HURN', 'BWXT', 'HCSG',
'NSC', 'XRAY', 'GBT', 'MAMS', 'SSYS', 'BBGI', 'AR', 'TRUP', 'MU', 'PEGA', 'BYND', 'PESI', 'MPC', 'VAR', 'NWY', 'VLRX', 'HLT',
'NWS', 'SWN', 'ED', 'JKHY', 'A', 'NERV', 'PSDV', 'TTWO', 'AWX', 'SAVE', 'UBER', 'GWGH', 'MTN', 'SLB', 'XYL', 'FLT', 'BLL', 'N
E', 'EA', 'SMSI', 'NVCR', 'RYN', 'HRS', 'TPCS', 'TROV', 'CIDM', 'KLAC', 'STX', 'RAVN', 'HDS', 'AAL', 'FISV', 'HAL', 'KITE',
'VEEV', 'PVAC', 'CXW', 'GMAN', 'TTGT', 'PCP', 'KMG', 'RPM', 'ZAGG', 'OA', 'MKSI', 'NLST', 'SCSC', 'MGNX', 'AOBC', 'CBOE', 'Jo
e', 'SANM', 'F', 'FKWL', 'FTI_old', 'LYFT', 'OXM', 'PKE', 'HCHC', 'SEDG', 'IPDN', 'GES', 'SPNE', 'ACF', 'VAL', 'FCX', 'CALM',
'INTT', 'NTGR', 'INTL', 'ADPT', 'CSL', 'JEC', 'NWL', 'TDG', 'EGAN', 'SYMC', 'ANGI', 'AKS', 'CUB', 'GFF', 'TOL', 'WLB', 'ATR',
'AON', 'CAMP', 'EGP', 'COVS', 'CR', 'NWE', 'TWOU', 'EXPR', 'CAKE', 'LSI_old', 'ES', 'TYME', 'EXP', 'REMI', 'TEVA', 'CELG', 'S
BAC', 'AEHR', 'SRT', 'TRC', 'RCUS', 'DMRC', 'JCOM', 'ITRI', 'FRAN', 'ETR', 'FTDR', 'JILL', 'COKE', 'BOOM', 'YGYI', 'BRO', 'QS
II', 'PPSI', 'RGEN', 'NGL', 'ITW', 'HRI', 'NI', 'ATVI', 'PI', 'ABG', 'EB', 'LYV', 'BOX', 'CMCSA', 'MON', 'NCLH', 'ANDV', 'SL
M', 'CRUS', 'SITE', 'FND', 'SMLP', 'MRVL', 'SPLS', 'WR', 'AMTD', 'TRNS', 'AC', 'APPF', 'SBGI', 'MLM', 'ABMC', 'GWRS', 'TEN',
'QSR', 'ABCD', 'BTAI', 'RAD', 'AEP', 'CCXI', 'CMD', 'PDEX', 'AMG', 'KANP', 'ISSC', 'ACGL', 'ACIA', 'WFT', 'COMM', 'CRWS', 'JW
A', 'CF', 'EFH', 'WDAY', 'TGI', 'ACLS', 'IDCC', 'TTC', 'HEMA', 'NLSN', 'VLGEA', 'IGT', 'MPX', 'SKX', 'SRCL', 'WSM', 'QRVO',
'BRRE', 'LCI', 'SEB', 'TDOC', 'PRSC', 'AVAV', 'LCUT', 'YUM', 'BMI', 'AIR', 'APA', 'HOG', 'FSCT', 'FII', 'Mark', 'OMC', 'AEE',
'KAR', 'IT', 'MC', 'NOC', 'CWH', 'CBLK', 'CYIG', 'ACET', 'NGVC', 'ARNA', 'MDLZ', 'TRGP', 'CSU', 'PDFS', 'MTRX', 'CNSL', 'WOO
F', 'BSTG', 'HCP', 'ILMN', 'SFS', 'GEL', 'INFU', 'EGHT', 'ALCO', 'MNKD', 'DWDP', 'AMP', 'GERN', 'ENV', 'WYN', 'JLL', 'MDR',
'LII', 'TWI', 'ULH', 'YELP', 'STJ', 'CLR', 'MJCO', 'IFF', 'XRX', 'GILD', 'ARA', 'TBI', 'KSU', 'PLPC', 'RAX', 'SNPS', 'BERY',
```

Wow that's a lot of stocks. However in obtaining this list, conversion to a Numpy array and then a set as we just did isn't strictly required, Pandas gives us a single function to do this, the *.unique()* function, which returns an array:

In[7]:

```
#Alternatively a DataFrame has a function .unique()
Income_Data['Ticker'].unique()
```

Out[7]:

```
array(['A', 'AA', 'AAC', ..., 'ZYNE', 'ZYXI', 'low'], dtype=object)
```

We can use logic learnt earlier to separate and arrange our data for use. For example, say we want to look at all company earnings data for the fiscal year 2017. We can do this by running each row through logic that returns *True* if we want it and *False* if it doesn't.

Luckily in Pandas, we don't have to create a loop to iterate through every row in our data. It's better to use the built-in Python functions where possible as it is faster to compute than building small loops from scratch. Using logic on a DataFrame series (column) to return a boolean returns a series of booleans as long as the DataFrame:

In[8]:

```
# You can use logic on dataframes to obtain a list of boolean values
# that say whether a rule is True or False for each row.
# Let's try and isolate financial data for the 2017 fiscal period.
# We will do a boolean check on items in the 'Fiscal Year' column:

IsIt2017 = (Income_Data['Fiscal Year'] == 2017)
print('The boolean logic on the Fiscal Year column is: ', IsIt2017)
```

Out[8]:

The boolean logic on the Fiscal Year column is:
0 False
1 False
2 False
3 False
4 False
...
15120 False
15121 False
15122 False
15123 False
15124 True
Name: Fiscal Year, Length: 15125, dtype: bool

This series of booleans is commonly called a 'boolean mask' as you can mask information you don't want from a DataFrame with it, for instance in the following screenshot we mask our original DataFrame table to only contain rows that have a 'Fiscal Year' value of 2017:

```
# We can use this list of booleans to make a smaller dataframe with only data from fiscal year 2017 as follows:
Income_Data[IsIt2017]
```

	Ticker	SimFinId	Currency	Fiscal Year	Fiscal Period	Report Date	Publish Date	Shares (Basic)	Shares (Diluted)	Revenue	...	Non-Operating Income (Loss)	Interest Expense, Net	Pretax Income (Loss), Adj.	A
9	A	45846	USD	2017	FY	2017-10-31	2017-12-21	322000000.0	326000000.0	4.472000e+09	...	-38000000.0	-57000000.0	803000000	
13	AA	367153	USD	2017	FY	2017-12-31	2018-02-26	184000000.0	187000000.0	1.165200e+10	...	-46000000.0	-104000000.0	1468000000	-309
24	AAL	68568	USD	2017	FY	2017-12-31	2018-02-21	489164000.0	491692000.0	4.220700e+10	...	-974000000.0	-959000000.0	3084000000	
34	AAN	441241	USD	2017	FY	2017-12-31	2018-03-01	70837000.0	72121000.0	3.383708e+09	...	-15122000.0	-18703000.0	257571000	-17
41	AAOI	671827	USD	2017	FY	2017-12-31	2018-02-28	19097355.0	20139105.0	3.823290e+08	...	-2425000.0	-637000.0	84526000	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
15090	ZUMZ	45730	USD	2017	FY	2018-02-28	2018-03-19	24679000.0	24878000.0	9.274010e+08	...	-357000.0	495000.0	48403000	
15099	ZVO	901866	USD	2017	FY	2017-12-31	2018-02-21	32058000.0	32794000.0	4.751130e+08	...	1511000.0	NaN	16619000	-8
15104	ZYNE	901704	USD	2017	FY	2017-12-31	2018-03-12	12914814.0	12914814.0	0.000000e+00	...	810705.0	519554.0	-32012304	
15113	ZYXI	171401	USD	2017	FY	2017-12-31	2018-02-28	32156000.0	33196000.0	2.343200e+07	...	-1450000.0	-1450000.0	7494000	
15124	low	186050	USD	2017	FY	2018-02-28	2018-04-02	839000000.0	840000000.0	6.861900e+10	...	-633000000.0	-633000000.0	5953000000	-464

1726 rows × 27 columns

After this boolean mask is applied the resulting DataFrame is only 1726 rows long. Such a logic operation on a DataFrame can be done with a single line of code, without having to create and store the boolean mask.

You can combine many DataFrame operations in a single line. For instance,

after this operation, you can extract out the first row by putting *[0]* at the end, or you could send the DataFrame through a new set of logic. If you wish to order this DataFrame use the *sort_values()* function:

```
# Let's order the stocks for financial year 2017 by net income.
# Within a DataFrame the rows can be ordered as follows:
Income_Data.sort_values(by=['Net Income'])
```

	Ticker	SimFinId	Currency	Fiscal Year	Fiscal Period	Report Date	Publish Date	Shares (Basic)	Shares (Diluted)	Revenue	...	Non-Operating Income (Loss)	Interest Expense, Net	Pretax Income (Loss), Adj.
5740	GE	244314	USD	2018	FY	2018-12-31	2019-02-26	8.691000e+09	8.691000e+09	1.216150e+11	...	-5.590000e+09	-5.059000e+09	4779000000
3367	COP	378120	USD	2008	FY	2008-12-31	2009-08-04	1.523432e+09	1.523432e+09	2.408420e+11	...	4.263000e+09	-9.350000e+08	31409000000
13052	TEVA	616649	USD	2017	FY	2017-12-31	2018-02-12	1.016000e+09	1.016000e+09	2.238500e+10	...	3.040000e+08	-8.950000e+08	4295000000
5077	F	249937	USD	2008	FY	2008-12-31	2009-08-05	2.273000e+09	2.272000e+09	1.435840e+11	...	-8.920000e+09	-9.737000e+09	-14895000000
2909	CHK	378111	USD	2015	FY	2015-12-31	2015-11-04	6.620000e+08	6.620000e+08	1.276400e+10	...	-4.050000e+08	-3.170000e+08	-852000000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
14892	XOM	121214	USD	2008	FY	2008-12-31	2009-08-05	4.941000e+09	4.961000e+09	4.595790e+11	...	-5.912000e+10	-6.730000e+08	83397000000
70	AAPL	111052	USD	2016	FY	2016-09-30	2016-10-26	5.470820e+09	5.500281e+09	2.156390e+11	...	1.348000e+09	NaN	61372000000
71	AAPL	111052	USD	2017	FY	2017-09-30	2017-08-02	5.217242e+09	5.251692e+09	2.292340e+11	...	2.745000e+09	NaN	64089000000
69	AAPL	111052	USD	2015	FY	2015-09-30	2015-10-28	5.753421e+09	5.793069e+09	2.337150e+11	...	1.285000e+09	NaN	72515000000
72	AAPL	111052	USD	2018	FY	2018-09-30	2018-11-05	4.955377e+09	5.000109e+09	2.655950e+11	...	2.005000e+09	NaN	72903000000

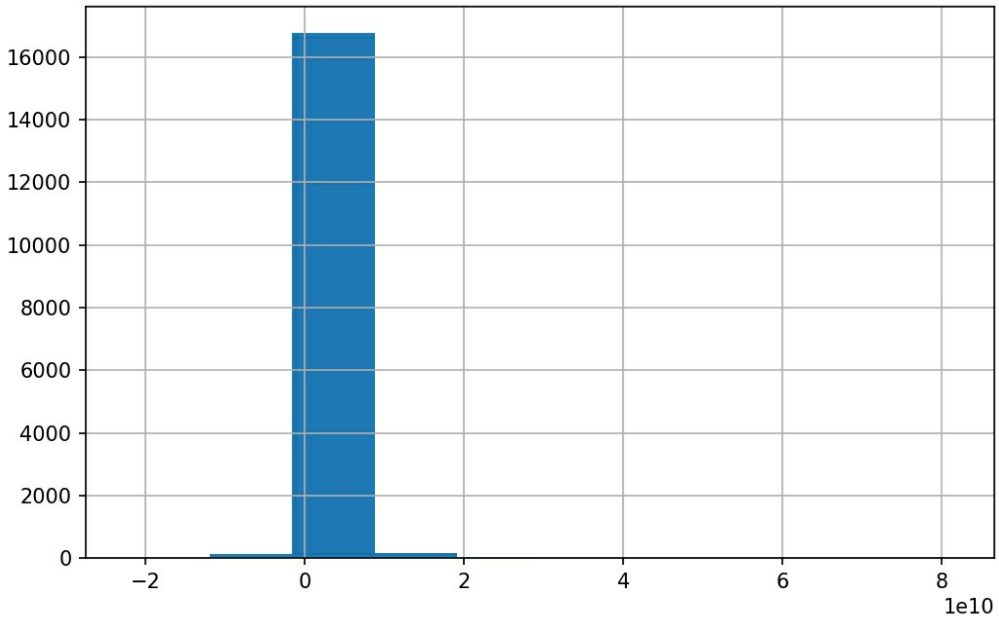
15125 rows × 27 columns

It is often useful to visualise this data in histograms, which in Jupyter Notebooks can be done in a single line with the *.hist()* command. With the data we have in our DataFrame we can view a histogram of the net income of all American companies:

In[9]:

```
# Lets plot the distribution of net income for all American companies:
Income_Data['Net Income'].hist()
```

Out[9]:



When using a function like this without any arguments the default settings are used, notice there are no axes labels or a title. This graph doesn't present the information that well either, as most companies earn very little relative to giants like AAPL, making the distribution very skewed. The histogram binning (that is, the column ranges) don't give us the resolution we might want.

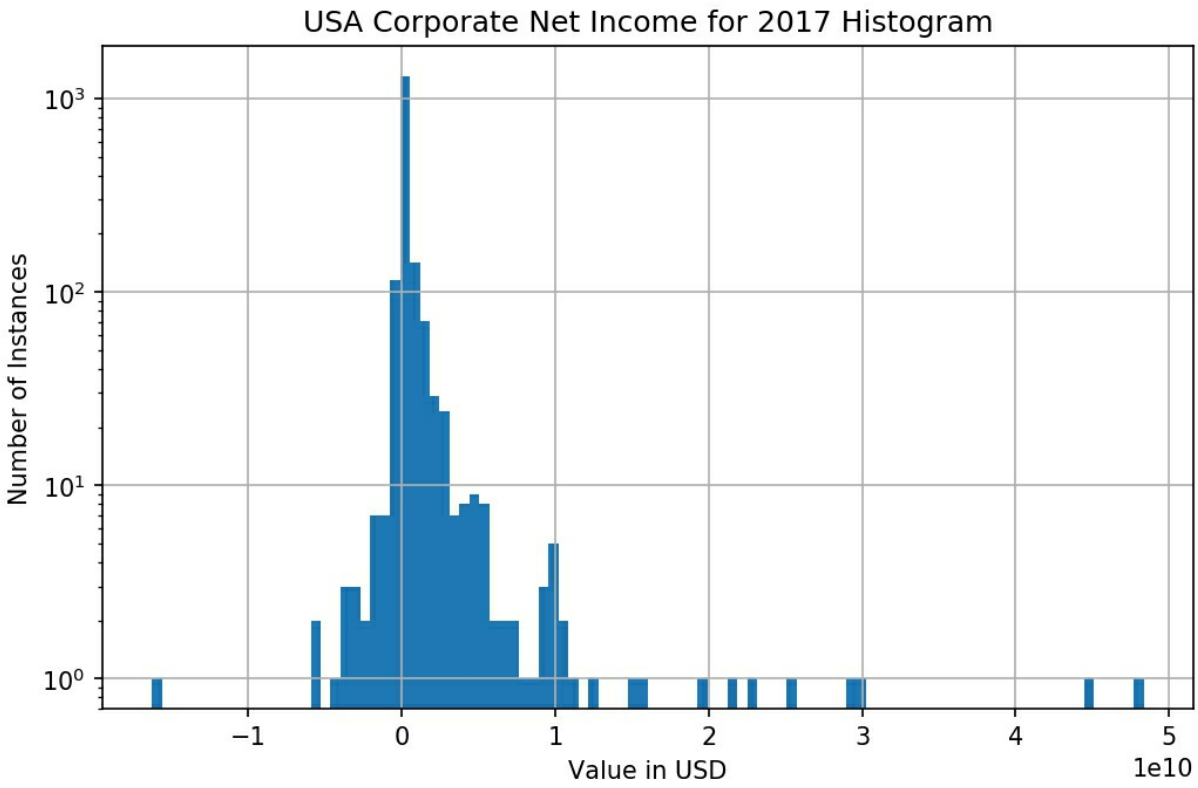
This histogram is also plotting the data for all fiscal years combined. Let's make the graph more interesting, we can plot it for only fiscal year 2017 results as before, and we can edit the histogram to be logarithmic so we can more easily see the extremes of the data. Furthermore, we can increase the number of bins so we can see more detail and give the axes titles:

In[10]:

```
# Plot graph with just 2017 FY data,
# logarithmic Y axis and more bins for greater resolution.
from matplotlib import pyplot as plt # Import plotting library
Income_Data[Income_Data['Fiscal Year'] == 2017]['Net Income'].hist(bins=100, log=True)
plt.title('USA Corporate Net Income for 2017 Histogram')
plt.xlabel('Value in USD')
plt.ylabel('Number of Instances')
print('Max value is: ', Income_Data[Income_Data['Fiscal Year'] == 2017]['Net Income'].max())
```

Out[10]:

Max value is: 48351000000



That's much better. Notice the two blips to the far right of the distribution, who are these companies that are making so much money? With some DataFrame manipulation, we can find out, just sort the table by net income, or find the *N* largest values with the DataFrame function *.nLargest()*. These companies are BRKA and AAPL.

You can extract the max/min values from a pandas series with *.max()* and *.min()* to see the extremes. Further useful statistical data can be calculated from any DataFrame with the *.describe()* function:

```
# Statistical data can be calculated with the dataframe .describe() function
Income_Data.describe()
```

	SimFinId	Fiscal Year	Shares (Basic)	Shares (Diluted)	Revenue	Cost of Revenue	Gross Profit	Operating Expenses	Selling, General & Administrative	Research & Development	...
count	15125.000000	15125.000000	1.503000e+04	1.503000e+04	1.490400e+04	1.377600e+04	1.491100e+04	1.506100e+04	1.439500e+04	6.035000e+03	...
mean	399399.405091	2013.913124	2.324829e+08	2.364339e+08	7.423264e+09	-5.205191e+09	2.611450e+09	-1.656050e+09	-1.135391e+09	-3.959251e+08	...
std	265720.843594	2.764324	6.077170e+08	6.202862e+08	2.444919e+10	1.864871e+10	7.904727e+09	4.924203e+09	3.947171e+09	1.262572e+09	...
min	18.000000	2007.000000	1.400000e+01	1.400000e+01	0.000000e+00	-3.853010e+11	-1.280800e+10	-1.071470e+11	-1.071470e+11	-2.141900e+10	...
25%	160965.000000	2012.000000	2.673825e+07	2.694063e+07	2.737512e+08	-3.082569e+09	1.184120e+08	-1.225592e+09	-8.121845e+08	-1.890505e+08	...
50%	371566.000000	2014.000000	6.677505e+07	6.800000e+07	1.415794e+09	-7.781640e+08	5.680430e+08	-3.638050e+08	-2.090000e+08	-5.279900e+07	...
75%	639822.000000	2016.000000	1.930758e+08	1.959935e+08	5.159512e+09	-1.431200e+08	1.964170e+09	-9.075300e+07	-4.737350e+07	-1.464700e+07	...
max	985521.000000	2018.000000	8.945000e+09	8.996000e+09	5.144050e+11	3.100000e+07	1.722200e+11	1.110132e+09	0.000000e+00	6.480000e+05	...

8 rows × 22 columns

This is a good way to get a few numbers for an overview of the data. Telling someone the mean, max/min and standard deviation is an easy way to describe a distribution, you get those figures for every column here.

You can create new DataFrames from filtered old DataFrames just like other object types, using =.

In[11]:

```
# You can create new dataframes from filtered old dataframes,
# let's separate the 2017 fiscal year data to a new DataFrame.
Income_Data_2017 = Income_Data[Income_Data['Fiscal Year'] == 2017]
```

And finally for this section, you can view the top and bottom parts of DataFrames with the *.head()* and *.tail()* functions. The Pandas documentation provides a cheat sheet should you want to do some DataFrame manipulation and don't know the function you need: *https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf*

Exercise 6 – Pandas Basics

If you're a stock market enthusiast and haven't done Python coding before, this is a great place to play around with the raw statistics without depending on a platform. Follow the Exercise 6 Jupyter Notebook, where we put together what you have learnt so far to make some meaningful DataFrame manipulations, look at some stock statistics and do some rudimentary stock screening and analysis of the financials of corporate America.

For everyday viewing of financial data there is no need to download csv files and load them up every time. The Pandas library has an extension that provides an easy way to quickly get stock data from online sources such as yahoo finance through the *datareader* function. This is a utility that will be useful for everyday use given that we would want to see how stocks are performing, and we might even want to track our portfolio without having to rely on tools our stockbroker provides us:

In[12]:

```
import pandas as pd
import pandas_datareader.data as pdr
from matplotlib import pyplot as plt

start = pd.to_datetime('2019-05-01')
end = pd.to_datetime('2020-05-01')
tickerData = pdr.DataReader('SPY', 'yahoo', start, end);

plt.plot(tickerData['Open']);
plt.grid()
plt.xlabel('Date')
plt.ylabel('SPY value')
```

Out[12]:

The documentation for Pandas *datareader* is available on: https://*pandas*-

datareader.readthedocs.io, where many other data sources are listed, with many requiring a subscription to access. Pandas *datareader* this is not part of the main Pandas library so don't expect the documentation to be as rigorous.

So far we have seen a DataFrame where the index column is an integer, this makes sense for lists of companies like we have seen form SimFin fundamental data, but for time series data like the output from the Pandas *datareader*, an index that is a date makes more sense, and is often the default output:

In[13]:

```
tickerData
```

Out[13]:

Date	High	Low	Open	Close	Volume	Adj Close
2019-05-01	294.950012	291.799988	294.720001	291.809998	71671900.0	282.375183
2019-05-02	292.700012	289.519989	291.679993	291.179993	65030200.0	281.765503
2019-05-03	294.339996	291.299988	292.820007	294.029999	56543700.0	284.523407
2019-05-06	293.309998	288.899994	289.250000	292.820007	107198100.0	283.352539
2019-05-07	290.809998	285.809998	290.149994	287.929993	144729900.0	278.620605
...	...	...	...	...	...	...
2020-04-27	288.269989	284.619995	285.119995	287.049988	77896600.0	283.444672
2020-04-28	291.399994	285.399994	291.019989	285.730011	105270000.0	282.141235
2020-04-29	294.880005	290.410004	291.529999	293.209991	118745600.0	289.527283
2020-04-30	293.320007	288.589996	291.709991	290.480011	122901700.0	286.831604
2020-05-01	290.660004	281.519989	285.309998	282.790009	125180000.0	279.238159

254 rows × 6 columns

To select specific rows from this data, we can use the index in a few ways. Firstly, *.iloc[]* is used to select rows in a similar way as for lists:

```
In [7]: tickerData.iloc[-5:] #last 5
```

Out[7]:

Date	High	Low	Open	Close	Volume	Adj Close
2020-04-27	288.269989	284.619995	285.119995	287.049988	77896600.0	283.444672
2020-04-28	291.399994	285.399994	291.019989	285.730011	105270000.0	282.141235
2020-04-29	294.880005	290.410004	291.529999	293.209991	118745600.0	289.527283
2020-04-30	293.320007	288.589996	291.709991	290.480011	122901700.0	286.831604
2020-05-01	290.660004	281.519989	285.309998	282.790009	125180000.0	279.238159

```
In [8]: tickerData.iloc[:5] #first 5
```

Out[8]:

Date	High	Low	Open	Close	Volume	Adj Close
2019-05-01	294.950012	291.799988	294.720001	291.809998	71671900.0	282.375183
2019-05-02	292.700012	289.519989	291.679993	291.179993	65030200.0	281.765503
2019-05-03	294.339996	291.299988	292.820007	294.029999	56543700.0	284.523407
2019-05-06	293.309998	288.899994	289.250000	292.820007	107198100.0	283.352539
2019-05-07	290.809998	285.809998	290.149994	287.929993	144729900.0	278.620605

```
In [9]: tickerData.iloc[207] # row 207
```

```
Out[9]: High         3.181100e+02
        Low          3.107000e+02
        Open         3.141800e+02
        Close        3.115000e+02
        Volume       1.947738e+08
        Adj Close    3.057895e+02
        Name: 2020-02-26 00:00:00, dtype: float64
```

For selecting specific rows, this can use useful if we know that our index proceeds through the integers 1, 2, 3,…,etc. however for a time series we may want to know the stock price at a specific date, or print the stock price (like the price graph for the S&P500 earlier) for a specific time. This is easier to do with the *.loc[]* function, for example to see the stock price on a specific date we can use:

In[14]:

```
tickerData.loc['2019-05-01']
```

Out[14]:

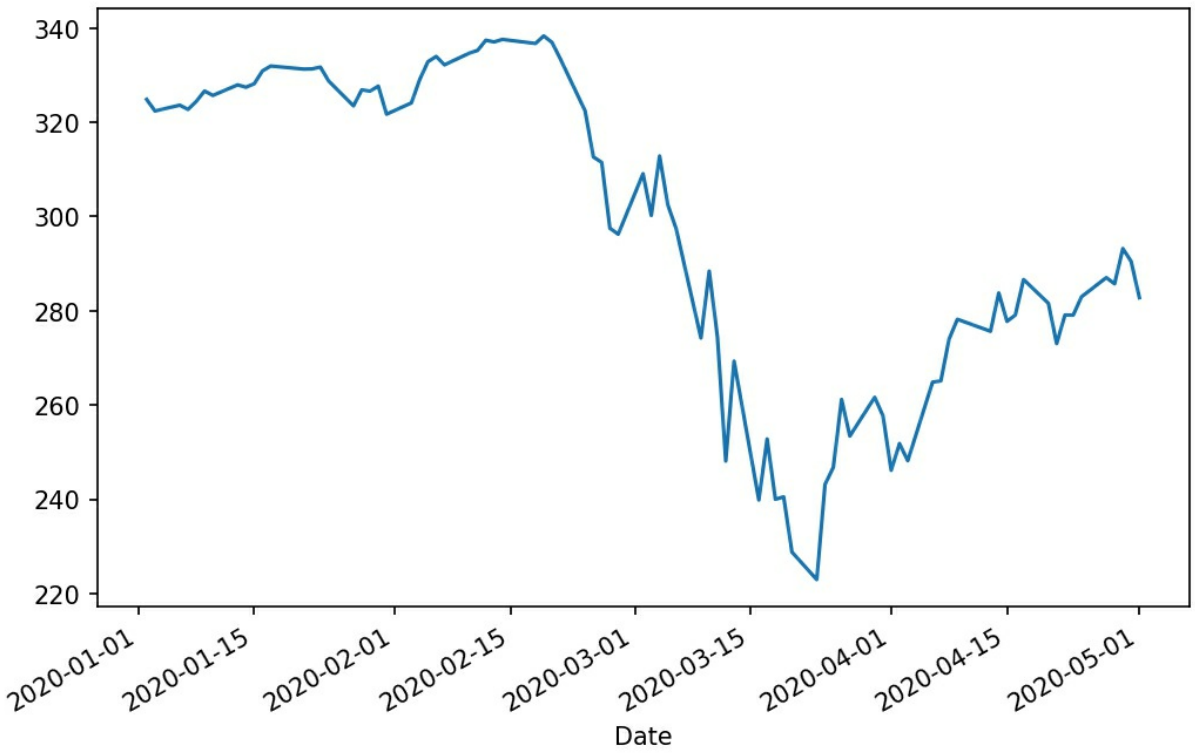
```
High         2.949500e+02
Low          2.918000e+02
Open         2.947200e+02
Close        2.918100e+02
Volume       7.167190e+07
Adj Close    2.823752e+02
Name: 2019-05-01 00:00:00, dtype: float64
```

We can plot our S&P500 data for a specific time period, for example starting on the 1st Jan 2021, we can use the following line:

In[12]:

```
tickerData.loc[tickerData.index>'2020-01-01']['Close'].plot()
```

Out[12]:



Chapter 3 - Machine Learning Introduction with Scikit-learn

With Numpy and Pandas fundamentals learnt, we can move on to learning about the Machine Learning algorithms we will be using in our investing AI. We will be using Machine Learning tools from the Scikit-Learn library.

The approach taken here is to teach the Machine Learning concepts by doing as much as possible, rather than learning all the theory at once before putting it into practice. We'll have a quick overview to put in place the broad concepts, before getting to grips with Machine Learning in more detail in code that you can follow along.

The depth to which Machine Learning is taught here is more than sufficient for developing and understanding the AI Investor. To understand Machine Learning specifically to a much greater depth, Aurélien Géron's *Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow* is written similar style to this book, though the required level of Python competence is higher.

What is Machine Learning?

Machine Learning is the use of computer algorithms to perform tasks without explicit programming to do so. Instead, they learn from experience, improving at their tasks with more experience.

Experience can come in a few forms, it can come from having more outside data to learn from, it can come from data generated from rules (like chess positions), it can also come from seeing already existing data in a new light.

Note this is separate from human-generated algorithmic rules, no matter how complex or how much a human might improve them over time. If we were to train an algorithm to play Tic-tac-toe with a thousand games of experience, and it ends up playing perfectly just like a human programmed tic-tac-toe program, even though the outputs end up the same, the former is Machine Learning and the latter isn't.

There are a lot of Machine Learning algorithms, and they can do a lot of

different things. Knowing how they are split into their generally accepted categories provides a good overview of what's possible.

Whether they are trained with or without human supervision identifies them as *supervised*, *unsupervised* or *semi-supervised* algorithms. A supervised algorithm might identify pictures of flowers, but a human will have identified and labelled the flower pictures in the training data beforehand, whereas an example of an unsupervised learning algorithm is identifying clusters of peoples interests, with no knowledge about the data except that distinct groupings might be found.

Learning continuously with new data or only from a batch of data separates *on-line learning* from *batch learning* algorithms. Youtube recommendations for videos is an example of on-line learning, it is learning what's best to recommend to you continuously based on your viewing history.

Model-based algorithms attempt to draw generalisations from the data, whereas *instance-based* learning simply compares new data to previous data. Instance based learning is a little like a student memorising all the answers from past exams instead of actually learning the concepts. Some success is still possible with this strategy.

Prediction Machine Learning algorithms are split in two, *classification algorithms* and *regression algorithms*. Classification algorithms predict the classification of something, like a human face in a photo or cancerous tissue in a medical scan. Regression algorithms predict a variable (or several), such as GDP of a country or the probability of winning a Poker hand.

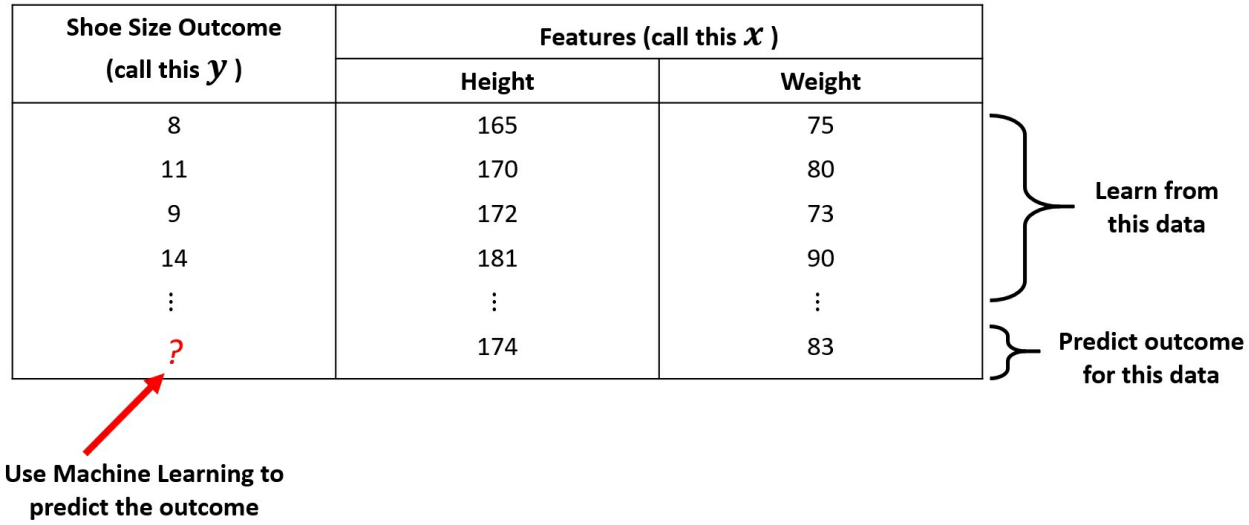
The Machine Learning models we will be using might be termed *model-based supervised batch learning regression prediction algorithms* in this book to make our investing AI, though we will have one instance-based algorithm. The term is a bit of a mouth full, so we'll just call them regression algorithms. We will be using them to predict the percentage change in stock prices over a year.

The Data

Of course, the data to be used to train a Machine Learning algorithm has to be computer readable.

The input data for the regression algorithms we will use are generally represented as a table. Whether it's a picture of something or a list of biometric data, from the Machine Learning algorithms perspective it's just a list of numbers (admittedly the vector of numbers might be larger for a picture than for mere waist and height measurements, a typical webcam photo has 2 million pixels, which would be flattened out to be an input row of 2 million numbers). The columns to this table, whatever data are stored in it, are called features.

This training data will need to have classifications or outcomes that we will later want to predict when they aren't available. For our algorithms the outcomes will be a single column of numbers, one for each for the input table rows. In classification problems these will be categories, in regression problems these will be numbers, like stock performance for example.



Training the Models

If we were to train our models with all the data in the table, we would have no way of knowing how well our models would perform in reality. The way we solve this issue is to split our table of data in two.

First, we train the model, giving it *y* and *X* for some of the rows in our data. We then test the model, using rows we haven't used in training. In testing we feed the model *X* testing information for which we know the real *y* answer, observing how close the *y* prediction is. This is called *Train/Test splitting* (more on this soon).

There are some caveats to model learning, of which similar may occur in human learning too. Imagine if a student revised so diligently for a maths exam that they thought the answer to every third question must be a multiple of 3, this kind of overgeneralising often occurs in Machine Learning models and is called *overfitting*. It happens when the complexity of your model is too high for the data that you have, foregoing broad generalisations that work for gritty details that don't help prediction at all.

The opposite, *underfitting*, can happen too. If you predict the height of a human over time with a linear model from birth, you might be surprised to see that you don't observe 30ft tall humans at age = 70 years. You need a more complex model than a linear one to fit the data to your algorithm.

We will see some graphical examples of these Machine Learning concepts in the next section, which should be easier to intuit.

Machine Learning Basics with Classification Algorithms

We will not be using classification algorithms in the final code, however anyone using Machine Learning tools should know them. We will explore some of these algorithms here to bring across the general concepts for Machine Learning before moving on to regression algorithms.

Decision Trees

Decision Trees are, as the name suggests, trees of decisions. The decisions to be made to classify your data are binary (yes/no, greater or less than) in the standard Scikit-Learn library implementation. Let's try out the Scikit-Learn Decision Tree classifier on data with two dimensions (two variables) so you can see how they work visually. Bear in mind this algorithm can work on many more dimensions if desired.

The Scikit-Learn library is well documented, you can see all the specifics for the Decision Tree as well as the other predictors and tools on in the online reference if you need to find a particular setting. For the Decision Tree it is: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

We will be using fabricated data based on Altman-Z scores to estimate the

probability of bankruptcy for a company. The Altman-Z score takes a variety of metrics on a company (such as EBIT/Total Assets) and weighting them to get a final score. The lower the score the higher the risk of going bankrupt.

The training data we will use is the *Altman_Z_2D.csv* file from the Github repository. This data has three columns, the first stating whether a company has gone bankrupt, and the other two are company ratios EBIT/Total Assets and MktValEquity/Debt (before bankruptcy). This is easy to see by loading the data with Pandas *read_csv* function and looking at the head of the data:

In[1]:

```
import pandas as pd # Importing modules for use.
import numpy as np
import matplotlib.pyplot as plt # For plotting scatter plot
data = pd.read_csv('Altman_Z_2D.csv', index_col=0) # Load the .csv data
data.head(5) # Taking a look at the data.
```

Out[1]:

	Bankrupt	EBIT/Total Assets	MktValEquity/Debt
0	False	27.693875	8.415582
1	False	-14.302305	8.878080
2	True	-20.515623	-8.742365
3	False	29.729424	4.878042
4	False	32.856383	1.107730

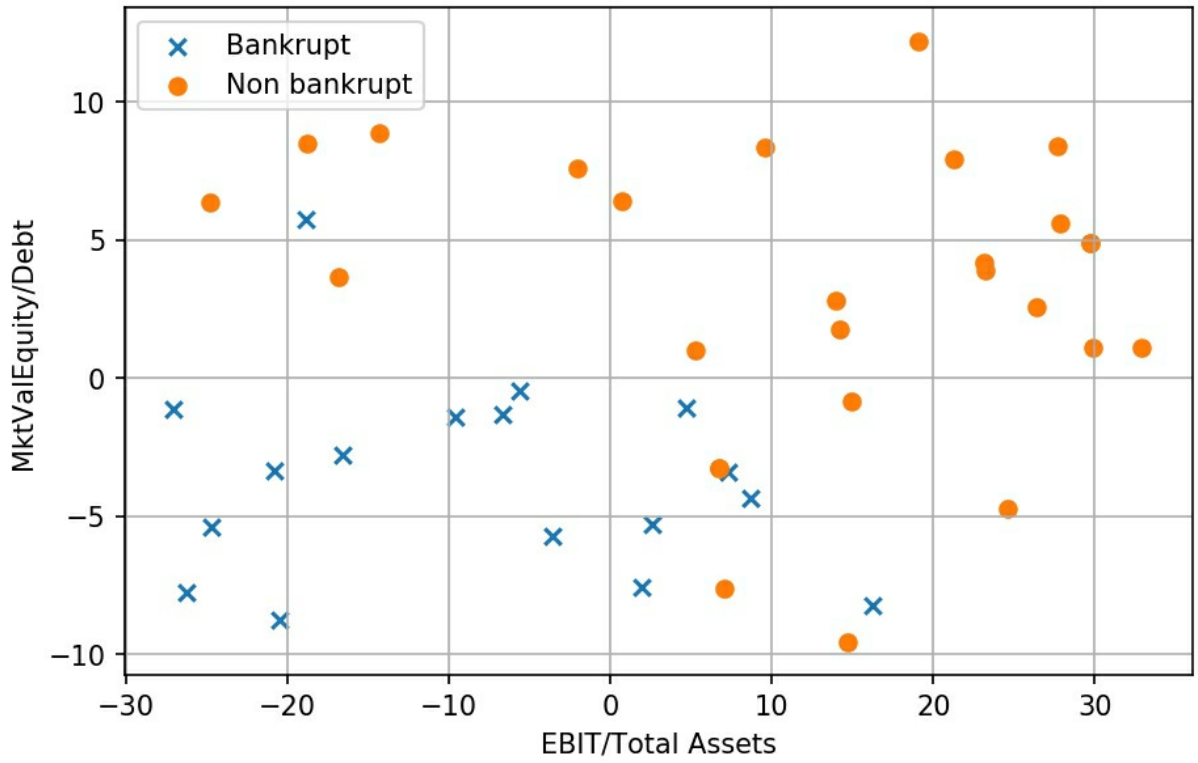
Plotting this data visually in a scatter plot makes the relationship to bankruptcy more obvious. Scatterplots are available from the Pandas DataFrame directly with *DataFrame.plot.scatter()*:

In[2]:

```
# Bankruptcy mask (list of booleans)
bankrupt_mask = data['Bankrupt'] == True

# Plot the bankrupt points
plt.scatter(data['EBIT/Total Assets'][bankrupt_mask],\
            data['MktValEquity/Debt'][bankrupt_mask],\
            marker='x')

# Plot the nonbankrupt points
plt.scatter(data['EBIT/Total Assets'][~bankrupt_mask],\
            data['MktValEquity/Debt'][~bankrupt_mask],\
            marker='o')

# Formatting
```

```
plt.xlabel('EBIT/Total Assets')
plt.ylabel('MktValEquity/Debt')
plt.grid()
plt.legend(['Bankrupt','Non bankrupt'])
```

Out[2]:



Let's use the Scikit-Learn library to make a Decision Tree from this data to identify companies that will go bust. First, we have to split up the data into a matrix, *X*, which contains all the feature values, and vector *Y* which contains the row classification labels, which are the True/False for bankruptcy. We will want to import the Decision Tree classifier from Scikit-Learn too.

In[3]:

```
# Split up the data for the classifier to be trained.
# X is data
# Y is the answer we want our classifier to replicate.
X = data[['EBIT/Total Assets','MktValEquity/Debt']]
Y = data['Bankrupt']

# Import Scikit-Learn
from sklearn.tree import DecisionTreeClassifier
```

We can now create a Decision Tree object. Here we specify the depth of the Decision Tree to 2, we will get more into this value later. As with all Scikit-

Learn models, the latest full documentation is available on the internet (*https://scikit-learn.org/*).

In[4]:

```
# Create a DecisionTreeClassifier object first
tree_clf = DecisionTreeClassifier(max_depth=2)

# Fit the Decision Tree to our training data of X and Y.
tree_clf.fit(X, Y)
```

Out[4]:

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini', max_depth=2,
max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort='deprecated',
random_state=None, splitter='best')
```

After fitting (or training) your Decision Tree classifier, it is imbued with rules that you can use on new data. Say for instance if you have a company of known EBIT/Total Assets and MktValEquity/Debt, your classifier should be able to predict future bankruptcy, making a prediction drawn from what your training data would suggest:

In[5]:

```
# Let's see if it predicts bankruptcy for a bad company
print('Low EBIT/Total Assets and MktValEquity/Debt company go bust?', tree_clf.predict([[-20,
-10]]))

# Let's try this for a highly values, high earning company
print('High EBIT/Total Assets and MktValEquity/Debt company go bust?', tree_clf.predict([[20,
10]]))
```

Out[5]:

```
Low EBIT/Total Assets and MktValEquity/Debt company go bust? [ True]
High EBIT/Total Assets and MktValEquity/Debt company go bust? [False]
```

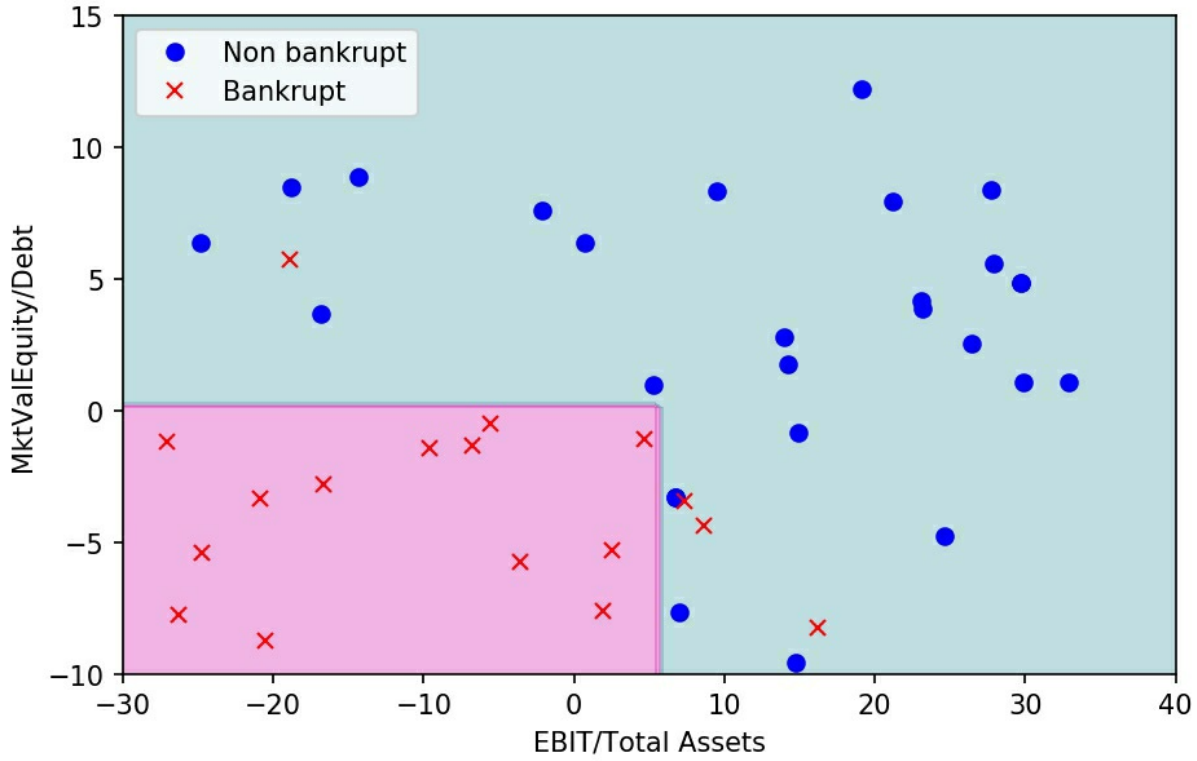
You can pass a DataFrame with several rows and two columns for *X* as well if you want your model to give an answer for a large number of companies. Let’s see what a contour plot of our tree’s predictions in the 2D space looks like.

In[6]:

```
# Contour plot.
from matplotlib.colors import ListedColormap
x1s = np.linspace(-30, 40, 100)
x2s = np.linspace(-10, 15, 100)
x1, x2 = np.meshgrid(x1s, x2s)
X_new = np.c_[x1.ravel(), x2.ravel()]
y_pred = tree_clf.predict(X_new).astype(int).reshape(x1.shape)
```

```
custom_cmap = ListedColormap(['#2F939F','#D609A8'])
plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
plt.plot(X['EBIT/Total Assets'][Y==False],\
         X['MktValEquity/Debt'][Y==False], "bo",
        X['EBIT/Total Assets'][Y==True],\
         X['MktValEquity/Debt'][Y==True], "rx")
plt.xlabel('EBIT/Total Assets')
plt.ylabel('MktValEquity/Debt')
```

Out[6]:



It seems our Decision Tree rules for predicting bankruptcy are quite simple, if each feature ratio (the *x* and *y* axes) are above a certain value the company isn't likely to go bust.

Earlier we fixed our Decision Tree depth with the setting *max_depth=2*. This fixes the depth of the Decision Trees rules. With a value of 2, the complexity of the boundary between bust/not bust is not going to be that complex. Our Decision Tree has made a binary split on our X-axis at around 5, and then another split in one of the remaining domains around a Y-axis value of 0. There are only two splits, which is the maximum depth we specified.

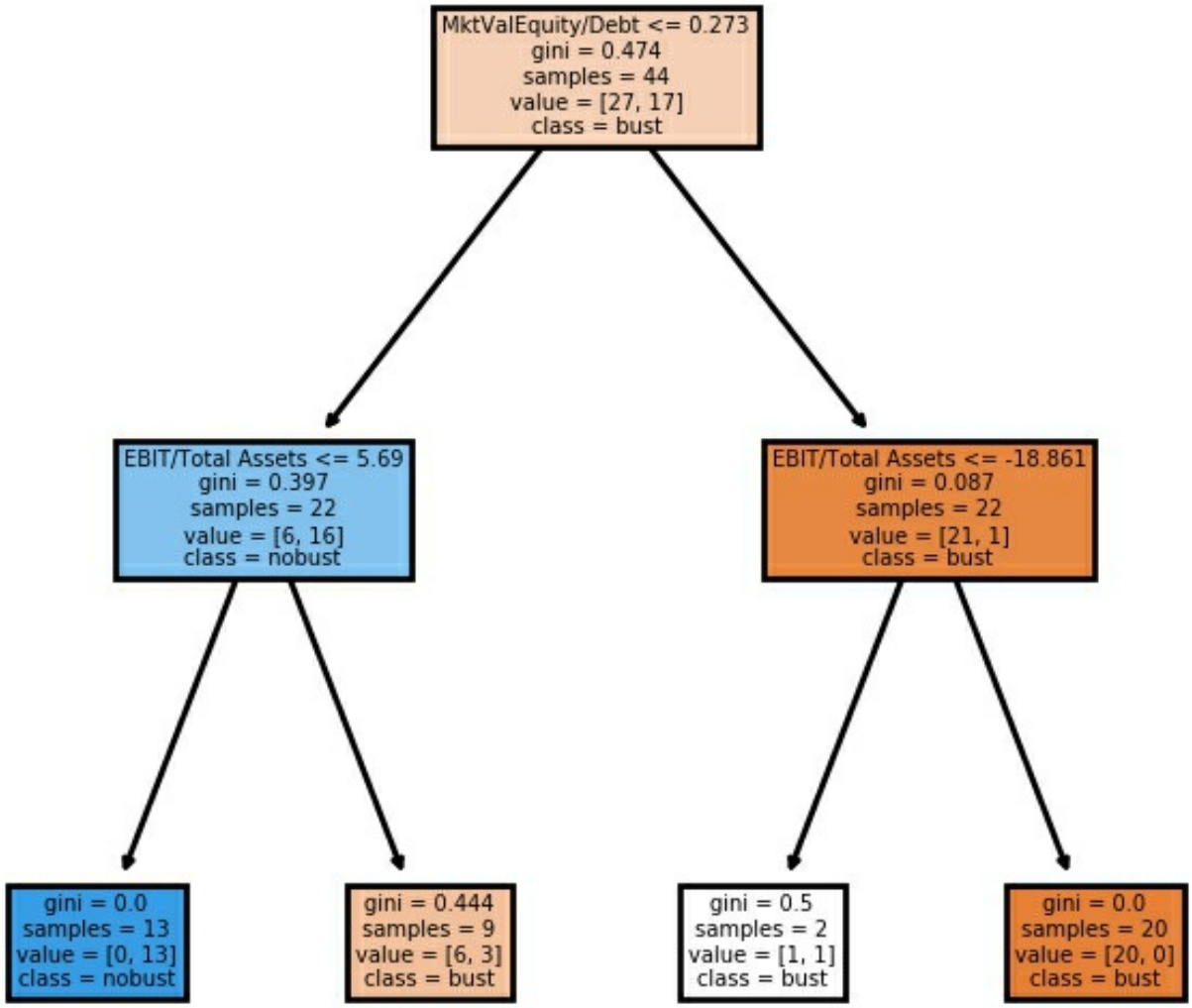
We can see the Decision Tree visually with Scikit-Learn using the *plot_tree*

function:

In[7]:

```
from sklearn import tree # Need this to see Decision Tree.
plt.figure(figsize=(5,5), dpi=300) # set figsize so we can see it
tree.plot_tree(tree_clf,
               feature_names=['EBIT/Total Assets','MktValEquity/Debt'],
               class_names=['bust', 'nobust'],
               filled = True);  # semicolon here to supress output
```

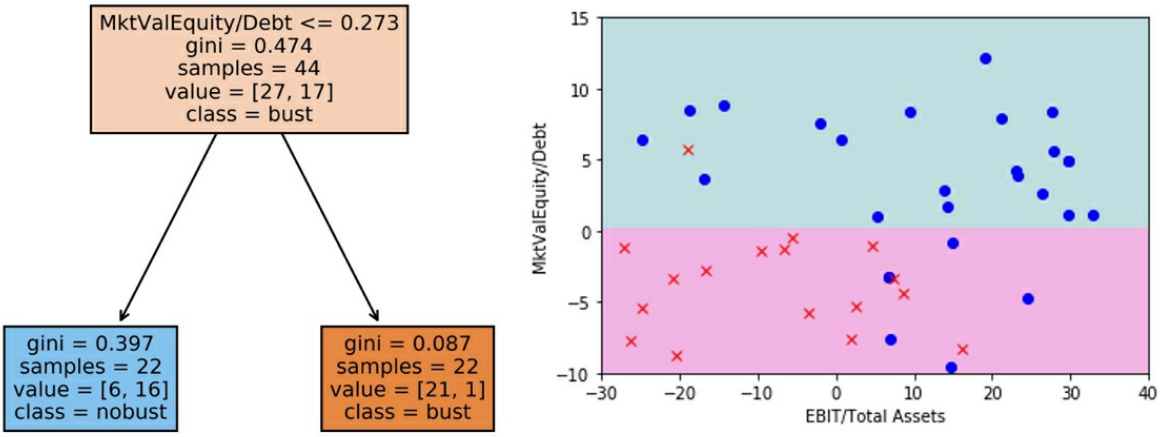
Out[7]:



Neat, we have an algorithm that predicts bankruptcy, and we didn't explicitly program it, furthermore, it can be trained with more data if we have it. Sure it is an algorithm so simple that a human can follow it by just looking up two numbers, but if we change the depth of the tree, and make it include a lot more than just two features, the classifier can easily be more complex than what a human can carry out.

How does a Decision Tree work?

A Decision Tree splits the data along one of the feature dimensions at each level. If we limit the *max_depth* of our Decision Tree to 1, we get a single split, resulting in a far simpler tree, and one that isn't that good at predicting bankruptcy:



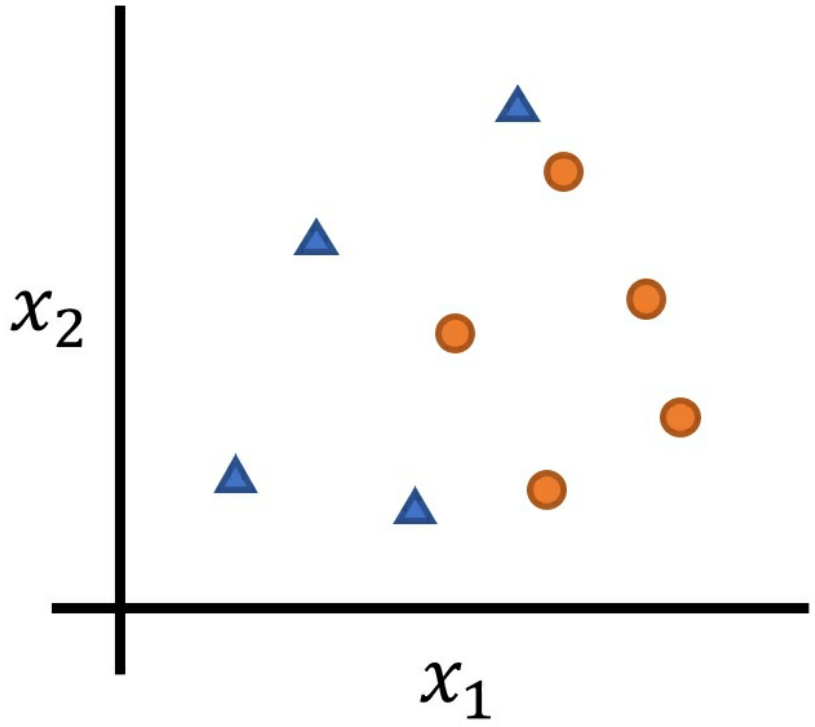
So how does the tree know where to make a split along an axis when it is fitting our data? We will need a quantitative measure of how good a split is, after which the best one can be chosen to make a new branch on our tree. This algorithm is called the Classification And Regression Tree (CART) algorithm. We'll walk through exactly how it works here, though if you are comfortable just using the Scikit-Learn library Decision Tree you can skip this part.

CART Algorithm

The measure being used in this algorithm is the *Gini Impurity* (you can see this value in the tree diagram). The Gini impurity is essentially the probability of mislabelling a random element in your set given that you know the distribution of all classes in the set (our classes here are True and False).

In this True/False scenario, the Gini Impurity is the probability of selecting a true company and labelling it false, plus the probability of selecting a false company and labelling it true.

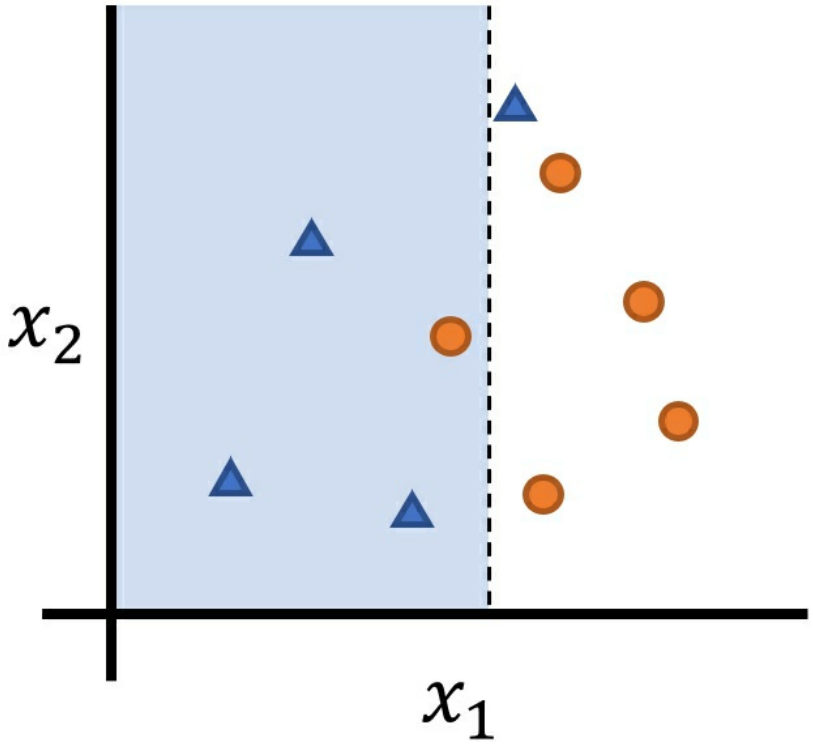
Let's calculate the Gini for a simple example, step by step. Here is a simplified plot of our bankruptcy True/False plot:



The Gini Impurity for this set is the probability of selecting a blue company (4/9) and labelling it orange (4/9) × (5/9), plus the probability of selecting orange (5/9) and labelling it blue (5/9) × (4/9).

This gives us a Gini Impurity of (4/9) × (5/9) + (5/9) × (4/9) = 0.4938.

This is close to what we would intuitively expect given nearly half are blue. Let's say we split our data so that there are many blues in an area we are interested in post-split, which is closer to what we want.



The Gini for that area is (1/4) × (3/4) + (3/4) × (1/4) = 0.375

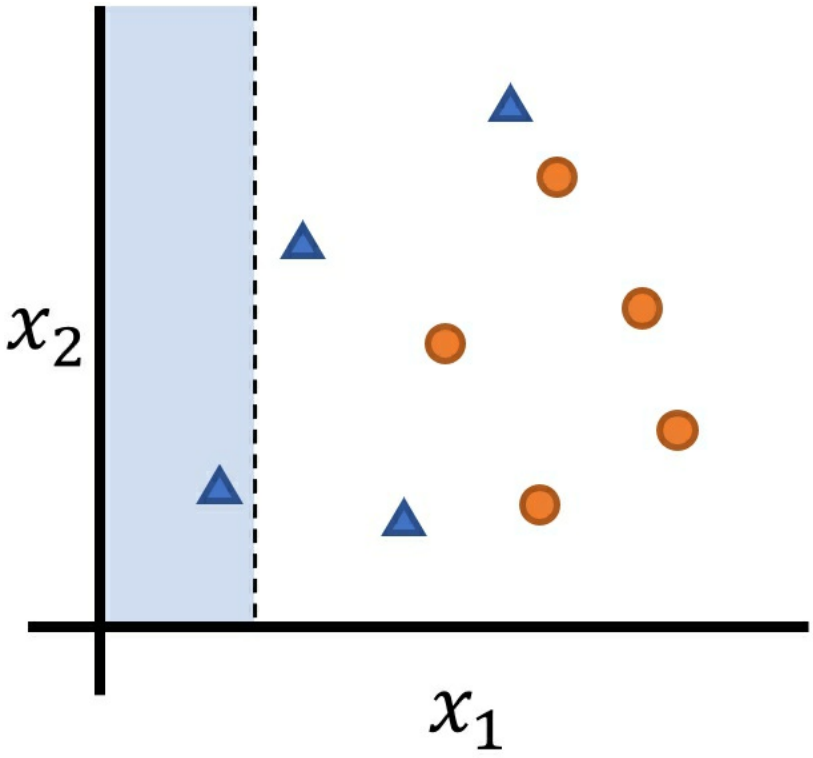
But what about the other side after the split? We must consider this side as well. In the algorithm, the Gini impurity is calculated for the other side too, and the results are weighted by the number of elements in each set that has its own Gini.

The Gini for the right-side area is (1/5) × (4/5) + (4/5) × (1/5) = 0.32

The weighted average is:

0.375 × 4/9 + 0.32 × 5/9 = 0.344

If we were to do a split in a less optimal place the weighted average Gini impurity is worse:



Gini left: (1/1) × (0/1) + (0/1) × (1/1) = 0

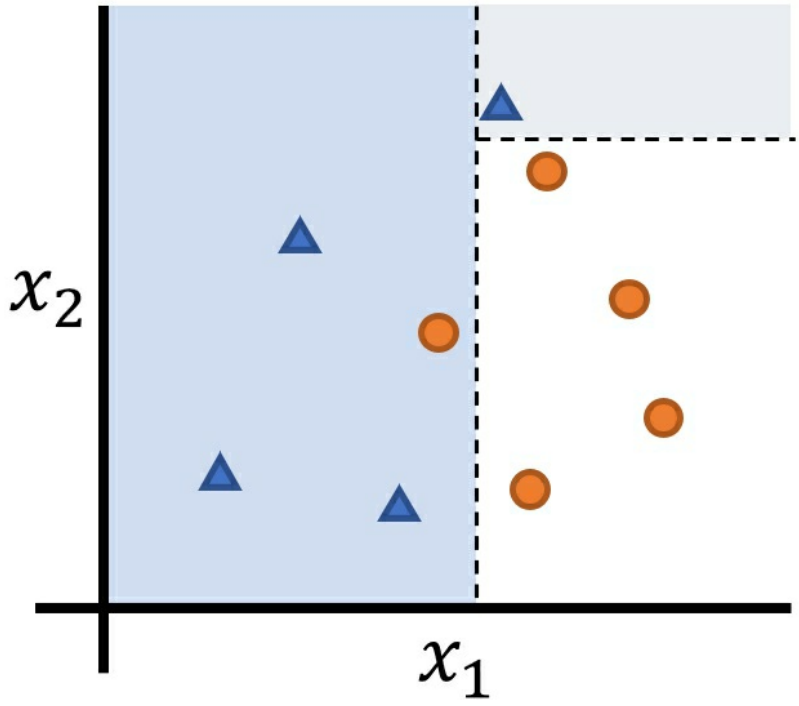
Gini right: (3/8) × (5/8) + (5/8) × (3/8) = 0.46875

The weighted average is then:

0 × 1/9 + 0.46875 × 8/9 = 0.4167

This is a higher Gini impurity value than the split down the centre. When the algorithm tests all the splits it can do, it will then choose the first example split over the second example. There are many ways to find the optimal split, but it is sufficient for our understanding to assume all splits are being tested for simple datasets like the ones we are using, so that the best split that our algorithm can make on a given axis is chosen.

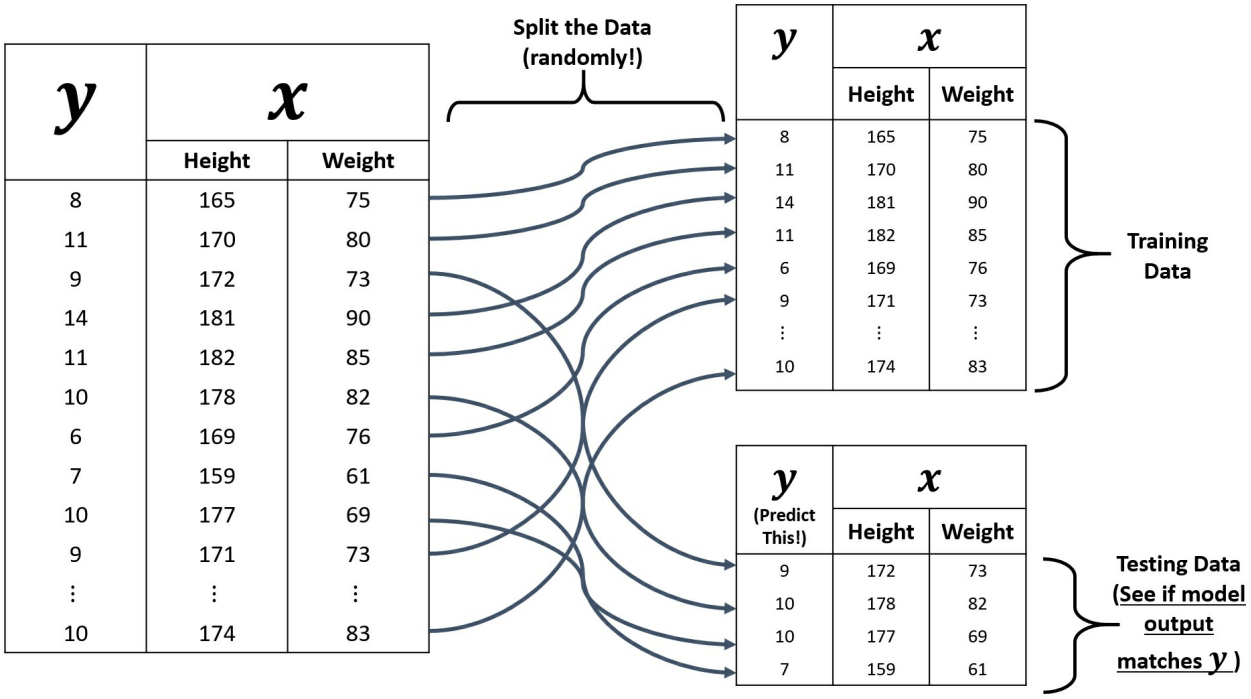
The splitting happens iteratively until you can't split any more, or you've reached the depth you wanted. Another level of depth might look something like this:



After the algorithm splits until it can't make any more splits, or when it reaches the maximum depth allowed, all the decisions are put together to get our Decision Tree.

Testing and Training sets

When training a Machine Learning algorithm, we want to be sure that it works when we give it new data. In the absence of new data, what can we do? We can separate the data into a testing set and a training set. We train our models (whatever they are) on the training set only, and then once it is trained, we test our model on the test set to see if it truly has predictive power.



It is important to sample the sets randomly to remove any kind of bias. Scikit-Learn has a function to do this for us called *train_test_split*, so there's no need to build code from scratch to do this. Arguments passed into it are *X* and *Y* (DataFrames), also *test_size* should be stated, which is a number between 0 and 1 which specifies the proportion to be taken as the test set. The *random_state* variable is a seeding random number, if you use the same number, the selected random set will stay the same each time.

When there isn't much data, like in this bankruptcy example, it is worth checking the histograms to make sure the test set bears a reasonable resemblance to the training data.

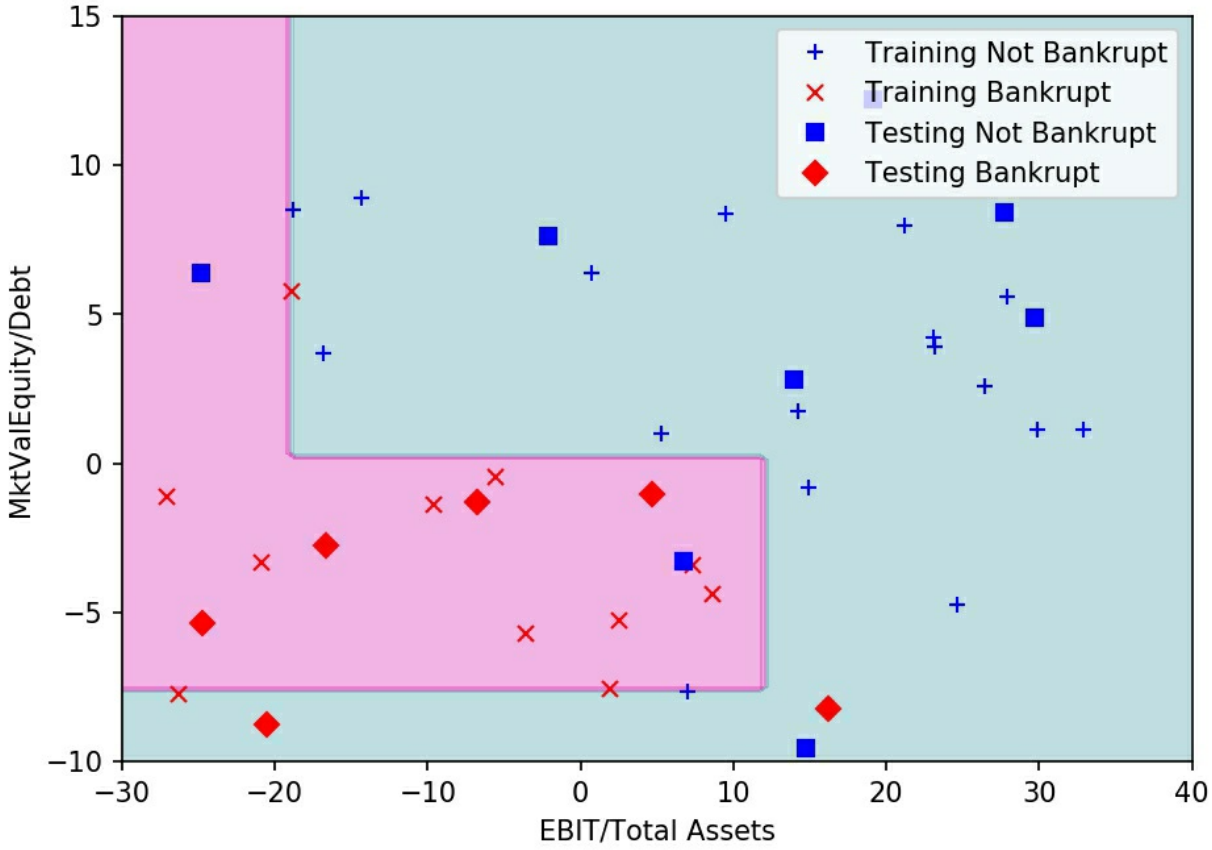
In[8]:

```
# Test/train splitting
from sklearn.model_selection import train_test_split # need to import
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=42)
# Have a look at the train and test sets.
print('X_train:', X_train.shape)
print('X_test:', X_test.shape)
print('y_train:', y_train.shape)
print('y_test:', y_test.shape)
```

Out[8]:

X_train: (29, 2)
X_test: (15, 2)
y_train: (29,)

y_test: (15,)



You can easily see the crosses which were used to train the model, and the boxes which we are testing the model with. Notice that the category split differs a little from before, our Decision Tree thinks that the whole left-hand region below -20 is the bankruptcy region. It is important to have a training set that is large enough to capture the relationships that may appear in the data.

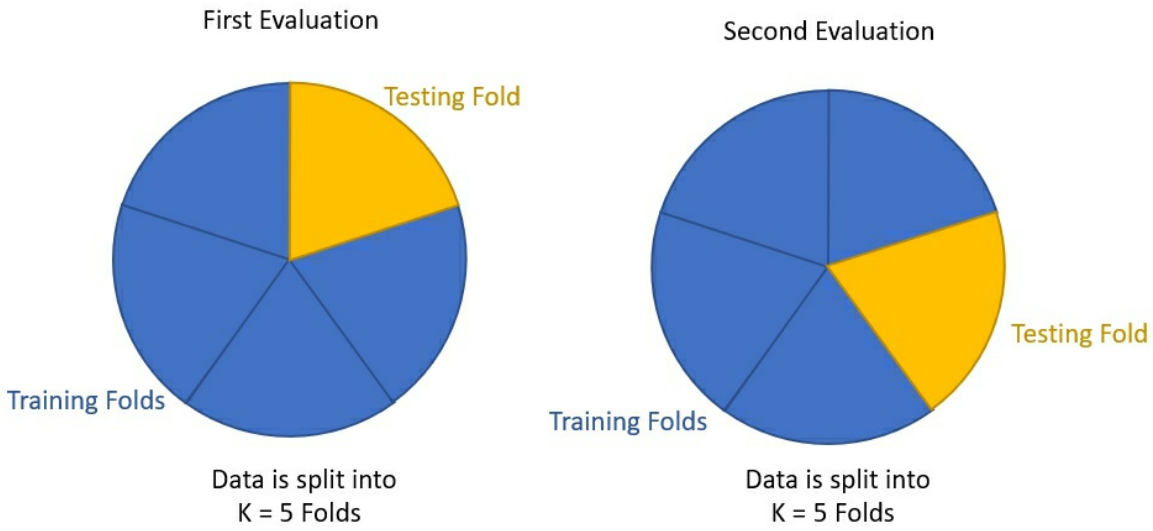
Cross-Validation

By splitting this data into training and testing sets, we aren't making full use of the data as we aren't using the testing data to train the model (if we did, what would we use to evaluate it then?) and we aren't doing any testing on the training data. A way around this issue is to split the data in half, and do a training/testing cycle twice, switching the data from the test set with data from the training set.

By switching these sections of data around we would get two versions of our model, which would give us a better idea of whether our algorithm is

working (we'd have 2 models that *should* have similar accuracy). We can repeat this splitting procedure more times if we want a better idea of whether the model is working or not.

This splitting is called K-fold cross-validation. Once our data is jumbled up, we split it into K folds (say, 5). We train the data on the first 4 folds, and we test the model on the last fold. We can then repeat the procedure and train the model on folds 2-5, and test it on fold 1. Once we have this done as many times as there are folds, we will have 5 models having a measure of accuracy each:



Scikit-Learn provides some functions to do this easily with a one-liner. Here we use *cross_val_score*, which will return to us the score of each of the evaluations. To get the predicted answers back we would use a similar function, *cross_val_predict* instead.

In[9]:

```
# Cross validation
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_clf2, X, Y, cv=3, scoring="accuracy")
print('Accuracy scores for the 3 models are:', scores)
```

Out[9]:
Accuracy scores for the 3 models are: [0.66666667 0.86666667 0.71428571]

If you have a lot of data, you can also make random splits to get your test/train data. It isn't a good idea to use here as we have few data points, but it is a good validation method to know. Scikitlearn calls this kind of cross-

validation *ShuffleSplit*, and is done in the same way as before, except the argument for *cv* is different.

In[10]:

```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_clf2,
                         X, Y,
                         cv=ShuffleSplit(n_splits=5,
                                         random_state=42,
                                         test_size=0.25,
                                         train_size=None),
                         scoring="accuracy")

scores # See scores
```

Out[10]:
array([0.72727273, 0.72727273, 0.81818182, 0.81818182, 0.63636364])

From these two cross-validation methods, it appears that our model gives an accuracy score of 80%+. Knowing the model correctly predicts the outcome 80% of the time seems fine, but with classification problems, we need to be a bit wary.

Measuring Success in Classification

If I made a predictor that disregarded all information by always telling you a company won't go bust, it would perform rather well. With the data that we currently have, it would make a correct prediction 60% of the time, and I wouldn't have to bother with Machine Learning at all, my prediction program would just have one line of code.

We have no information about false positives, false negatives, and so on. There are many ways of evaluating performance in Machine Learning, for classification algorithms asking for something like the mean squared error doesn't make sense, whilst it would make sense for regression algorithms (more than that later).

If we have a table with horizontal columns being the actual outcomes (True, False) and the rows being predicted outcomes, recording the number of cases of True Positives, False Positives etc. in the table, we would have a Confusion Matrix.

The Confusion Matrix

	Actual Positive	Actual Negative
Predicted Positive	True Positive (TP)	False Positive (FP)
Predicted Negative	False Negative (FN)	True Negative (TN)

This matrix is useful to us in understanding the stats of any classification problem, like the effectiveness of a virus test in a pandemic, for example. We will need the confusion matrix to see how well our classifier is performing in predicting corporate bankruptcy. We can get this matrix with the Scikit-Learn library with the *confusion_matrix* function.

In[1]:

```
from sklearn.model_selection import cross_val_predict
# First get the predictions of Trues and Falses
scores = cross_val_predict(tree_clf2, X, Y, cv=3)

from sklearn.metrics import confusion_matrix
# Compare these predictions with the known correct answers
confusion_matrix(scores, Y)
```

Out[1]:

```
array([[23,  8],
       [ 4,  9]], dtype=int64)
```

With these four numbers, there are several useful ways to measure accuracy, notably precision and recall, where precision is TP/(TP+FP) and recall is TP/(TP+FN).

As a performance metric, precision is giving you the percentage actual positives over all predicted positives, which is a good measure if getting a false positive is something you really don't want.

Recall measures the positives you've predicted as a percentage of the total actual positives in your population, so it is a good measure if you really don't want a positive missed, where perhaps you can accept a few false positives.

We will only concern ourselves with the accuracy score as we won't be using classification algorithms in our final investing AI, though the concepts shown so far are enough for a beginner to start simple classification projects.

Hyperparameters, Underfitting and Overfitting

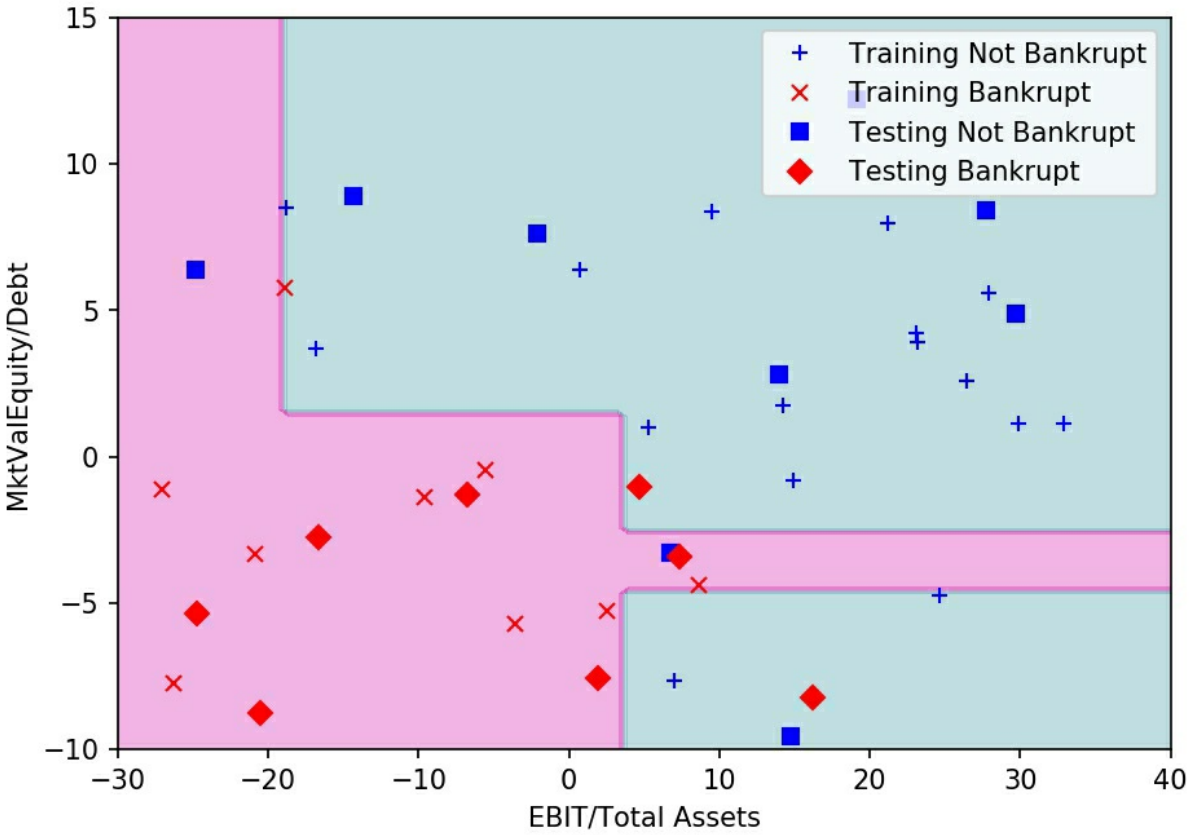
Model-based Machine Learning algorithms have parameters that can be set to change the way they learn. This is distinct from all the other dials that change for the machine to learn as new data comes in during training. For Decision Trees, the main hyperparameter that we can change is the maximum depth of the Decision Tree.

So far, we have fixed the *max_depth* parameter of our Decision Tree to 2 or 3, making a simple Decision Tree that you can easily see in a tree diagram. If we increase this depth our Decision Tree becomes more complex, let's try a *max_depth* of 4:

In[2]:

```
# create a DecisionTreeClassifier object first
tree_clf2 = DecisionTreeClassifier(max_depth=4)
# Fit the Decision Tree to our TRAINING data of X and Y.
tree_clf2.fit(X_train, y_train)
```

Out[2]:

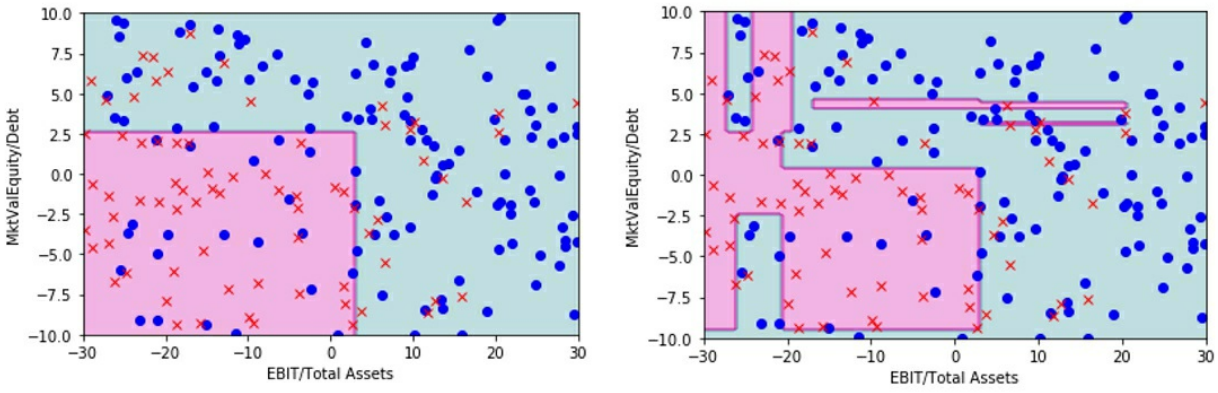


So far so good, the accuracy still looks fine, though we can start to see a strange rule appearing, it appears the algorithm thinks any company with a stock equity/debt ratio of -4 is certain to go bankrupt. Let's try this on our Decision Tree for a larger set of our data, our *Altman_Z_2D_Large.csv* data. On the left is the tree result with a max depth of 2, and to the right with a max depth of 6.

In[3]:

```
# create a DecisionTreeClassifier object first
tree_clf = DecisionTreeClassifier(max_depth=6)
# Fit the Decision Tree to our training data of X and Y.
tree_clf.fit(X, Y)
[…]
```

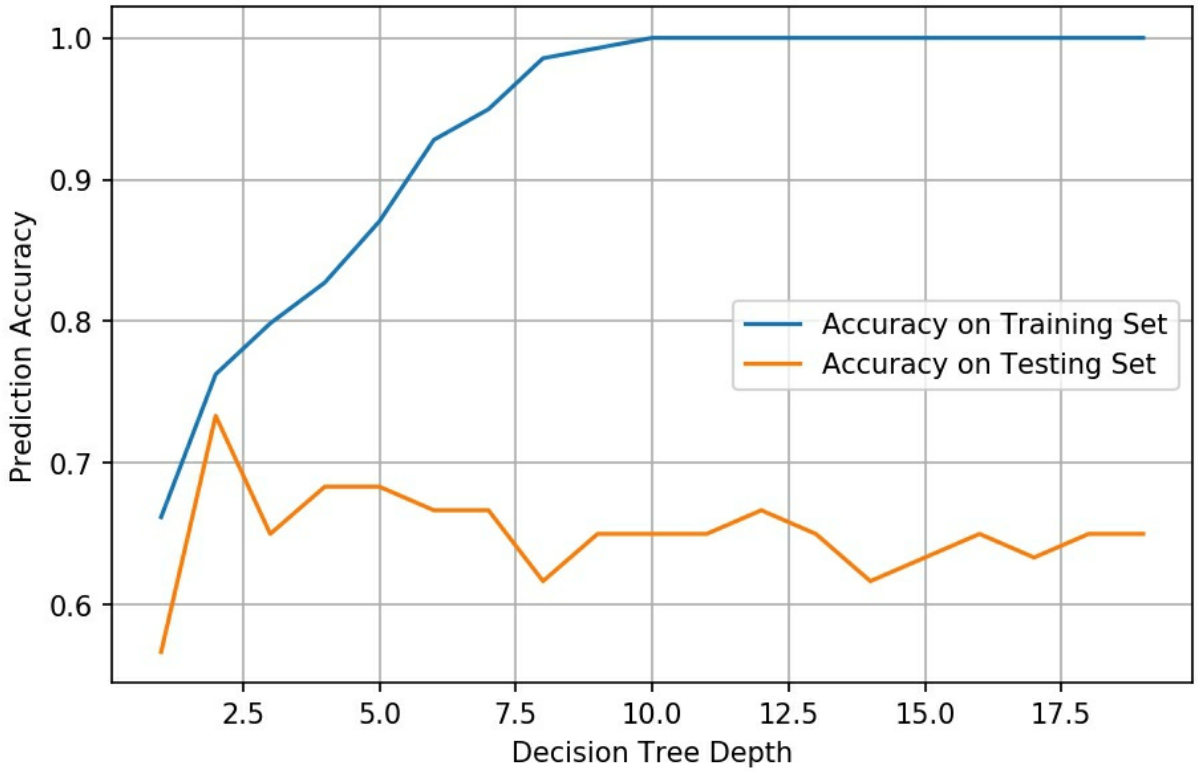
Out[3]:



We can see that as we increase the max depth of our Decision Tree, the error is dropping (less crosses are left out). However, the tree rules start to categorise things too well for our training data, at the expense of the test data, as can be seen in the red areas in our contour plot capturing peculiar areas of our feature space.

When testing out this Decision Tree with a test set, the error that we would get back is going to be worse than the error we would get with the training data. This is called overfitting.

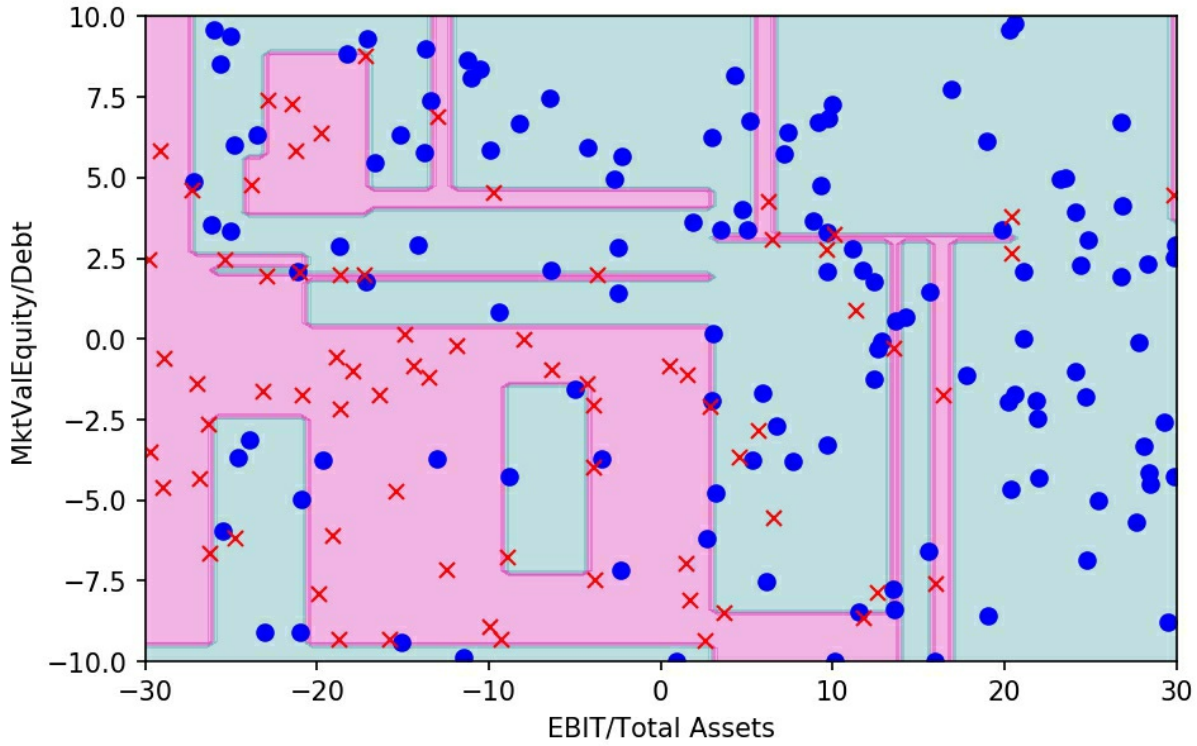
Plotting out the accuracy of our Decision Tree on the training set compared with the testing set when increasing the depth of our Decision Tree yields an interesting result. Notice that the accuracy in predicting the training set results is quite high, increasing as we increase the tree depth. Also notice that the accuracy in predicting the testing set results starts well, but as the tree gets more complex, its predictive accuracy decreases with this set.



This kind of graph, where we observe the performance of a predictor whilst changing one of the hyperparameters is called a Validation Curve. This validation curve is quite useful for us to gain insight into how our model is operating, and for knowing what settings are best for predictive ability.

The accuracy at high tree depth is bad here. This is a classical result of overfitting our training data with too high a Decision Tree depth, rendering our Decision Tree rather worse at prediction on the testing data than we would have thought given how well it matches the training data (where it thinks it makes perfect predictions).

If you were to make predictions with this model, you would much rather have an accuracy of near 0.75 with a Decision Tree depth of 2 instead of an accuracy of less than 0.6 with a more complex Decision Tree. To get an idea of just how bad the overfitting is, take a look at the contour plot when the Decision Tree depth is set to 10:



This kind of result can happen with many Machine Learning algorithms. Your machine is learning too much from the specifics of your training data that it forgets the predictive power of more general trends. If you were using this Decision Tree to inform you, you would fix the Decision Tree depth at two or three, after seeing the validation curve, as we know that the accuracy of your model is at its highest there. The code for these diagrams is covered in the next exercise.

Exercise 7 – Underfitting/Overfitting with Decision Tree Classification

Have a go at the Jupyter Notebook for Exercise 7, where you will have a go at investigating overfitting for prediction of company bankruptcy.

Support Vector Machine Classification

Support Vector Machines are the most complex Machine Learning algorithm we will be using. Here we will only cover the main concepts behind how they work, in sufficient depth to be comfortable adding them to our toolbox of predictors.

The concept of a support vector machine in either regression or classification is centred on an N-dimensional hyperplane which is described by the equation:

$$0 = B_0 + B_1 X_1 + B_2 X_2 + ...$$

Where X_i are the feature dimensions, like the (*EBIT/Total Assets*) feature we trained our Decision Tree with. If we only have two features, this is just the equation of a 2D straight line.

This plane has as many dimensions as features, so if for instance the only features we feed into it is (*EBIT/Total Assets*) and (*MktValEquity/Debt*) then the hyperplane will be:

$$0 = B_0 + B_1 X_1 + B_2 X_2$$

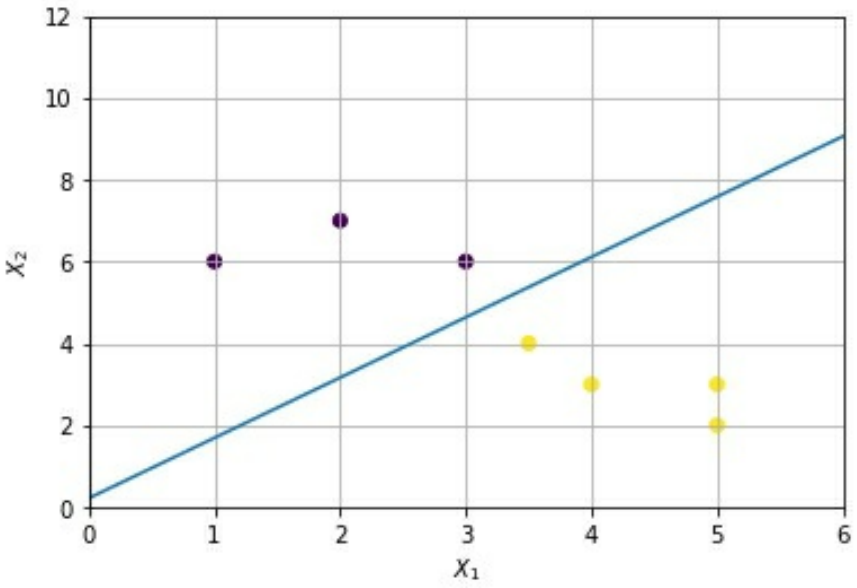
which would be the equation of a line on a 2D graph. Anything above the line is category 1, anything below the line is category 2.

If we take an example point and feed its values of features X_1, X_2 into our classifier $Y = B_0 + B_1 X_1 + B_2 X_2 + \ldots$, if Y is greater than 0, that point is above the hyperplane, and so is in category 1. Conversely, if Y is < 0, the point is below the hyperplane and so is in category 2.

You can use the $Y=mX+c$ formula for a line on a 2D plane of X_1, X_2, the formula would be:

$$X_1 = -\frac{B_1}{B_2} X_2 + \frac{B_0}{B_2}$$

The way this works should be easy to intuit visually, here is an SVM classifier that has been trained on 6 points, with the separating hyperplane plotted:



If there were 3 features, the hyperplane would be a 2D plane in a 3D cube with axes being the 3 feature dimensions. Visualising the hyperplane in more than three dimensions isn't easy. You can have as many dimensions as you want.

The aim with classification support vector machines is to place the line, that is to find the values of B_0, B_1, etc. such that our hyperplane separates the categories. These are the two numbers that are tweaked to fit the data that the model sees until it has learned from all the data we have. Thereafter our machine can categorise items by plugging X_1 values into $Y = B_0+B_1X_1+B_2X_2+$ … and checking whether Y is above or below 0.

The simplest way to find a hyperplane for classification is with a Perceptron, which is a primitive kind of support vector machine.

Let's build a Perceptron. First re-write the hyperplane equation into vector notation. Support Vector Machines are easier to understand using this notation, which you will see later.

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$$

Here b is called the bias, and is the same as our value of B_0 just now. $\boldsymbol{x}$ is our feature vector, which is a row of all our X_1, X_2, X_3, … values. $\boldsymbol{w}$ is the weight vector, which is a row of all our B_1, B_2, B_3, …values. The superscript T means Transpose, which is a matrix operation that just flips our weight vector to be

a column so that with vector multiplication our equation is the same as our original hyperplane equation.

$f(\boldsymbol{x})$ is here to denote that this is a function of $\boldsymbol{x}$ depending on what $\boldsymbol{x}$ we put in. If $f(\boldsymbol{x})>0$ the point is in category 1, If $f(\boldsymbol{x})<0$ the point is in category 2, and the equation of the hyperplane is $0 = \boldsymbol{w}^T\boldsymbol{x}+b$.

We will call the vector of training categories $\boldsymbol{y}$. This is a vector of values being +1 or -1 corresponding to our $\boldsymbol{x}$ vector. The perceptron learning algorithm is as follows:

1. All values in $\boldsymbol{w}$ start as 0. Start b as 0.
2. Go through all our data points.
    a. At each point compare the current $f(\boldsymbol{x})$ prediction (positive or negative) to the actual classification $\boldsymbol{y}$.
        i. If that point is misclassified, then change the values in $\boldsymbol{w}$. $\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \alpha\boldsymbol{x}\boldsymbol{y}$
3. Repeat step 2 until everything is predicted from $f(\boldsymbol{x})$ is in the right classification $\boldsymbol{y}$.

Here are some sample $\boldsymbol{x}$ and $\boldsymbol{y}$ vectors as Numpy arrays, this make up the 6 points you say in the previous graph, ready to have a Perceptron classify the groups:

In[1]:

```
# Can try two different arrangements of x
#x = np.array([[1,2], [3,6], [4,7], [5,6], [1,3], [2.5,4], [2,3]])
x = np.array([[5,2], [3,6], [2,7], [1,6], [5,3], [3.5,4], [4,3]])
y_orig = np.array([-1,1,1,1,-1,-1, -1])
x # View x matrix
```

Out[1]:

```
array([[5. , 2. ],
       [3. , 6. ],
       [2. , 7. ],
       [1. , 6. ],
       [5. , 3. ],
       [3.5, 4. ],
       [4. , 3. ]])
```

The following is the Perceptron algorithm coded up, you may notice that all it is doing is going through each point and nudging our hyperplane in the right direction if the prediction is wrong. This is essentially our model learning to find a good separation by making the current point it is considering fit in the

right category.

In[2]:

```
## Perceptron
L = y_orig.size #Length of y(input) vector

X1 = x[:,0].reshape(L,1)#Separate X1 and X2 to see the algorithm clearly
X2 = x[:,1].reshape(L,1)
y=y_orig.reshape(L,1)

w0, w1, w2 = 1, 0, 0 #Creating our weights, start at 0 for 1st iteration

count = 0 # Counter to go through each point
iteration = 1 # Number of iterations.
# We will just do a fixed number of iterations here

alpha = 0.01 # Learning rate
while(iteration < 1000):
    y_pred =  w0 + w1 * X1 + w2 * X2 # The current prediction
    prod = y_pred * y # Calculate product to see whether correct
            #(If >1, the prediction is correct)
    for val in prod: # go through all of product
        if(val <= 1): # if the prediction incorrect
            w0 = w0 + alpha * y[count] #nudge w vect in right direction
            w1 = w1 + alpha * y[count] * X1[count]
            w2 = w2 + alpha * y[count] * X2[count]

        count += 1
    count=0
    iteration += 1

print('w0',w0)
print('w1',w1)
print('w2',w2)

# Final perceptron answers (less than 0 is category 1, greater than 0 is category 2)
y = w0 + w1*X1 + w2*X2
```

Out[2]:

```
w0 [0.8]
w1 [-1.305]
w2 [0.69]
```

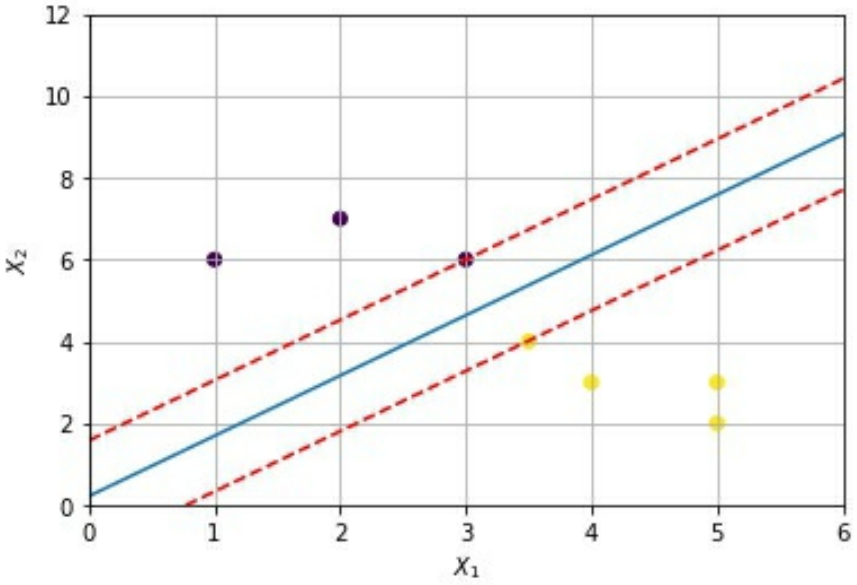
The points that are closest to the separation plane are called support vectors (Though we technically have a perceptron rather than a support vector machine). After some iterations, these vectors are the ones that come to define the split, because the points which are on the correct side aren't going to move the hyperplane, which we see in the *if(val <= 1):* statement.

Equations $0 = \boldsymbol{w}^T\boldsymbol{x}+b+1$ and $0 = \boldsymbol{w}^T\boldsymbol{x}+b-1$ will give us parallel lines that pass through our support vectors. The equations for these lines are similar to the hyperplane equation, they sit just next to it, either side. Plotting these out make it obvious which points are our support vectors that define our separation.

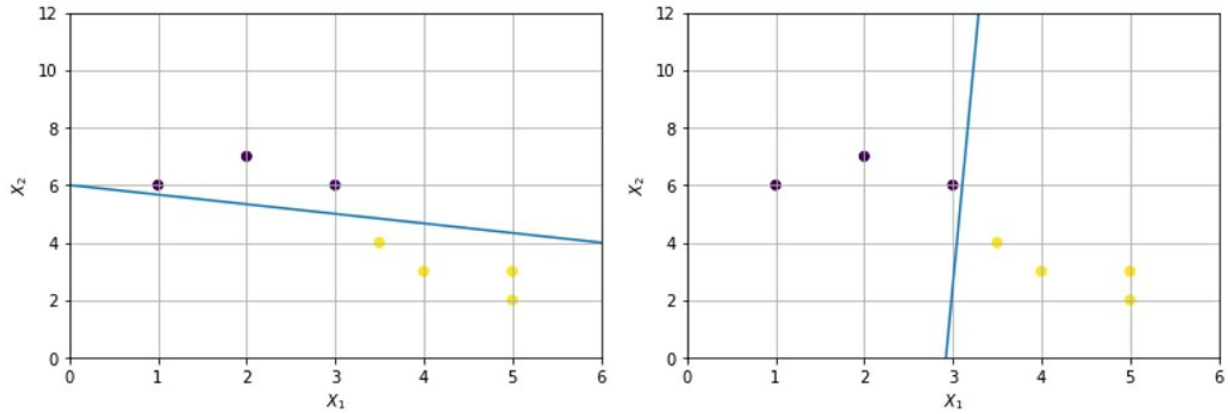
In[3]:

```
# Plot the predictions.
plt.scatter(x[:,0], x[:,1], c = (y<0).reshape(1,-1)[0] ) #c=color, which is the class(greater or less than 0)

# Plot the line
q = np.array([0,7]) # 2 points on x axis.
x_ = -(w1/w2).reshape(1,-1)*q - w0/w2 # Calculated hyperplane (a line)

# f(x) = w.x+b+1 support vector line
x_p = -(w1/w2).reshape(1,-1)*q - w0/w2 - 1/w2

# f(x) = w.x+b+1 support vector line
x_n = -(w1/w2).reshape(1,-1)*q - w0/w2 + 1/w2

plt.plot(q, x_[0])
plt.plot(q, x_p[0], 'r--')
plt.plot(q, x_n[0], 'r--')
plt.xlim([0,6])
plt.ylim([0,12])
plt.grid()
plt.xlabel(r'$X_{1}$')
plt.ylabel(r'$X_{2}$')
```

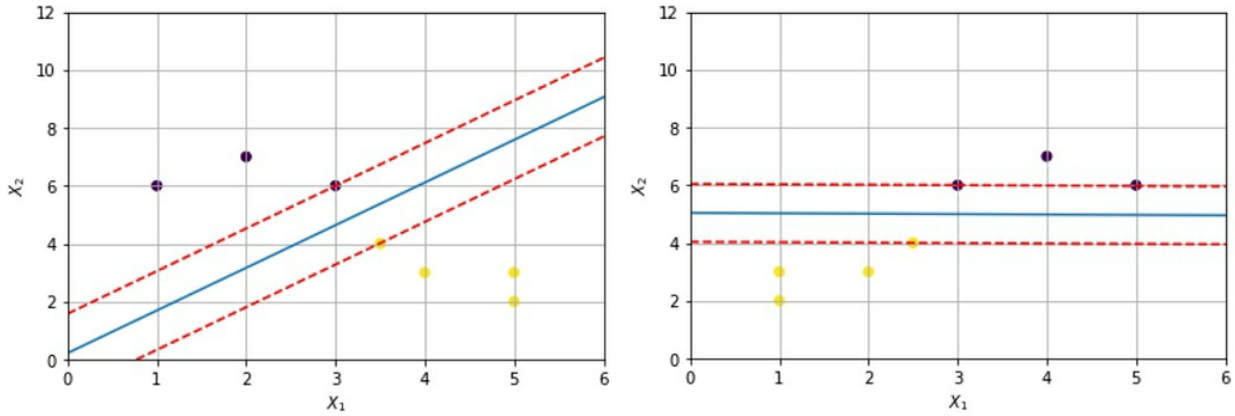
Out[3]:



A problem with Perceptrons is that they settle on the first plane that fits. We could end up with a hyperplane that sits on the edge of a category rather than being reasonably distant from the points of both categories (giving us better predictions).



Support Vector Machines get around this problem by adding a bit more to the basic Perceptron algorithm, adding some components to try and maximise the distance between the support vectors, maximising the distance means maximising the street width you can see when plotting the support vectors with the hyperplane.



The left graph looks optimal, the plane maximises the distance from the support vectors. The right graph is sub-optimal, the distance between the support vectors and the plane can be increased if the plane was diagonal, widening the street. Mathematically.

The next parts of this section will get a bit mathematical, though there is no real need to understand it exactly if you know how to use the predictor from the library, and test its effectiveness. The width of this street is:

$$\frac{2}{\|\boldsymbol{w}\|},$$

which we want to maximise (we won't go over the vector mathematics here). Usually this desire is expressed as a minimisation problem, aiming to minimise:

$$\|\boldsymbol{w}\|^2$$

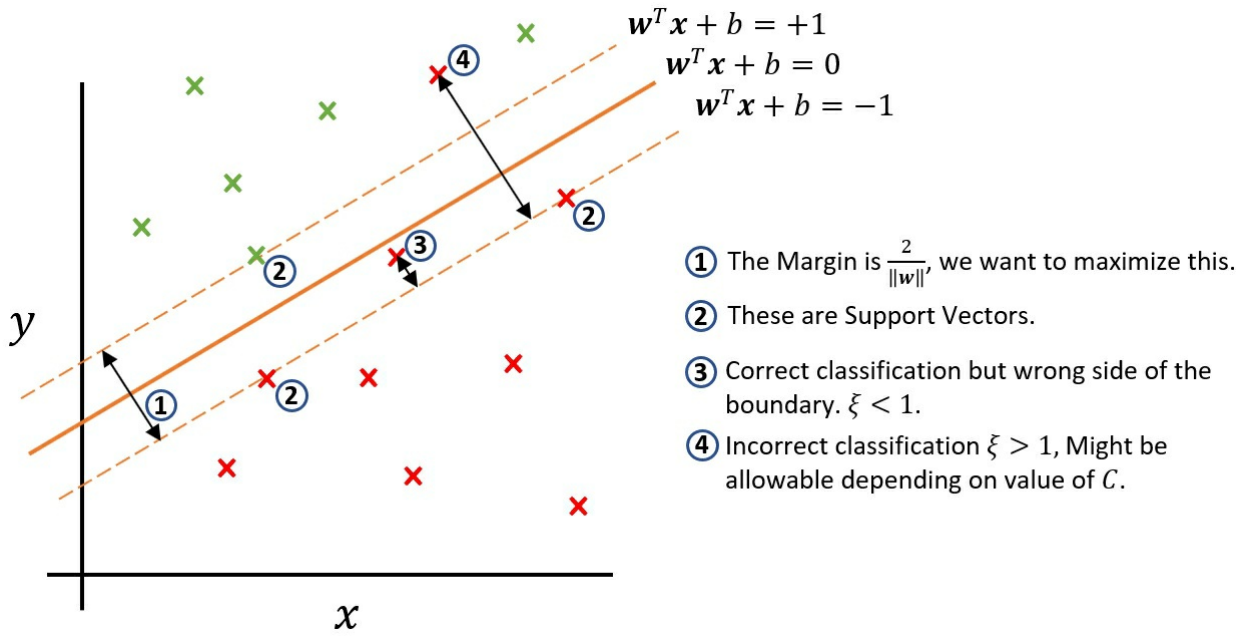
The minimisation problem being solved is expressed in this way in the Scikit-Learn documentation and elsewhere:

$$\min_{\boldsymbol{w}} \|\boldsymbol{w}\|^2 \text{ subject to } \boldsymbol{y}\left(\boldsymbol{w}^T\boldsymbol{x}+b\right) \geq 1$$

Remember the $\boldsymbol{w}^T\boldsymbol{x}+b$ part will output a negative number for category 2 and a positive number for category 1, so multiplying this by $\boldsymbol{y}$ will make this always positive if the hyperplane is correctly placed.

So what essentially being said in plain English is: "Maximise the hyperplane street width (by minimising $\|\boldsymbol{w}\|^2$), but I want the classifier to work perfectly for all the training data (expressed by $\boldsymbol{y}(\boldsymbol{w}^T\boldsymbol{x}+b)\geq 1$) at the same time." You will create a simpler own minimization algorithm from scratch later on.

Support Vector Machines can be made to accommodate "soft margins" where points are allowed to be placed incorrectly. This is done with an additional component to the optimisation problem with slack variables, usually denoted ξ_i, one for each point. For points that are in the right area $\xi_i = 0$, and for any point that is past its side of the street, a penalty is $1 - \boldsymbol{y}(\boldsymbol{w}^T\boldsymbol{x}+b)$. This gives a large penalty for incorrectly classified points that are far out but allows ones near the street.



The minimisation problem stays mostly the same with slack variables added. The value *C* is a hyperparameter which modulates the trade-off between the street width and the slack variable penalty, it can be turned off if you set it to 0.

$$\min_{\boldsymbol{w}} \|\boldsymbol{w}\|^2 + C \sum_{n}^{i=1} \xi_i \quad \text{subject to} \quad \boldsymbol{y}\left(\boldsymbol{w}^T \boldsymbol{x} + b\right) \geq 1 - \xi_i$$

The Scikit-Learn library user guide states the implementation in their SVM code (*https://scikit-learn.org/stable/modules/svm.html*) as:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

Subject to:

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i$$
$$\zeta_i \geq 0, i = 1, ..., n$$

Some variables are labelled differently but it should be easy to get the gist. It is a classification hyperplane boundary wrapped up as a minimisation problem.

Support vector machines need to find the best $\boldsymbol{w}$ and b values for the hyperplane whilst *simultaneously* optimising the street width and accounting for the soft margin. The algorithm combines the two into a single problem to be optimised after a lot of matrix mathematics (so we won't go there).

The use of iteration to minimise error will be clearer in the next chapter where you will create a regression algorithm by coding a minimisation solver yourself from scratch. The minimisation problem for SVM is a lot more complex, but as users of the library we generally just need to know what is being optimised, and that an iterative method is done for us under the hood.

Exercise 8 – Using Scikit-Learns SVM Classifier

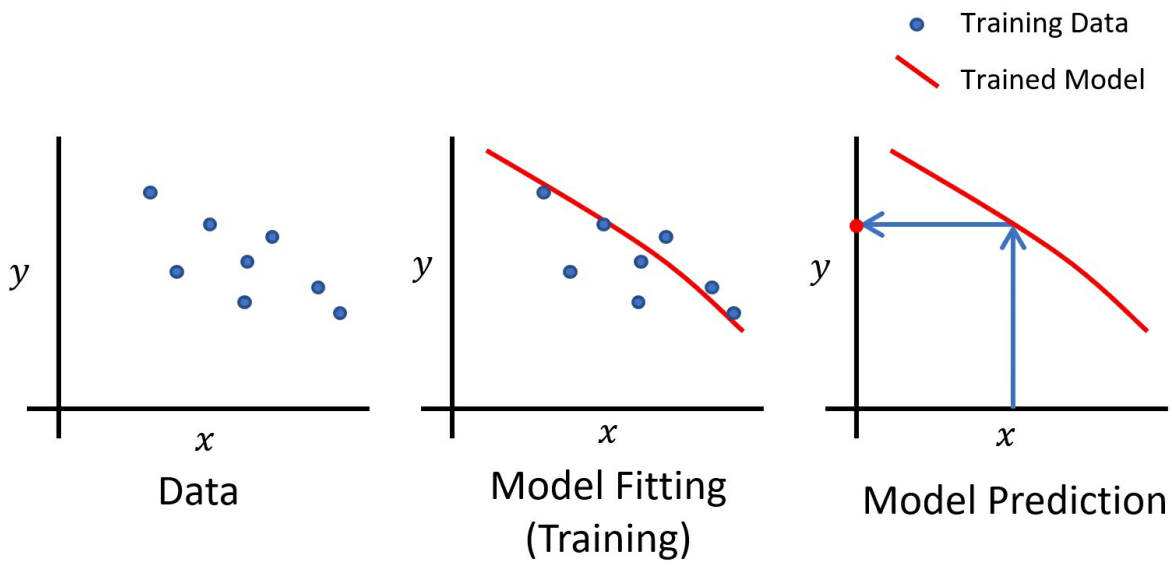
If this is the first time you have seen Support Vector Machines, this might seem overwhelming. Fear not, because using one from the library is straightforward, and the example exercise will give you an idea of how you might use it.

Regression Machine Learning Algorithms

The other kind of Machine Learning algorithms are regression algorithms. Instead of classifying things, these algorithms predict numbers, such as temperature, pressure, price of something etc.

We will be using these in our AI stock picker to predict stock performance, so these algorithms are more relevant to us, however the concepts we learnt with the classifiers carry over to all Machine Learning algorithms (train/test splitting of data, hyperparameters, overfitting etc.). For the investing AI, we will be trying out a group of regressors and picking the best ones to be used for stock selection.

With only one feature, the basic regression models are simple to understand. The model fits the data (learns/trains from the data), and then once it has learnt the broad rules underlying the data, it (hopefully) can make predictions:



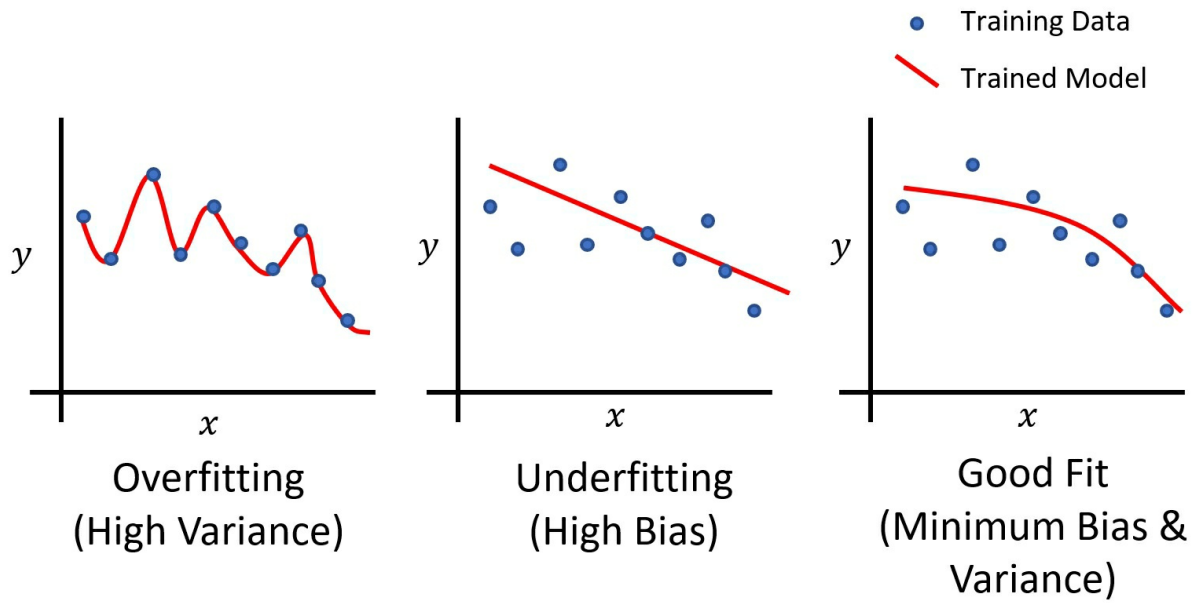
Of course, this looks simple with one feature dimension, but it can be very powerful where there are a lot of features that you want your model to make a prediction from.

When a model is fitted to the data, just like with classification algorithms, there is always the possibility of underfitting or overfitting the data. It has been proven that two components of error relate to this.

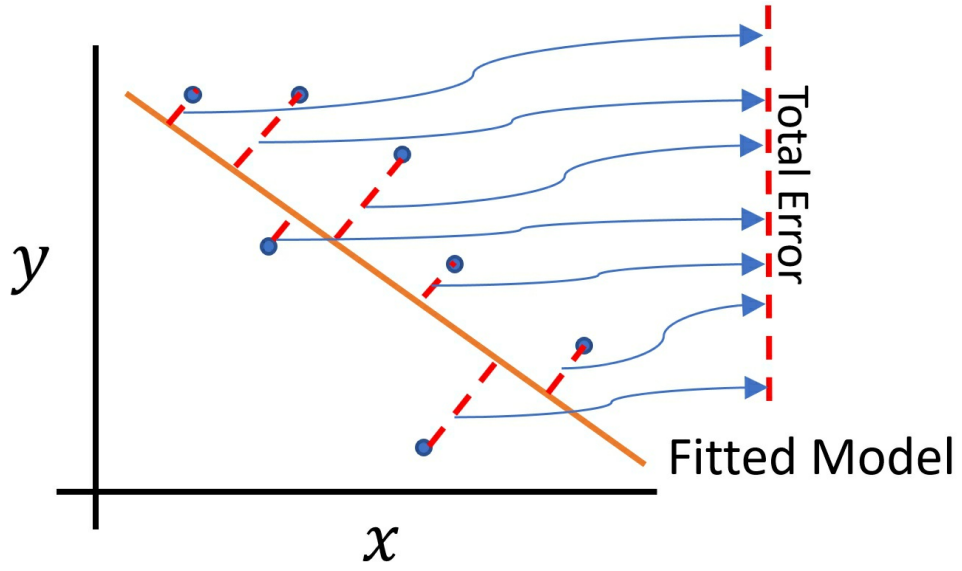
Firstly there is the variance component of error, if your model overfits and accounts too much for the variations in the data, it can't see the forest for the trees. The second component of error is bias, which is a measure of how inflexible the model is to the information.

These errors are somewhat analogous to errors in human thinking. If your model has too much of a bias you will end up with under underfitting, just like people having too biased a view, they don't change their prediction much when new information comes in. With too much variance, it is as if they fully believe everything someone tells them without considering broader context. And of course, if the learning data is not representative, incorrect conclusions may be reached.

A model with high variance is relatively complex, graphically we can see it snaking its way around all the points, and a model with high bias is relatively simple.



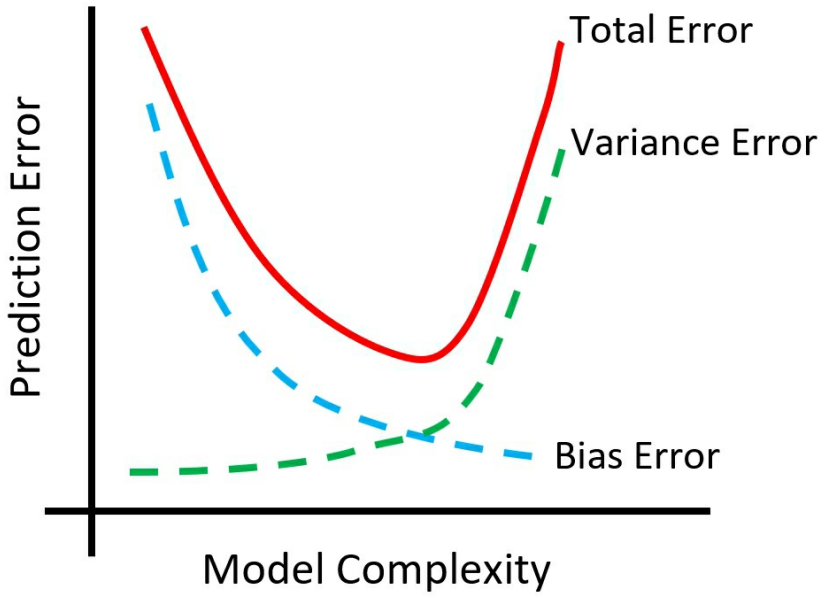
There is a sweet spot between bias and variance that will give the best prediction accuracy of your model. To measure the accuracy of a regression model typically the Mean Squared Error (MSE) or Mean Average Error (MAE) is used. These error measures add up all the errors and find an average in some way:



This diagram isn't exactly what is going on in finding the error, there is still an averaging step required, but it is a good representation to build intuition, and this measure of error would work too.

The complexity of the model needs to be complex enough to accommodate

the variance in the data, whilst staying simple enough to accommodate the broad data trends that are present. The total error is the sum of these two errors:



With this in mind, we know that we need to tweak the model complexity to get the best prediction performance out of them, which is balancing out the two errors.

Linear Regression

Linear regression is the simplest commonly used regression Machine Learning algorithm. It predicts your response variable *y* based on one or more independent variables *X*, fitting a linear relationship between the two. Let's take a look at a simple example, data with one independent variable provided in the *LinRegData.csv* file.

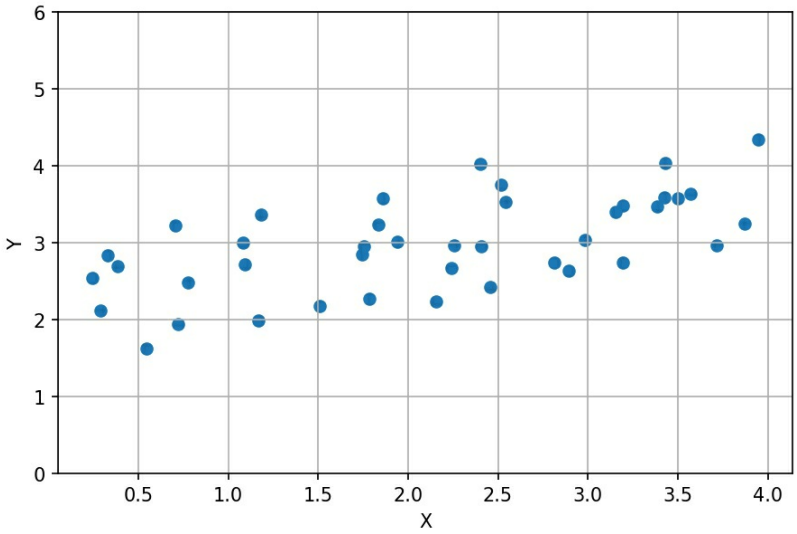
In[1]:

```
data = pd.read_csv('LinRegData.csv') # Read in our data
data.head() # See the first few rows to understand the data
```

Out[1]:

	X	Y
0	3.717656	2.969227
1	2.240286	2.674028
2	2.541955	3.532068
3	3.570501	3.636004
4	0.288653	2.125236

Seeing the full data on a graph:



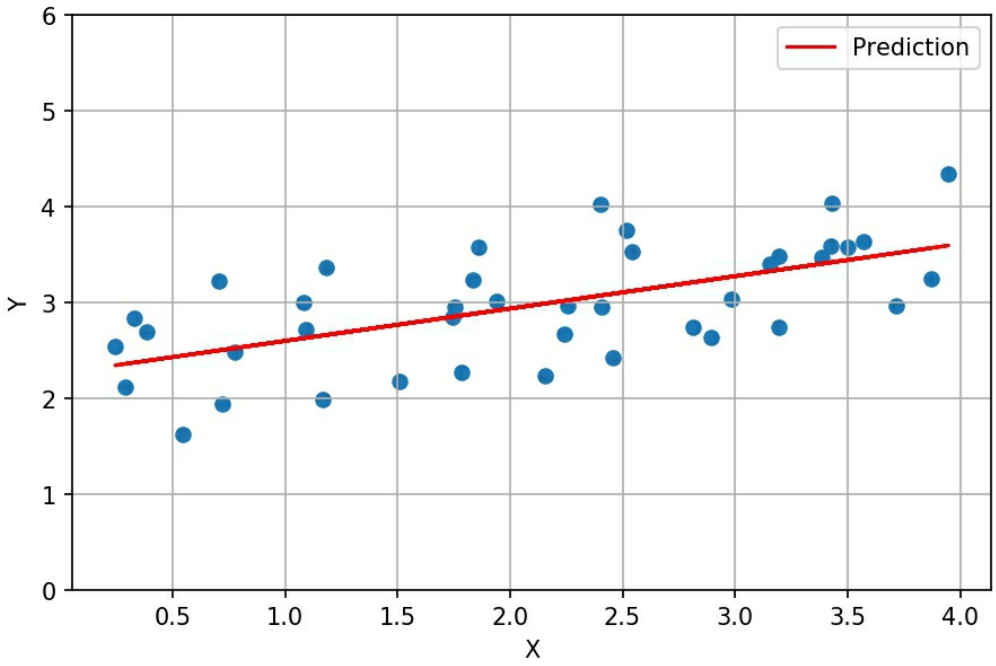
Let's fit a regression model to it. Just like most of the other Scikit-Learn models we import, the *LinearRegression* model from *sklearn.linear_model* can be used straight out of the box with no algorithmic understanding to model this data:

In[2]:

```
# Use Scikit-Learn to do linear regression
from sklearn.linear_model import LinearRegression
linearRegressor = LinearRegression()

# LinearRegressor object wants arguments as numpy objects
x = X.values.reshape(-1,1)
```

```
y = Y.values.reshape(-1,1)
linearRegressor.fit(x, y)
y_pred = linearRegressor.predict(x)
plt.scatter(X, Y) # scatter plot, learning data

# Plot of linear regression prediction on top (red line)
plt.plot(x, y_pred, 'red')

# Plot formatting.
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()
plt.ylim([0,6]);
plt.legend(['Prediction'])
```

Out[2]:



You can clearly see the prediction is a linear fit of the *X*, *y* data to give a prediction. Let's make our own Linear Regressor so we can understand what is happening under the hood.

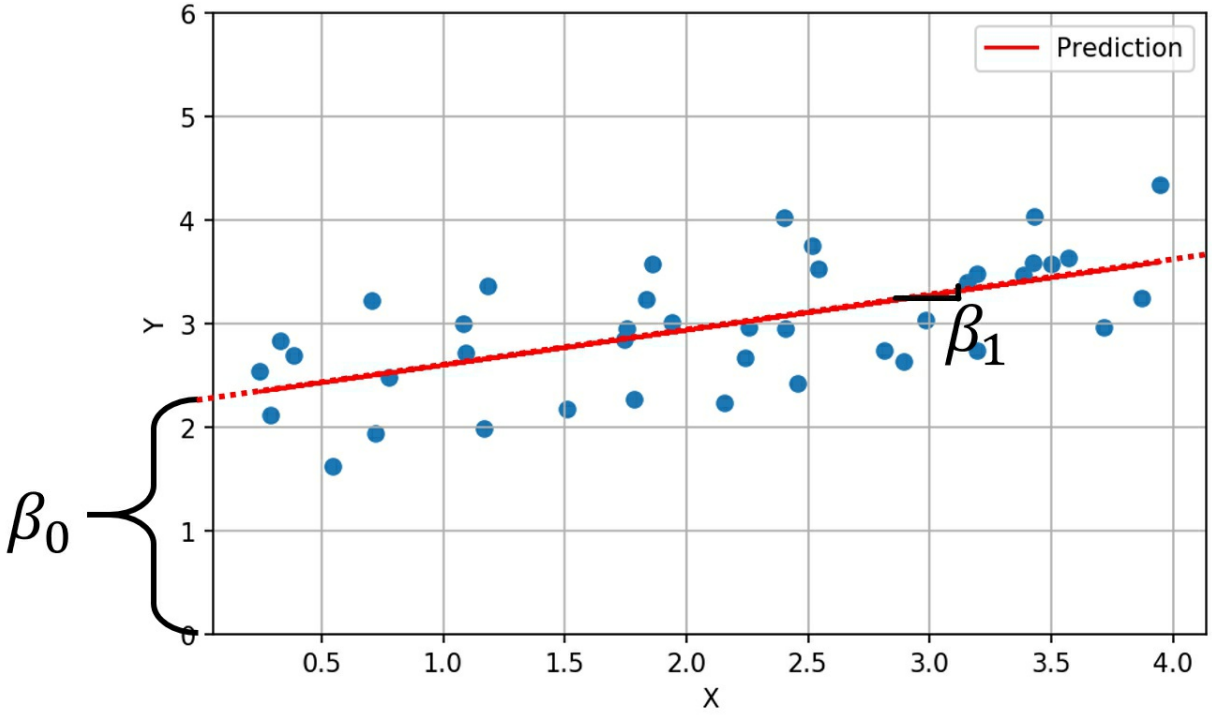
Building a Linear Regressor

For simple linear regression with one feature, the model is fitting the linear equation (where $\hat{y}$ is the prediction of *y*).)

$$\hat{y} = \beta_0 + \beta_1 x$$

To the data by learning the best values of β_0 and β_1 to give as best prediction

of y as it can. This should be familiar from school as the equation for a straight line on in 2D as $y=mX+c$. When making predictions with more than one input feature, the predictor is simply extended $\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots$

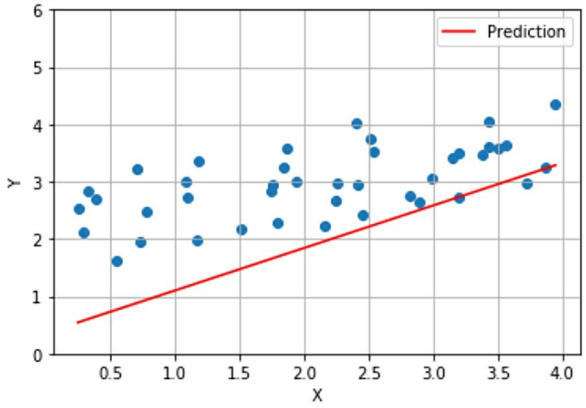


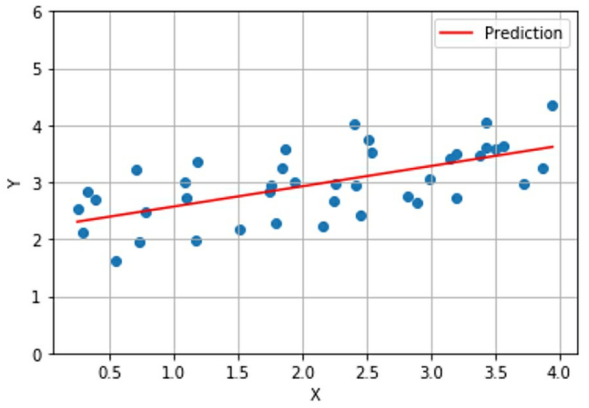
To fit this predictor to the training data we first need a quantitative measure of our prediction inaccuracy. With this measure, we can go about reducing the inaccuracy algorithmically.

This function that quantifies this error is called the *Cost Function*, and typically the mean squared error is used. The mean squared error is as the name suggests, the mean (average) of the square of all the errors.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The Scikit-Learn library has a function that does this for us, *mean_squared_error()* (import with *from sklearn.metrics import mean_squared_error*). Here are two Linear Regressors, one good one bad, with mean squared errors shown, so you can clearly see the kind of errors that might appear.






```
mean_squared_error(Y, Y_pred)
```

```
1.5233268755796858
```

```
mean_squared_error(Y, Y_pred)
```

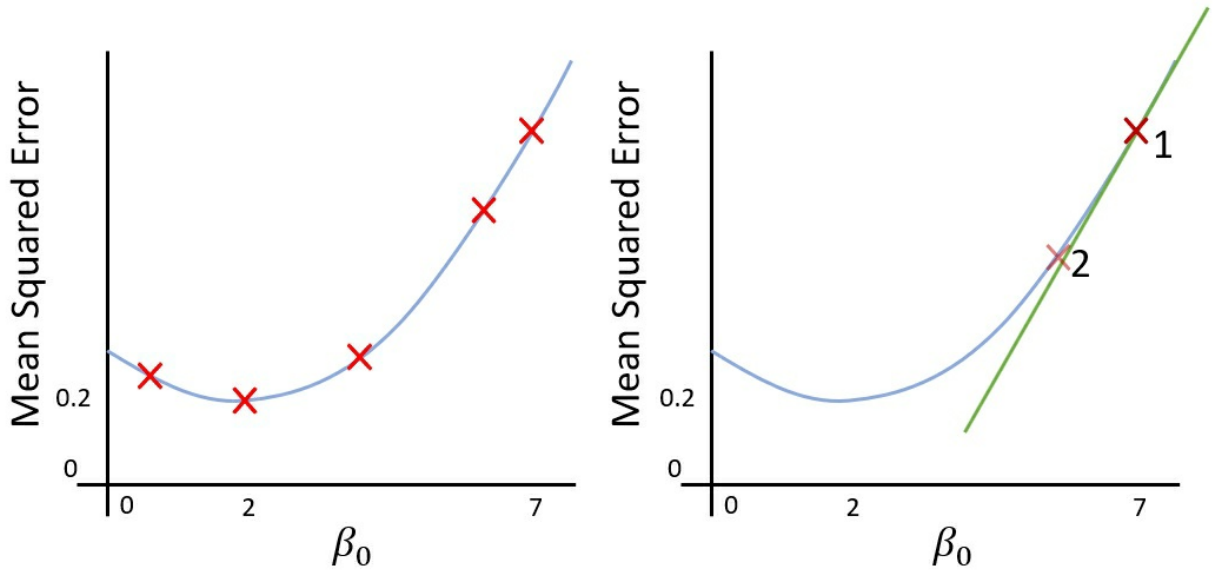
```
0.23076576689233347
```

Notice that with our model there will always be some error. This is the best compromise between bias and variance error we can get with this model.

With the error measure in mind, we can make an algorithm to learn how to tweak β_0 and β_1 to minimise this error. The algorithm will work by first choosing values of β_0 and β_1 (which are very likely going to be wrong on the first try), and subsequently updating them to get a lower mean squared error. At some point, the error can't be reduced any further, at which point we will have our prediction algorithm done ready to go.

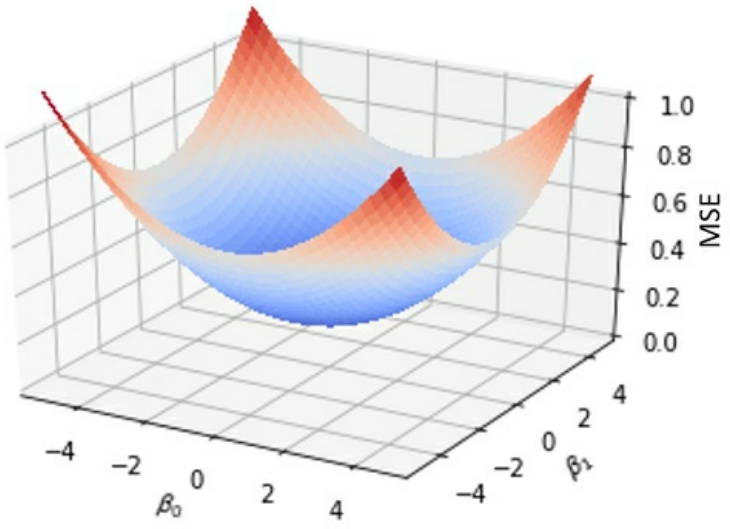
To know a good direction to tweak β_0 and β_1 in we need to use some algebra and differentiation. To aid our intuition of how this will be approached, let's just consider β_0, which we know graphically from the earlier figure. Assume β_1 is roughly correct for the time being. If β_0 is 2, we know that the fit is reasonable, giving a low RMSE. However, we may start with a random number like 7, which would give us a high MSE.

If we were to plot more values of MSE with β_0 we would see a curve. A similar thing would happen with β_1 also. The algorithm aims to reach the bottom of the curve. To do this the *Gradient Descent* method is used.



By differentiating the MSE equation we can find the rate of change of the MSE with respect to β_0 (the green line). As we know that the slope downhill will bring us closer to our answer, we update our value of β_0 (point 1) to be somewhere in the direction of the slope (point 2) and try again.

We need to do this with β_1 as well, as it has a similar effect on the MSE. If we imagine β_0 and β_1 being planar dimensions, their MSE graphs would be cross-sections of a bowl shape in 3D if the third dimension is the MSE:



Luckily, we are only working with 2 dimensions (β_0 and β_1). This would be much harder to visualise with more dimensions. To find our gradient we need

to do partial differentiation (which is basically finding the slope of our bowl along a plane of a fixed β_0 or β_1 value).

If your maths knowledge doesn't extend to partial differential equations, just know that we are trying to get to the bottom of the bowl, which will give us the best values of β_0 and β_1, which makes our line fit best to make predictions. You will see the code for it in a few pages if you don't understand the equations. The MSE (mean squared error) is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Which is the same as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x - y_i)^2$$

The partial derivatives of this error with respect to the values of β_0 and β_1 are:

$$\frac{\partial MSE}{\partial \beta_0} = \frac{2}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x - y_i) = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

$$\frac{\partial MSE}{\partial \beta_1} = \frac{2}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x - y_i)\, x = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\, \beta_1$$

The new value of β_0 will be the current β_0 minus some value relating to the gradient (as we want to go downhill).

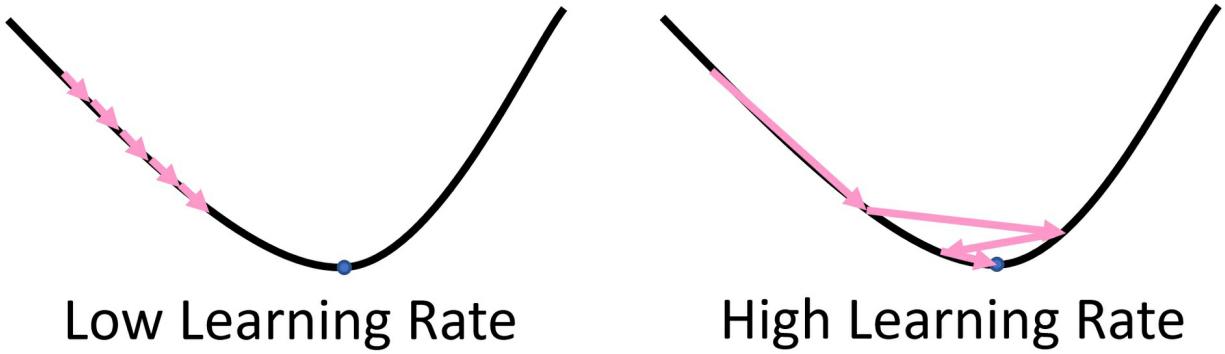
$$\beta_0^{new} = \beta_0^{new} - \eta\frac{\partial MSE}{\partial \beta_0}$$

$$\beta_1^{new} = \beta_1^{new} - \eta\frac{\partial MSE}{\partial \beta_1}$$

Where η is a variable we set, called the learning rate. Finding the bottom of

our 2D space with gradient descent is a bit like finding the bottom of a crater in pitch dark, you can only really use the gradient to guide you and at any point, you won't really know if you have reached the bottom.

If you stop to gauge the gradient at every step, you'll be more sure of it once you've reached the bottom, but it'll take a long time. If you run, you may run past the lowest point between times you run and times you stop and gauge the gradient, but you'll probably be faster. This is a trade-off you have to make with linear regression.



The equations stated earlier can easily be put into code, with a loop to subsequently improve the estimates for β_0 or β_1:

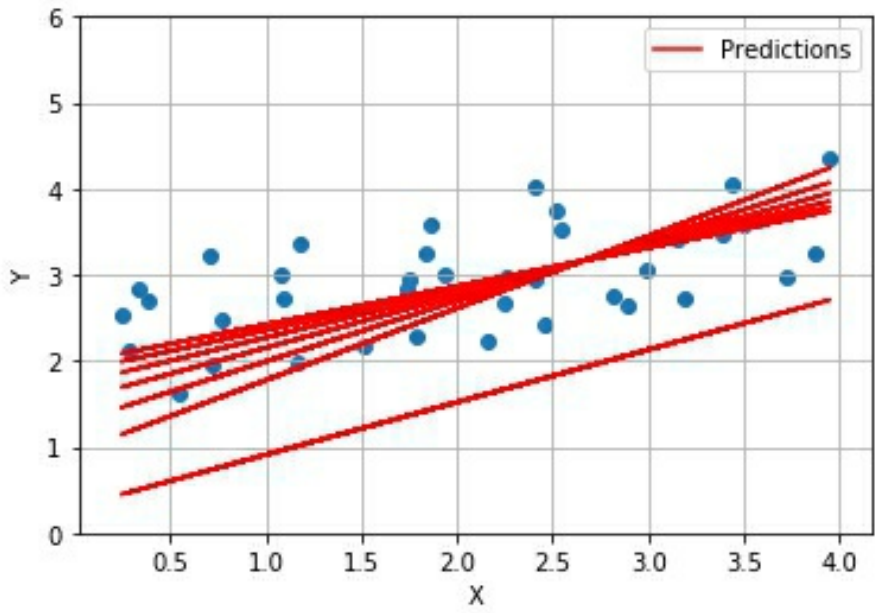
In[3]:

```
# Simple linear regression Gradient Descent
eta = 0.05  # Learning Rate 0.01
iterations = 100  # The number of iterations to perform 1e3
n = len(X) # Number of elements
beta0, beta1 = 0, 0 # start with random numbers

# Performing Gradient Descent
for i in range(iterations):
    beta0 = beta0 - eta * (2/n) * sum(beta0 + beta1 * X - Y)
    beta1 = beta1 - eta * (2/n) * sum((beta0 + beta1 * X - Y) * X)

print('Final values of Beta0 and Beta1 are:', beta0, beta1)

Y_pred = beta0 + beta1 * X # Do the prediction
```

Out[3]:

Final values of Beta0 and Beta1 are: 2.0228762355020855 0.4283872160417386

As β_0 and β_1 change you can see the model learning by plotting the prediction within the loop:



Machine Learning is ultimately an optimisation problem (so is life, if you think about it), so finding a minimum error given adjustable model parameters is a common one. Finding the minimum of a multivariable function is a deep subject by itself, though by building your own gradient descent algorithm it should be clearer what these solvers are doing.

Many solvers can be chosen for most predictors in the Scikit-Learn library. When using the regressors out of the box, the library automatically chooses the best model based on the data.

Exercise 9 – Linear Regression

Download the Jupyter Notebook for Exercise 9 where you will be creating your own multivariable Linear Regressor to predict stock return over a year (from real data!).

Linear Regression – Regularised

In Exercise 9 we started to use real stock data for prediction, and it seemed almost able to predict stock performance, however, only two features were used in the regression: price/earnings ratio and the return on equity. As there are many more metrics that describe a company, and as we want the best prediction we can get with the data we have, we should run our regression on all reasonable features we can get our hands on.

Extra data for the companies in Exercise 9 are provided in *Exercise_9_stock_data_performance_fundamentals.csv*. Let's take a look at the DataFrame keys:

In[4]:

```
data = pd.read_csv('Exercise_9_stock_data_performance_fundamentals.csv', index_col=0) # Read in our data
data.keys()
```

Out[4]:

Index(['EV/EBIT', 'Op. In./(NWC+FA)', 'P/E', 'P/B', 'P/S', 'Op. In./Interest Expense', 'Working Capital Ratio', 'RoE', 'ROCE', 'Debt/Equity', 'Debt Ratio', 'Cash Ratio', 'Asset Turnover', 'Gross Profit Margin', 'Perf'], dtype='object')

```
data = pd.read_csv('Exercise_9_stock_data_performance_fundamentals.csv', index_col=0) # Read in our data
data
```

	EV/EBIT	Op. In./(NWC+FA)	P/E	P/B	P/S	Op. In./Interest Expense	Working Capital Ratio	RoE	ROCE	Debt/Equity	Debt Ratio	Cash Ratio	Asset Turnover	
3400	6.381478	0.204089	10.875822	1.955389	1.043914	507.500000	1.683476	0.179792	0.205403	0.746217	2.340092	0.489970	1.896559	0.
12853	13.063094	0.320015	18.033607	2.593587	2.290410	23.015477	2.739353	0.143820	0.143781	0.619297	2.614733	1.013029	4.911931	0.
11564	22.946430	0.049592	-11.771037	2.470096	0.154558	0.518199	2.304271	-0.209845	0.043381	6.343013	1.157654	0.384641	9.211404	0.
6584	9.117308	0.124919	12.254908	100.000000	0.406767	-200.000000	1.989239	5.000000	0.115158	100.000000	1.003977	0.179304	7.571732	0.
2810	-41.561149	0.418362	20.870461	0.850370	1.098445	39.277419	1.379839	0.040745	0.039916	3.682290	1.271570	1.117836	26.533528	1.
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
5034	83.505399	-0.509146	-179.701327	4.152762	1.604563	0.875874	0.739027	-0.023109	0.016279	4.303501	1.232434	0.453732	7.999567	0.
2151	15.319187	0.223177	22.101117	4.375731	1.225972	29.056159	0.643573	0.197987	0.187116	1.095606	1.912737	0.125216	2.354439	0.
7622	11.633947	0.578705	19.023766	3.430473	3.802736	-200.000000	4.495777	0.180326	0.265560	0.154412	7.476183	1.687236	5.452243	0.
3578	11.929398	1.061325	18.686352	2.997386	1.498863	6.799893	1.222132	0.160405	0.159912	1.318212	1.758603	0.207343	8.240939	0.
2253	10.948177	0.179916	14.896027	6.627833	0.528175	3.420576	0.763239	0.444940	0.167598	5.742966	1.174126	0.154096	2.394690	0.

78 rows × 15 columns

Let's put that data into the Scikit-Learn Linear Regressor and see if it gives a reasonable prediction of the subsequent stock performance with a test set size of 0.2.

In[5]:

```
# Use Scikit-Learn to do linear regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
linearRegressor = LinearRegression()
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=1234)

linearRegressor.fit(X_train, y_train)
y_pred = linearRegressor.predict(X_train)

print('train error', mean_squared_error(y_train, y_pred))
print('test error', mean_squared_error(y_test, linearRegressor.predict(X_test)))
```

Out[5]:
train error 0.12926017769660428
test error 3.8944563775967076

So here we have a problem, the prediction error between the training set and the test set is quite large. The model has overfitted our training data, recall the earlier overfitting diagram, if the model line is passing through every point, the error will be 0, but the prediction for unseen data will be high. This renders the prediction worse than when we just used two features previously.

Learning Curves

A good tool to use in understanding whether overfitting is going on besides the error is a learning curve. A learning curve is simply a plot of the error for the training and the test set (y-axis) as the size of the training set increases (x-axis).

The following is a function that will plot this out with our Linear Regressor. Here we will output the square root of the mean squared error, the RMSE. This is another common error measure which is useful to us here in understanding the error values. It is approximately the typical error you expect from the model (you square root a number you just squared, bringing you back to the number you started with).

In[6]:

```
# Learning curve for any Regressor
def learningCurve(myModel, X, Y, randomState):

    # We will take these lists as output
    testErr, trainErr, trainSize = [],[],[]


    X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                            test_size=0.2,
                                            random_state=randomState)
    trainSize = range(1, len(X))
    for i in trainSize:
        myModel.fit(X_train[:i], y_train[:i])
        y_pred = myModel.predict(X_train[:i])
        trainErr.append(mean_squared_error(y_train[:i], y_pred))
        testErr.append(mean_squared_error(y_test, myModel.predict(X_test)))
    return np.sqrt(testErr), np.sqrt(trainErr), trainSize
```

Plotting the learning curve for our regressor yields some interesting information. The model fit appears to be a good one with ~37 rows used in

the training set, however, when the *random_state* was set to 123 instead of 1234, that bump doesn't look as good.

As we are training and testing this data on less than 100 rows of data, we are getting a some difference in results depending on how the test and train data is sampled from our full set (with *random_state*). Feel free to test out different random states with the Exercise 9 Jupyter Notebook.

Notice that the error versus the training set starts quite low and slowly rises. This shouldn't be surprising as a plot with only 2 points is going to have our straight-line go through them, and as we add more points, the line will be struggling to accommodate all the points, eventually, the error will plateau.

The error against the testing set starts high, as a line that is trained with a few points isn't likely to be a good fit for unseen data, however as it sees more points, it learns more about the data and ends up with a good fit at a plateau. This would be akin to being through the centre of all of our points in the diagram from earlier.

Generally, we are finding a large difference between the error vs. training set and error vs. testing set, but in learning curves what is important is to observe how the model is learning with more experience. As both errors reach a plateau, we are reaching the best we can with the model we are using. The optimal fit will always have some variance error that we can't do anything about.

When there are many observed features in the model, overfitting is something to be wary of, even with linear models. What you would see with overfitting

on the learning curve graph is a low error versus the training set, with the testing set error staying high.

We can overcome this by constraining the model weightings (the β values) in some way within the cost function. These are called regularisation methods. This way we can throw lots of feature data at it (we want it to learn from as much as possible) while having a bit more leeway with respect to overfitting.

Regularised Linear Regression - Ridge Regression

In ridge regression the MSE cost function is edited to become:

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \alpha{\beta_i}^2$$

The value α is a hyperparameter you have to tune, obviously if you set it to 0 you will end up with vanilla linear regression. It is just an extra term to MSE which can be dialled up with a higher α value. This is only used within the model to make it learn from the data differently, measuring errors will still be done with MSE as before.

What this extra term is doing is allocating a squared cost component to all values of beta. Intuitively, it is dampening out values of β_i that are high relative to the others as it iterates to find the minimum. If for example, our bowl had one of the axis values of β_i correlating very high with a feature, the bowl would resemble a gutter, and the gradient descent method would make us drop down into the gutter on that axis and roll down slowly on the other axis.

In higher numbers of dimensions, our gradient descent could get stuck in the gutter for a while and think it has found the optimum when in fact it hasn't. By lessening the impact of a few of the large β_i convergence values we give the other β_i coefficients time to contribute to the final predictor.

Ridge regression is known as L2 regularisation.

Regularised Linear Regression - Lasso Regression

The other regularised regression method is called Lasso regression. Lasso Regression is similar except that the additional β_i cost is the absolute value

only. The value λ is another hyperparameter that has to be set to scale the effect of regularisation.

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\,|\beta_i|$$

This kind of regularisation is known as L1 regularisation. Though the difference in the cost function equations look subtle, they can have large consequences on how the linear model works. With Lasso regression some of the features can be forced to become 0, being eliminated altogether.

Regularised Linear Regression - Elastic-Net Regression

Elastic-Net is a combination of Ridge and Lasso, this gives you a bit more flexibility in tweaking a linear model to make more accurate predictions.

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \alpha{\beta_i}^2 + +\lambda\,|\beta_i|$$

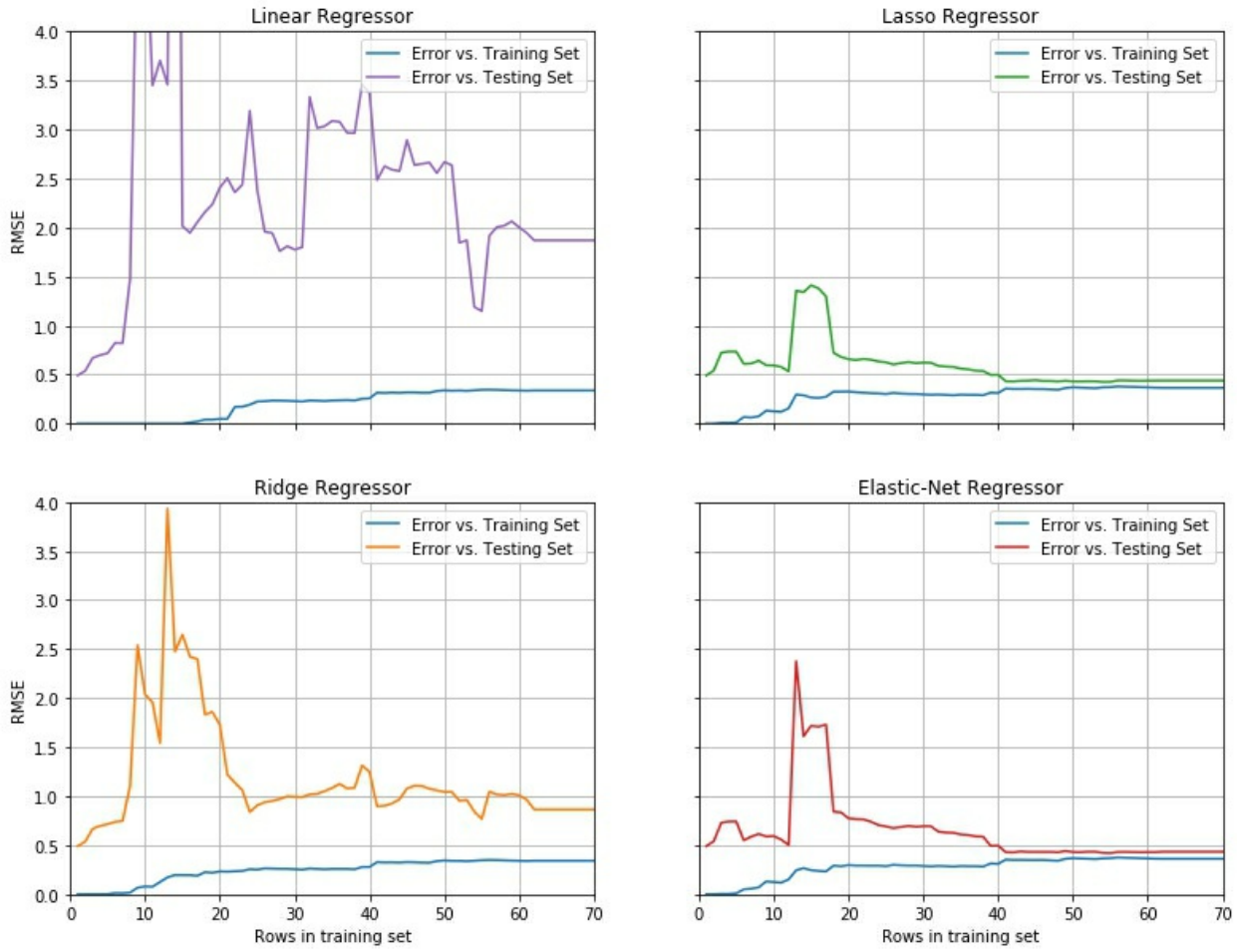
Comparing Linear Models

Let's see if our regularised linear models can do any better with this complex data.

Running all 4 regressors with the same Train/Test sets reveals that we can get improved predictions with regularisation. Train/Test splitting was done with *random_state = 1234*.

Regressor	RMSE vs. Training Data	RMSE vs. Testing Data
Linear	0.129	3.894
Ridge	0.133	0.752
Lasso	0.146	0.116
Elastic-Net	0.144	0.119

Taking a look at the learning curves for all the models:

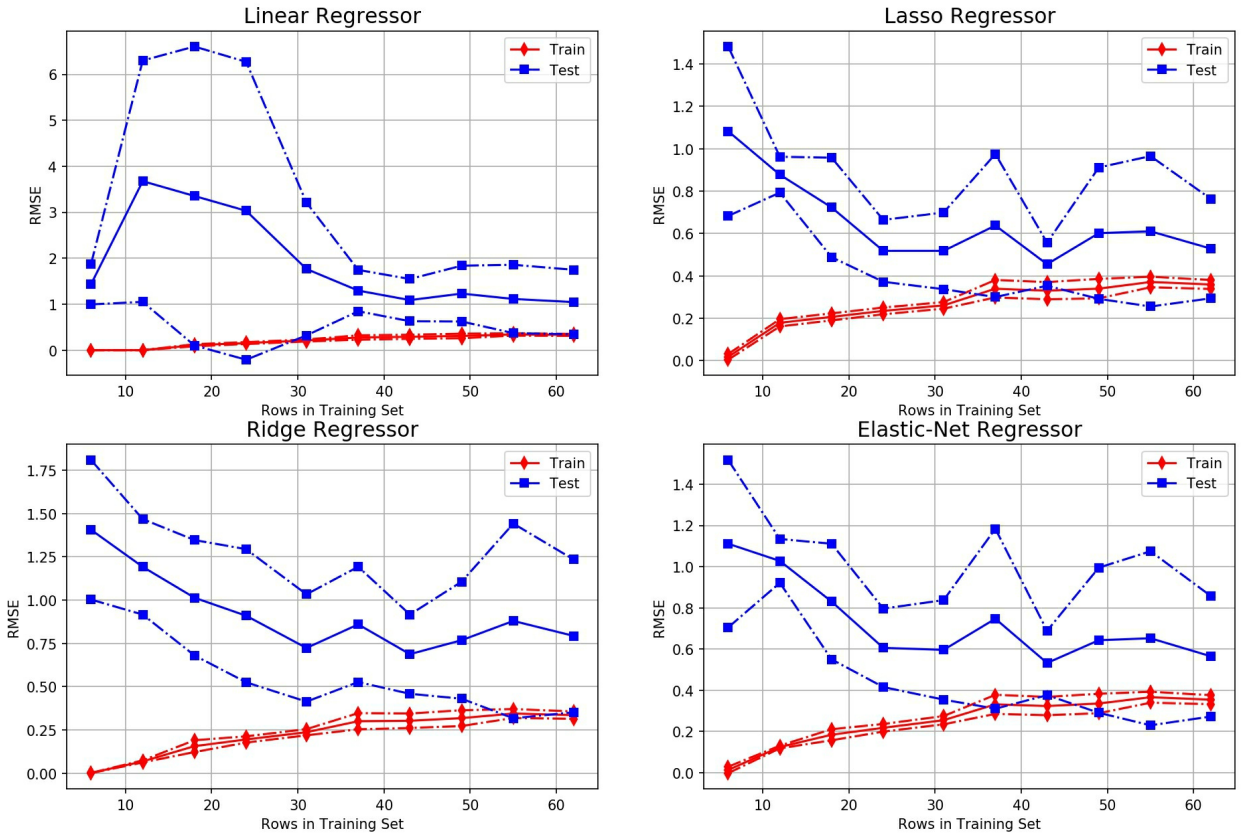


The performances are varied, though the final errors are still relatively high, a RMSE of 0.4 means our typical prediction is 0.4 out, which is quite large for stocks. However, there is a silver lining in that we have gained some insight now: our machines are learning, and we know that some liner models are better than others.

In a project, we would pick the best regressor as our model for predicting stock performance, and then tweak some model hyperparameters systematically (such as L1 and L2 activation λ and α) to get the best predictor we can.

We can see that the learning curves are not smooth, this isn't ideal for knowing how well the models are learning from the data. To help this we could also try a large number of train/test splits to average out the errors over a large number of runs, to get an idea of the repeatability. Example code to do this is in this sections notebook, using the Scikit-Learn *learning_curve*

function rather than our own *learningCurve* function. The resulting plots below are the same plots, but with different train/test split samplings. There is a large distribution of results, but thankfully the test error is decreasing with increasing training instances, so we know our regressor is working.



So far we have only taken 100 rows of company data, and haven't done more than input a hypothetical stock into our predictors, we will get to obtaining, processing and using the full set of this stock data for later on when properly making the investing AI.

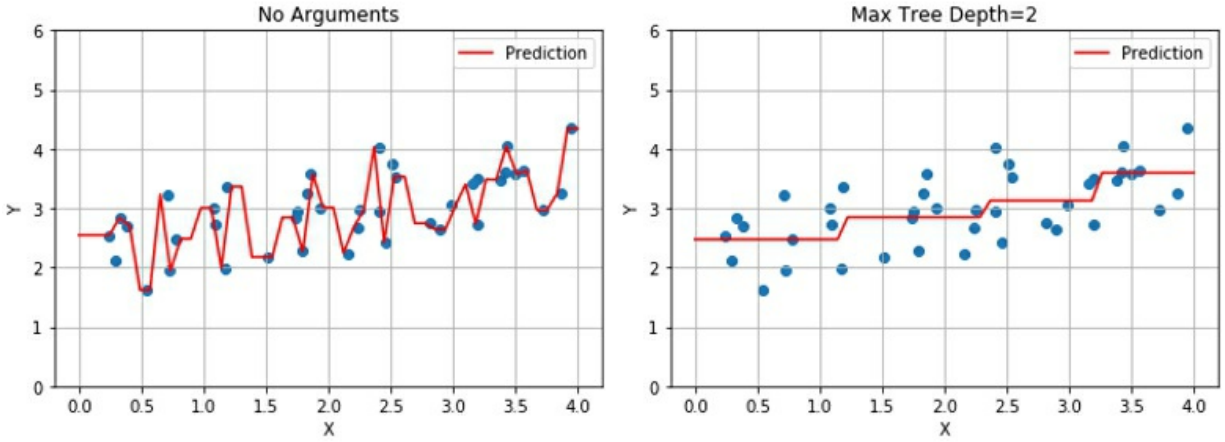
Decision Tree Regression

Decision Trees can also be used for regression predictions. The *DecisionTreeRegressor* object from Scikit-Learn can create a regression Decision Tree for us out of the box. As with the classification Decision Trees, it is a good idea to set the hyperparameter for tree depth directly to avoid overfitting. If we create the Decision Tree regressor without any arguments it tends to overfit, we can clearly see overfitting with our example *LinRegData.csv* data:

In[1]:

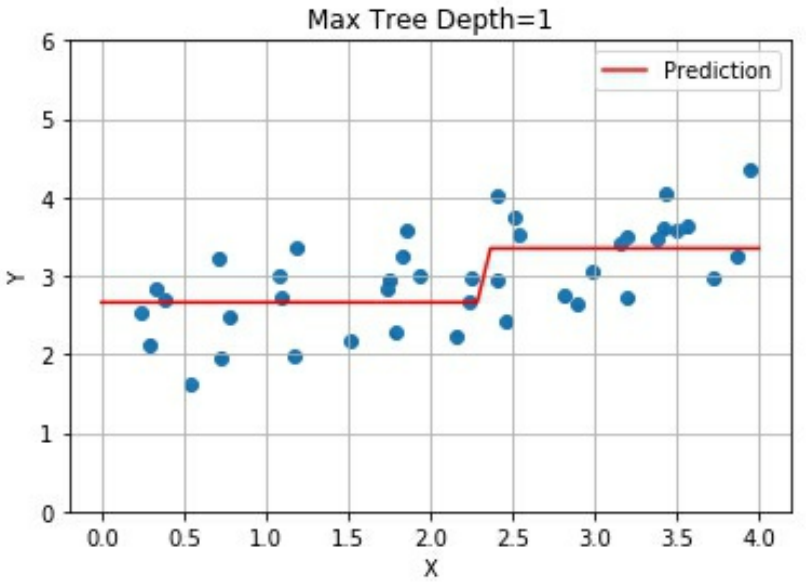
```
from sklearn.tree import DecisionTreeRegressor
treeRegressor = DecisionTreeRegressor()
[…]
```

Out[1]:

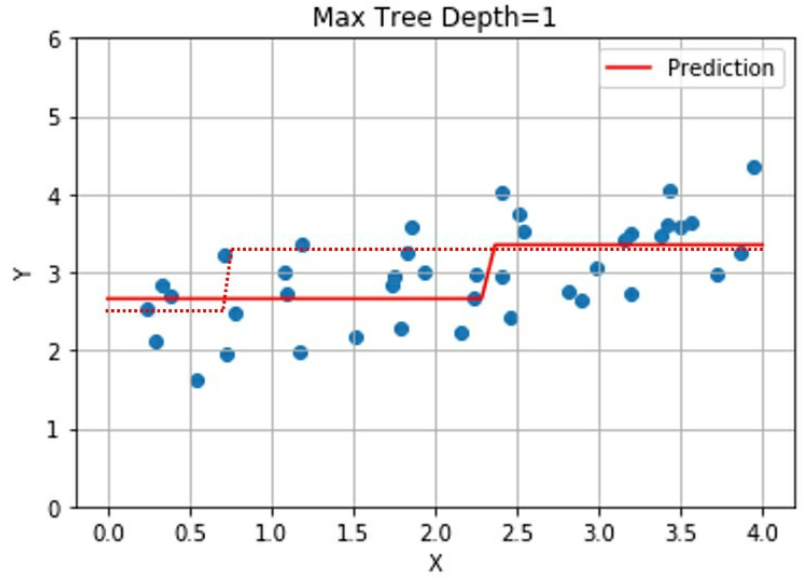


It’s a good idea to look at the learning curves to know when this model is overfitting the training data when there are a lot of feature dimensions to the data you can’t visualise the model like the 2D graphs here.

The CART algorithm for regression differs slightly from the classification version. Firstly, the prediction value will be the average of the training data in each category. Secondly, the splitting algorithm does not make a split based on the lowest Gini coefficient. Instead, it makes a split where the weighted average mean squared error would be lowest. It should be intuitively easy to see with a Decision Tree depth of 1 with our example data.



We can see the tree prediction of the points from *X* at roughly 2 being the average, and above *X=2.5* the average of those points is certainly higher. When moving the split point along the *x*-axis, say if it were more towards *X =1.0* (the dotted line) the weighted average MSE of this split would be quite high. We can try all the splits to find the one of lowest MSE to make the first split with the Decision Tree.



Support Vector Machine Regression

Recall that the classification support vector machine classifies items depending on which side of a hyperplane the item is on, where we allow

some points to be misclassified with the use of slack variables ζ_i, and we want to maximise the street width.

This problem can be flipped on its head and used as a regression algorithm by incentivising the hyperplane to gather as many points as it can in the margin rather than outside the street by changing what the slack variable does. The Scikit-Learn library user guide states the problem as follows (*https://scikit-learn.org/stable/modules/svm.html#svm-regression*):

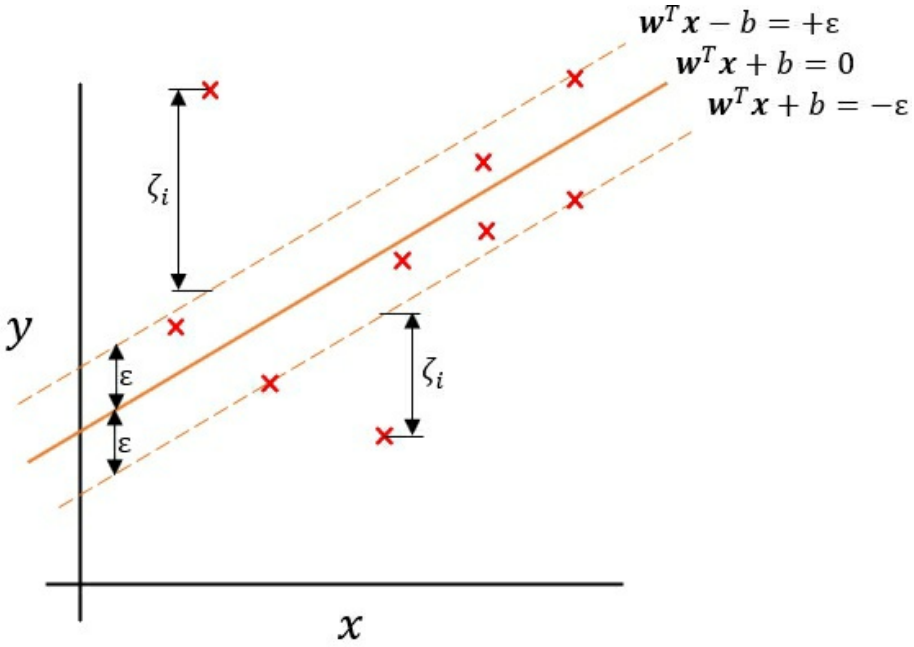
$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)$$

Subject to:

$$\begin{aligned} y_i - w^T \phi(x_i) - b &\leq \varepsilon + \zeta_i, \\ w^T \phi(x_i) + b - y_i &\leq \varepsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* &\geq 0, i = 1, ..., n \end{aligned}$$

Notice now that the inequality signs are ≤ instead of ≥ for the classifier. The optimisation problem is inverted.

The variable ζ_i^* is the same as for ζ_i except it accounts for points below the ε line, and ζ_i only accounts for points above the line. Here ε is a hyperparameter that has to be set, it specifies the width of the street. The values of ζ_i and ζ_i^* are 0 for any points inside the margin and are as large as the distance from the margin, so the penalty is proportional to the distance our point is away from the margin. Variable C is another hyperparameter to be set which gives the trade-off between the tolerance of the outside points (that have ζ_i or ζ_i^* values) and the position of the hyperplane and margin (the street).



As with the support vector classifier, once the optimisation problem is set up, the problem is transformed into a single problem to optimise after a lot of matrix algebra. Much like the gradient descent method we made earlier the process is an iterative one (albeit more complex). As users of the library, we will trust that the library is doing what it says and optimising the stated problem. Of course, we can leave this regressor out in the final regressor comparisons for the investing AI if you only want regressors that you fully understand.

Exercise 10 – Support Vector Regression with Scikit-Learn Library

In the Exercise 10 Notebook, some example code for the Scikit-Learn *SVR* regressor is shown. Non-linear fitting of data with support vector machines is demonstrated as well, through explanation of their operation is out of the scope of this book. Results for non-linear *SVR* are included for more advanced users and those that are happy to use it without understanding its operation.

K Nearest Neighbours Regressor

The K-Nearest Neighbours algorithm is perhaps the simplest Machine

Learning algorithm. No learning is required, in fact, the training data itself is nearly all of the model. It is a memory-based algorithm.

Essentially it is a kind of averaging. For a prediction, the input features are used to find the *K* nearest neighbours (*K* is a chosen hyperparameter integer), after which the average of the *y* result is returned as the prediction.

To find the nearest neighbours a measure of distance is required. For this the Euclidian distance is typically used, that is the square root of the sum of squares of the difference for each feature, a bit like finding the hypotenuse of a triangle. Of course, you can throw as many features as you want at this regressor, which wouldn't give such an intuitive measure of distance beyond three dimensions.

$$Distance = \sqrt{\sum_{i=1}^{n} (X_i - x_i)^2}$$

Where X_i are feature values from the data, and x_i are input features values.

Let's try this out with some data manually to get the idea on a very simple problem. If we want to get a prediction for features x_1, x_2= *5,4*, and we set *K = 4*, and our data is:

Row number	y	X1	X2
0	9	1	8
1	8	9	5
2	2	5	7
3	6	7	3
4	2	3	6
5	4	1	2
6	1	4	5

The Euclidian distance to our input can be calculated in steps:

Row number	y	X1	X2	$(X_1 - x_1)^2$	$(X_2 - x_2)^2$	$Distance = \sum_{i=1}^{n} (X_i - x_i)^2$
0	9	1	8	16	16	5.65
1	8	9	5	16	1	4.12
2	2	5	7	0	9	3
3	6	7	3	4	1	2.23
4	2	3	6	4	4	2.82
5	4	1	2	16	4	4.47
6	1	4	5	1	1	1.41

Now we take the 4 nearest neighbours (rows 6, 3, 4 and 2) and average the value for y:

Row Number	y
6	1
3	6
4	2
2	2
Average:	2.75

So our prediction should be 2.75. Indeed, this is the answer the Scikit-Learn library gives us:

In[1]:

```
# Hand-make example data from worked example
X = pd.DataFrame([[1,8],
                  [9,5],
                  [5,7],
                  [7,3],
                  [3,6],
                  [1,2],
                  [4,5]], columns=['X1','X2'])
y = pd.DataFrame([9,8,2,6,2,4,1], columns=['y'])
KNN = KNeighborsRegressor(n_neighbors=4).fit(X, y)
print('K-Nearest Neighbours prediction value is:', KNN.predict([[5,4]]).ravel())
```

Out[1]:

K-Nearest Neighbours prediction value is: [2.75]

There are a multitude of ways that the distance between two sets of numbers may be measured, such as the Manhattan distance (which is like the streets in

New York). Scikit-Learn implemented KNN with Minkowski distance, which is a generalisation which can be tweaked with a *p* value. It is equivalent to Euclidian distance with *p=2* or Manhattan distance with *p=1* (if you do not set it the default is *p=2*).

Besides the value of *p*, the other obvious hyperparameter that may be tweaked is *K*. It is worth looking at learning curves to find the best hyperparameters.

Ensemble Methods

We have now tried out most of the basic Machine Learning algorithms. To get better predictions out we will try and combine them in some way. To see how this works, consider first the following question. How many jellybeans are in this jar?

(Photo from rob791, pixabay.com)

This question is a common one many children encounter at school for teaching estimation and volume calculation. There are likely to be a variety of answers coming from the children; there are various ways to go about estimating the number of jellybeans, and even if everyone used the same estimation method the interpretation of the visual data underlying the estimation calculation will differ between participants, giving a variety of answers.

However, if you combine all the estimates in some way, for example by averaging them, you often get a better predictor than with any one individual. This was a famous finding by Francis Galton when analysing answers to a contest at a country fair where people had to guess the weight of an ox. This “Wisdom of the crowd” is the basis of ensemble methods, and can be harnessed in several ways, aggregating the answers of different algorithms to obtain a better prediction.

There are roughly two kinds of ensemble algorithms, the first is to aggregate the predictions of several independent algorithms together, such as averaging or voting, the second is to have one predictor give its prediction, and have another take that answer and improve upon it, improving the prediction sequentially with many predictors in a chain, each correcting the priors work.

As an example of the first kind of ensemble algorithm, we will be using Decision Trees, combining the predictions of trees in parallel by averaging, called a Random Forest. As an example of the second kind, combining them in series, we will use boosting. We will be using these to create the investing AI.

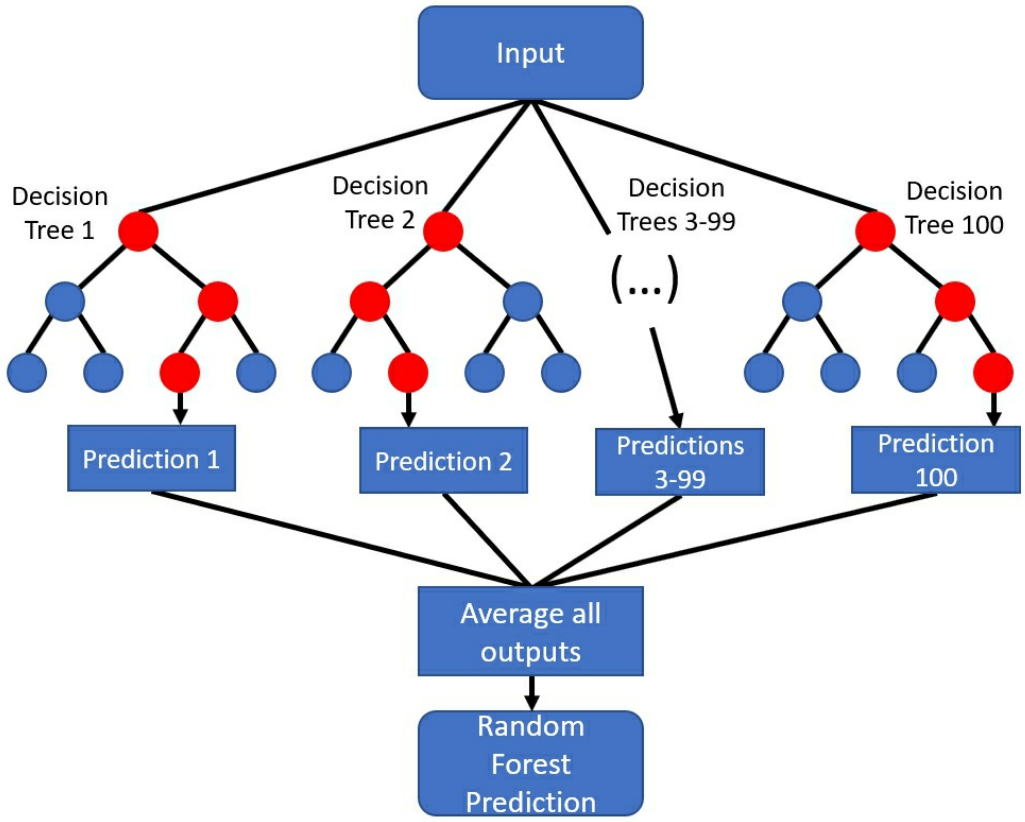
There are a great deal more ensemble methods available in Scikit-Learn which could be explored, e.g. XGBoost, Adaboost, Stacking etc. We will only concern ourselves with gradient boosted Decision Trees and Random Forests.

Random Forrest Regressor

The Random Forest is an ensemble that uses the Decision Tree as a building block, performing a kind of averaging over a large number of Decision Trees to overcome the greedy effects of a single Decision Tree (where it might overfit data rather quickly, it is called greedy).

The data each of the trees are trained on will have to differ in some way, otherwise we will end up with 100 identical trees. To do this, when we sample the training data each of our trees, a tree is trained with a randomly sampled subset of that training data, which might have one third left out, furthermore at the creation of each branch, a random subset of the features are chosen to find the best splits on. When we come to making a prediction, the average of all the trees (all of which will be grown differently) is taken as the final prediction:

$$f(x)_{avg} = \frac{1}{N}\sum_{i=1}^{n} f^{i}(x)$$



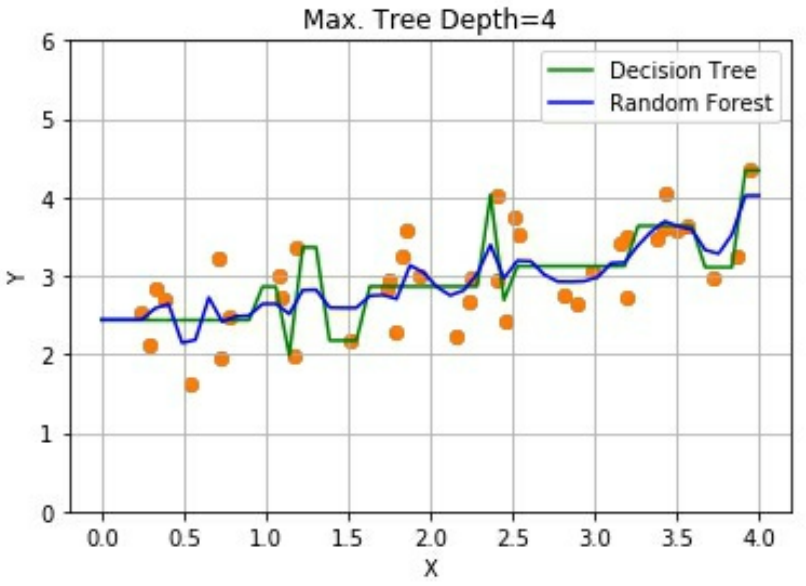
The hyperparameters available to the Random Forest are the same as that for Decision Trees, with the addition of the number of trees in the forest, and the maximum number of features considered in splits.

Scikit-Learn has a Random Forest class we can use directly, the regression version of it is called the *RandomForestRegressor*. Trying this regressor on

our *LinRegData.csv* file makes is quite clear that it is a better predictor than a single Decision Tree when the tree depth is relatively high.

In[1]:

```
# Read in our data, X and Y are series
data = pd.read_csv('LinRegData.csv')

# Scikit-Learn wants arguments as numpy arrays of this shape
x=data['X'].values.reshape(-1,1)
y=data['Y'].values.reshape(-1,1)

from sklearn.tree import DecisionTreeRegressor
treeRegressor = DecisionTreeRegressor(max_depth=4)
from sklearn.ensemble import RandomForestRegressor
RFRegressor = RandomForestRegressor(max_depth=4, n_estimators=100)
treeRegressor.fit(x, y)
RFRegressor.fit(x, y)

# Make the predictions over the entire of X to see the tree splits
x1 = np.linspace(0, 4, 50).reshape(-1, 1)
y_pred_tree = treeRegressor.predict(x1)
y_pred_forest = RFRegressor.predict(x1)

plt.scatter(x, y, color='orange') # scatter plot, learning data

# plot of regression prediction on top (green line)
plt.plot(x1, y_pred_tree, 'green')

# plot of regression prediction on top (blue line)
plt.plot(x1, y_pred_forest, 'blue')

plt.xlabel('X') # formatting
plt.ylabel('Y')
plt.grid()
plt.ylim([0,6]);
plt.xlim([0,4]);
plt.legend(['Single Decision Tree', 'Random Forest (100 Trees)'])
plt.title('Regression Max. Tree Depth=4');
```



We can see that the overfitting that happened with the Decision Tree is less likely with the Random Forest (less overfitting spikes in the prediction), even though each of the trees in our forest has a depth of 4.

Gradient Boosted Decision Trees

Gradient boosting is an ensemble method which combines predictors in a serial manner instead of parallel as in the Random Forest.

This can be made manually with individual Decision Trees. The first Decision Tree makes a prediction, after which a second tries to correct the errors between the prediction and the training data. It does this by taking in as its own training data the first trees prediction subtracted from the original training data, which is called the residual.

As many trees as you like can be strung together in this way. To make a prediction, the corrections of all the subsequent trees are added to the first tree. Gradient Boosting can be done with a variety of base algorithms.

The mechanics of what is happening is easy to visualise in 2D with a single feature and a single variable and individual Decision Trees.

In[1]:

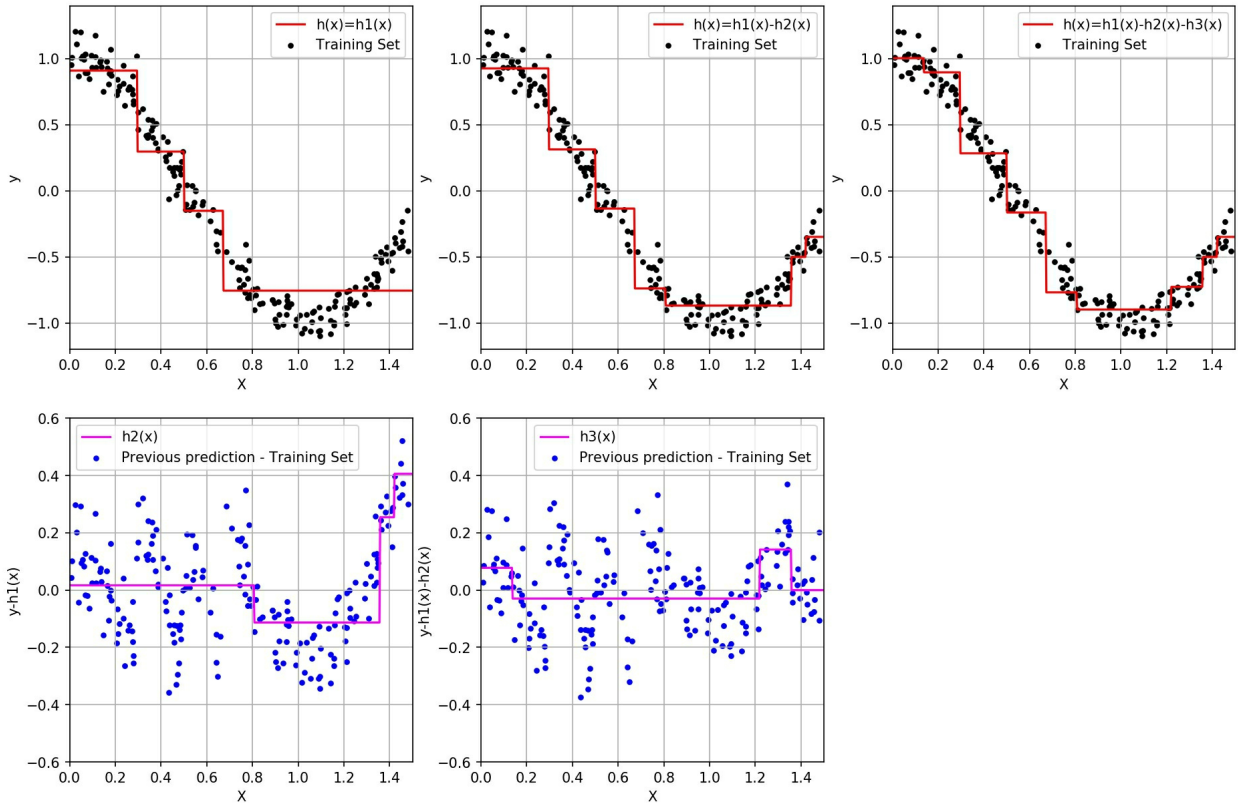
```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=123)
tree_reg1.fit(X, y)
```

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=123)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=123)
tree_reg3.fit(X, y3)

y4 = y3 - tree_reg3.predict(X)
tree_reg4 = DecisionTreeRegressor(max_depth=2, random_state=123)
tree_reg4.fit(X, y4)
```

In[2]:

```
def plotRow1(x, y, x1, y1, limitx, limity, legendDat):
    plt.scatter(X, y, color='black', s=10);
    plt.plot(x1, y1 ,'red')
    plt.xlim(limitx)
    plt.ylim(limity)
    plt.grid()
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend([legendDat,'Training Set'])

def plotRow2(x, y, x1, y1, limitx, limity, ylab, legendDat, lincol, dotcol):
    plt.scatter(X, y, color=dotcol, s=10);
    plt.plot(x1, y1 , lincol)
    plt.xlim(limitx)
    plt.ylim(limity)
    plt.grid()
    plt.xlabel('X')
    plt.ylabel(ylab)
    plt.legend([legendDat,'Previous prediction - Training Set'])

x1 = np.linspace(0, 1.5, 500).reshape(-1,1)
plt.figure(figsize=(15,10))

limitx=[0,1.5]
limity=[-1.2, 1.4]
plt.subplot(231)
plotRow1(X, y, x1, tree_reg1.predict(x1), limitx, limity, 'h(x)=h1(x)')
plt.subplot(232)
plotRow1(X, y, x1, tree_reg1.predict(x1)+tree_reg2.predict(x1), limitx, limity, 'h(x)=h1(x)-h2(x)')
plt.subplot(233)
plotRow1(X, y, x1, tree_reg1.predict(x1)+tree_reg2.predict(x1)+tree_reg3.predict(x1), limitx, limity,
'h(x)=h1(x)-h2(x)-h3(x)')

plt.subplot(234)
plotRow2(X, y2, x1, tree_reg2.predict(x1), [0,1.5], [-0.6,0.6], 'y-h1(x)', 'h2(x)', 'fuchsia', 'blue')
plt.subplot(235)
plotRow2(X, y3, x1, tree_reg3.predict(x1), [0,1.5], [-0.6,0.6], 'y-h1(x)-h2(x)', 'h3(x)', 'fuchsia', 'blue')
```

Out[2]:

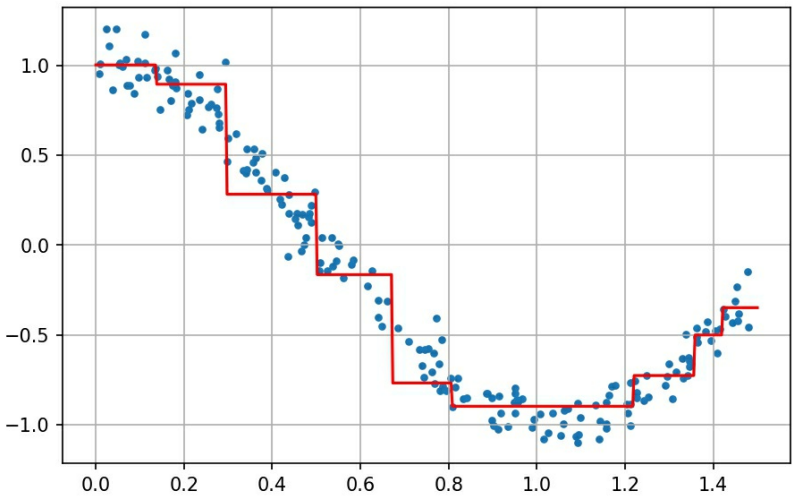


We don't actually have to write all this code to create a boosted Decision Tree, Scikit-Learn comes with an object to do the boosting operation for us, called the *GradientBoostingRegressor*, which is actually identical to the written out code:

In[3]:

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0,
random_state=42)
gbrt.fit(X, y)
plt.scatter(X, y, s=10);
plt.plot(x1, gbrt.predict(x1))
```

Out[3]:



Feature Scaling and Machine Learning Pipelines

Many of the Machine Learning algorithms we have covered are sensitive to the data scaling. For instance, if one of the features ranges from 0 to 1,000,000 and another of the features ranges from -5 to +5, our K-Nearest Neighbour algorithm would be looking very intensely at one feature dimension, and almost not at all in the other when calculating the distances to the neighbours. A similar issue arises in our linear regression gradient descent program. To overcome this problem, we perform feature scaling or transformations to our data before training our models with them.

These scalers generally standardise the data by centring the mean around 0 as well as scaling the variance to be the same across all the features. There are a great number of scalers/transformers that can be applied to a dataset. We will be using Scikit-Learns standard scaler and the power transformer in our investing AI.

Scikit-Learn provides a good overview guide of scalers available that are worth exploring when starting new Machine Learning projects *https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html*.

Standard Scaler

The standard scaler goes through each feature and computes the mean and the standard deviation. After this, it subtracts the mean from every row in the feature and divides the result by the standard deviation. After this, each

feature in our data will be a distribution that is centred around 0 which has a standard deviation of 1.

The equation for the mean is:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Which is just the average. The equation for the standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2}$$

Which is just a measure of how much your numbers differ from each other. Let's try this out with our 300 row stock data.

In[1]:

```
data = pd.read_csv("stock_data_performance_fundamentals_300.csv",\
                   index_col=0)
x = data.drop(columns='Perf')
y = data['Perf']
```

Out[1]:

```
x.describe()
```

	EV/EBIT	Op. In./(NWC+FA)	P/E	P/B	P/S	Op. In./Interest Expense	Working Capital Ratio	RoE	ROCE	Debt/Equity	Debt Ratio	Cash Ratio
count	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000	309.000000
mean	10.723351	0.260785	18.061302	5.034919	8.825682	30.553998	2.507297	0.087411	0.135981	3.058334	2.512936	1.217449
std	72.215214	0.809383	54.378952	13.035311	49.938987	206.738424	2.176221	0.847172	0.142394	11.049073	1.992864	2.046078
min	-500.000000	-5.000000	-500.000000	-50.000000	0.000810	-200.000000	0.051627	-5.000000	0.000000	0.000000	0.487378	0.003299
25%	7.655582	0.070594	9.355681	1.693638	0.906209	-1.509469	1.247292	0.024674	0.037945	0.503998	1.450366	0.181262
50%	12.956057	0.223177	17.356074	2.868679	1.796910	3.674477	1.916651	0.107910	0.110295	1.004943	1.931926	0.531027
75%	19.509391	0.418362	25.520310	4.788118	4.062314	16.537313	2.978365	0.209479	0.181868	2.001237	2.769044	1.372320
max	500.000000	5.000000	359.767138	100.000000	500.000000	800.000000	15.224228	5.000000	1.042717	100.000000	14.806901	15.093623

We can see the mean and the standard deviation for each feature with the *.describe()* pandas DataFrame function.

Notice the mean and the standard deviation are all different. We know our Machine Learning algorithms can be trained on this data and used because we have done it in earlier exercises, but it is sub-optimal.

The Scikit-Learn standard scaler has to be imported for us to use from *sklearn.preprocessing*. It is an object which returns to us a Numpy array of transformed data if we pass our DataFrame into it with the *fit_transform* function.

In[2]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler() # Create Standard Scaler Object
x_s=scaler.fit_transform(x) # Fit the scaler, this returns a numpy array
x_s = pd.DataFrame(x_s, columns=x.keys())
```

```
x_s.describe()
```

	EV/EBIT	Op. In./(NWC+FA)	P/E	P/B	P/S	Op. In./Interest Expense	Working Capital Ratio	RoE	ROCE	Debt/Equity	
count	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.090000e+02	3.(
mean	8.623091e-18	-3.161800e-17	8.623091e-18	1.006027e-16	-5.748728e-18	2.874364e-17	-8.048219e-17	-1.724618e-17	1.322207e-16	2.874364e-18	-2.
std	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.001622e+00	1.(
min	-7.083712e+00	-6.510290e+00	-9.542325e+00	-4.228836e+00	-1.769997e-01	-1.117006e+00	-1.130240e+00	-6.014913e+00	-9.565149e-01	-2.772445e-01	-1.(
25%	-4.254984e-02	-2.353633e-01	-1.603514e-01	-2.567412e-01	-1.588402e-01	-1.553435e-01	-5.799269e-01	-7.417551e-02	-6.896044e-01	-2.315560e-01	-5.
50%	3.096754e-02	-4.654091e-02	-1.298982e-02	-1.664521e-01	-1.409755e-01	-1.302280e-01	-2.718491e-01	2.423638e-02	-1.806786e-01	-1.861442e-01	-2.
75%	1.218620e-01	1.950033e-01	1.373897e-01	-1.896404e-02	-9.553847e-02	-6.790910e-02	2.168127e-01	1.443229e-01	3.227734e-01	-9.582809e-02	1.
max	6.786247e+00	5.864840e+00	6.293981e+00	7.297035e+00	9.851442e+00	3.727871e+00	5.853062e+00	5.808218e+00	6.378127e+00	8.787969e+00	6.

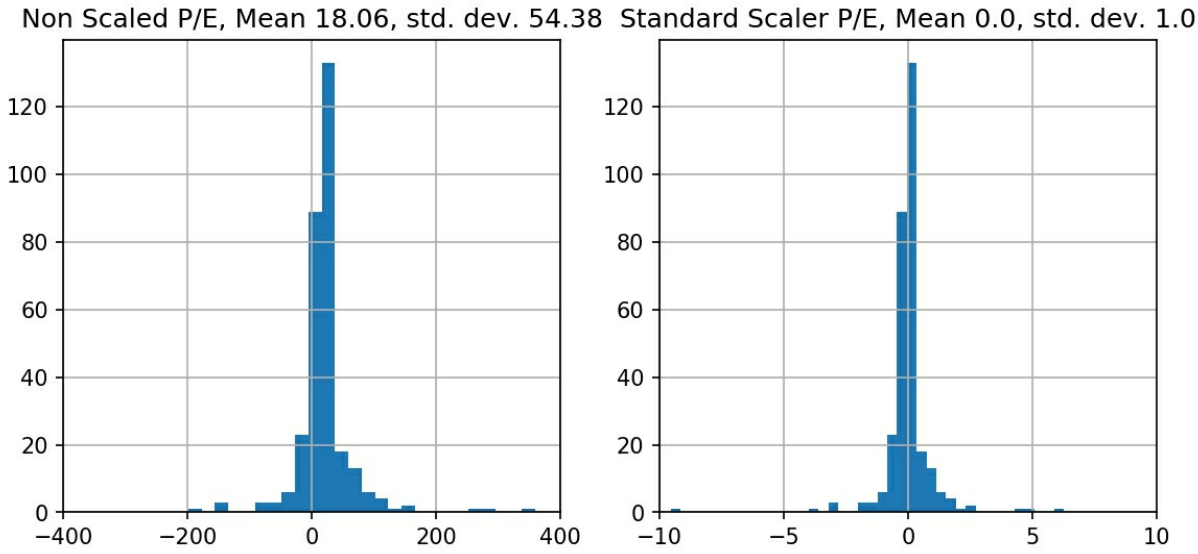
Looking at the *.describe()* table after transformation, notice now all the features have changed to have a mean that is very nearly 0, with standard deviations that are very nearly 1. Plotting this data out as a histogram for a feature makes it visually obvious as to what the standard scaler is doing if you look at the x-axis. Let's plot out P/E and see the before and after scaling disributions.

In[3]:

```
myKey = x.keys()[2] # Can change key number, 2 is P/E ratios

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
x[myKey].hist(bins=40)
plt.title('Non Scaled {}, Mean {}, std. dev. {}'.format(\
        myKey, round(x[myKey].mean(),2), round(x[myKey].std(),2)))
plt.xlim([-400, 400]);

plt.subplot(1,2,2)
x_s[myKey].hist(bins=40)
plt.title('Standard Scaler {}, Mean {}, std. dev. {}'.format(\
```

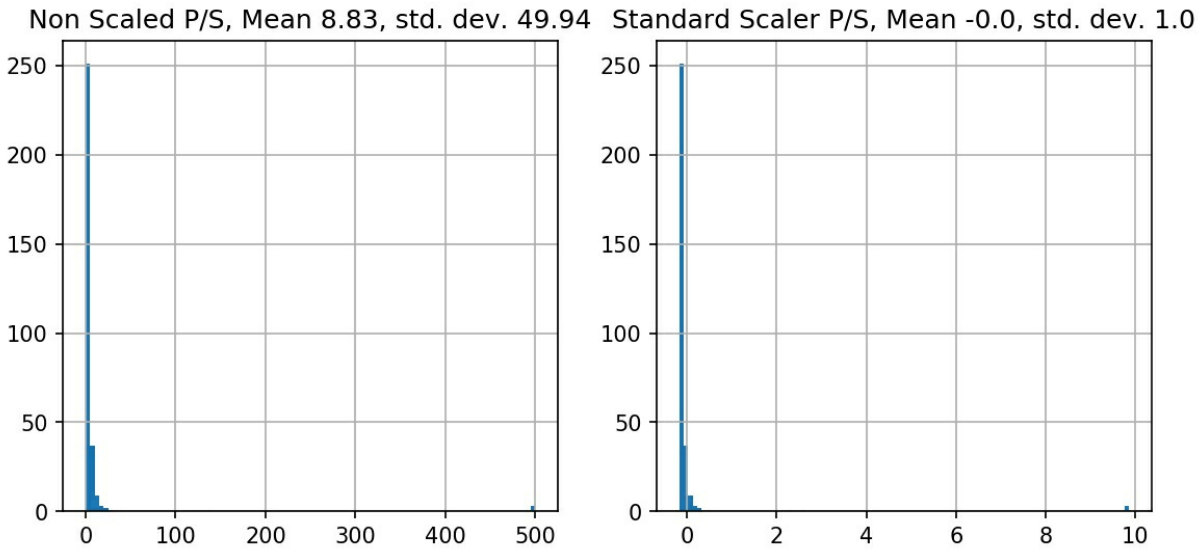
```
        myKey, round(x_s[myKey].mean(),2), round(x_s[myKey].std(),2)))
plt.xlim([-10, 10]);
```

Out[3]:



On these graphs you can see that the x axis range has changed somewhat, and that the peak has been shifted to be right where 0 is. Notice the mean and standard deviation changing (printed out in the plot titles).

There are however some features where using the standard scaler won't give us a nice distribution for our algorithms to work from. Taking at a look at the Price/Sales feature reveals quite an ugly distribution, even after using the standard scaler:



The problem is that there are some extreme values to the far right of the distribution. To clean up the data without excluding the outliers on the far right, we need to transform the distribution.

Power Transformers

Transformers are like the scaler except that they transform our data in a non-linear fashion (not with just multiplication and addition). With them we can get closer to the normal distribution even with starting distributions like that for our Price/Sales distribution. We will use the Scikit-Learn *PowerTransformer*.

Transforms are simply functions where you pass in a series of your variables, and it outputs new transformed variables. They are equations where you can reverse the transform easily. The *PowerTransformer* defaults to the Yeo-Johnson transform, which is written (in the Scikit-Learn documentation and elsewhere):

$$x_i^{(\lambda)} = \begin{cases} [(x_i+1)^\lambda - 1]/\lambda & \text{if } \lambda \neq 0, x_i \geq 0, \\ \ln(x_i+1) & \text{if } \lambda = 0, x_i \geq 0 \\ -[(-x_i+1)^{2-\lambda} - 1]/(2-\lambda) & \text{if } \lambda \neq 2, x_i < 0, \\ -\ln(-x_i+1) & \text{if } \lambda = 2, x_i < 0 \end{cases}$$

Basically, our input feature values are x_i, and the result of the transformation is $x_i^{(\lambda)}$, where the value of λ is a number we can tweak (usually between 5 and -5). When using the Scikit-Learn power transformer the value of λ is chosen to try and make our resulting distribution as close to the normal distribution as possible.

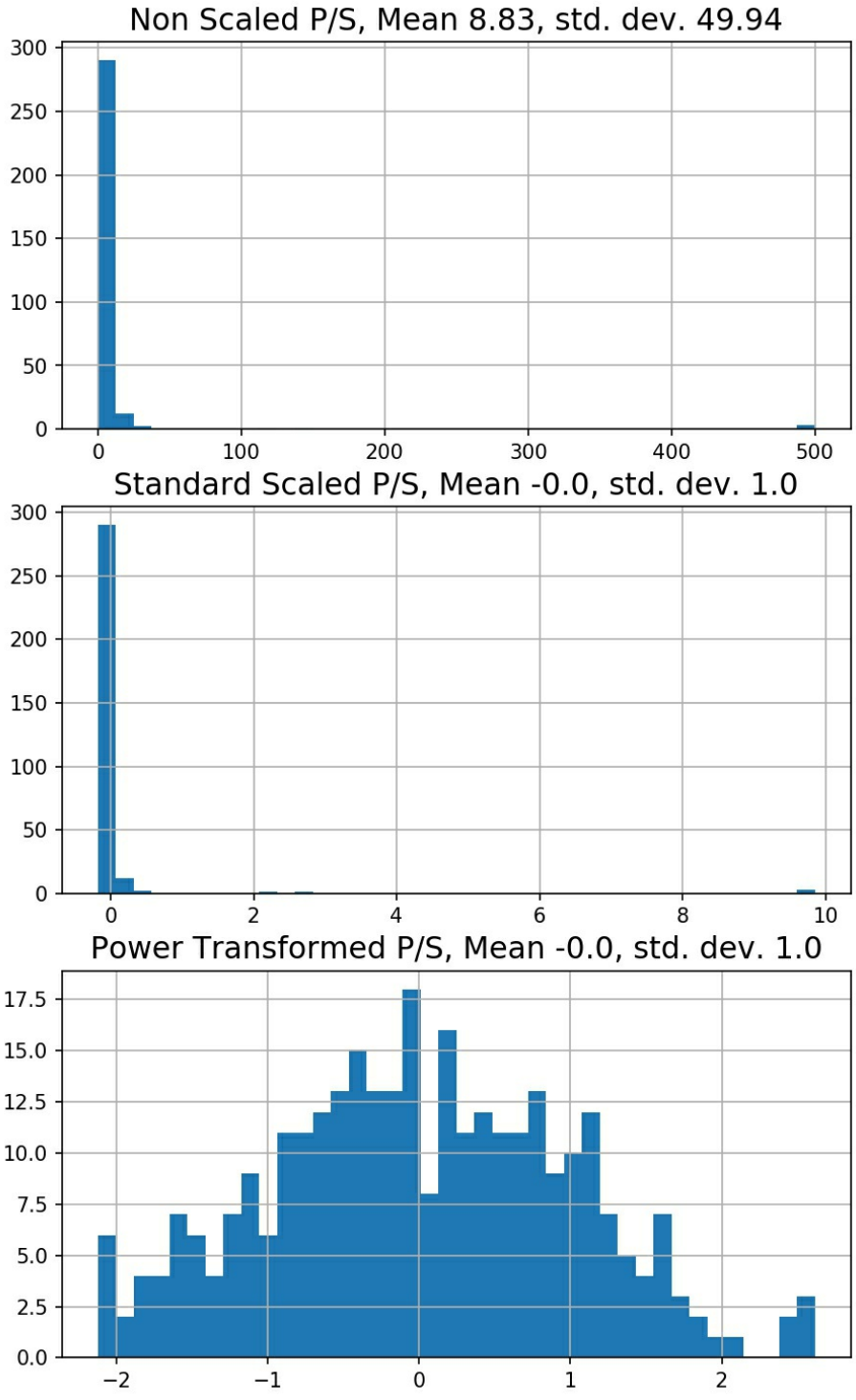
The *PowerTransformer* is used in much the same way as the standard scaler

In[4]:

```
from sklearn.preprocessing import PowerTransformer
transformer = PowerTransformer()
x_t=transformer.fit_transform(x)
```

The difference in the resulting distributions is really obvious, the power transformer certainly makes our price/sales data more gaussian bell-curve

looking, better for our algorithms to learn from.



Scikit-Learn Pipelines

Often we will want to apply scalers/transformers and models in sequence,

which we call a pipeline. Scikit-Learn supplies a pipeline object that we can put our models/transformers into so that we can treat the entire pipeline as a predictor without needing to remember to put in the transformation processes or the sequence of models each time we use them.

In[1]:

```
my_rand_state = 42 # Try and random state for splitting the data and see the difference in test error.
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=my_rand_state)

#Standard Linear Regressor
linearRegressor = LinearRegression()
linearRegressor.fit(X_train, y_train)

print('train error',\
      mean_squared_error(y_train, linearRegressor.predict(X_train)))
print('test error',\
      mean_squared_error(y_test, linearRegressor.predict(X_test)))
```

Out[1]:

```
train error 0.25596952740060863
test error 0.1478164012720238
```

In[2]:

```
#Standard Linear Regressor With PowerTransformer Pipeline
pl_linear = Pipeline([
    ('PowerTransformer', PowerTransformer()),
    ('linear', LinearRegression())])

pl_linear.fit(X_train, y_train)

print('train error',\
      mean_squared_error(y_train, pl_linear.predict(X_train)))
print('test error', \
       mean_squared_error(y_test, pl_linear.predict(X_test)))
```

Out[2]:

```
train error 0.2623233238134826
test error 0.12394151376001815
```

Chapter 4 - Making the AI Investor

Now that we have a good understanding of the models that we need from the Scikit-Learn library, we can put the concepts together and go about formulating an investing AI based on them. We will approach the problem from a fundamental perspective, as we wish to invest in stocks rather than trade them, using publicly available company financial data and the current market cap to judge the quality of an investment.

Classically, the way to invest from a fundamental perspective is to first value a company through quantitative or qualitative means, and then purchase the stock when the market cap of this company is below this value. The purchase should be made at a sufficient margin of safety to compensate for valuation mistakes and the vicissitudes of the market, selling when there is no margin of safety for a profit.

We can't directly go about value investing as our Machine Learning tools don't improve our ability to value a company, however, we do have stock market cap data (company market valuation) over time and we do have public stock fundamental financial data. What we can do is correlate the change in company value to the fundamental financial data, assuming that the value of a company discerned from financial data will likely be realised after a certain time frame.

In investing, the time needed for value to be realised by the market is known to be quite long, commonly of the order of quarters. For the investing AI in this book, we will make our lives easier by setting the algorithms to predict performance over the course of a year, between annual reports (10-K forms from the SEC).

The first goal then is to obtain financial data and stock performance data, and to use the algorithms we have to see if we have any predictive power between financial data "now" and the stock performance "a year from now". The financial data will be used to create our features (this is our X matrix) and the stock performance over the year after that financial data was released will be our y vector.

If the models are found to have predictive power, we will use the models to

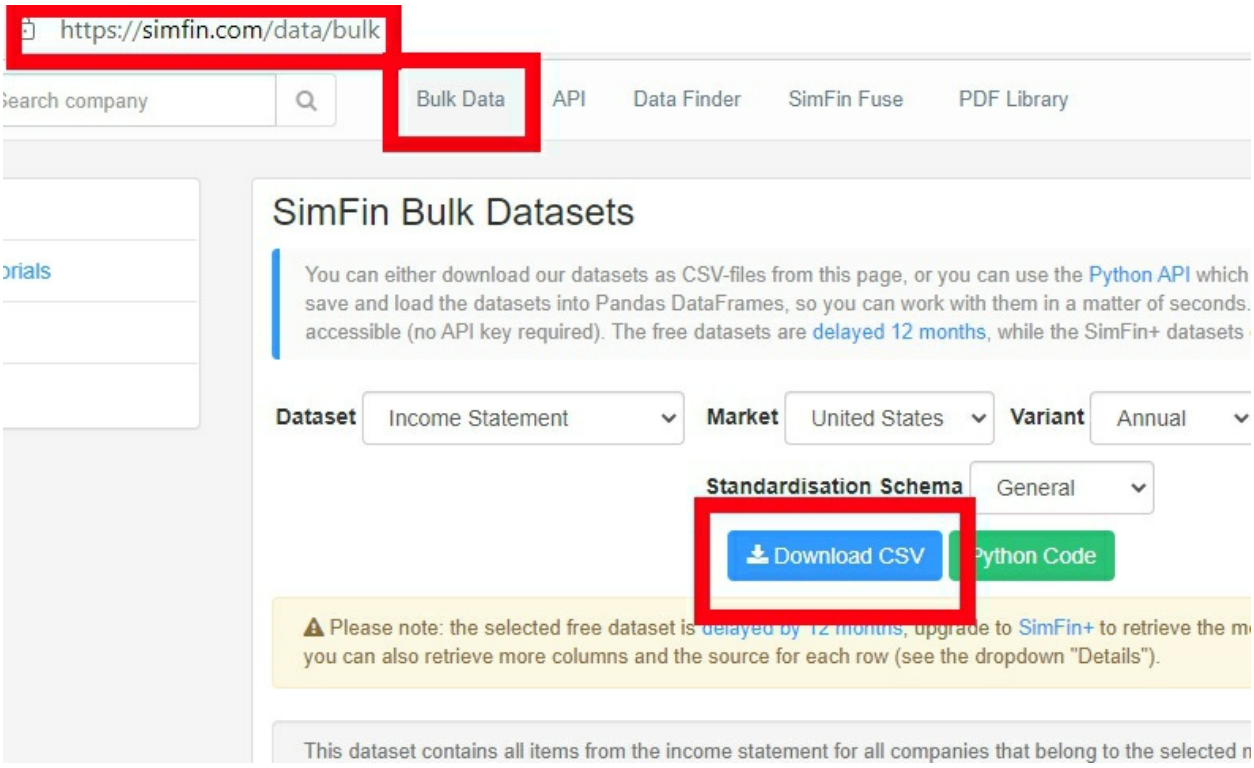
create value investing AIs, after which we will then backtest the investing methods with our historical data to identify the best performing algorithms. We will then pick the winner and tweak it further for performance improvements.

Obtaining and Processing Fundamental Financial Stock data

Getting Data and Initial Filtering

Raw company financial data is available directly from the SEC (*https://www.sec.gov/dera/data/financial-statement-data-sets.html*) though the process of cleaning up that data is an arduous one. We will be getting our financial data instead from *SimFin.com* as they have done the hard work for us and cleaned up the data. Data more than a year old from *SimFin.com* is free, and the very latest data in the same format can be obtained from them for a small fee.

These next steps can be followed with either the free dataset that SimFin provides, which is delayed 12 months, or the full data, which costs 20€ at time of writing. It is a good idea to see and understand the code first with the free data.



The data comes in *.csv* format. Download the general annual income statement, balance sheet and cash flow data for the United States market, with "normal" details in the drop-down menu. When reading this data into Jupyter with Pandas we will need to specify the data delimiter as SimFin separates columns with semi-colon rather than a comma or space.

Alternatively you can download the data directly with Python using the code provided on the SimFin website (Click on the 'Python Code' button in the screenshot above). Avoid this and just download from the website directly if you do not know how to use the Python pip package manager.

Let's make a function to get this data loaded, and merge the income statement, balance sheet and cash flow data into a single monolithic DataFrame. The columns that contain date information should be converted from strings (the default setting) to a date format for our code to easily refer to later. Be sure to put the file address correct for your machine.

In[1]:

```
def getXDataMerged(myLocalPath='C:/Users/G50/Stock_Data/SimFin2021/'):
    '''
    For combining fundamentals financial data from SimFin,
    or SimFin+ (https://simfin.com/) without API.
    Download Income Statement, Balance Sheet and Cash Flow files,
```

```
    Place in a directory and give the directory path to the function.
    Assumes standard filenames from SimFin.
    Returns a DataFrame of the combined result.
    Prints file infos.
    '''
    incomeStatementData=pd.read_csv(myLocalPath+'us-income-annual.csv',
                                    delimiter=';')
    balanceSheetData=pd.read_csv(myLocalPath+'us-balance-annual.csv',
                                 delimiter=';')
    CashflowData=pd.read_csv(myLocalPath+'us-cashflow-annual.csv',
                             delimiter=';')

    print('Income Statement CSV data is(rows, columns): ',
          incomeStatementData.shape)
    print('Balance Sheet CSV data is: ',
          balanceSheetData.shape)
    print('Cash Flow CSV data is: ' ,
          CashflowData.shape)


    # Merge the data together
    result = pd.merge(incomeStatementData, balanceSheetData,\
                on=['Ticker','SimFinId','Currency',
                    'Fiscal Year','Report Date','Publish Date'])

    result = pd.merge(result, CashflowData,\
                on=['Ticker','SimFinId','Currency',
                    'Fiscal Year','Report Date','Publish Date'])

    # dates in correct format
    result["Report Date"] = pd.to_datetime(result["Report Date"])
    result["Publish Date"] = pd.to_datetime(result["Publish Date"])

    print('Merged X data matrix shape is: ', result.shape)

    return result

X = getXDataMerged()
```

Out[1]:

```
Income Statement CSV data is(rows, columns):  (19866, 28)
Balance Sheet CSV data is:  (19866, 30)
Cash Flow CSV data is:  (19866, 28)
Merged X data matrix shape is:  (19866, 74)
```

Great, now we have a DataFrame with nearly all the fundamental company financial data, though this dataset excludes banks and insurers (which is a good thing). It doesn't contain absolutely everything as the raw data itself from the SEC isn't perfect. In Data Science the base data is frequently imperfect and dirty and requires a lot of cleaning.

We can see the columns to the data at our disposal by looking at the DataFrame keys.

```
In [6]: x.keys()

Out[6]: Index(['Ticker', 'SimFinId', 'Currency', 'Fiscal Year', 'Fiscal Period_x',
               'Report Date', 'Publish Date', 'Shares (Basic)_x', 'Shares (Diluted)_x',
               'Revenue', 'Cost of Revenue', 'Gross Profit', 'Operating Expenses',
               'Selling, General & Administrative', 'Research & Development',
               'Depreciation & Amortization_x', 'Operating Income (Loss)',
               'Non-Operating Income (Loss)', 'Interest Expense, Net',
               'Pretax Income (Loss), Adj.', 'Abnormal Gains (Losses)',
               'Pretax Income (Loss)', 'Income Tax (Expense) Benefit, Net',
               'Income (Loss) from Continuing Operations',
               'Net Extraordinary Gains (Losses)', 'Net Income', 'Net Income (Common)',
               'Fiscal Period_y', 'Shares (Basic)_y', 'Shares (Diluted)_y',
               'Cash, Cash Equivalents & Short Term Investments',
               'Accounts & Notes Receivable', 'Inventories', 'Total Current Assets',
               'Property, Plant & Equipment, Net',
               'Long Term Investments & Receivables', 'Other Long Term Assets',
               'Total Noncurrent Assets', 'Total Assets', 'Payables & Accruals',
               'Short Term Debt', 'Total Current Liabilities', 'Long Term Debt',
               'Total Noncurrent Liabilities', 'Total Liabilities',
               'Share Capital & Additional Paid-In Capital', 'Treasury Stock',
               'Retained Earnings', 'Total Equity', 'Total Liabilities & Equity',
               'Fiscal Period', 'Shares (Basic)', 'Shares (Diluted)',
               'Net Income/Starting Line', 'Depreciation & Amortization_y',
               'Non-Cash Items', 'Change in Working Capital',
               'Change in Accounts Receivable', 'Change in Inventories',
               'Change in Accounts Payable', 'Change in Other',
               'Net Cash from Operating Activities',
               'Change in Fixed Assets & Intangibles',
               'Net Change in Long Term Investment',
               'Net Cash from Acquisitions & Divestitures',
               'Net Cash from Investing Activities', 'Dividends Paid',
               'Cash from (Repayment of) Debt', 'Cash from (Repurchase of) Equity',
               'Net Cash from Financing Activities', 'Net Change in Cash'],
              dtype='object')
```

We now have our company financial data, from which we can soon use to construct our *X* matrix. We aren't likely to yield much by simply shoving this data straight into the Machine Learning algorithms. Instead, we need to curate the information and tease out some features that we know should correlate well with stock performance based on our expertise. This is commonly called feature engineering.

Before doing some feature engineering, let's try and get our *y* vector set up. *y* will be the stock performance from 10-K report to 10-K report, a full year of stock performance. We will have to make this from raw stock price data.

Load up the stock price performance data from *SimFin.com*, converting the date column to a date data format.

In[2]:

```
def getYRawData(my_local_path='C:/Users/G50/Stock_Data/SimFin2021/'):
    '''
    Read stock price data from SimFin or SimFin+ (https://simfin.com/),
    without API.
    Place in a directory and give the directory path to the function.
    Assumes standard filenames from SimFin.
    Returns a DataFrame.
    Prints file info.
    '''
    dailySharePrices=pd.read_csv(my_local_path+
                                 'us-shareprices-daily.csv',
                                 delimiter=';')

    dailySharePrices["Date"]=pd.to_datetime(dailySharePrices["Date"])
    print('Stock Price data matrix is: ',dailySharePrices.shape)
    return dailySharePrices
```

After reading this data you may realise that there is a lot of data here, 5.2 million rows! Taking a closer look, we can see that this DataFrame pretty much the stock price for every day as far back as 2007. Here's the DataFrame with a boolean mask to see GOOG stock.

```
d = getYRawData()
d[d['Ticker']=='GOOG']
```

```
Stock Price data matrix is:  (5289952, 10)
```

	Ticker	SimFinId	Date	Open	Low	High	Close	Adj. Close	Dividend	Volume
0	GOOG	18	2014-03-27	568.000	552.92	568.00	558.46	558.46	NaN	13100
1	GOOG	18	2014-03-28	561.200	558.67	566.43	559.99	559.99	NaN	41100
2	GOOG	18	2014-03-31	566.890	556.93	567.00	556.97	556.97	NaN	10800
3	GOOG	18	2014-04-01	558.710	558.71	568.45	567.16	567.16	NaN	7900
4	GOOG	18	2014-04-02	565.106	562.19	604.83	567.00	567.00	NaN	146700
...	...	...	...	...	...	...	...	...	...	...
1319	GOOG	18	2019-06-24	1119.610	1111.01	1122.00	1115.52	1115.52	NaN	1395696
1320	GOOG	18	2019-06-25	1112.660	1083.80	1114.35	1086.35	1086.35	NaN	1546913
1321	GOOG	18	2019-06-26	1086.500	1072.24	1092.97	1079.80	1079.80	NaN	1810869
1322	GOOG	18	2019-06-27	1084.000	1075.29	1087.10	1076.01	1076.01	NaN	1004477
1323	GOOG	18	2019-06-28	1076.390	1073.37	1081.00	1080.91	1080.91	NaN	1693450

1324 rows × 10 columns

We need to get the performance percentage for every stock for every year from this data, to get our corresponding rows to our *X* matrix. Doing this from the 1st of January each year wouldn't be a good idea because when we do the training, we will need to compare financial data *when it was available to the market* to the performance over the next 12 months.

We could do this by getting a stock price once a year for each ticker and calculating percentage returns that way. Unfortunately, there are some missing days in this data (nothing is perfect). As a compromise we can get the stock price as near to the day we want as we can.

We should refrain from writing spaghetti code. Let's break down the operations, first we need a function that gets our stock price for us for a single day. Let's call it *getYPriceDataNearDate* and let's pass to this function the ticker we want the price of, our desired day, the number of days we will consider for our window, and a modifier to change our start day if we desire. We will have the function return a list containing the ticker, stock price, date and the volume (because some stocks barely trade, we'll want to exclude those soon).

To extract the data for the dates we want from the raw *y* data for the specified stock, we'll use a boolean mask. We are going to get a list of prices in the window, so we'll take the top one to return as the answer.

The logic to get the bool mask in plain English is essentially "Give me rows that are between the date I want and that date plus a little bit, whilst having the ticker I want".

In[3]:

```
def getYPriceDataNearDate(ticker, date, modifier, dailySharePrices):
    '''
    Return just the y price and volume.
    Take the first day price/volume of the list of days,
    that fall in the window of accepted days.
    'modifier' just modifies the date to look between.
    Returns a list.
    '''
    windowDays=5
    rows = dailySharePrices[
        (dailySharePrices["Date"].between(pd.to_datetime(date)
                                          + pd.Timedelta(days=modifier),
                                          pd.to_datetime(date)
                                          + pd.Timedelta(days=windowDays
                                                         +modifier)
                                         )
        ) & (dailySharePrices["Ticker"]==ticker)]

    if rows.empty:
        return [ticker, np.float("NaN"),\
                np.datetime64('NaT'),\
                np.float("NaN")]
    else:
        return [ticker, rows.iloc[0]["Open"],\
                rows.iloc[0]["Date"],\
                rows.iloc[0]["Volume"]*rows.iloc[0]["Open"]]
```

Just in case there are any hiccups, we will put an if statement in to return *null* values if nothing can be found in the date window, there are a lot of stocks and we wouldn't want our function crashing our program halfway through. Let's try out our function on some stocks. Here's AAPL at some point in 2012, and a month later, using the *modifier* argument as 30.

```
In [17]: getYPriceDataNearDate('AAPL', '2012-05-12', 0, d)
Out[17]: ['AAPL', 80.3671, Timestamp('2012-05-14 00:00:00'), 1012127183.9799999]

In [18]: getYPriceDataNearDate('AAPL', '2012-05-12', 30, d)
Out[18]: ['AAPL', 83.96, Timestamp('2012-06-11 00:00:00'), 1772949735.9999998]
```

The results match historical prices online for a bunch of stocks. It is worth doing sense checks here when so much will depend on this one function later on.

Now we can use that function to get our *y* data. There will be as many *y* rows as there are rows in *X*. We will want to get the stock performance over a year, so we'll be running our function twice for every row of *X* (beginning and end of the year) and computing the return. Let's put this in a function and break down what we want as input and output.

We'll pass the whole of *X* and the *y* raw data as arguments, as well as the modifier which we will use as the time in days between start price and end price (In Python all arguments are passed by reference so if you're not used to this don't worry). The function will return the *y* data we want for each row of *X* which will be the returned result from *getYPriceNearDate()* we just wrote.

We'll have the function iterate through *X*, and for each row of *X* run our function twice and add the answers to a list, which we return.

Appending large lists and iterating over DataFrames is generally a bad idea because it is slow, we will preallocate this matrix in our computer's memory to try and mitigate the adverse effects, however, because we have to access our stock price data in a sequential way this will take a long time regardless.

There are faster ways of doing this operation, however, these functions are partly written in the interests of readability and we only need to do this once anyway. We do the preallocation with this line:

```
y = [[None]*8 for i in range(len(x))]
```

And when we iterate through *X*, we will take the data we want from *X* to correspond to *y*, so we will pass items from *X* into our *getYPriceNearDate* function. This makes our *getYPricesReportDateAndTargetDate()* function. It

is a long name but this function is doing most of our work and it is easy to understand what it does just by its name.

In[4]:

```
def getYPricesReportDateAndTargetDate(x, d, modifier=365):
    '''
    Takes in all fundamental data X, all stock prices over time y,
    and modifier (days), and returns the stock price info for the
    data report date, as well as the stock price one year from that date
    (if modifier is left as modifier=365)
    '''
    # Preallocation list of list of 2
    # [(price at date) (price at date + modifier)]
    y = [[None]*8 for i in range(len(x))]

    whichDateCol='Publish Date'# or 'Report Date',
    # is the performance date from->to. Want this to be publish date.

    # Because of time lag between report date
    # (which can't be actioned on) and publish date
    # (data we can trade with)
    i=0
    for index in range(len(x)):
        y[i]=(getYPriceDataNearDate(x['Ticker'].iloc[index],
                                    x[whichDateCol].iloc[index],0,d)
              +getYPriceDataNearDate(x['Ticker'].iloc[index],
                                     x[whichDateCol].iloc[index],
                                     modifier, d))
        i=i+1

    return y
```

Now that we have done the hard work within the functions, we can run our code in a visually cleaner and easier to understand way by calling the functions when we need the results. Here the merged stock financial data is stored (it's good to have a .csv file for each bit of processing done so we can backtrack easily if needed), and the *y* data as a matrix is made with our *getYPricesReportDateAndTargetDate* function.

In[5]:

```
X = getXDataMerged()
X.to_csv("Annual_Stock_Price_Fundamentals.csv")
```

Out[5]:

```
Income Statement CSV data is(rows, columns):  (19866, 28)
Balance Sheet CSV data is:  (19866, 30)
Cash Flow CSV data is:  (19866, 28)
Merged X data matrix shape is:  (19866, 74)
```

In[6]:

```
# We want to know the performance for each stock,
# each year, between 10-K report dates.
#takes VERY long time, several hours.
#because of lookups in this function:
y = getYPricesReportDateAndTargetDate(x, d, 365)
```

Out[6]:
Stock Price data matrix is: (5159358, 10)

This step takes a very long time as the code is looking through our *entire* 5 million line matrix to find the stock price data within a time window for *every* stock. We only need to run this once.

Taking a look at the resulting DataFrame to check we have everything we need, we have 19,866 stock return values, from ticker A to ZYXI. It is worth checking the dates correspond to reporting dates by looking at the actual information in 10-K reports from the SEC. You can get these from *www.sec.gov/edgar/searchedgar/companysearch.html*. If our returns data from *y* doesn't start at dates after the 10-K company data is published, our AI will not work properly.

```
y
```

	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
0	A	16.78	2008-12-19	2.483239e+08	A	29.60	2009-12-21	9.319856e+07
1	A	29.60	2009-12-21	9.319856e+07	A	40.65	2010-12-21	1.469619e+08
2	A	40.10	2010-12-20	1.613504e+08	A	33.99	2011-12-20	1.247977e+08
3	A	33.73	2011-12-16	1.053320e+08	A	39.98	2012-12-17	1.381149e+08
4	A	40.58	2012-12-20	1.545570e+08	A	57.47	2013-12-20	1.697664e+08
...	...	...	...	...	...	...	...	...
19861	ZYXI	0.34	2016-03-31	3.400000e+01	ZYXI	0.31	2017-03-31	2.170000e+02
19862	ZYXI	0.34	2017-04-18	3.502000e+03	ZYXI	3.46	2018-04-18	1.887119e+05
19863	ZYXI	5.00	2018-02-28	5.846550e+05	ZYXI	4.96	2019-02-28	2.961021e+05
19864	ZYXI	5.60	2019-02-26	2.684080e+05	ZYXI	13.09	2020-02-26	3.442356e+06
19865	ZYXI	12.38	2020-02-27	4.285139e+06	ZYXI	NaN	NaN	NaN

19866 rows × 8 columns

After running this step, put the list data into a DataFrame and save it as a csv.

In[7]:

```
y = pd.DataFrame(y, columns=['Ticker', 'Open Price', 'Date', 'Volume',\
                             'Ticker2', 'Open Price2', 'Date2', 'Volume2'
                             ])
y.to_csv("Annual_Stock_Price_Performance.csv")
```

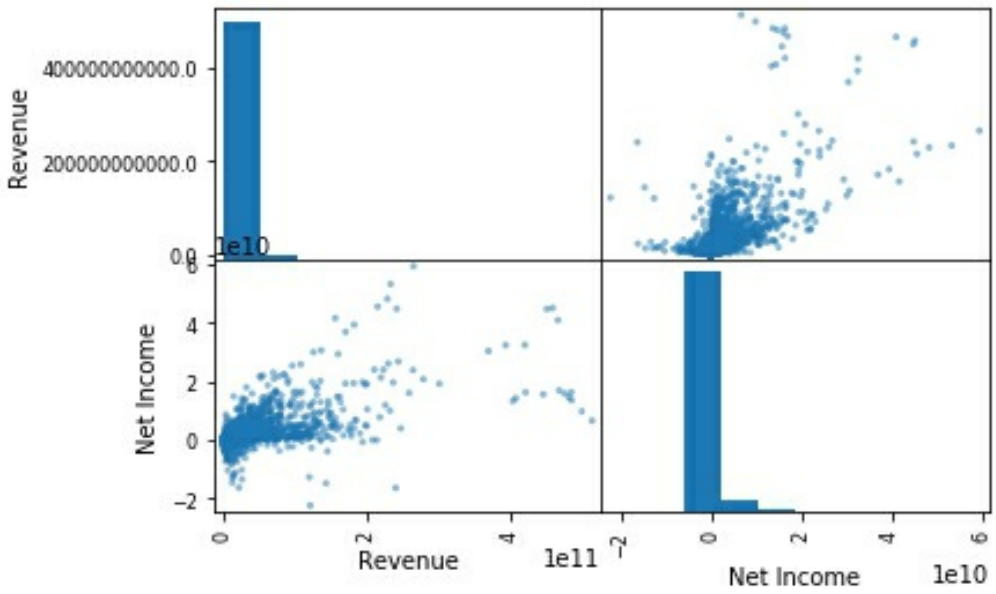
We now have the needed *y* data, we just need to find the percentage change in stock price between the two columns for every row.

Back to the *X* matrix, which should be familiar as we saw a part of it at the beginning of this guide. This is good data to explore the US stock market with some plots. For example, here is a scatter plot of revenue vs. net income. Intuitively there should be a strong correlation, interestingly it seems like companies are grouped between high revenue/low income and low revenue/high-income groups.

In[8]:

```
from pandas.plotting import scatter_matrix
attributes=["Revenue","Net Income"]
scatter_matrix(x[attributes]);
```

Out[8]:



Our data will need to be cleaned of rows we do not want our algorithms to analyse, like stocks with very low numbers in the *Volume* column, it turns out there are a few in the data with less than 10,000 shares traded in a day:

```
y[(y['Volume']<1e4) | (y['Volume2']<1e4)]
```

	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
32	AAN	15.6667	2010-02-03	0.0000	AAN	19.500	2011-02-03	7121400.000
128	ABCD	1.1965	2013-03-08	4277.4875	ABCD	2.100	2014-03-10	150225.600
153	ABMC	0.1200	2011-02-14	240.0000	ABMC	0.160	2012-02-14	496.000
154	ABMC	0.1600	2012-02-14	496.0000	ABMC	0.220	2013-02-13	220.000
155	ABMC	0.1990	2013-03-28	348.2500	ABMC	0.115	2014-03-28	714.725
...	...	...	...	...	...	...	...	...
15461	ZYXI	0.5600	2013-03-28	30968.0000	ZYXI	0.430	2014-03-28	43.000
15462	ZYXI	0.4300	2014-03-28	43.0000	ZYXI	0.140	2015-03-30	518.000
15463	ZYXI	0.1400	2015-03-31	182.0000	ZYXI	0.340	2016-03-30	0.000
15464	ZYXI	0.3400	2016-03-31	34.0000	ZYXI	0.310	2017-03-31	217.000
15465	ZYXI	0.3200	2017-04-17	5696.0000	ZYXI	3.250	2018-04-17	78900.250

619 rows × 8 columns

We also need to get rid of rows with no volume at the target date *Volume2*, rows with no prices and *X* rows with no shares outstanding (yes, somehow this is possible).

In our *X* matrix, we would also like to create a column for market capitalisation as this is something we will use a lot when we try and engineer some features that we want our algorithms to look at, like price/earnings ratios. When we get rid of rows in the *X* matrix, we have to get rid of the corresponding rows from the other matrix, otherwise our data won't match up:

In[9]:

```
# Issue where no share price
bool_list1 = ~y["Open Price"].isnull()

# Issue where there is low/no volume
bool_list2 = ~((y['Volume']<1e4) | (y['Volume2']<1e4))

# Issue where dates missing(Removes latest data too, which we can't use)
bool_list3 = ~y["Date2"].isnull()

y=y[bool_list1 & bool_list2 & bool_list3]
X=X[bool_list1 & bool_list2 & bool_list3]
```

```
# Issues where no listed number of shares
bool_list4 = ~X["Shares (Diluted)_x"].isnull()
y=y[bool_list4]
X=X[bool_list4]

y=y.reset_index(drop=True)
X=X.reset_index(drop=True)
```

Save CSVs for this data after this filtering, it looks like there are roughly 15,000 rows now (for 2020 there were 13,000).

In[10]:

```
X.shape
```

Out[10]:
(15588, 75)

In[11]:

```
y.shape
```

Out[11]:
(15588, 8)

In[12]:

```
X.to_csv("Annual_Stock_Price_Fundamentals_Filtered.csv")
y.to_csv("Annual_Stock_Price_Performance_Filtered.csv")
```

Feature Engineering (and more filtering)

The stock fundamentals data we have at this point needs to be used to generate company features we want our algorithms to learn from. Just as when us humans analyse companies it is more helpful to consider relative values like return on equity, price/sales ratio or interest expense/operation income instead of the absolute size of any number like the revenue or debt, it is more useful for our Machine Learning algorithms to consider these values in the *X* matrix from the outset.

There are a large number of features that we could engineer, all known accounting ratios could be used, as well as any ratios that can be dreamt up that may give us insight into how any company is run. The most important are going to be ratios that are related to the market cap of the stock, like price/earnings ratio, because these ratios tell us something about the company relative to the price we are paying for the stock.

This is a good place to make up new company features that you think will

correlate with business success, making your investing AI unique with the possibility of increasing the performance of the performance predictor.

You could of course just try all the reasonable permutations and throw it at the algorithms to find something. Here we will use a large number that make sense intuitively, ratios that have appeared in other formulaic investing books for the public such as *What Works on Wall Street, The Little Book That Beats The Market, Contrarian Investment Strategies*, etc.

Our feature engineered data will need to be cleaned, as there are null values in the company financial data we have at this point, furthermore our feature engineered ratios will give us a few infinities as no doubt there are some companies with no earnings, or no debt, etc. which will give us infinity in some ratios.

We will start the code in this chapter in a new Jupyter Notebook *2_Process_X_Y_Learning_Data.ipynb*.

In[13]:

```
# In this notebook we will use x_ for x fundamentals,
# and X is the input matrix we want at the end.
x_=pd.read_csv("Annual_Stock_Price_Fundamentals_Filtered.csv",
                index_col=0)
y_=pd.read_csv("Annual_Stock_Price_Performance_Filtered.csv",
                index_col=0)
```

We will change the null values to be 0s instead of deleting entire rows as the null values are there because the company has no accounting value for those items, which can be checked by looking at the 10-K report for a line of data where a null value exists (Try the EDGAR website, https://www.sec.gov/edgar.shtml).

In[14]:

```
def fixNansInX(x):
    '''
    Takes in x DataFrame, edits it so that important keys
    are 0 instead of NaN.
    '''
    keyCheckNullList = ["Short Term Debt" ,\
            "Long Term Debt" ,\
            "Interest Expense, Net",\
            "Income Tax (Expense) Benefit, Net",\
            "Cash, Cash Equivalents & Short Term Investments",\
            "Property, Plant & Equipment, Net",\
```

```
            "Revenue",\
            "Gross Profit"]
    x[keyCheckNullList]=x[keyCheckNullList].fillna(0)
```

Let's start the feature engineering. There are a few columns we will want to use that need to be made before we make the *X* matrix proper. Create the columns of enterprise value and earnings before interest and tax.

In[15]:

```
def addColsToX(x):
    '''
    Takes in x DataFrame, edits it to include:
        Enterprise Value.
        Earnings before interest and tax.

    '''
    x["EV"] = x["Market Cap"] \
    + x["Long Term Debt"] \
    + x["Short Term Debt"] \
    - x["Cash, Cash Equivalents & Short Term Investments"]

    x["EBIT"] = x["Net Income"] \
    - x["Interest Expense, Net"] \
    - x["Income Tax (Expense) Benefit, Net"]
```

Now we get to what is probably the most important part, making the features that are ratios (and other metrics) from which we want our Machine Learning algorithms to find relationships from. You can have more that are aimed at company valuation like Price/Earnings or Enterprise Value/Earnings if you want your investing AI to be more like a value investor, or you can make the AI concentrate more on the quality of a company with measures like Return on Equity or the Dupont Analysis ratios if you want it to be more of a growth style investor.

We will make a new DataFrame for these features which will be the actual *X* matrix for our Machine Learning process. There are a lot of features here as we want as many as possible to make the most of our information:

In[16]:

```
# Make new X with ratios to learn from.
def getXRatios(x_):
    '''
    Takes in x_, which is the fundamental stock DataFrame raw.
    Outputs X, which is the data encoded into stock ratios.
    '''
    X=pd.DataFrame()
```

```
# EV/EBIT
X["EV/EBIT"] = x_["EV"] / x_["EBIT"]

# Op. In./(NWC+FA)
X["Op. In./(NWC+FA)"] = x_["Operating Income (Loss)"] \
/ (x_["Total Current Assets"] - x_["Total Current Liabilities"] \
   + x_["Property, Plant & Equipment, Net"])

# P/E
X["P/E"] = x_["Market Cap"] / x_["Net Income"]

# P/B
X["P/B"] = x_["Market Cap"] / x_["Total Equity"]

# P/S
X["P/S"] = x_["Market Cap"] / x_["Revenue"]

# Op. In./Interest Expense
X["Op. In./Interest Expense"] = x_["Operating Income (Loss)"]\
/ - x_["Interest Expense, Net"]

# Working Capital Ratio
X["Working Capital Ratio"] = x_["Total Current Assets"]\
/ x_["Total Current Liabilities"]

# Return on Equity
X["RoE"] = x_["Net Income"] / x_["Total Equity"]

# Return on Capital Employed
X["ROCE"] = x_["EBIT"]\
/ (x_["Total Assets"] - x_["Total Current Liabilities"] )

# Debt/Equity
X["Debt/Equity"] = x_["Total Liabilities"] / x_["Total Equity"]

# Debt Ratio
X["Debt Ratio"] = x_["Total Assets"] / x_["Total Liabilities"]

# Cash Ratio
X["Cash Ratio"] = x_["Cash, Cash Equivalents & Short Term Investments"]\
/ x_["Total Current Liabilities"]

# Asset Turnover
X["Asset Turnover"] = x_["Revenue"] / \
                        x_["Property, Plant & Equipment, Net"]

# Gross Profit Margin
X["Gross Profit Margin"] = x_["Gross Profit"] / x_["Revenue"]

### Altman ratios ###
# (CA-CL)/TA
X["(CA-CL)/TA"] = (x_["Total Current Assets"]\
                    - x_["Total Current Liabilities"])\
```

```
                    /x_["Total Assets"]

    # RE/TA
    X["RE/TA"] = x_["Retained Earnings"]/x_["Total Assets"]

    # EBIT/TA
    X["EBIT/TA"] = x_["EBIT"]/x_["Total Assets"]

    # Book Equity/TL
    X["Book Equity/TL"] = x_["Total Equity"]/x_["Total Liabilities"]

    return X
```

The final process we need to do on our *X* matrix is to remove extremes from the data. We want our learning algorithms to have nice distributions to work with. If you try and plot a histogram of some of the columns without cleaning the data, you will get an error saying that some values are infinite. Imagine running the gradient descent program you made earlier and giving it infinity as one of the inputs (it wouldn't be pretty). With the pandas library you can clip the distributions with the *.clip()* function.

We will impose our own max/min values to out *X* features, looking at a histogram of that column as we go to see if the distribution looks reasonable after transformation.

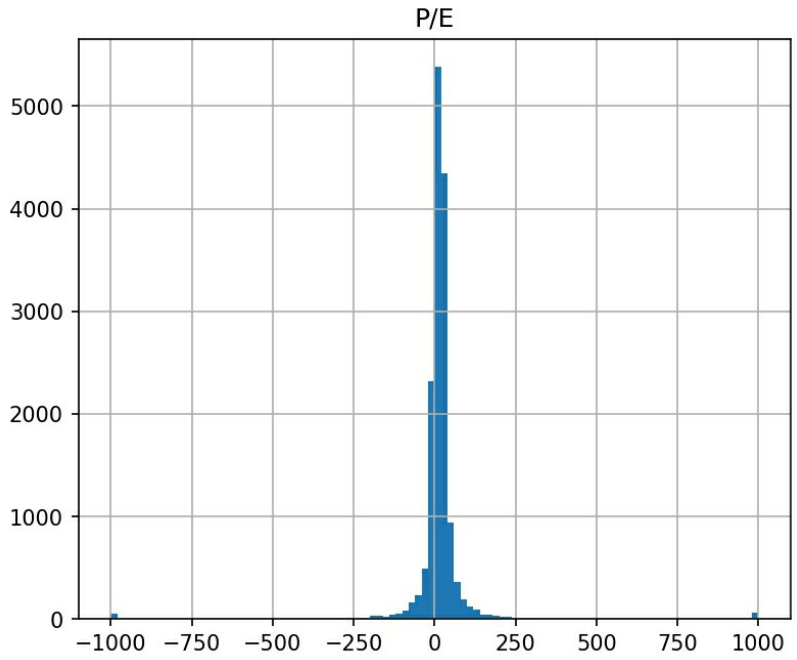
When using Machine Learning algorithms it would normally be a better idea to create a general way to remove the outliers of a dataset or to implement better scaling. In our case, we know that when dealing with company ratios e.g. P/E, a value of a million or infinity can easily be substituted with a number like five hundred with little loss of meaning.

Here is the distribution of P/E after we are finished with it. As some companies have 0 earnings, there were many infinity values beforehand, as can be seen at the tip of the distribution (Run this after creating variable *x*):

In[17]:

```
k=x.keys()[2] #14 max
x[k].hist(bins=100, figsize=(5,5))
plt.title(k);
```

Out[17]:



Good limits for the other ratios have been worked out for you in the *fixXRatios()* function. This feature cleaning step is important in attaining good performance for the predictor. Here we go through all the keys in our *X* matrix and slice the distribution to suit our needs with *.clip()* (Don't forget there is always the pandas cheat sheet for DataFrame functions available at the official website *https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf*).

In[18]:

```
def fixXRatios(X):
    '''
    Takes in X, edits it to have the distributions clipped.
    The distribution clippings are done manually by eye,
    with human judgement based on the information.
    '''
    X["RoE"].clip(-5, 5, inplace=True)
    X["Op. In./(NWC+FA)"].clip(-5, 5, inplace=True)
    X["EV/EBIT"].clip(-500, 500, inplace=True)
    X["P/E"].clip(-1000, 1000, inplace=True)
    X["P/B"].clip(-50, 100, inplace=True)
    X["P/S"].clip(0, 500, inplace=True)
    X["Op. In./Interest Expense"].clip(-600, 600, inplace=True)
    X["Working Capital Ratio"].clip(0, 30, inplace=True)
    X["ROCE"].clip(-2, 2, inplace=True)
    X["Debt/Equity"].clip(0, 100, inplace=True)
    X["Debt Ratio"].clip(0, 50, inplace=True)
    X["Cash Ratio"].clip(0, 30, inplace=True)
    X["Gross Profit Margin"].clip(0, 1, inplace=True) #how can be >100%?
    X["(CA-CL)/TA"].clip(-1.5, 2, inplace=True)
```

```
    X["RE/TA"].clip(-20, 2, inplace=True)
    X["EBIT/TA"].clip(-2, 1, inplace=True)
    X["Book Equity/TL"].clip(-2, 20, inplace=True)
    X["Asset Turnover"].clip(0, 500, inplace=True)
```

Here is a nice plotting function to see all of the distributions (I won't print all the graphs here).

In[19]:

```
# Make a plot of the distributions.
cols, rows = 3, 5
plt.figure(figsize=(5*cols, 5*rows))

for i in range(1, cols*rows):
    if i<len(X.keys()):
        plt.subplot(rows, cols, i)
        k=X.keys()[i]
        X[k].hist(bins=100)
        plt.title(k);
```

Now that we have our base functions the final program to create our *X* matrix is simply:

In[20]:

```
# From x_ (raw fundamental data) get X (stock fundamental ratios)
fixNansInX(x_)
addColsToX(x_)
X=getXRatios(x_)
fixXRatios(X)
```

Great, our *X* matrix is ready to be used for some Machine Learning, save the DataFrame:

In[21]:

```
X.to_csv("Annual_Stock_Price_Fundamentals_Ratios.csv")
```

We will create our final *y* vector by finding the percentage change between the open price at the start and end period from *y_*.

In[22]:

```
def getYPerf(y_):
    '''
    Takes in y_, which has the stock prices and their respective
    dates they were that price.
    Returns a DataFrame y containing the ticker and the
    relative change in price only.
```

```
    '''
    y=pd.DataFrame()
    y["Ticker"] = y_["Ticker"]
    y["Perf"]=(y_["Open Price2"]-y_["Open Price"])/y_["Open Price"]
    y["Perf"].fillna(0, inplace=True)
    return y

# From y_(stock prices/dates) get y (stock price change)
y=getYPerf(y_)
```

Our *y* vector is now a list of performance percentages:

```
In [201]: y
```

Out[201]:

	Ticker	Perf
0	A	0.253328
1	A	0.373311
2	A	-0.152369
3	A	0.185295
4	A	0.416215
...	...	...
13002	low	0.517206
13003	low	0.018521
13004	low	0.096215
13005	low	0.029423
13006	low	0.246709

13007 rows × 2 columns

Now we have real market data and stock financial features all cleaned up ready to train and test our Machine Learning algorithms with.

In[23]:

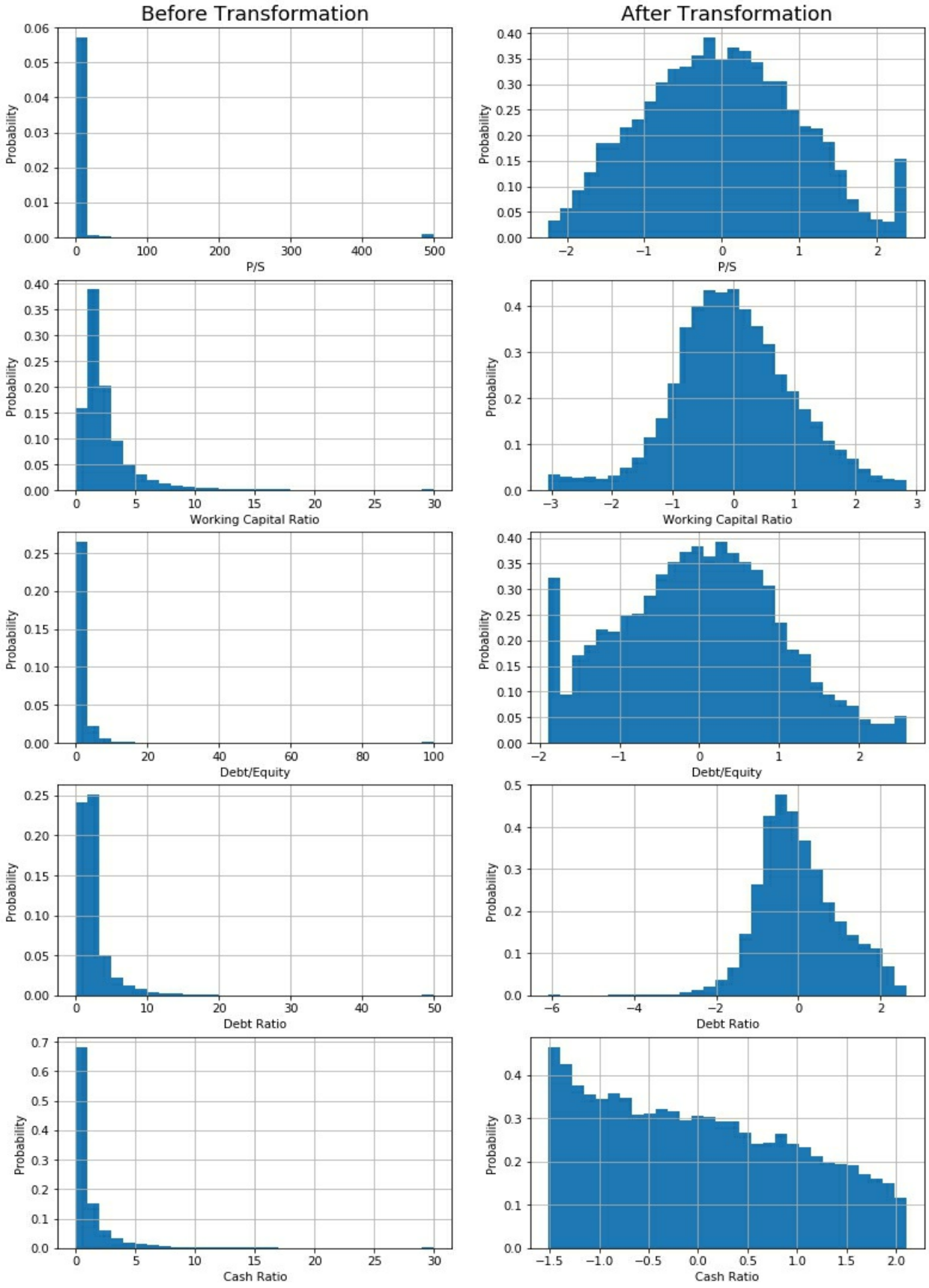
```
y.to_csv("Annual_Stock_Price_Performance_Percentage.csv")
```

These distributions look much closer to the gaussian distribution after we put

them through a power transformer, which is what we want. It's worth viewing these histograms to get an idea of what distributions the Machine Learning algorithms will be working with:

In[24]:

```
# Write code to plot out all distributions of X in a nice diagram
from sklearn.preprocessing import PowerTransformer
transformer = PowerTransformer()
X_t=pd.DataFrame(transformer.fit_transform(X), columns=X.keys())

def plotFunc(n, myDatFrame):
    myKey = myDatFrame.keys()[n]
    plt.hist(myDatFrame[myKey], density=True, bins=30)
    plt.grid()
    plt.xlabel(myKey)
    plt.ylabel('Probability')

plt.figure(figsize=(13,20))
plotsIwant=[4,6,9,10,11]

j=1
for i in plotsIwant:
    plt.subplot(len(plotsIwant),2,2*j-1)
    plotFunc(i,X)
    if j==1:
        plt.title('Before Transformation',fontsize=17)
    plt.subplot(len(plotsIwant),2,2*j)
    plotFunc(i,X_t)
    if j==1:
        plt.title('After Transformation',fontsize=17)
    j+=1

plt.savefig('Transformat_Dists.png', dpi=300)
```


Before Transformation
After Transformation
Probability
P/S
Working Capital Ratio
Debt/Equity
Debt Ratio
Cash Ratio

Using Machine Learning to Predict Stock Performance

Now we can see if the Scikit-Learn Machine Learning models can give us any predictive ability with our stock data. We will try out all the models we have covered so far, the more the better. We should take note of any models that show predictive ability, saving them for later, when we will perform more rigorous backtesting. Firstly, reading in our *X* and *y* data:

In[1]:

```
def loadXandyAgain():
    '''
    Load X and y.
    Randomises rows.
    Returns X, y.
    '''
    # Read in data
    X=pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios.csv",
                  index_col=0)
    y=pd.read_csv("Annual_Stock_Price_Performance_Percentage.csv",
                  index_col=0)
    y=y["Perf"] # We only need the % returns as target

    # randomize the rows
    X['y'] = y
    X = X.sample(frac=1.0, random_state=42) # randomize the rows
    y = X['y']
    X.drop(columns=['y'], inplace=True)

    return X, y

X, y = loadXandyAgain()
```

Let's check out the mean baseline return, we want our algorithms to select stocks which produce returns at least be above this return.

In[2]:

```
y.mean()
```

Out[2]:
0.13882683275381503

This is higher than the S&P500 index return, which went from ~1000 to ~3000 in 10 years ending in 2020, a ~11% return. This is because the S&P500 is weighted towards large companies, because there are more stocks

here than the S&P500, and because survivorship bias is present in this data. Asking for a theoretical return above say 25% might be adequate compensate us for the risk of default and the higher mean return. The issue can of course be eliminated by purchasing better stock data.

Let's see what our algorithms can do. The results printed here might not be exactly replicated if you follow the code exactly as the data from SimFin is updated over time. Even if the results are negative, stick with it until the end, we don't know what the likely performance is until we do some testing.

Linear Regression

Linear regression is arguably the simplest model so we'll start with that. Let's do a single train/test split and train the regressor, just to see if our data works. There's no specif reason a *test_size* of 0.1 is chosen, training with 14,000 rows and testing with 1500 rows just sounds reasonable.

In[3]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.1,
                                                    random_state=42)
print('X training matrix dimensions: ', X_train.shape)
print('X testing matrix dimensions: ', X_test.shape)
print('y training matrix dimensions: ', y_train.shape)
print('y testing matrix dimensions: ', y_test.shape)
```

Out[3]:

```
X training matrix dimensions:  (14029, 18)
X testing matrix dimensions:  (1559, 18)
y training matrix dimensions:  (14029,)
y testing matrix dimensions:  (1559,)
```

We will put the *Powertransformer* in a pipeline with our Linear Regressor.

In[4]:

```
# Linear
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PowerTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error

pl_linear = Pipeline([
                      ('Power Transformer', PowerTransformer()),
                      ('linear', LinearRegression())
                      ])
```

```
pl_linear.fit(X_train, y_train)

y_pred = pl_linear.predict(X_test)
print('Train MSE: ',
      mean_squared_error(y_train, pl_linear.predict(X_train)))
print('Test MSE: ',
      mean_squared_error(y_test, y_pred))
```

Out[4]:
train mse: 0.4029998242279224
test mse: 0.3040293597007813

Stock returns are going to be around something like 10% in a year, depending on the stock, and here we are getting an error of roughly 30%. This is like having every prediction off by 30%, which isn't very accurate.

At least it makes some prediction, the training error should of course be below the error from testing. This might not be the case for you. When we try a new *train_test_split* with a different *random_state*, we find that the error terms change quite a lot. (Market data is fickle, if it were too easy everyone would do it and the relationship would disappear). At this point we also aren't so sure about how big the train/test split should be.

To remedy the first problem we can run do a train/test run many times, making a different train/test split each time and seeing if on average our model is genuinely learning anything from the data, even something fleeting. To remedy the second problem, we can construct the learning curve for our regression model (where we increase the size of the learning set and see how the prediction ability of the model changes as we increase the amount of learning material).

The Scikit-Learn online documentation provides a good sample piece of code we can use to solve our issue: (https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-py). We only need to take the parts we need from that code (their example looks a bit long-winded). It is always a good idea to take a look at the documentation of a programming library you are using.

To construct the learning curve following the example we use the *learning_curve()* function from the Scikit-Learn library, here the nice part for us is that the cross-validation setting doesn't have to be K-folds, it can instead

be *ShuffleSplit()*, where we randomise the rows before taking train/test data. *ShuffleSplit()* suits us here because just using linear regression on our data for prediction is hit-and-miss for a single run, by averaging a large number of runs we can get an idea of prediction ability.

The Scikit-Learn implementation of *learning_curve()* also allows you to do the train/test runs in parallel to speed things up. To do so, set the *n_jobs* variable to the number of CPU cores you want to use. Not everyone has an expensive CPU so the setting is fixed at 4 here.

We will take the scores from *learning_curve()* and palce them in a DataFrame for us to easily see the results.

In[5]:

```
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit

sizesToTrain = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 0.9]
train_sizes, train_scores, test_scores, fit_times, score_times = \
    learning_curve(pl_linear, X, y, cv=ShuffleSplit(n_splits=100,
                                                    test_size=0.2,
                                                    random_state=42),
                   scoring='neg_mean_squared_error',
                   n_jobs=4, train_sizes=sizesToTrain,
                   return_times=True)

#Create a DataFrame of results
results_df = pd.DataFrame(index=train_sizes)
results_df['train_scores_mean'] = np.sqrt(-np.mean(train_scores,axis=1))
results_df['train_scores_std'] = np.std(np.sqrt(-train_scores), axis=1)
results_df['test_scores_mean'] = np.sqrt(-np.mean(test_scores, axis=1))
results_df['test_scores_std'] = np.std(np.sqrt(-test_scores), axis=1)
results_df['fit_times_mean'] = np.mean(fit_times, axis=1)
results_df['fit_times_std'] = np.std(fit_times, axis=1)

results_df # see results
```

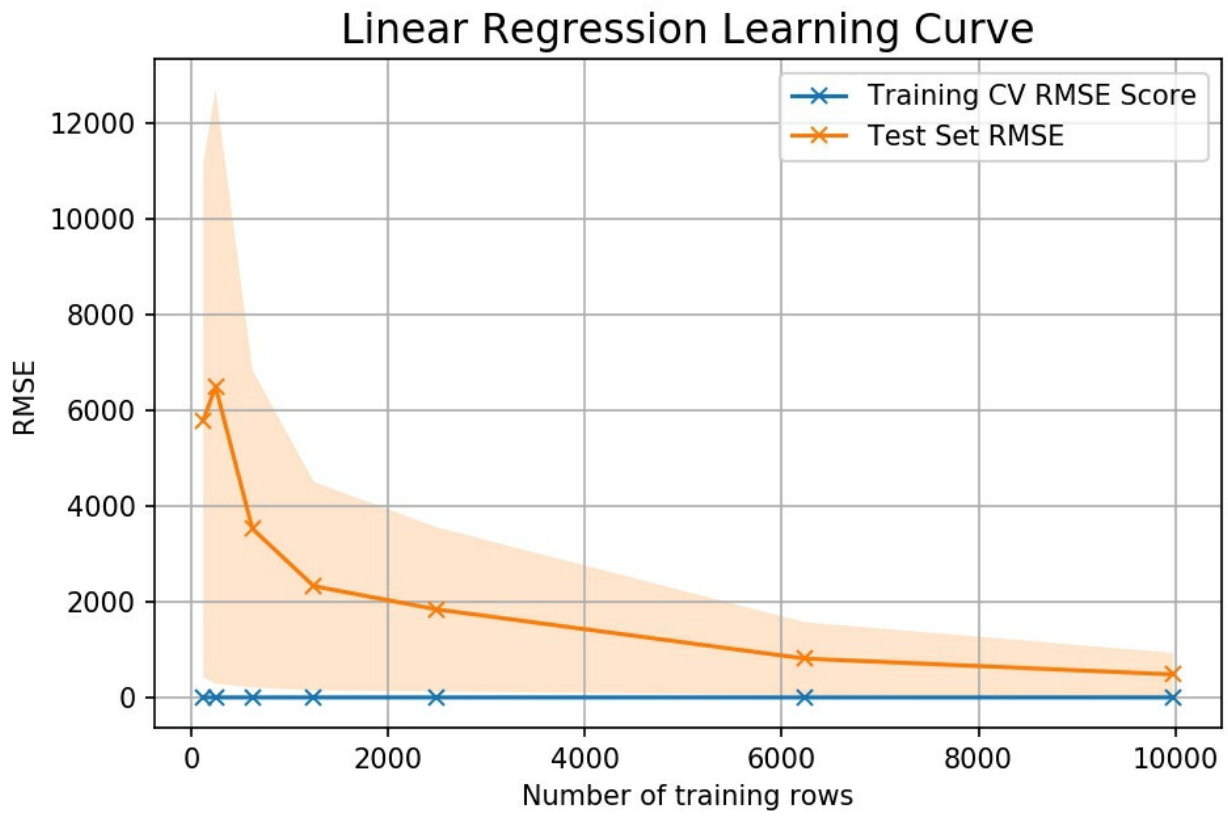
Out[5]:

	train_scores_mean	train_scores_std	test_scores_mean	test_scores_std	fit_times_mean	fit_times_std
124	0.469482	0.125088	5792.721898	5371.911839	0.044464	0.009262
249	0.513085	0.128952	6498.078872	6209.732892	0.049661	0.009391
623	0.527395	0.088025	3530.542594	3320.070351	0.064168	0.010495
1247	0.555333	0.094145	2328.304371	2180.004972	0.081160	0.010043
2494	0.587386	0.091263	1842.494000	1717.668292	0.118509	0.015390
6235	0.608916	0.061745	813.476609	760.003905	0.220840	0.021571
9976	0.618780	0.039564	481.662526	449.254250	0.321444	0.027817

The DataFrame index is the number of rows we used to train the model. Plotting out our learning curve we see that there is quite a bit of variation in the results. Of the many train/test runs done the standard deviation is recorded and the band of results is plotted so we can see the variation.

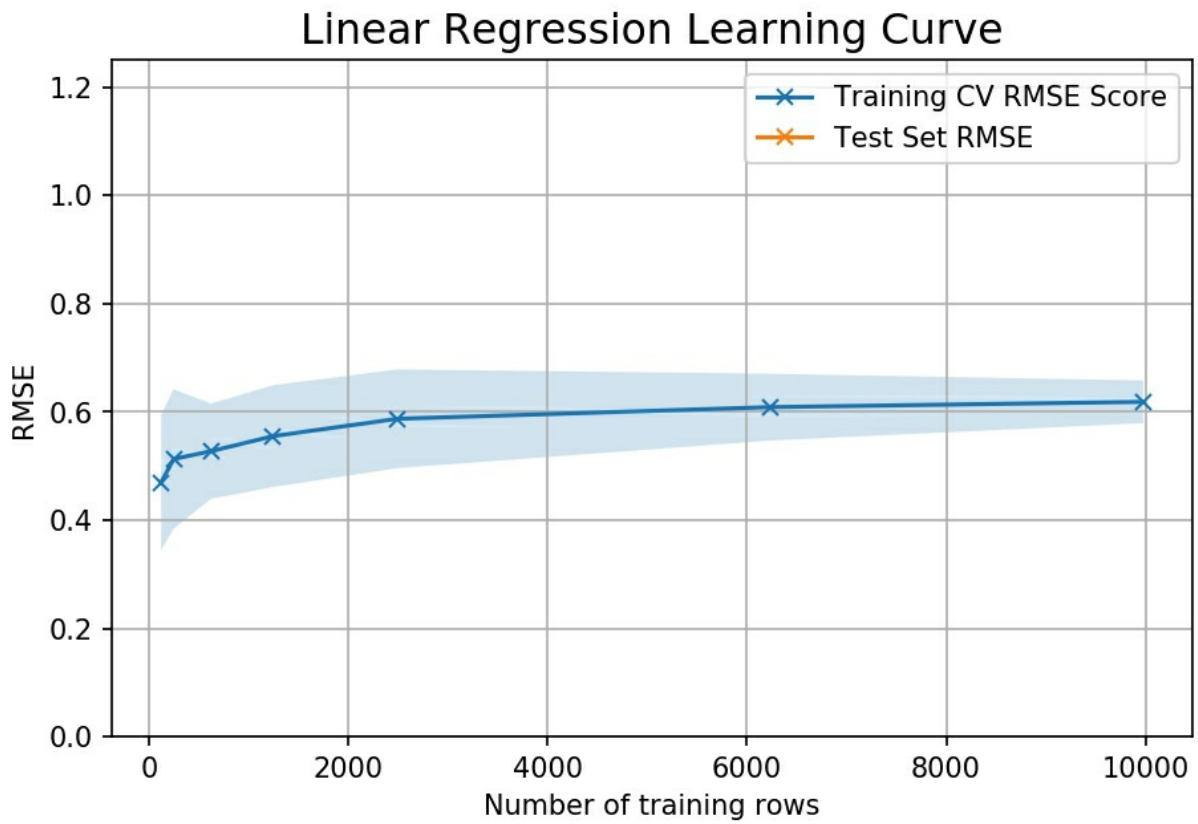
In[6]:

```
df['train_scores_mean'].plot(style='-x')
df['test_scores_mean'].plot(style='-x')

plt.fill_between(df.index,\
                 df['train_scores_mean']-df['train_scores_std'],\
                 df['train_scores_mean']+df['train_scores_std'],\
                 alpha=0.2)
plt.fill_between(df.index,\
                 df['test_scores_mean']-df['test_scores_std'],\
                 df['test_scores_mean']+df['test_scores_std'],\
                 alpha=0.2)
plt.grid()
plt.legend(['Training CV RMSE Score','Test Set RMSE'])
plt.ylabel('RMSE')
plt.xlabel('Number of training rows')
plt.title('Linear Regression Learning Curve', fontsize=15)
#plt.ylim([0, 1.25]);
```



Although the accuracy is low and there is a lot of variation with the Linear Regressor, it makes a learning curve with our data showing some learning ability, which just might be all we need.

When looking at the training curve by itself, we can justifiably use from 4000 to 10,000+ training rows (test_size of 0.4 makes the training set size about 10,000 rows). The graph below is the same graph just zoomed into the blue line.



To get a better look at the prediction results. Let's plot a scatter plot, with the x-axis being the predicted return of the test data, and the y-axis being the actual test data return.

With this kind of plot, the perfect prediction would yield a straight line along y=x, going top right to bottom left at 45 degrees. Obviously our points aren't going to end up on that line (as we know the errors are large) however it will give us a better idea of the predictive ability.

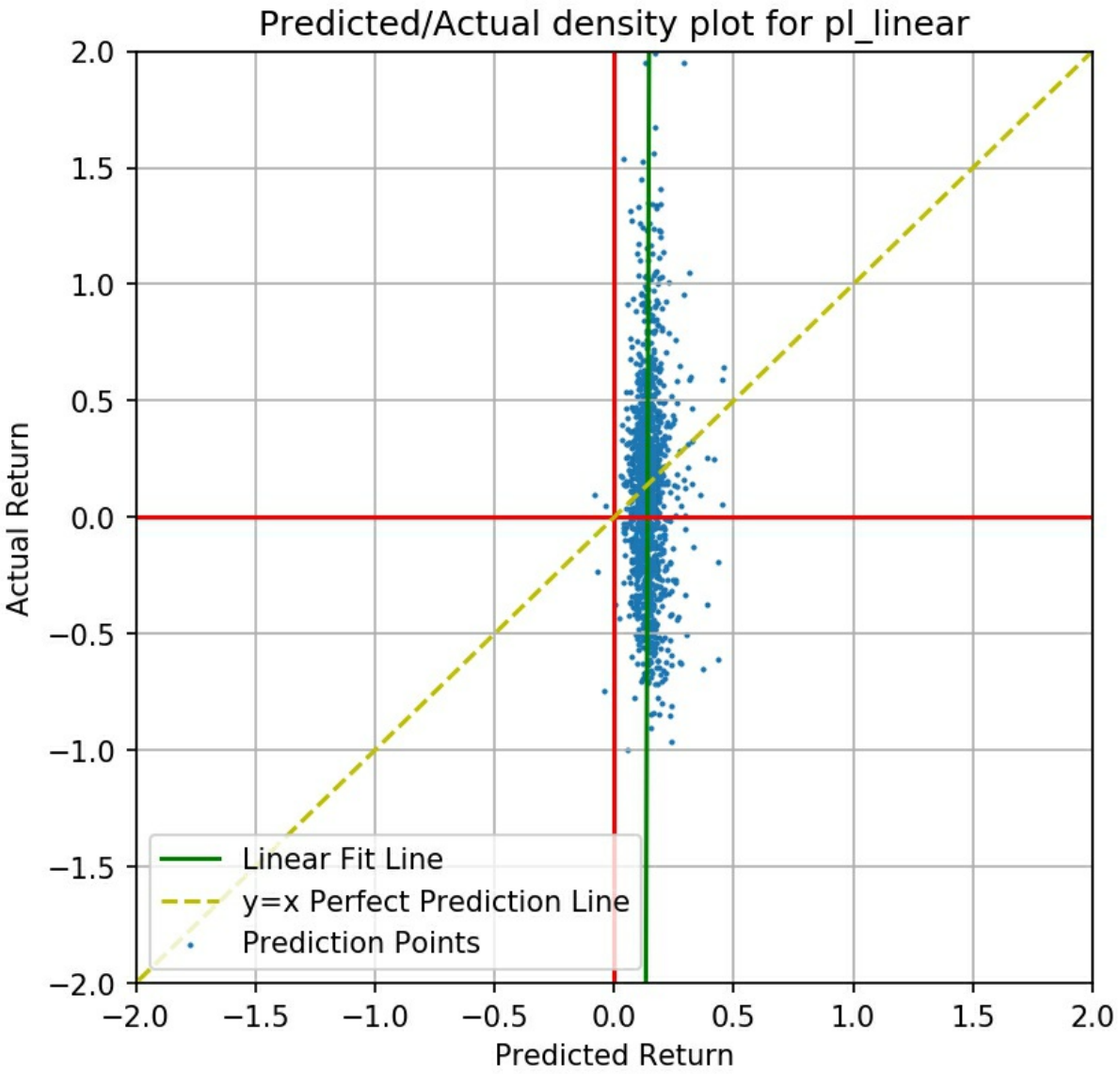
On this scatter plot we will also plot a linear regression line of the data. If this line looks something like y=-x our model would be completely wrong, and we will need to go back to the drawing board. We'll put this plot in a function for us to use when we want it, there are a lot of Scikit-Learn models we want to test and view the graph for.

In[7]:

```
# Output scatter plot and contour plot of density of points to see
# if prediciton matches reality
# Line of x=y is provided, perfect prediction would have all density
# on this line
# Also plot linear regression of the scatter
```

```
def plotDensityContourPredVsReal(model_name, x_plot, y_plot, ps):
    # Plotting scatter
    plt.scatter(x_plot, y_plot, s=1)
    # Plotting linear regression
    # Swap X and Y fit because prediction is quite centered
    # around one value.
    LinMod = LinearRegression().fit(y_plot.reshape(-1, 1), \
                                    x_plot.reshape(-1, 1))
    xx=[[-5],[5]]
    yy=LinMod.predict(xx)
    plt.plot(yy,xx,'g')
    # Plot formatting
    plt.grid()
    plt.axhline(y=0, color='r', label='_nolegend_')
    plt.axvline(x=0, color='r', label='_nolegend_')
    plt.xlabel('Predicted Return')
    plt.ylabel('Actual Return')
    plt.plot([-100,100],[-100,100],'y--')
    plt.xlim([-ps,ps])
    plt.ylim([-ps,ps])
    plt.title('Predicted/Actual density plot for {}'.format(model_name))
    plt.legend(['Linear Fit Line',\
                'y=x Perfect Prediction Line','Prediction Points'])

plotDensityContourPredVsReal('pl_linear', y_pred, y_test.to_numpy(), 2)
```

Out[7]:



A few things come out at us when we see the predictions plotted out like this, the distribution of actual returns (y-axis) is much wider than the predicted returns (x-axis), giving us a concentrated pillar of points on the plot.

This concentration of predicted returns at around x=0.1 tells us that there is a large bias that predicts a positive return. This is understandable as the stock market generally goes up at around 10% a year, and the relationships between our *X* matrix values and the stock returns are probably hard for our regressor to find, giving us this large bias. This would be a bit like predicting no rain every day, which would actually be a rather good prediction even though you aren't taking other information into account.

On closer inspection, the points seem to lean right ever so slightly. It is a far cry from the perfect $y=x$ prediction line, but at least it isn't totally random. Resampling and retraining the Linear Regressor doesn't seem to make that relationship disappear.

Let's see the ability of this prediction quantitatively by taking out the top 10 predicted stock returns and comparing them to the real returns. Visually this is like scanning the graph from right to left and taking the 10 points furthest to the right. Picking points based on the x-axis is all we can do, as these are our predictions, in the real world the y-axis information is obviously hidden from us.

Let's combine these 10 stocks into a portfolio and see what the predicted return would be, compared with the actual return. Eagle-eyed readers will notice that the time periods are unlikely to match up. These portfolios are synthetic things we are using to test our algorithms. This will be resolved later on when backtesting.

In[8]:

```
# See top 10 stocks and see how the values differ
# Put results in a DataFrame so we can sort it.
y_results = pd.DataFrame()
y_results['Actual Return'] = y_test
y_results['Predicted Return'] = y_pred

# Sort it by the prediced return.
y_results.sort_values(by='Predicted Return',
                      ascending=False,
                      inplace=True)
y_results.reset_index(drop=True,
                      inplace=True)

print('Predicted Returns:', \
       list(np.round(y_results['Predicted Return'].iloc[:10],2)))
print('Actual Returns:', \
       list(np.round(y_results['Actual Return'].iloc[:10],2)))

print('\nTop 10 Predicted Returns:', \
      round(y_results['Predicted Return'].iloc[:10].mean(),2) , '%','\n')
print('Actual Top 10 Returns:', \
      round(y_results['Actual Return'].iloc[:10].mean(),2) , '%','\n')

# See bottom 10 stocks and see how the values differ
print('\nBottom 10 Predicted Returns:', \
    round(y_results['Predicted Return'].iloc[-10:].mean(),2) , '%','\n')
print('Actual Bottom 10 Returns:', \
```

```
    round(y_results['Actual Return'].iloc[-10:].mean(),2) , '%','\n')
```

Out[8]:
Predicted Returns: [0.46, 0.45, 0.45, 0.43, 0.43, 0.42, 0.39, 0.39, 0.37, 0.36]
Actual Returns: [0.64, 0.59, 0.05, -0.61, -0.19, 0.25, -0.38, 0.26, -0.65, 0.09]

Top 10 Predicted Returns: 0.42 %

Actual Top 10 Returns: 0.01 %

Bottom 10 Predicted Returns: -0.01 %

Actual Bottom 10 Returns: 0.13 %

It seems that the regressors predictive ability isn't very good, but we did see a slight diagonal skew to our prediction/actual scatter plot so it is worth investigating further. Let's get these top/bottom results with different sampled train/test random states and see if these results are repeatable.

In[9]:

```
def observePredictionAbility(my_pipeline, X, y):
    '''
    For a given predictor pipeline.
    Create table of top10/bottom 10 averaged,
    10 rows of 10 random_states.
    to give us a synthetic performance result.
    Prints Top and Bottom stock picks
    '''
    Top10PredRtrns, Top10ActRtrns=[], []
    Bottom10PredRtrns, Bottom10ActRtrns=[], []

    for i in range (0,10):
        # Pipeline and train/test
        X_train, X_test, y_train, y_test =\
        train_test_split(X, y, test_size=0.1, random_state=42+i)
        my_pipeline.fit(X_train, y_train)
        y_pred = my_pipeline.predict(X_test)

        # Put results in a DataFrame so we can sort it.
        y_results = pd.DataFrame()
        y_results['Actual Return'] = y_test
        y_results['Predicted Return'] = y_pred

        # Sort it by the prediced return.
        y_results.sort_values(by='Predicted Return',
                              ascending=False,
                              inplace=True)
        y_results.reset_index(drop=True,
                              inplace=True)
```

```
        # See top 10 stocks and see how the values differ
        Top10PredRtrns.append(
            round(np.mean(y_results['Predicted Return'].iloc[:10])*100,
                2))
        Top10ActRtrns.append(
            round(np.mean(y_results['Actual Return'].iloc[:10])*100,
                2))

        # See bottom 10 stocks and see how the values differ
        Bottom10PredRtrns.append(
            round(np.mean(y_results['Predicted Return'].iloc[-10:])*100,
                2))
        Bottom10ActRtrns.append(
            round(np.mean(y_results['Actual Return'].iloc[-10:])*100,
                2))


print('Predicted Performance of Top 10 Return Portfolios:',
      Top10PredRtrns)
print('Actual Performance of Top 10 Return Portfolios:',
      Top10ActRtrns,'\n')
print('Predicted Performance of Bottom 10 Return Portfolios:',
      Bottom10PredRtrns)
print('Actual Performance of Bottom 10 Return Portfolios:',
      Bottom10ActRtrns)

print('--------------\n')

print('Mean Predicted Std. Dev. of Top 10 Return Portfolios:',
      round(np.array(Top10PredRtrns).std(),2))
print('Mean Actual Std. Dev. of Top 10 Return Portfolios:',
      round(np.array(Top10ActRtrns).std(),2))
print('Mean Predicted Std. Dev. of Bottom 10 Return Portfolios:',
      round(np.array(Bottom10PredRtrns).std(),2))
print('Mean Actual Std. Dev. of Bottom 10 Return Portfolios:',
      round(np.array(Bottom10ActRtrns).std(),2))


print('--------------\n')
#PERFORMANCE MEASURES HERE
print(\
'\033[4mMean Predicted Performance of Top 10 Return Portfolios:\033[0m',\
      round(np.mean(Top10PredRtrns), 2))
print(\
'\t\033[4mMean Actual Performance of Top 10 Return Portfolios:\033[0m',\
      round(np.mean(Top10ActRtrns), 2))
print('Mean Predicted Performance of Bottom 10 Return Portfolios:',\
      round(np.mean(Bottom10PredRtrns), 2))
print('\tMean Actual Performance of Bottom 10 Return Portfolios:',\
      round(np.mean(Bottom10ActRtrns), 2))

print('--------------\n')
```

```
    pass

observePredictionAbility(pl_linear, X, y)
[…]
```

Out[9]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 43.12
        Mean Actual Performance of Top 10 Return Portfolios: 37.05
Mean Predicted Performance of Bottom 10 Return Portfolios: -16386.59
        Mean Actual Performance of Bottom 10 Return Portfolios: 13.57
--------------
```

Feel free to try out different *test_size* numbers and see how this affects the portfolio returns. With a large *test_size*, the regressor will have little data to train from, though it will be able to select from a larger universe of stocks. Conversely, with a small *test_size*, our regressor will have a lot of content to learn from, though the universe of stocks to put into the portfolio would be small. A *test_size* of 0.1-0.2 is fine.

The accuracy may be low for our regressor, and it seems to be expecting some pretty dire stock returns for the lowest predictions (it predicts -16386% returns for the worst stocks when the most you can lose is 100%), however for stock picking we may be satisfied that the predicted trends are correct, certainly the top 10 predictions average a higher value than the bottom 10 predictions. You can run the loop for more than 10 iterations to see if the result sticks.

This model has definite potential to be used as part of our investing AI. We will backtest this model under closer scrutiny later, there are many more Machine Learning algorithms we can try out.

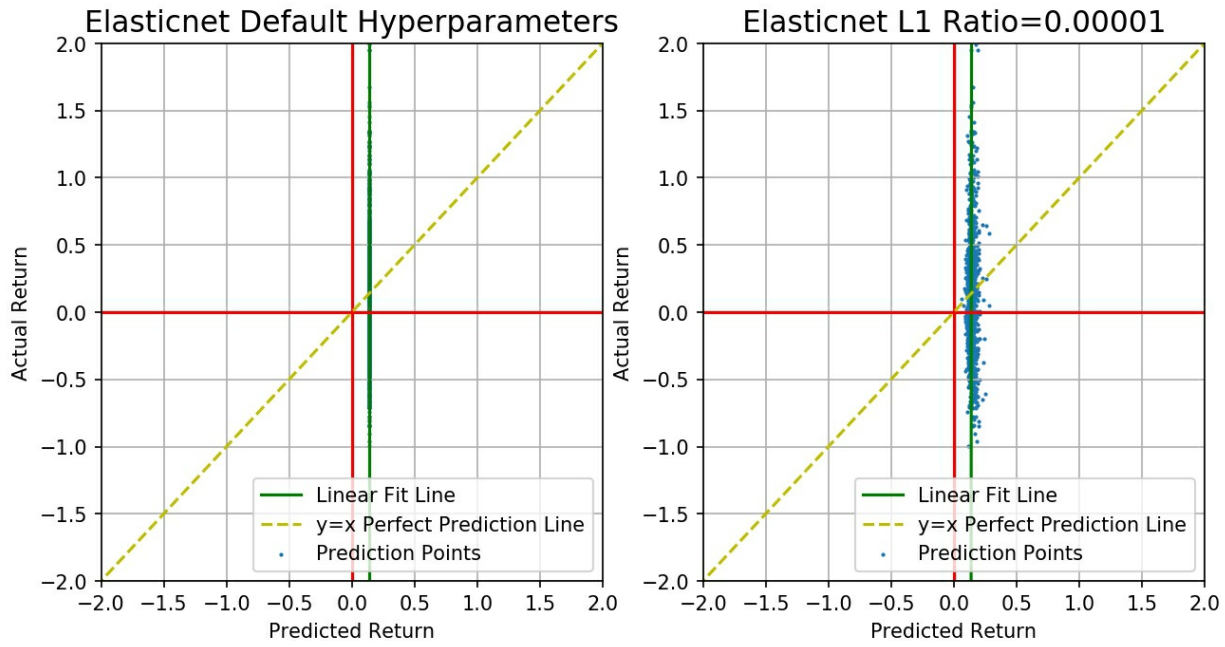
Elastic-Net Regression

Naturally progressing from Standard linear regression, we will try elastic net (which is just regularised linear regression) next using the same pipeline. Hopefully some regularisation will improve the prediction performance. This can easily be done by copying the code and simply replacing the Linear Regressor with elastic-net.

Regularising our Linear Regressor by using elastic-net appears to be a bad idea as the model is forcing a lot of weights to 0 with the default

regularisation, with our predictor is just outputting a return of ~12% all the time. We may as well have just taken the average 12% return and given that as our prediction and forget the algorithm.

Let's tweak some hyperparameters to see if we can get a better prediction with Elastic-Net (recall these are L1, L2, alpha).



Better results are possible by forcing the L1 ratio to be very low, which is forcing it to use Ridge regression only. Increasing L1 doesn't give us any improvement. The only other parameter we could tweak is alpha (default as 1) which turns off the elastic-net regularisation components. Turning this down makes our predictor effectively become the Linear Regressor.

In[10]:

```
# ElasticNet
from sklearn.linear_model import ElasticNet
from sklearn.preprocessing import PowerTransformer

pl_ElasticNet = Pipeline([
    ('Power Transformer', PowerTransformer()),
    ('ElasticNet', ElasticNet(l1_ratio=0.00001))
])

pl_ElasticNet.fit(X_train, y_train)
y_pred_lowL1 = pl_ElasticNet.predict(X_test)
from sklearn.metrics import mean_squared_error
print('train mse: ', \
```

```
      mean_squared_error(y_train, pl_ElasticNet.predict(X_train)))
print('test mse: ', \
      mean_squared_error(y_test, pl_ElasticNet.predict(X_test)))

import pickle
pickle.dump(pl_ElasticNet, open("pl_ElasticNet.p", "wb" ))
```

Checking out the predictive ability of this regressor with the function we made earlier:

In[11]:

```
observePredictionAbility(pl_ElasticNet, X, y)
```

Out[11]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 25.81
        Mean Actual Performance of Top 10 Return Portfolios: 35.68
Mean Predicted Performance of Bottom 10 Return Portfolios: -56219.01
        Mean Actual Performance of Bottom 10 Return Portfolios: 14.95
--------------
```

Your results may vary. Here in a similar manner to our Linear Regressor results, there is some predictive ability there so we will subject Lasso regression to backtesting. 60% might seem too good to be true, and indeed it is, remember the regressor is picking from the top 10 stocks for the entire 10 year period, performances in the backtest should be lower.

K-Nearest Neighbours

We'll try K-Nearest Neighbours next. This is a simple, but powerful Machine Learning algorithm, especially given that it doesn't actually 'learn' anything. The code stays nearly the same as for our Linear Regressor and our lasso regressor, we use the same pipeline with our power transformer.

In[12]:

```
from sklearn.neighbors import KNeighborsRegressor
pl_KNeighbors = Pipeline([
    ('Power Transformer', PowerTransformer()),
    ('KNeighborsRegressor', KNeighborsRegressor(n_neighbors=40))
])

pl_KNeighbors.fit(X_train, y_train)
y_pred = pl_KNeighbors.predict(X_test)
from sklearn.metrics import mean_squared_error
print('train mse: ', mean_squared_error(y_train, pl_KNeighbors.predict(X_train)))
print('test mse: ', mean_squared_error(y_test, y_pred))
```

Out[12]:
train mse: 0.3907707658880228
test mse: 0.30775430494312667

It would be nice for us to justify the value we fix for the hyperparameter K. Recall that we can find a good value for this kind of variable with Validation curves. We will plot the train and test set errors with increasing values of K, hoping to see a point where these lines start to have zero gradient. First let's run the loops and store the results in a DataFrame:

In[13]:

```
knn_validation_list = []
numNeighbours = [4,8,16,32,64,100]
runNum = 40

for i in numNeighbours:
    print('Trying K='+str(i))
    for j in range(0, runNum):
    #Get a new train/test split
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.1,
                                                    random_state=42+j)
        pl_KNeighbors = Pipeline([
            ('Power Transformer', PowerTransformer()),
            ('KNeighborsRegressor', KNeighborsRegressor(n_neighbors=i))
            ]).fit(X_train, y_train)

        y_pred = pl_KNeighbors.predict(X_test)

        resultThisRun = [i,
            mean_squared_error(y_train, pl_KNeighbors.predict(X_train)),
            mean_squared_error(y_test, y_pred)]

        knn_validation_list.append(resultThisRun)

knn_validation_df=pd.DataFrame(knn_validation_list,
                               columns=['numNeighbours',
                                        'trainError',
                                        'testError'])
knn_validation_df
```

Out[13]:

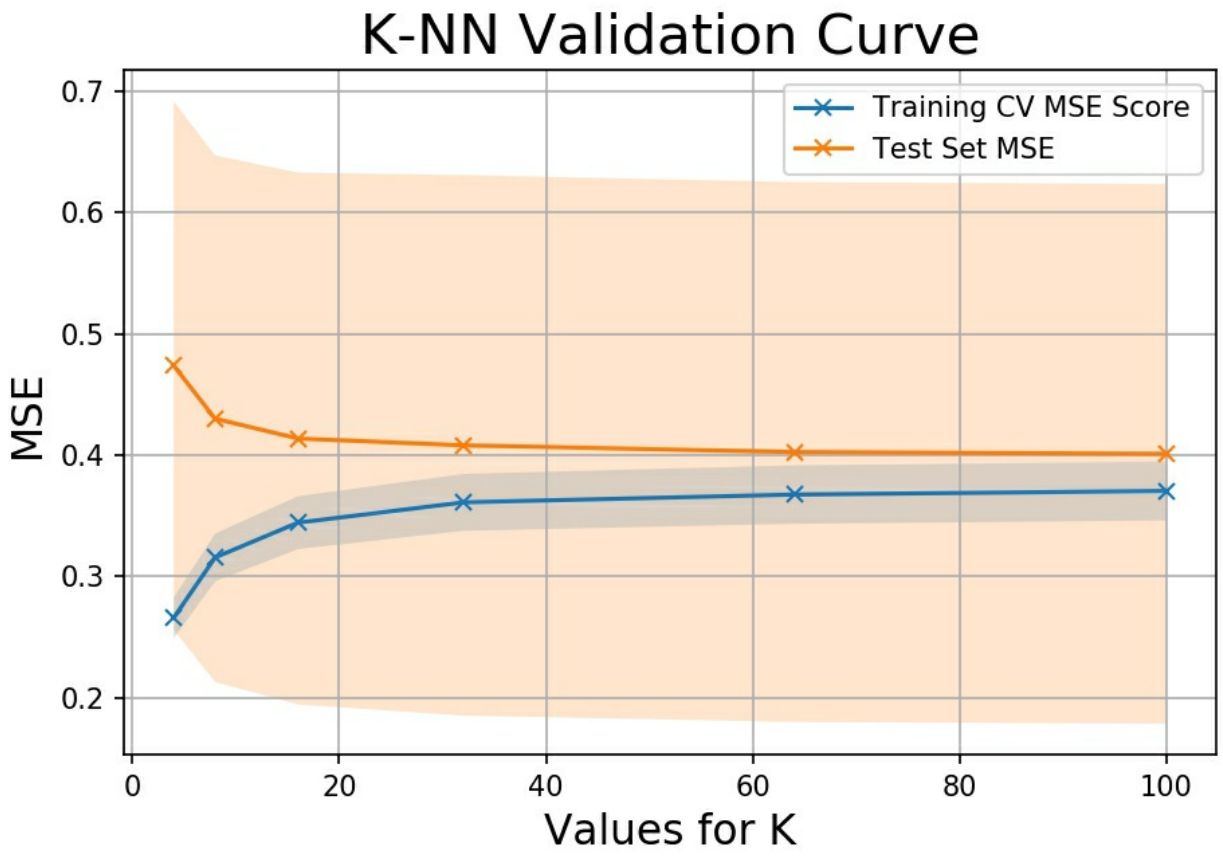
	numNeighbours	trainError	testError
0	4	0.254972	0.620575
1	4	0.292280	0.401202
2	4	0.296965	0.369807
3	4	0.283583	0.451889
4	4	0.306689	0.348510
...	...	...	...
235	100	0.407679	0.263474
236	100	0.329603	0.576972
237	100	0.330126	0.575280
238	100	0.396877	0.307621
239	100	0.325842	0.593721

240 rows × 3 columns

From this point we can combine our data into a validation curve plot.

In[14]:

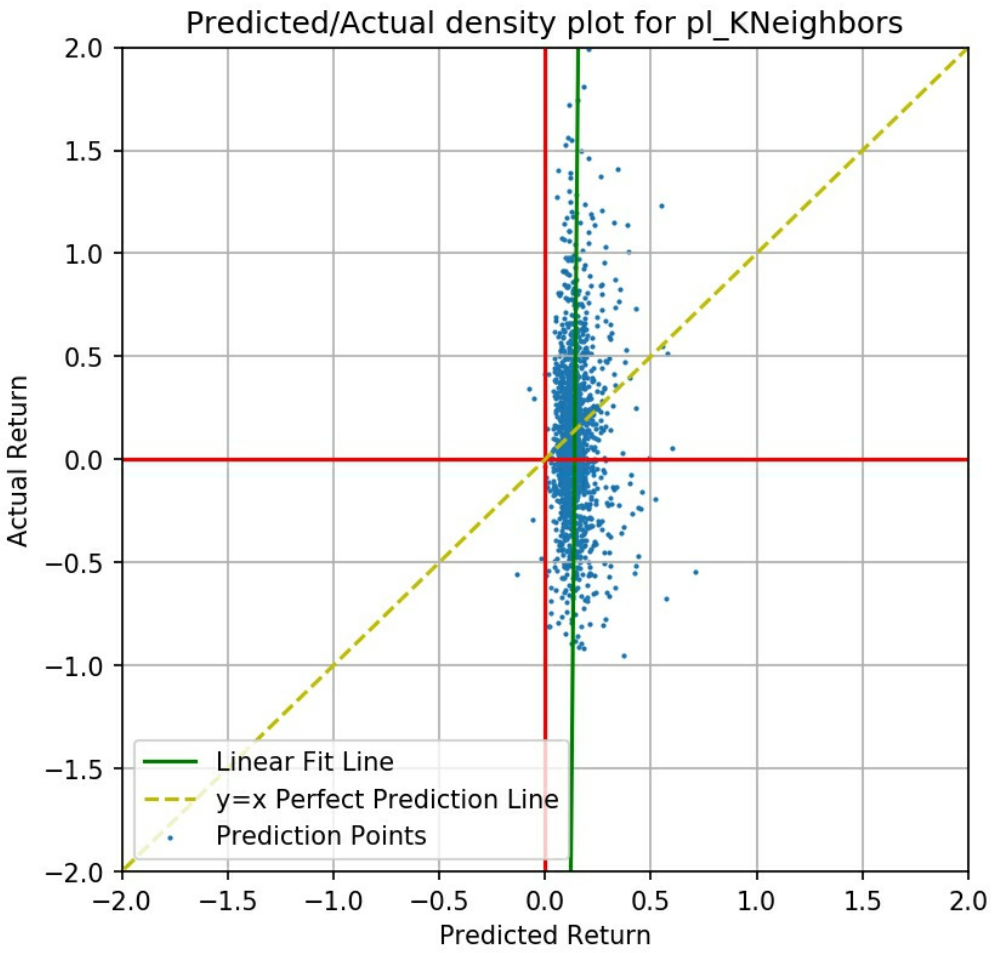
```
knn_results_df['trainErrorsMean'].plot(style='-x', figsize=(9,7))
knn_results_df['testErrorsMean'].plot(style='-x')

plt.fill_between(knn_results_df.index,\
                 knn_results_df['trainErrorsMean']
                 -knn_results_df['trainErrorsStd'],
                 knn_results_df['trainErrorsMean']
                 +knn_results_df['trainErrorsStd'], alpha=0.2)
plt.fill_between(results_df.index,\
                 knn_results_df['testErrorsMean']
                 -knn_results_df['testErrorsStd'],
                 knn_results_df['testErrorsMean']
                 +knn_results_df['testErrorsStd'], alpha=0.2)
plt.grid()
plt.legend(['Training CV MSE Score','Test Set MSE'])
plt.ylabel('MSE', fontsize=15)
plt.xlabel('Values for K', fontsize=15)
plt.title('K-NN Validation Curve', fontsize=20)
plt.ylim([0, 0.8]);
```



We are seeing a lot of variation here, but the performance in MSE terms definitely improves with increasing values of K on average. We'll take a value of *K=40* going forward. As with the linear regression we are running this with many samplings, plotting this individually sometimes gives us strange looking graphs, which is understandable given the variation, though on average we see it should level out at around *K=40*.

In[15]:

```
plotDensityContourPredVsReal('pl_KNeighbors', y_pred, y_test.to_numpy(),2)
```



As with the Linear Regressors, let's using the same top/bottom prediction test to see if there is some predictive power present.

In[16]:

```
observePredictionAbility(pl_KNeighbors, X, y)
```

Out[16]:

Mean Predicted Performance of Top 10 Return Portfolios: 55.64

Mean Actual Performance of Top 10 Return Portfolios: 32.66

Mean Predicted Performance of Bottom 10 Return Portfolios: -5.31

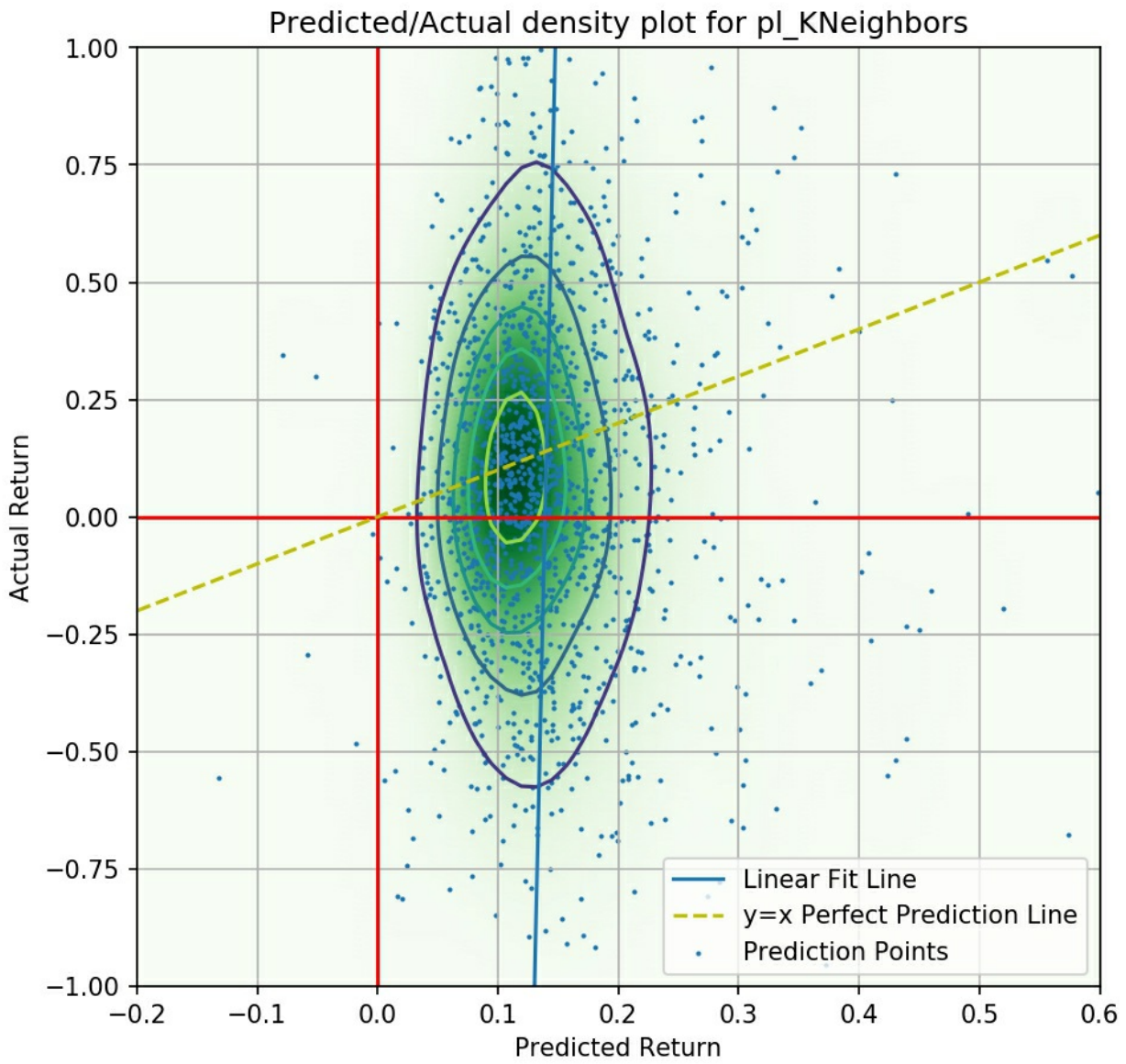
Mean Actual Performance of Bottom 10 Return Portfolios: -2.53

Our hypothetical top 10 predicted stock portfolio would return 32%, on average that sounds not too bad by itself, but the variation is quite bad. At least the top 10 stock picks are outperforming the bottom 10 stock picks on average.

For an intuitive understanding of the predictions, let's plot out a contour plot of the density of the scatter plot points. Here the *scipy.stats kde* (Kernel Density Estimator) object is used to lay down a contour plot over our scatterplot.

In[17]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import kde # Need for Kernel Density Calculation
from sklearn.linear_model import LinearRegression

def plotDensityContourPredVsRealContour(model_name, x_plot, y_plot, ps):
#x_plot, y_plot = y_pred, y_test.to_numpy()
    resolution = 40
    # Make a gaussian kde on a grid
    k = kde.gaussian_kde(np.stack([x_plot,y_plot]))
    xi, yi = np.mgrid[-ps:ps:resolution*4j, -ps:ps:resolution*4j]
    zi = k(np.vstack([xi.flatten(), yi.flatten()]))
    # Plotting scatter and contour plot
    plt.pcolormesh(xi,
                   yi,
                   zi.reshape(xi.shape),
                   shading='gouraud',
                   cmap=plt.cm.Greens)
    plt.contour(xi, yi, zi.reshape(xi.shape) )
    plt.scatter(x_plot, y_plot, s=1)
    # Plotting linear regression
    LinMod = LinearRegression()
    LinMod.fit(y_plot.reshape(-1, 1), x_plot.reshape(-1, 1),)
    xx=[[-2],[2]]
    yy=LinMod.predict(xx)
    plt.plot(yy,xx)
    # Plot formatting
    plt.grid()
    plt.axhline(y=0, color='r', label='_nolegend_')
    plt.axvline(x=0, color='r', label='_nolegend_')
    plt.xlabel('Predicted Return')
    plt.ylabel('Actual Return')
    plt.plot([-100,100],[-100,100],'y--')
    plt.xlim([-ps*0.2,ps*0.6]) #
    plt.ylim([-ps,ps])
    plt.title('Predicted/Actual density plot for {}'.format(model_name))
    plt.legend([
        'Linear Fit Line','y=x Perfect Prediction Line',
        'Prediction Points'])

plotDensityContourPredVsRealContour('pl_KNeighbors', y_pred, y_test.to_numpy(),1)
```



Our contour plot gives displays a ‘hill’ in the centre where all our scatter points are densest. This hill resembles a potato shape on the graph with its centre around the average returns we would expect of stocks, which mostly seems to be around 10%. Luckily for us, the potato isn’t symmetrical, it leans a little to the right, very slightly.

If we are picking stocks that are to the right-hand side of our graph, we should be slightly more likely to pick high return stocks. As we are picking 10 stocks from that group we are protected from the really bad predictions through diversification, whilst keeping the performance advantage assuming we don’t go through some paradigm shift in the way corporations and

markets work. We will take this predictor through to the next section.

Support Vector Machine Regressor

A word of caution on this regressor: it takes a very long time to train so it would be a good idea to skip this code if you are following this book throught eh Jupyter notebooks. Given the complexity of support vector machine regression it's not all that surprising it takes so long to train.

Let's try a support vector machine regressor using the default hyperparameters:

In[18]:

```
from sklearn.svm import SVR

pl_svm = Pipeline([
    ('Power Transformer', PowerTransformer()),
    ('SVR', SVR())
])

pl_svm.fit(X_train, y_train)
y_pred = pl_svm.predict(X_test)
from sklearn.metrics import mean_squared_error
print('mse: ', mean_squared_error(y_test, y_pred))
from sklearn.metrics import mean_absolute_error
print('mae: ', mean_absolute_error(y_test, y_pred))
```

Out[18]:

mse: 0.561148780499458
mae: 0.33737759100211656

An error of around 0.3 is in line with what our other regressors are doing, lets see if the support vector machine performs well. To see how consistent this algorithm is, and the expected returns, we'll use our prediction function.

In[19]:

```
observePredictionAbility(pl_svm, X, y)
```

Out[19]:

Mean Predicted Performance of Top 10 Return Portfolios: 26.9

Mean Actual Performance of Top 10 Return Portfolios: 12.53

Mean Predicted Performance of Bottom 10 Return Portfolios: -142381.15

Mean Actual Performance of Bottom 10 Return Portfolios: -6.98

Several hyperparameters can be optimised in a regression support vector machine. We have the parameter *C* (The outlier penalty modifier), *epsilon* (The width of the street or tube which our support vector machine will be placing) and when we are using a radial basis function for non-linear fitting *gamma* needs to be set (*gamma* modifies the pointedness of the radial basis function).

We can try out a grid search to find the optimal parameters of *C*, *epsilon* and *gamma* (when the RBF kernel is used). Although this is very slow, this method is recommended by Scikit-Learn creators themselves (*https://scikit-learn.org/stable/modules/svm.html#parameters-of-the-rbf-kernel*) and is an exhaustive search covering every parameter. This particular search took a whole day to compute, luckily you don't have to follow through with the grid search yourself as your author has found the optimal parameters for you. Running this function is best done in parallel to use all the horsepower fromy our CPU to reduce computing time. Set the *n_jobs* variable to the number of CPU cores you want to use.

When doing the grid search, we shouldn't just aim for lowest RMSE to make the stock performance prediction more accurate. The reason for this is that even though the predictions may be more accurate, we don't want accuracy per se. What we actually want is the performance of the highest predicted return stocks.

This is where we deviate from doing normal data science, we are using Machine Learning tools to create a stock picker, so the accuracy of our model isn't actually what we want to optimise for, though it is a good proxy, we want stock performance.

If predictions of the top 10 stock returns of 50% when the actual returns are 100%, this will give us a large error, however, a return prediction of 10% matching an actual return of 12% will give us a small error. The grid search will favour the parameters that give us the smaller error over our superior return, yet inaccurate predictors.

We could create our own grid search function and program in our own criteria for prediction success by aiming for the highest 10 positive returns. Or perhaps the highest and lowest ten being far apart, or those methods combined with prediction accuracy, or anything else your imagination may

come up with.

Changing the way you evaluate your predictors will influence the style of investment the AI will perform. If you are the kind of person who likes tweaking things this is a good place to do it.

For the sake of simplicity we will do a kind of grid search by just modifying our *observePredictionAbility* function to see what parameters give us the highest returns, and use that as the support vector machine candidate for backtesting.

In[20]:

```
def getMyPredictionAbility(my_pipeline, X, y):
    '''
    For a given predictor pipeline.
    Create table of top10 stock picks averaged,
    and average that over several runs,
    to give us a synthetic performance result.
    '''
    Top10PredRtrns, Top10ActRtrns=[], []
    Bottom10PredRtrns, Bottom10ActRtrns=[], []

    for i in range (0,10):
        # Pipeline and train/test
        X_train, X_test, y_train, y_test =\
        train_test_split(X, y, test_size=0.1, random_state=42+i)
        my_pipeline.fit(X_train, y_train)
        y_pred = my_pipeline.predict(X_test)

        # Put results in a DataFrame so we can sort it.
        y_results = pd.DataFrame()
        y_results['Actual Return'] = y_test
        y_results['Predicted Return'] = y_pred

        # Sort it by the prediced return.
        y_results.sort_values(by='Predicted Return',
                              ascending=False,
                              inplace=True)
        y_results.reset_index(drop=True,
                              inplace=True)

        # See top 10 stocks and see how the values differ
        Top10PredRtrns.append(round(
                              np.mean(
                              y_results['Predicted Return'].iloc[:10])
                              * 100, 2))
        Top10ActRtrns.append(round(
                             np.mean(
                             y_results['Actual Return'].iloc[:10])
```

```
                        * 100, 2))

        # See bottom 10 stocks and see how the values differ
        Bottom10PredRtrns.append(round(
                                np.mean(
                               y_results['Predicted Return'].iloc[-10:])
                                * 100, 2))
        Bottom10ActRtrns.append(round(
                                np.mean(
                               y_results['Actual Return'].iloc[-10:])
                                * 100, 2))

    return round(np.mean(Top10ActRtrns),2), \
           round(np.array(Top10ActRtrns).std(),2)
```

Out[20]:

```
return:13.85 std.dev. 3.58 For Kernel: linear C: 1 Gamma:0 Epsilon 0.05
return:11.78 std.dev. 3.42 For Kernel: linear C: 1 Gamma:0 Epsilon 0.1
return:12.7 std.dev. 3.03 For Kernel: linear C: 1 Gamma:0 Epsilon 0.2
return:13.85 std.dev. 3.6 For Kernel: linear C: 10 Gamma:0 Epsilon 0.05
return:11.78 std.dev. 3.43 For Kernel: linear C: 10 Gamma:0 Epsilon 0.1
return:12.7 std.dev. 3.02 For Kernel: linear C: 10 Gamma:0 Epsilon 0.2
return:13.06 std.dev. 3.52 For Kernel: linear C: 100 Gamma:0 Epsilon0.05
return:12.04 std.dev. 3.44 For Kernel: linear C: 100 Gamma:0 Epsilon 0.1
[…]
```

Optimal hyperparameters found in this grid search are:

kernel='rbf', C=100, gamma=0.1, epsilon=0.1

Depending on your data the results may differ slightly. Trying these out on our support vector regressor:

In[21]:

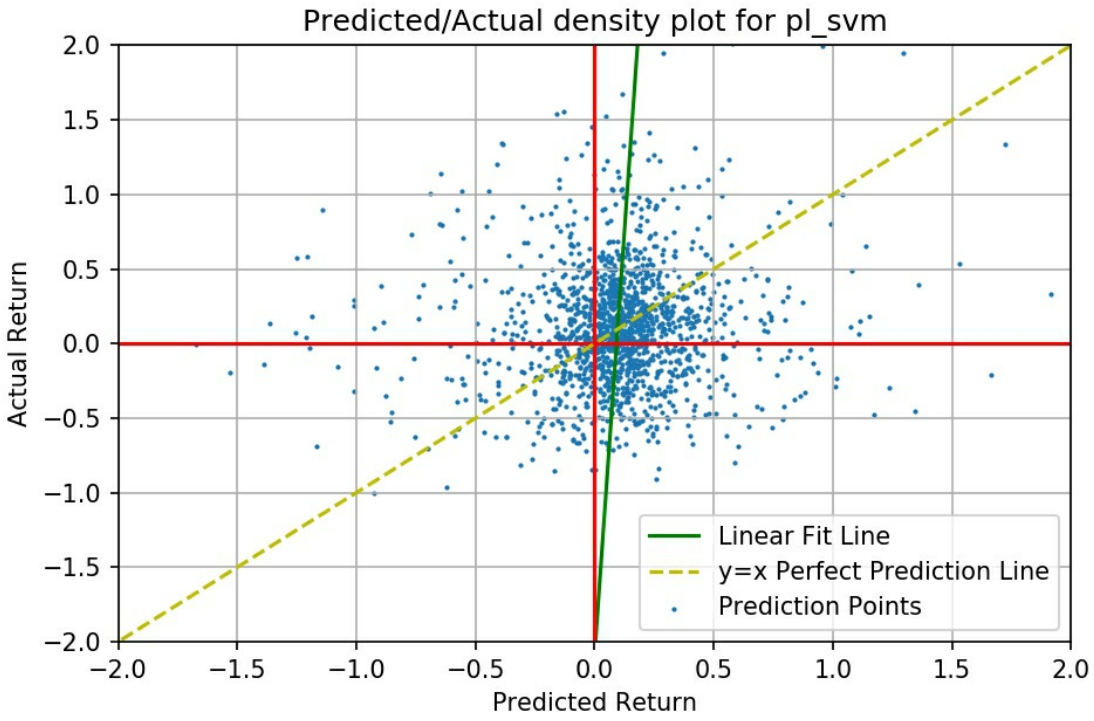
```
[…]
pl_svm = Pipeline([
    ('Power Transformer', PowerTransformer()),
    ('SVR', SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1))
])
[…]
observePredictionAbility(pl_svm, X, y)
```

Out[21]:

```
Mean Predicted Performance of Top 10 Return Portfolios: 274.76
        Mean Actual Performance of Top 10 Return Portfolios: 38.34
Mean Predicted Performance of Bottom 10 Return Portfolios: -168.03
        Mean Actual Performance of Bottom 10 Return Portfolios: 12.63
```

With optimised hyperparameters we get slightly higher performance. As with

the other regressors the support vector machine regressor overpredicts and underdelivers, but it gives a promising result overall: We have a difference between top and bottom stocks of 25%, and the scatter plot shows a tendency for points to lean to the right, so it's worth trying this regressor our later.



Decision Tree Regressor

To improve the ecology of our algorithm selection, we will use Decision Trees and Random Forests next. These algorithms do not need the power transformer so we'll take that out. Trying out the default *decisionTreeRegressor*:

In[22]:

```
from sklearn.tree import DecisionTreeRegressor

pl_decTree = Pipeline([
    ('DecisionTreeRegressor',
     DecisionTreeRegressor(random_state=42)) # no need scaler
])

pl_decTree.fit(X_train, y_train)
y_pred = pl_decTree.predict(X_test)
from sklearn.metrics import mean_squared_error
print('train mse: ',
      mean_squared_error(y_train,
```

```
                        pl_decTree.predict(X_train)))
print('test mse: ',
      mean_squared_error(y_test, y_pred))
```

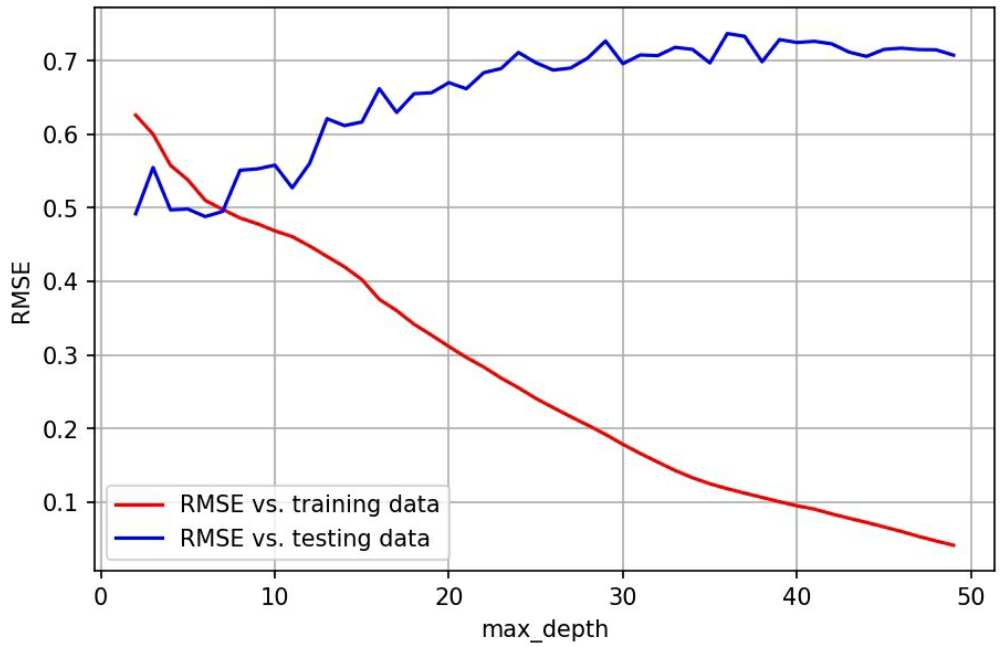
Out[22]:
train mse: 2.7399716575099673e-10
test mse: 1.1552766810449335

With the training error near zero and the test error at ~1.15, it's evident we have overfitting on our hands, just like with the tutorial earlier on. We might need to fix the *max_depth* hyperparameter for our Decision Tree to prevent overfitting if the prediction ability isn't optimal. To find a good value we'll take a look at the validation curve with increasing *max_depth*.

In[23]:

```
train_errors, test_errors, test_sizes=[], [], []

for i in range(2,50):
    pl_decTree = Pipeline([
    ('DecisionTreeRegressor', \
      DecisionTreeRegressor(random_state=42, max_depth=i))])
    pl_decTree.fit(X_train, y_train)

    y_pred = pl_decTree.predict(X_test)
    train_errors.append(mean_squared_error(y_train, \
                                           pl_decTree.predict(X_train)))
    test_errors.append(mean_squared_error(y_test, y_pred))
    test_sizes.append(i)

plt.plot(test_sizes, np.sqrt(train_errors),\
         'r',test_sizes, np.sqrt(test_errors),'b')

plt.legend(['RMSE vs. training data','RMSE vs. testing data'])
plt.grid()
plt.ylabel('RMSE');
plt.xlabel('max_depth');
```

Out[23]:



This validation curve doesn't look too good, the test set error shows no sign of decreasing with increasing tree depth, in fact, it is almost always increasing, and the training error is little better, simply decreasing over time, eventually overfitting the data.

In choosing our *max_depth* we can't choose a value of 40+ because the Decision Tree is obviously overfitting there with a training error of 0. A lower tree depth is desirable, however, we can't choose a very low number, as there are 15 columns to our X matrix, which we would like to make the most of in prediction.

For the moment let's use a *tree_depth* of 10, Any less and we will be utilising too little of our data, yet the error between test and training sets is reasonably close. Let us put the model through our prediction ability function and see how it performs.

In[24]:

```
observePredictionAbility(pl_decTree, X, y)
```

Out[24]:

Mean Predicted Performance of Top 10 Return Portfolios: 388.34

Mean Actual Performance of Top 10 Return Portfolios: 16.94

Mean Predicted Performance of Bottom 10 Return Portfolios: -61.29

Mean Actual Performance of Bottom 10 Return Portfolios: 4.11

The top 10 portfolios state that our model thinks it can get 388%+ returns no matter the sampling, yet the actual returns average ~16%. The variance is quite high too, if we print out the individual portfolio results from *observePredictionAbility*:

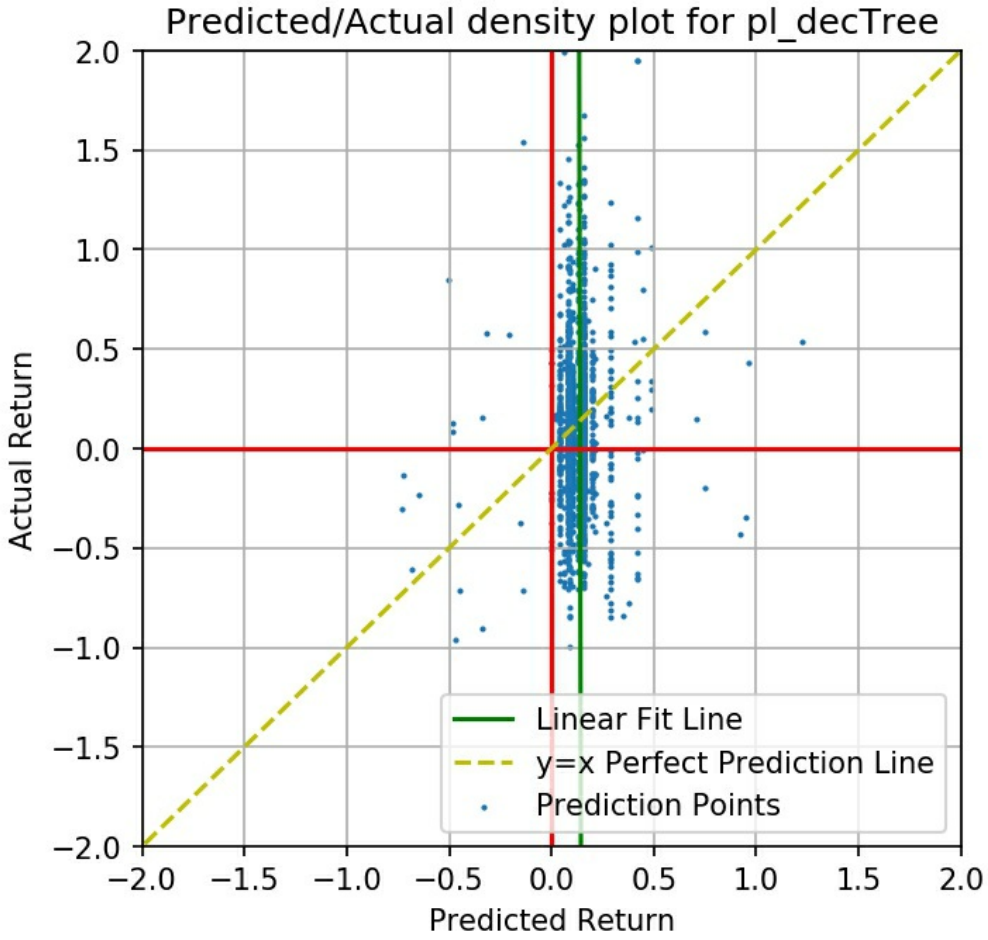
Actual Performance of Top 10 Return Portfolios: [0.95, 2.69, -5.94, 57.4, 67.45, 76.42, -5.55, 5.77, -10.7, -19.07]

Taking a look at the prediction scatterplot for one of the samples, the randomness in our results is showing up visually:

In[25]:

```
plotDensityContourPredVsReal('pl_decTree', y_pred, y_test.to_numpy(),2)
```

Out[25]:



We can easily see the low depth Decision Tree giving stepped predictions, which is unlikely to be what we want.

It is not all bad news, at least we are not getting a mono result like with ElasticNet regression. We should bear in mind that the test set here is only 10% of all of our data, which is quite a handicap for a stock picker, human or AI. Let's try a few values of *max_depth* in a loop just like we did with our support vector machines and see if we can improve the stock selection performance.

In[26]:

```
for my_depth in [4,5,6,7,8,9,10,15,20,30,50]:
    decTree = DecisionTreeRegressor(random_state=42, max_depth=my_depth)
    performance, certainty = getMyPredictionAbility(decTree, X, y)
    print('Tree max_depth:', my_depth,
          'Average Return:',  performance,
          'Standard Deviation:', certainty)
```

Out[26]:

```
Tree max_depth: 4 Average Return: 25.33 Standard Deviation: 37.34
Tree max_depth: 5 Average Return: 38.19 Standard Deviation: 62.07
Tree max_depth: 6 Average Return: 46.63 Standard Deviation: 65.1
Tree max_depth: 7 Average Return: 41.45 Standard Deviation: 54.91
Tree max_depth: 8 Average Return: 34.47 Standard Deviation: 53.89
Tree max_depth: 9 Average Return: 15.94 Standard Deviation: 26.49
Tree max_depth: 10 Average Return: 16.94 Standard Deviation: 33.76
Tree max_depth: 15 Average Return: 24.97 Standard Deviation: 33.35
Tree max_depth: 20 Average Return: 29.16 Standard Deviation: 45.88
Tree max_depth: 30 Average Return: 26.37 Standard Deviation: 28.03
Tree max_depth: 50 Average Return: 31.58 Standard Deviation: 36.24
```

These are interesting results. A Decision Tree with a depth of only 6 delivers a very high average return, but it is *extremely* variable. The deviation peters out at a *max_depth* of about 9, at which point adding more trees doesn't seem to do much. The general trend is for the return to decrease with increasing tree depth, and for the standard deviation of the results to decrease too.

We can justifiably use a *max_depth* value of 15 for the Decision Tree, which gives us a good return, and it isn't as variable as for predictors with only 4 trees. Given how variable our best results are, we would like to keep the variability down. After training with this tree depth the results are:

In[27]:

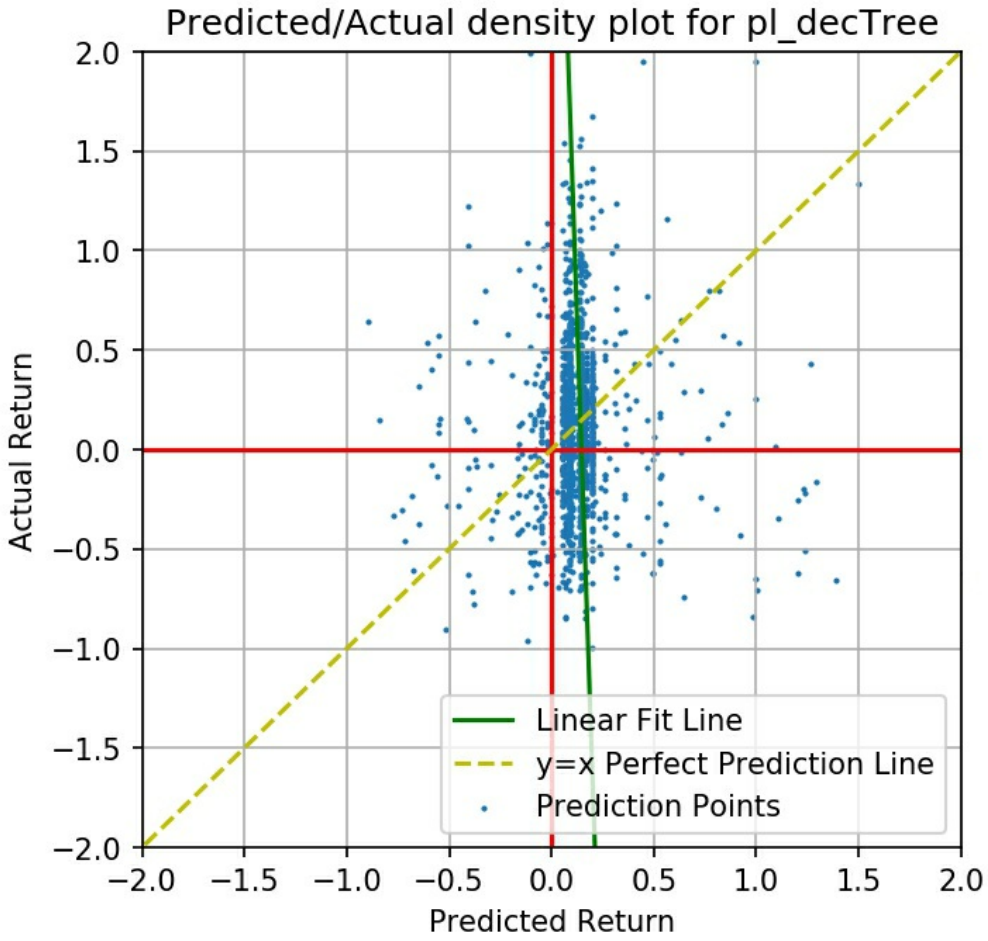
```
[…]
observePredictionAbility(pl_decTree, X, y)
```

Out[27]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 486.42
```

Mean Actual Performance of Top 10 Return Portfolios: 24.97
Mean Predicted Performance of Bottom 10 Return Portfolios: -80.73
Mean Actual Performance of Bottom 10 Return Portfolios: 7.21

This result is not bad considering the result variance was reduced at little cost. The Decision Tree, having more depth, gives a more scattered set of predictions than before:



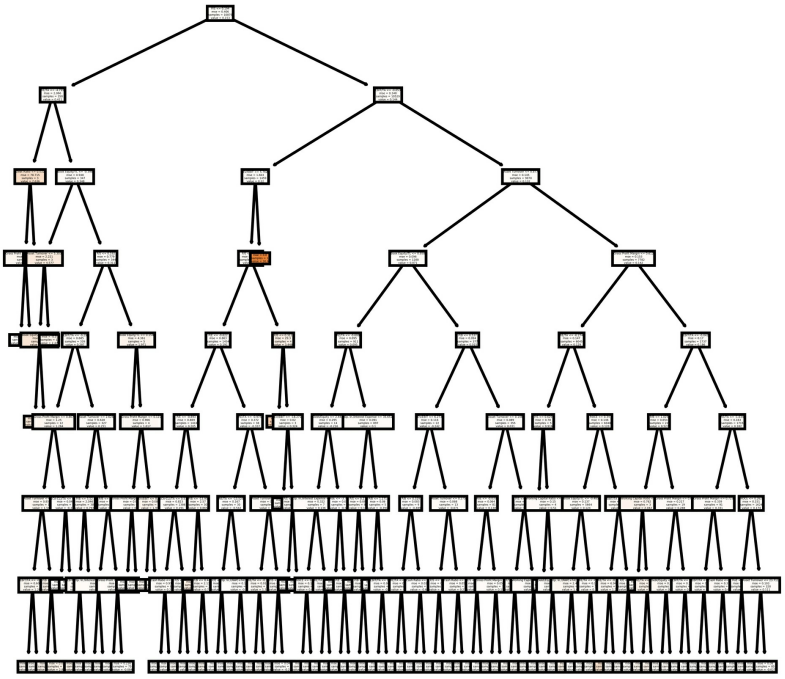
Under the Hood

If you want to take a look under hood at the Decision Tree to see how it is predicting returns, you may print it out with the same functions as you learnt with the Decision Tree classifier. You will see a lot of boxes with rules just like for the classifier saying things like “If P/E is greater than 11.67” etc. As the tree is relatively large you need to view the generated picture of it in another program outside of Jupyter Notebook to see what ratios are used in

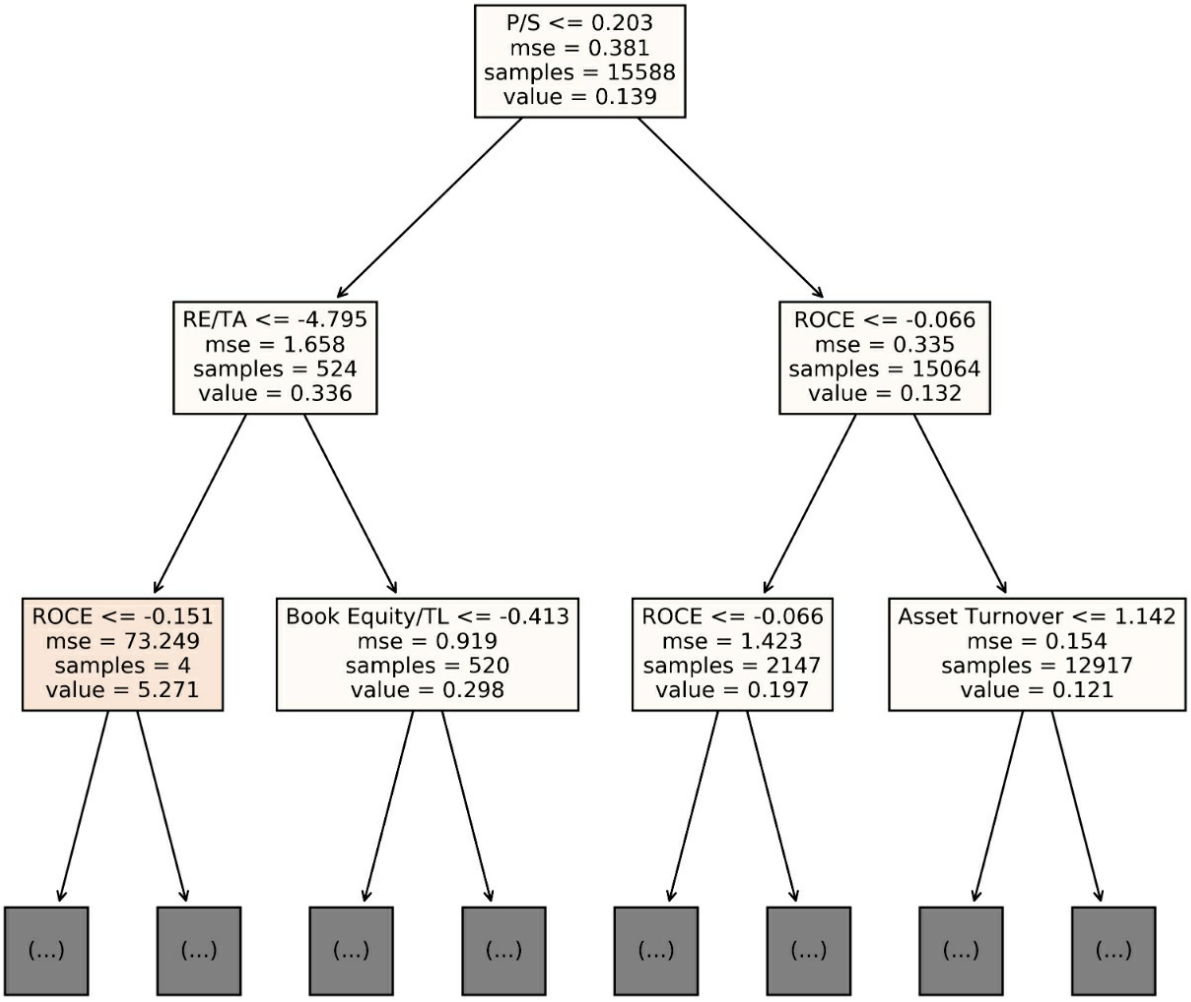
the tree.

In[28]:

```
reg_decTree = DecisionTreeRegressor(random_state=42, max_depth=20)
reg_decTree.fit(X_train, y_train)

from sklearn import tree # Need this to see Decision Tree.
plt.figure(figsize=(5,5), dpi=1000) # set figsize so we can see it

tree.plot_tree(reg_decTree, feature_names = x.keys(),  filled = True);  # semicolon here to supress output

plt.savefig('RegDecTree.png')
```



You might think it looks a little bit like how your brain might go about picking stocks when looking at company metrics. If we zoom in on the first few splits we can see something enlightening coming from our algorithm:



Interestingly the first branch our Decision Tree grows is a filter on the Price/Sales ratio, so the tree thinks this feature is a significant predictor of success. Through other statistical methods, this was the same finding as James O'Shaughnessy in his book *What Works on Wall Street* and David Dreman in *Contrarian Investment Strategies*.

Your Decision Tree may make branch splits on different features, in the last edition of this book the first split was done on Price/Sales ratio too, but the lower ratios differ, certainly ROCE had less of an impact then. Any trees you grow will differ depending on the information you give them. Given that the Decision Tree is showing some prediction ability, it is worth keeping our Decision Tree in portfolio backtesting.

Random Forest Regressor

If the Decision Tree can tease out some prediction ability, a forest made of them should be able to do better.

There are a two hyperparameters we need to set, the tree depth and the number of trees. For the number of trees in the forest we'll leave it at the default of 100 (We will do fine tuning in backtesting), and we'll run a loop to figure out a good value for tree depth of each tree in our forest.

In[1]:

```
from sklearn.ensemble import RandomForestRegressor
#for my_depth in [4, 6, 10, 16, 24]: # faster
for my_depth in range(4,21):
    rForest = RandomForestRegressor(random_state=42, max_depth=my_depth)
    performance, certainty = getMyPredictionAbility(rForest, X, y)
    print('Tree max_depth:', my_depth,
          'Average Return:', performance,
          'Standard Deviation:', certainty)
```

Out[1]:

```
Tree max_depth: 4 Average Return: 47.38 Standard Deviation: 62.52
Tree max_depth: 5 Average Return: 60.6 Standard Deviation: 59.6
Tree max_depth: 6 Average Return: 58.64 Standard Deviation: 65.03
Tree max_depth: 7 Average Return: 60.2 Standard Deviation: 61.1
Tree max_depth: 8 Average Return: 40.49 Standard Deviation: 64.12
Tree max_depth: 9 Average Return: 37.11 Standard Deviation: 58.64
Tree max_depth: 10 Average Return: 40.34 Standard Deviation: 59.46
Tree max_depth: 11 Average Return: 50.05 Standard Deviation: 78.67
Tree max_depth: 12 Average Return: 42.55 Standard Deviation: 62.1
Tree max_depth: 13 Average Return: 34.73 Standard Deviation: 61.61
Tree max_depth: 14 Average Return: 43.12 Standard Deviation: 69.77
Tree max_depth: 15 Average Return: 41.14 Standard Deviation: 60.89
Tree max_depth: 16 Average Return: 50.91 Standard Deviation: 74.12
Tree max_depth: 17 Average Return: 42.97 Standard Deviation: 62.81
Tree max_depth: 18 Average Return: 45.19 Standard Deviation: 69.55
Tree max_depth: 19 Average Return: 41.18 Standard Deviation: 63.83
Tree max_depth: 20 Average Return: 38.26 Standard Deviation: 65.79
```

It seems as though the depth of our trees has little influence on the performance. These are the best results we've had so far. We'll use a tree depth of 10 going forward as this lowers computing time and works well relative to other estimators, we can do further tweaking after backtesting and finding the models we want.

In[2]:

```
from sklearn.ensemble import RandomForestRegressor
rfregressor = RandomForestRegressor(random_state=42, max_depth=10)
rfregressor.fit(X_train, y_train)
```

```
y_pred = rfregressor.predict(X_test)

print('train mse: ',
      mean_squared_error(y_train,
                         rfregressor.predict(X_train)))
print('test mse: ',
      mean_squared_error(y_test,
                         y_pred))
```

Out[2]:
train mse: 0.20971718040583018
test mse: 0.2622593995036744

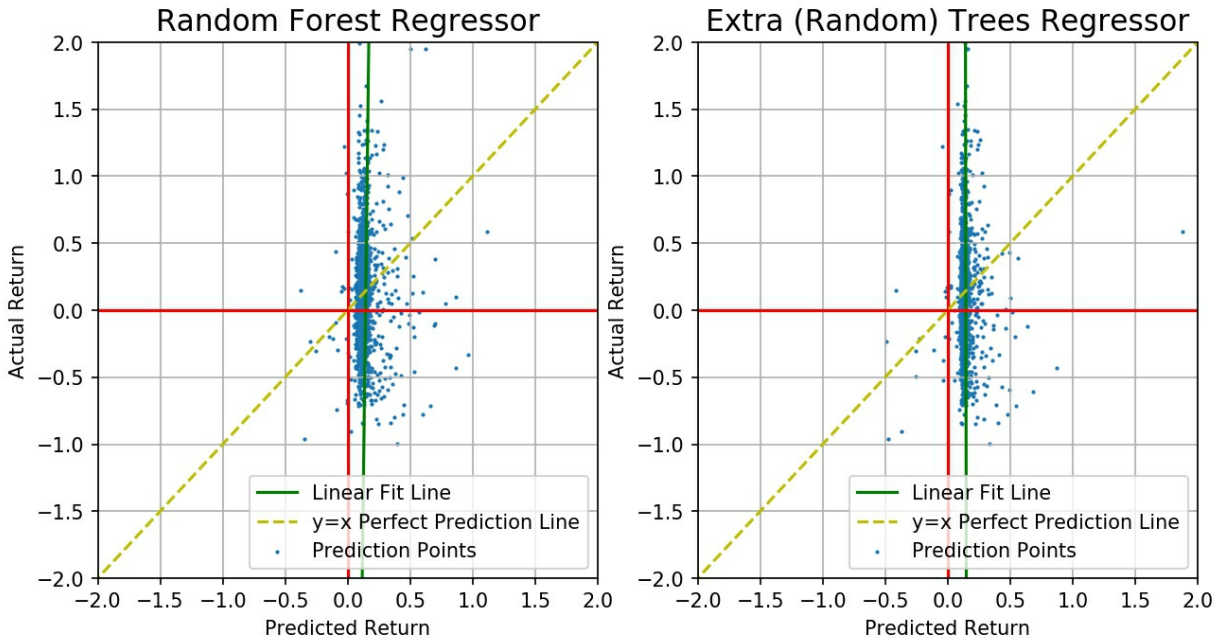
The error is also among the lowest we've had so far. Perhaps our prediction ability can be enhanced further by using extra-random trees.

In[3]:

```
from sklearn.ensemble import ExtraTreesRegressor
ETregressor = ExtraTreesRegressor(random_state=42, max_depth=10)
ETregressor.fit(X_train, y_train)
y_pred_ET = ETregressor.predict(X_test)

print('train mse: ',
      mean_squared_error(y_train,
                         ETregressor.predict(X_train)))
print('test mse: ',
      mean_squared_error(y_test,
                         y_pred_ET))
```

Out[3]:
train mse: 0.21982249603645046
test mse: 0.2933175190727553

It doesn't seem like there is too much of a difference in errors. Taking a look at the scatter plots, the aim with extra trees for lower variance and higher bias can be clearly seen as the predicted returns are much more concentrated than for the Random Forest regressor.



Let’s observe the prediction abilities of both kinds of Random Forests.

In[4]:

```
observePredictionAbility(rfregressor, X, y)
```

Out[4]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 130.71
        Mean Actual Performance of Top 10 Return Portfolios: 40.34
Mean Predicted Performance of Bottom 10 Return Portfolios: -18.3
        Mean Actual Performance of Bottom 10 Return Portfolios: -10.46
--------------
```

In[5]:

```
observePredictionAbility(ETregressor, X, y)
```

Out[5]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 127.39
        Mean Actual Performance of Top 10 Return Portfolios: -0.6
Mean Predicted Performance of Bottom 10 Return Portfolios: -23.53
        Mean Actual Performance of Bottom 10 Return Portfolios: -8.55
--------------
```

Oh dear, our predictive ability is a lot lower with the extra random trees. It’s not all that surprising considering how the extratrees regressor differs from the standard Random Forest. The extra trees method goes a step further than Random Forests by not only making splits on a random subset of features,

but the thresholds of the splits are random too. Recall that for our Decision Tree important features like Price/Sales were quite near the trunk, making them of great importance.

It is more likely that the Random Forest will have these features present earlier on making important splits in the right places, but with extra random trees a feature like Price/Sales, for example, is likely to be split at a sub-optimal value.

We'll keep the standard Random Forest for the next stage and discard the extra random trees regressor.

Random Forest Feature Importance

Analogous to observing which features are important in a Decision Tree, we can see feature importance in the Random Forest with the *feature_importances* attribute.

In[6]:

```
#Can see the importance of each feature in Random Forest.
ks, scores=[],[]
for k, score in zip(x.keys(), rfregressor.feature_importances_):
    print(k, round(score,3))
    ks.append(k)
    scores.append(score)
```

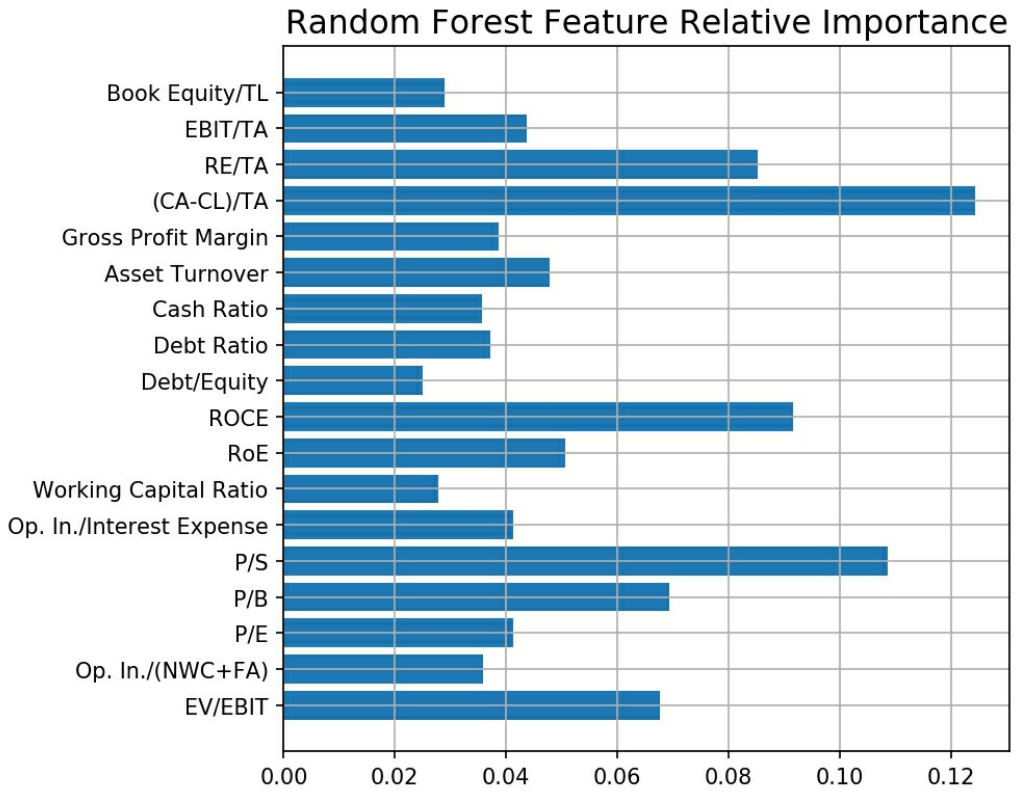
Out[6]:

EV/EBIT 0.068
Op. In./(NWC+FA) 0.036
P/E 0.041
P/B 0.069
P/S 0.109
Op. In./Interest Expense 0.041
Working Capital Ratio 0.028
RoE 0.051
ROCE 0.092
Debt/Equity 0.025
Debt Ratio 0.037
Cash Ratio 0.036
Asset Turnover 0.048
Gross Profit Margin 0.039
(CA-CL)/TA 0.124
RE/TA 0.085
EBIT/TA 0.044
Book Equity/TL 0.029

In[7]:

```
plt.figure(figsize=(6,6))
plt.barh(ks,scores)
plt.grid()
```

Out[7]:



This tells us some great information about features we should be looking for in stocks, whether you are analysing companies manually to invest in or making some systematic investing strategy like we are here. The relative importance of these features appears to match our Decision Tree quite well if you look back at the tree diagram.

What's more, these ratios make intuitive sense and match a lot of systematic stock selection literature. Near the top we have Price/Sales ratio, supposedly the most important ratio according to *What Works on Wall Street*, return on capital, which is known to loom large in Warren Buffett's investing style from Berkshire Hathaway Letters to Shareholders and books such as *Buffetology* (A bit like return on equity, but ROCE helps estimate how much further investment would return). Also the Price/Book ratio appears, somehow at the top. Quite fitting since the importance of this ratio was highlighted by Ben Graham to the investing public in *The Intelligent Investor*.

This graph will differ depending on what data you used to grow your forest, typically measures like Debt/Equity and Working Capital Ratio will be of lower importance than Price/Sales or Price/Book.

Gradient Boosted Decision Tree

The last regressor we will use is a Gradient Boosted Decision Tree, using Scikit-Learns GradientBoostingRegressor class, we'll use the same tree depth and the default number of estimators for a quick test:

In[8]:

```
from sklearn.ensemble import GradientBoostingRegressor

pl_GradBregressor = Pipeline([
    ('GradBoostRegressor', GradientBoostingRegressor(n_estimators=100,\
                                                     learning_rate=0.1,\
                                                     max_depth=10,\
                                                     random_state=42,\
                                                     loss='ls')) ])
[…]
observePredictionAbility(pl_GradBregressor, X, y)
```

Out[8]:

```
--------------
Mean Predicted Performance of Top 10 Return Portfolios: 220.23
        Mean Actual Performance of Top 10 Return Portfolios: 16.28
Mean Predicted Performance of Bottom 10 Return Portfolios: -39.73
        Mean Actual Performance of Bottom 10 Return Portfolios: 1.45
--------------
```

That's an OK return out of the gradient boosted regressor. There are many more ensemble methods you can try on this data, the number of process permutations you could have this data go through is limitless.

We won't be exploring any further models as we now have a large group of them that show promise. At this point, it should be easy to see how to train other kinds of Scikit-Learn models yourself on the data, and many ways of analysing the results have been demonstrated. Further venturing in this area might be fun if you are the tinkering type., for example, a simple way to combine your regressors for a single Frankenstein monster predictor can be made easily with Scikit-Learns voting regressor on the following link: (https://scikit-learn.org/stable/modules/ensemble.html#voting-regressor).

Checking Our Regression Results So Far

We now have a group of regressors which work to some extent when trained with our data. Let's take see the results of all these regressors we have used so far, and the best performance we achieved them. Remember that the mean of the entire *y* vector is a return of 13% and that we have survivorship bias in this data. Table results are in percentage:

Summary Results of Top10/Bottom10 Selected Stock Portfolios. Average of 10 Runs. (Feb 2021 Data)						
Model	Top 10 predicted	Top10 actual	Top 10 Actual Standard Deviation	Bottom 10 predicted	Bottom 10 actual	Bottom 10 actual Standard Deviation
Linear Regression	43.12	37.05	83.96	-∞	13.57	17.8
Lasso Regression	25.81	35.68	80.8	-∞	14.95	14.65
K-NN	55.64	32.66	60.68	-5.31	-2.53	15.13
SVM	274.76	38.34	60.38	-168.03	12.63	19.99
Decision Tree	486.42	24.97	33.35	-80.73	7.21	28.34
RandomForest	130.71	40.34	59.46	-18.3	-10.46	24.42
ExtraTrees	127.39	-0.6	25.93	-23.53	-8.55	24.32
Gradient Boosted Decision Tree	220.23	16.28	21.4	-39.73	1.45	20.45

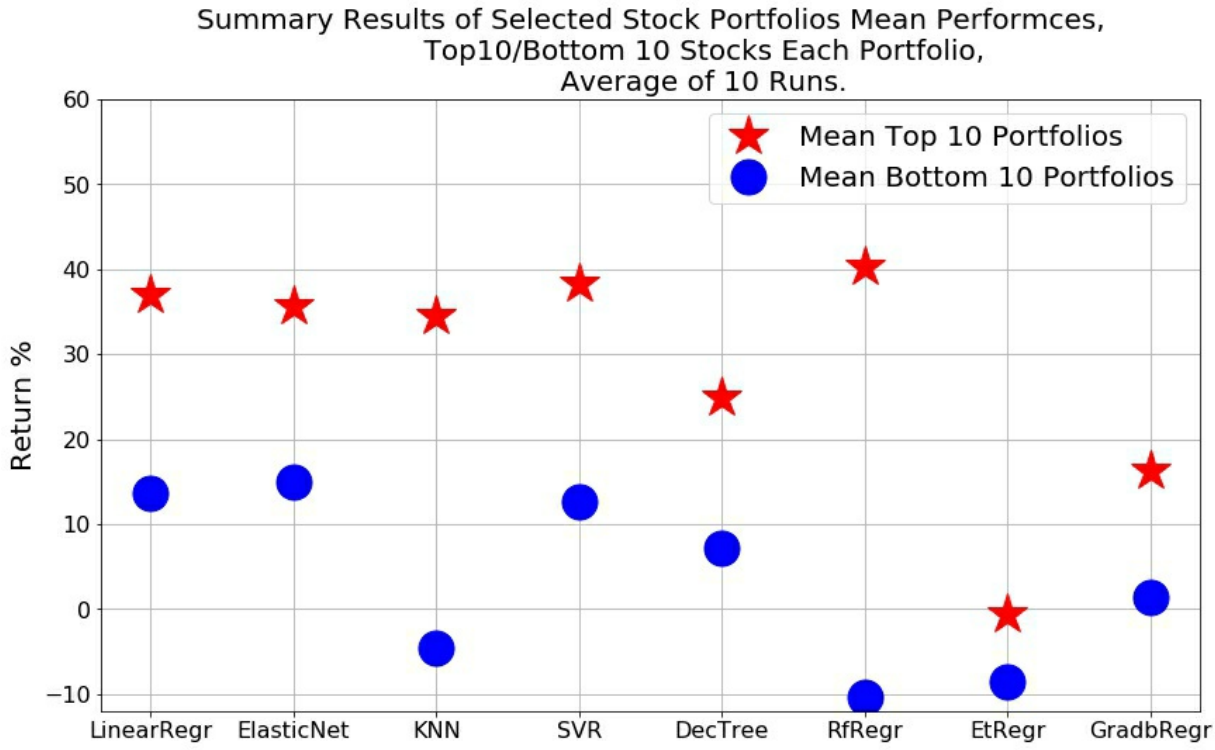
We are most interested in the *Top 10 actual* column, as this tells us how high the returns are for the top 10 predicted return stock picks on average.

Most of our predictors overpromise and underdeliver, though at least they all make predictions in the correct direction. This table differs a little from the 2020 edition of this book, however the general themes still hold. As with last year, the standard deviation of the results is quite large, and the Random Forest regressor appears to do quite well. It is easier to visualise our results data in a plot:

In[9]:

```
df_best, df_worst = pd.DataFrame(), pd.DataFrame()

# Run all our regressors (might take awhile.)
# We do this so we can plot easily with the pandas library.
# Make sure using the final hyperparameters.
df_best['LinearRegr'], df_worst['LinearRegr']=observePredictionAbility(
    pl_linear, X, y, returnSomething=True, verbose=False)

df_best['ElasticNet'], df_worst['ElasticNet']=observePredictionAbility(
```

```
    pl_ElasticNet, X, y, returnSomething=True, verbose=False)

df_best['KNN'], df_worst['KNN']=observePredictionAbility(
    pl_KNeighbors, X, y, returnSomething=True, verbose=False)

df_best['SVR'], df_worst['SVR']=observePredictionAbility(
    pl_svm, X, y, returnSomething=True, verbose=False)

df_best['DecTree'], df_worst['DecTree']=observePredictionAbility(
    pl_decTree, X, y, returnSomething=True, verbose=False)

df_best['RfRegr'], df_worst['RfRegr']=observePredictionAbility(
    rfregressor, X, y, returnSomething=True, verbose=False)

df_best['EtRegr'], df_worst['EtRegr']=observePredictionAbility(
    ETregressor, X, y, returnSomething=True, verbose=False)

df_best['GradbRegr'], df_worst['GradbRegr']=observePredictionAbility(
    pl_GradBregressor, X, y, returnSomething=True, verbose=False)


# Plot results out
# Warning: the results are quite variable,
we are only looking at the means here.
# Comment out the code to see with standard deviations.

plt.figure(figsize=(14,8))
df_best.mean().plot(linewidth=0,
                    marker='*',
                    markersize=30,
                    markerfacecolor='r',
                    markeredgecolor='r',
                    fontsize=16)

df_worst.mean().plot(linewidth=0,
                     marker='o',
                     markersize=25,
                     markerfacecolor='b',
                     markeredgecolor='b')

plt.legend(['Mean Top 10 Portfolios',
            'Mean Bottom 10 Portfolios'],
           prop={'size': 20})
plt.ylim([-12, 60])
plt.title('Results of Selected Stock Portfolios Mean Performces,\n\
            Top10/Bottom 10 Stocks Each Portfolio,\n\
            Average of 10 Runs.', fontsize=20)
plt.ylabel('Return %', fontsize=20)
plt.grid()
```



What we want to see is a good separation from the top 10 predicted return stock portfolios (stars), and the bottom 10 predicted return stock portfolios (circles). What isn't plotted here, but is worth plotting is some information about the distribution of the results, like with our learning curves earlier. Some code is present in the notebook that can be commented out to see the standard deviations plotted out on the same graph, though the graph gets a little busy with this information on.

As with last year the Extra Random Decision Tree model performed the worst, we will discard this model for the next stage. The rest of the models show some promise.

The issue of survivorship bias will be addressed in the next section, though even without adjusting for that, these predictors seem to perform well enough that a portfolio picked with them can afford for a company or two to go to zero over time (Unless it is picking companies that were likely to go bankrupt but didn't).

An effect in our favour is that the stocks being chosen are less likely to go bust as the criteria for top prediction performance stock selection chooses stocks that are generally better than average, as we have seen under the hood

with our Decision Trees and Random Forests. A similar effect happens with *The Little Book that Beats The Market*, though that method only uses two features and the top stocks there generally look like they are going to go belly-up.

You can experiment with this stock performance prediction code, now that all the models are trained. If you input a hypothetical company in as an X vector, what a value investor might intuitively suspect is generally true. Give the predictor an imaginary company with a return on equity of 1000% and a price/earnings ratio of 1 and you're going to get high predicted returns.

We know that the average return is about 12%, let's see what ratios the average company in our dataset would have. These ratios should stay relatively constant from year to year.

In[10]:

```
X.mean()
```

Out[10]:

```
EV/EBIT                  13.489216
Op. In./(NWC+FA)          0.196079
P/E                      18.031356
P/B                       4.210863
P/S                      13.662130
Op. In./Interest Expense -23.351751
Working Capital Ratio     2.686535
RoE                       0.050905
ROCE                      0.056053
Debt/Equity               2.378435
Debt Ratio                2.697024
Cash Ratio                1.373297
Asset Turnover           34.535357
Gross Profit Margin       0.468239
(CA-CL)/TA                0.217532
RE/TA                    -0.272173
EBIT/TA                   0.034760
Book Equity/TL            1.645187
```

If we put this through one of our predictors, we should get an answer around the average return. The Random Forest was probably the best predictor so let's see what return that predicts:

In[11]:

```
avg_company = pd.DataFrame(x.mean().values.reshape(1,-1), \
                columns=x.keys()) # Put in a DataFrame for prediction
rfregressor.predict(avg_company)
```

Out[11]:
array([0.15609572])

It’s not perfect, but it is close. If we put in place of a few of those ratios some numbers that investors might think are more favourable, like a lower P/E ratio, etc., we often get a higher return as the predicted output from out model.

Let’s, try out some made-up company ratios and see if the model can separate a dog from a golden goose (or at least, what *we* think is a dog or a golden goose).

In[12]:

```
# Try making a bunch of numbers reflect
# higher earnings relative to price
good_company = pd.DataFrame(X.mean().values.reshape(1,-1),
                            columns=X.keys())
good_company['EV/EBIT']=5
good_company['Op. In./(NWC+FA)']=0.4
good_company['P/E']=3
good_company['P/B']=4
good_company['P/S']=13
good_company['EBIT/TA']=1

rfregressor.predict(good_company)
```

Out[12]:
array([0.16338807])

In[13]:

```
# Let's try the same in the opposite direction
bad_company = pd.DataFrame(X.mean().values.reshape(1,-1),
                           columns=X.keys())
bad_company['EV/EBIT']=900
bad_company['Op. In./(NWC+FA)']=-0.5
bad_company['P/E']=30
bad_company['P/B']=300
bad_company['P/S']=400
bad_company['EBIT/TA']=0.04

rfregressor.predict(bad_company)
```

Out[13]:
array([0.09677982])

This may not reflect a real company: some of the ratios don’t make sense in combination when we change just a few ratios like this, but at least we know our models' predictions are in roughly the right direction.

Try out some other models and company ratios yourself and see what predicted stock performance your machines return to you, not all the models do as well as the Random Forest. Furthermore, sometimes counter-intuitive results come out, for example a company with lower relative earnings might have a higher predicted return, possibly because there are many growth companies that the predictor has learnt from.

Chapter 5 - AI Backtesting

Now that we have our stock performance prediction code, we need to test it out and see if we can make AI Investors with them. We want to compare the returns from choosing the annual predicted highest return stocks to our S&P500 benchmark, as well as the kind of volatility that might be expected.

For simplicity we will make our AI investors take the top 7 predicted stock returns from each model, near the financial reporting date, and hold those stocks for a year, repeating the cycle every year. The investment weighting will be equal across the 7 stocks at the start. This number of stocks was chosen as it is a reasonable balance between projected returns and diversification, and for some investors, high numbers of stocks will begin to incur high fees. Feel free to change this number and in your own backtesting.

The data from *SimFin.com* goes back to 2009, so each backtest will encompass 11 years of stock picking. Any stock data which our models are trained on should not be in the universe of stocks they can choose from in the backtest. This presents us with a data splitting issue, too little backtesting data and our stock picker will have a small universe of stocks to choose from, limiting returns from what should be possible, on the other hand, too little training data and our Machine Learning algorithms will perform badly. We will test a few splits to understand the trends and so know how well the stock pickers are likely to perform in reality.

As we want to know the volatility as well as the return, we need to store the stock price for each stock in the portfolio every day or week of the year that stock is held. Combining these stock price levels will give us the portfolio value over time, from which we can calculate volatility.

To be sure of our investing AIs abilities we will need to iteratively run backtests with new train/test sampling each time. The result will be a distribution of backtest returns, as well as a distribution of backtesting volatilities. If these distributions are satisfactory, with the majority of backtests giving above-market returns at acceptable volatility, we will have achieved our aim. We will do this in the next section, for the moment we will focus on getting the backtest program made for a given train/test split.

The issue of survivability bias in our data can be eliminated by purchasing better data (we only used freely available or cheap internet data for this guide). Alternatively, the inclusion of Altman Z scores in our algorithms can mitigate the issue, which we will cover in this chapter.

A priority for the code is to make things readable and understandable, this often means opting for the slower code design choice. We'll do this because the trade-off is worthwhile: Waiting overnight for a large batch of backtests to be run is palatable, yet the increase in the number of people who can follow these instructions, and understand what they are writing, is large. Where an operation gets complex, we will break down the steps in lower-level functions. There will be a cascade of functions within functions, so we can focus on each function individually.

The backtest program will work as follows:

First Split the data into train/test sets and train the model on the training set.

For a given year, in March, have the regressors predict stock returns and pick the top 7 highest return predictions from the test set to create an equally weighted portfolio.

Record the weekly portfolio value change.

Repeat for the 10 years we have data for. The value on the last day gives us the portfolio performance and the volatility of the portfolio is computed from the portfolio market value over time.

To be clear, what we want out of our code should be a function that gives us a backtest portfolio value over time, which hopefully will look something like this when plotted out:

To make things easy going forward we will compartmentalise the code into functions that we can easily focus on individually, to make things easier to mentally digest. Feel free to approach the code explanation in either a top-down perspective by reading in order, starting with the highest-level functions, or a bottom-up perspective by going a few pages ahead and reading the sections in reverse. Given that some code is present that you would only know is needed later, it is worth approaching the code from both perspectives. Alternatively, you can just copy the code and figure things out hands-on.

Using an Example Regressor

We will train an example Linear Regressor to run the backtester on:

In[1]:

```
X, y = loadXandyAgain()

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.5,
                                                    random_state=42)

# Save CSVs
# For the backtester to get correct test data
# in case want to see the data.
X_train.to_csv("Annual_Stock_Price_Fundamentals_Ratios_train.csv")
X_test.to_csv("Annual_Stock_Price_Fundamentals_Ratios_test.csv")
y_train.to_csv("Annual_Stock_Price_Performance_Percentage_train.csv")
```

```
y_test.to_csv("Annual_Stock_Price_Performance_Percentage_test.csv")

# Linear
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PowerTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
import pickle # To save the fitted model

pl_linear = Pipeline([('Power Transformer', PowerTransformer()),
    ('linear', LinearRegression())]).fit(X_train, y_train)

y_pred = pl_linear.predict(X_test)

print('train mse: ',
      mean_squared_error(y_train, pl_linear.predict(X_train)))
print('test mse: ',
      mean_squared_error(y_test, y_pred))

pickle.dump(pl_linear, open("pl_linear.p", "wb" ))
```

Out[1]:
train mse: 0.48490895851905697
test mse: 0.27626250426988774

Building the First Bits

Let’s load in the data to get the ball rolling:

In[2]:

```
# X AND Y
# The backtester needs dates from the old y vector
# to plot the stock prices.

# Financial ratios
X = pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios.csv",
                index_col=0)

# Annual stock performances, with date data.
y_withData=pd.read_csv("Annual_Stock_Price_Performance_Filtered.csv",
                       index_col=0)

# Convert to date
y_withData["Date"] = pd.to_datetime(y_withData["Date"])
y_withData["Date2"] = pd.to_datetime(y_withData["Date2"])

# X AND Y (splitting for train/test done previously for trained model)
X_train=pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios_train.csv",
                    index_col=0)
X_test=pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios_test.csv",
                   index_col=0)
```

```
y_train=pd.read_csv("Annual_Stock_Price_Performance_Percentage_train.csv                ",index_col
y_test=pd.read_csv("Annual_Stock_Price_Performance_Percentage_test.csv",
                   index_col=0)

# Get y_withData to correspond to y_test
y_withData_Test=pd.DataFrame()
y_withData_Test=y_withData.loc[y_test.index, :]

# Convert string to datetime
y_withData_Test["Date"] = pd.to_datetime(y_withData_Test["Date"])
y_withData_Test["Date2"] = pd.to_datetime(y_withData_Test["Date2"])
```

Here the DataFrame *y_withData_Test* corresponds to the *X_test* DataFrame. These will be the data used to do the backtesting with (predict returns with *X_test*, select the predicted best stocks, then get the portfolio actual performance with data from *y_withData_Test*). Throughout this chapter you can execute a cell with just the variable typed out to see the data:

```
In [8]: y_withData_Test.head() # y data corresponding to y targets
```

Out[8]:

	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
4235	DISCA	70.55	2013-02-19	1.293252e+08	DISCA	80.33	2014-02-19	2.181682e+08
12287	SEE	20.49	2010-03-01	2.415361e+07	SEE	27.49	2011-03-01	2.780614e+07
8308	LGIH	63.70	2018-02-27	8.367779e+07	LGIH	58.89	2019-02-27	1.676310e+07
13733	TOWR	28.40	2017-02-28	3.835846e+06	TOWR	26.90	2018-02-28	2.583906e+06
11601	RECN	16.43	2019-07-19	6.332713e+06	RECN	11.53	2020-07-20	8.064197e+05

Since we want to know our portfolio value over time, we will need the daily stock price data for all stocks from *SimFin.com*, let's keep the code to load this data inside a function. This file is rather big, it is six million rows of data, so it might take a second.

In[3]:

```
# Daily stock price time series for ALL stocks. 5M rows.
# Some days missing.
def getYRawData(directory='C:/Users/G50/Stock_Data/SimFin2021/'):
    '''
    Can set directory to look for file in.
    Get daily stock price time series for ALL stocks.
    5M rows. Some days missing.
    Returns DataFrame
    '''
    daily_stock_prices=pd.read_csv(directory+'us-shareprices-daily.csv',
                                   delimiter=';')
    daily_stock_prices["Date"]=pd.to_datetime(daily_stock_prices["Date"])
```

```
    print('Reading historical time series stock data, matrix size is: ',
        daily_stock_prices.shape)

    return daily_stock_prices
```

Out[3]:
Reading historical time series stock data, matrix size is: (6482905, 10)

What we want is the value of our portfolio over the entire backtest. We will create a main function to return this to us as a DataFrame. Let's call this function *getPortTimeSeries* so we know this is the function that "Gets the portfolio time series".

getPortTimeSeries

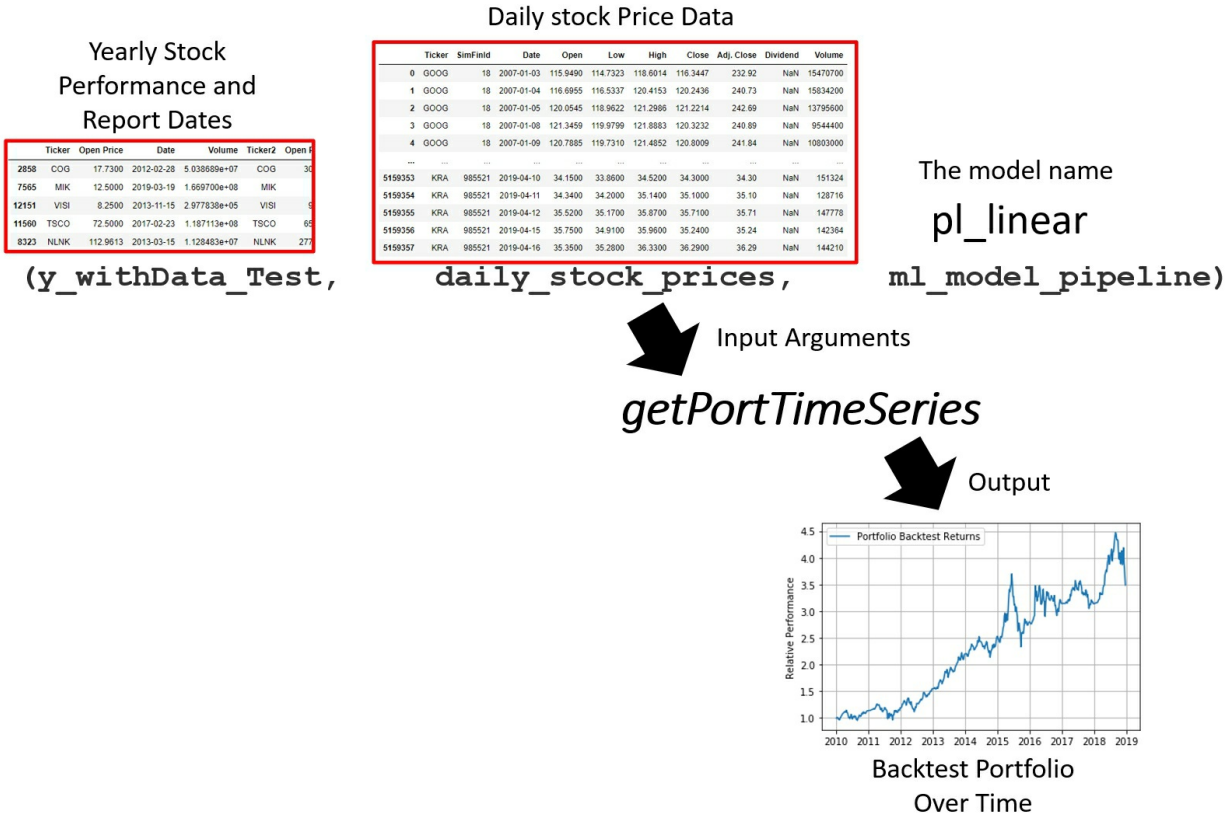
This *getPortTimeSeries* function will need to know what model we want to use, so we will pass that as an argument. Let's also pass the daily stock prices and the *y* data containing buy and sell dates, we'll need those to know how stocks will move.

This function will be called to run our backtest program after all our code is written:

In[4]:

```
trained_model_pipeline = pickle.load(open("pl_linear.p", "rb" ))

backTest = getPortTimeSeries(y_withData_Test, X_test,
                             daily_stock_prices_data,
                             trained_model_pipeline)
```

We want the output to be a DataFrame (here called *backTest*) which will have the portfolio relative value over time in a Pandas series (starting at 1.0 on day 1). We will pass into the function the ingredients we know we need to get us our backtest, namely the *X_test* DataFrame containing all our stock ratios, the *y_withData_Test* DataFrame containing beginning and end stock prices, as well as the publication dates of the 10-K reports, the daily stock prices and our trained stock prediction model. There is a verbose boolean setting to switch print-out information on or off (personal preference).



We are essentially performing the same operation every year of the backtest. Let's make a function called *getPortTimeSeriesForYear* and do a loop through that within our *getPortTimeSeries* function. We'll loop through a date range rather than a python list as we want to do some arithmetic between dates later. The date range we are looping through is *dr*, and you can see *getPortTimeSeriesForYear* being called in the loop with mostly the same inputs as *getPortTimeSeries*, except with the current year being passed too.

In[5]:

```
def getPortTimeSeries(y_withData, X, daily_stock_prices,
                      ml_model_pipeline, verbose=True):
    '''
    Returns DataFrames of selected stocks/portfolio performance since
    2009. Needs X and y(with data), the daily_stock_prices DataFrame,
    the model pipeline we want to test.
    X is standard X for model input.
    y_withData is the stock price before/after df with date information.
    Input X and y must be data that the model was not trained on.
    '''
    # set date range to make stock picks over
    dr=pd.date_range(start='2009-01-01', periods=11, freq='Y')
```

```
    # For each date in the date_range, make stock selections
    # and plot the return results of those stock selections
    port_perf_all_years = pd.DataFrame()
    perfRef=1 # performance starts at 1.
    for curr_date in dr:

        [comp, this_year_perf, ticker_list] = \
        getPortTimeSeriesForYear(curr_date, y_withData, X,\
                                 daily_stock_prices, ml_model_pipeline)

        if verbose: # If you want text output
            print("Backtest performance for year starting ",\
                  curr_date, " is:",\
                  round((this_year_perf.iloc[-1]-1)*100,2), "%")
            print("With stocks:", ticker_list)
            for tick in ticker_list:
                print(tick, "Performance was:",\
                      round((comp[tick].iloc[-1]-1)*100,2), "%" )
            print("-----------------------------------------------")

        # Stitch performance for every year together
        this_year_perf = this_year_perf * perfRef

        port_perf_all_years = pd.concat([port_perf_all_years,\
                                         this_year_perf])

        perfRef = this_year_perf.iloc[-1]

    # Return portfolio performance for all years
    port_perf_all_years.columns = ["Indexed Performance"]
    return port_perf_all_years
```

Let's specify what we want *getPortTimeSeriesForYear* to take in, and return. We want *getPortTimeSeriesForYear* to return the daily portfolio value for that year, as well as the list of stocks our algorithm chose, and the returns of those individual stocks. We'll have the function give us this information as a list. We will print the stock selection and performance for each year in this loop.

Within our loop of backtest years, once we have the DataFrame for relative stock performance for a given year, we add that data to previous years by scaling the beginning performance index to be the last value of the stock index. The final DataFrame is returned *port_perf_all_years*.

This loop is a good place to have text output, we would like to know what stocks the AI selected for that year, and what their returns were. Our verbose boolean is used in case we want to turn off the backtest printout.

Looping Through Each Year of a Backtest

getPortTimeSeriesForYear

For *getPortTimeSeriesForYear* input we wanted to pass the following data:

(date_starting, y_withData, daily_stock_prices, model_file)

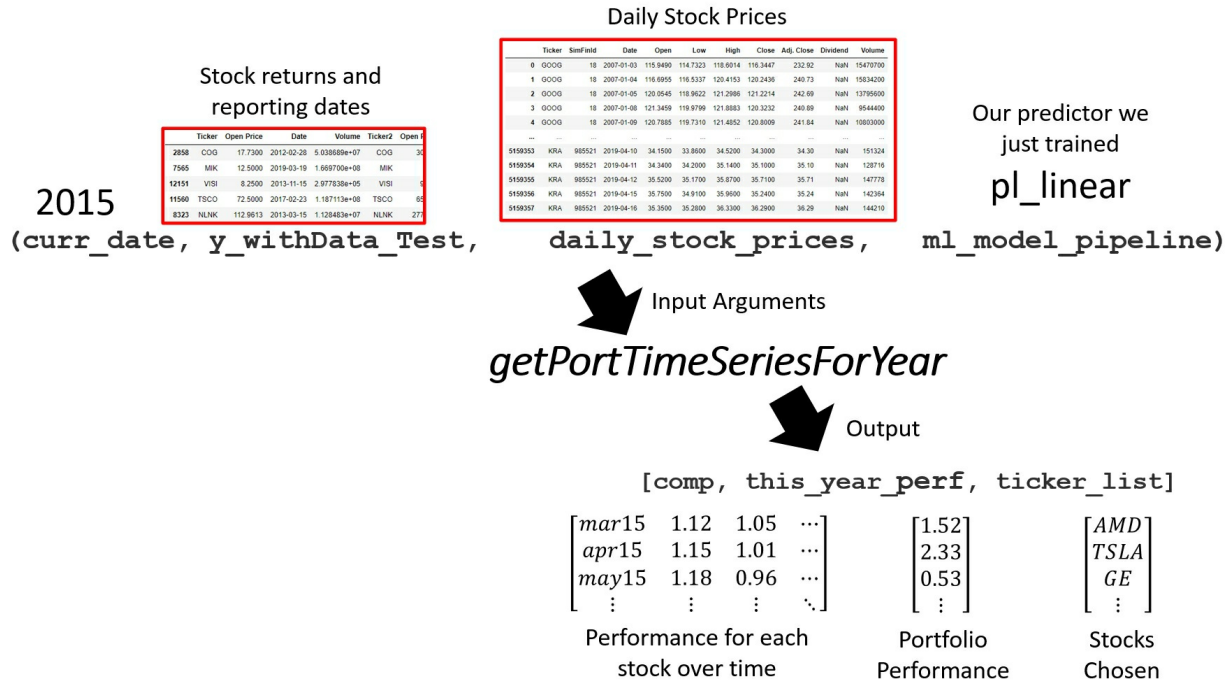
These are for the current year in the loop to tell it which year to start from, the stock return *y* data (which contains the financial report date, we can't select stocks before the data is published unless you have a time machine), Our daily stock price data and the name of our regressor model.

The desired outputs for *getPortTimeSeriesForYear* as:

[comp, this_year_perf, ticker_list]

These outputs are the composite performance over the course of the year (a DataFrame of the stocks and their returns), the overall portfolio performance over the year, and the list of stocks the model chose.

We'll output *comp* and *this_year_perf* as DataFrames with the index being the dates and the series being the relative price (indexed to 1 at the start of the year). This diagram should make things clearer:



The first thing we will do in this function is cut down our *y* data to only include the year we are interested in. We do this by looking at the report date of the financial data (from *y*) and checking that it is within a few months of the date we supplied. We know that annual reports are available to us around February, we will make a window of acceptable financial report publication dates to select stocks from (two months either side of the new year).

This will give us a boolean, called *thisYearMask* which can also be used on the *X* data (remember the rows correspond), so we can get *X* data for a particular year:

In[6]:

```
# Get performance only for time frame we care about,
# mask original datausing the start date
    thisYearMask = y_withData["Date"].between(\
            pd.to_datetime(date_starting) - pd.Timedelta(days=60),\
            pd.to_datetime(date_starting) + pd.Timedelta(days=60) )
```

The second thing is to use the model pipeline to make stock return predictions for the current year.

```
    # Get return prediction from model
    y_pred = ml_model_pipeline.predict(X[thisYearMask])
```

The third thing to do is to take this list of stock predictions and cut out the 7 best to become our portfolio for that year. We do this by making the prediction a DataFrame to select the 7 best with a boolean mask.

```
y_pred = pd.DataFrame(y_pred)

    #bool list of top stocks
    bl_bestStocks =(y_pred[0]>y_pred.nlargest(8,0).tail(1)[0].values[0])
    ticker_list=y[thisYearMask].reset_index(drop=True)\
                [bl_bestStocks]["Ticker"].values
```

the variable *y* here is a DataFrame of the ticker and performance only, calculated with this line:

```
y = getYPerf(y_withData)
```

Which we will write this function for:

In[7]:

```
def getYPerf(y_):
    y=pd.DataFrame()
```

```
    y["Ticker"] = y_["Ticker"]
    y["Perf"]=(y_["Open Price2"]-y_["Open Price"])/y_["Open Price"]
    y[y["Perf"].isnull()]=0
    return y
```

Great. Now we know exactly what stocks we will have in our portfolio for a given backtest year, which is one of our three outputs for *getPortTimeSeriesForYear*. We now need to get the other two outputs, the value of each of these stocks over the course of the year, and an aggregation of them to get the portfolio performance overall for the year.

We will go deeper and make another function to give us the stock values over time. Our data sometimes skips days, so we will settle for weekly value updates. If a day is missing, we can get the closest to that day. Let's call this function *getStockTimeSeries*.

To *getStockTimeSeries*, we will pass an index of dates we want the stock value for, the *y* values (holding stock return data as well as the financial reporting date), the boolean mask filtering for stocks in the current year, and the daily stock price data.

We want *getStockTimeSeries* to return a DataFrame which has the index of our time steps of 7 days, and the series data will be columns of the stock tickers, with those stock prices filling up the table.

Our *getPortTimeSeriesForYear* function now looks like this:

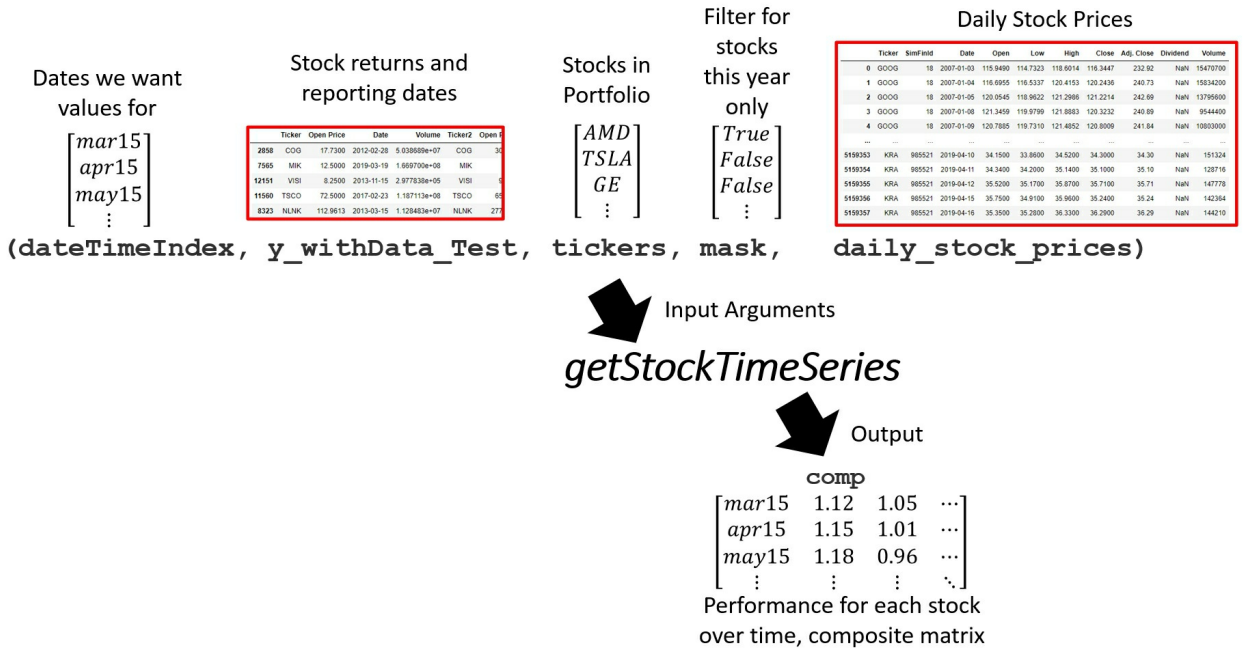
In[8]:

```
def getPortTimeSeriesForYear(date_starting, y_withData, X,
                             daily_stock_prices, ml_model_pipeline):
    '''
    Function runs a backtest.
    Returns DataFrames of selected stocks/portfolio performance,
    for 1 year.
    y_withData is annual stock performances (all backtest years)
    date_starting e.g. '2010-01-01'
    daily_stock_prices is daily(mostly) stock price time series for
    all stocks
    '''

    # get y dataframe as ticker and ticker performance only
    y = getYPerf(y_withData)

    # Get performance only for time frame we care about,
    # mask original data using the start date
    thisYearMask = y_withData["Date"].between(\
```

```
        pd.to_datetime(date_starting) - pd.Timedelta(days=60),\
        pd.to_datetime(date_starting) + pd.Timedelta(days=60) )

    # Get return prediction from model
    y_pred = ml_model_pipeline.predict(X[thisYearMask])

    # Make it a DataFrame to select the top picks
    y_pred = pd.DataFrame(y_pred)

    # Bool list of top stocks
    bl_bestStocks=(y_pred[0]>y_pred.nlargest(8,0).tail(1)[0].values[0])

    # DatetimeIndex
    dateTimeIndex = pd.date_range(\
                        start=date_starting, periods=52, freq='W')

    # 7 greatest performance stocks of y_pred
    ticker_list = y[thisYearMask].reset_index(drop=True)\
                    [bl_bestStocks]["Ticker"].values

    # After we know our stock picks, we get the stock performance
    # Get DataFrame index of:
    # time stamp, series of stock prices, keys=tickers
    stockRet = getStockTimeSeries(dateTimeIndex, y_withData,
                                  ticker_list, thisYearMask,
                                  daily_stock_prices)

    # Get DataFrame of relative stock prices from
    # 1st day(or close) and whole portfolio
    stockRetRel = getPortfolioRelativeTimeSeries(stockRet)
    return [stockRetRel, stockRetRel["Portfolio"], ticker_list]
```

getStockTimeSeries

What *getStockTimeSeries* does is get our stock time series as a DataFrame:



Essentially this is a function that breaks down the *daily_stock_prices* matrix to only get the stocks that we want at the dates that we specify, which are weekly.

As the same operations will be done for each stock, we will make a function to get our data for a single stock and iterate through all the stocks in a for loop, appending them to a DataFrame for the output of the *getStockTimeSeries* function.

In[9]:

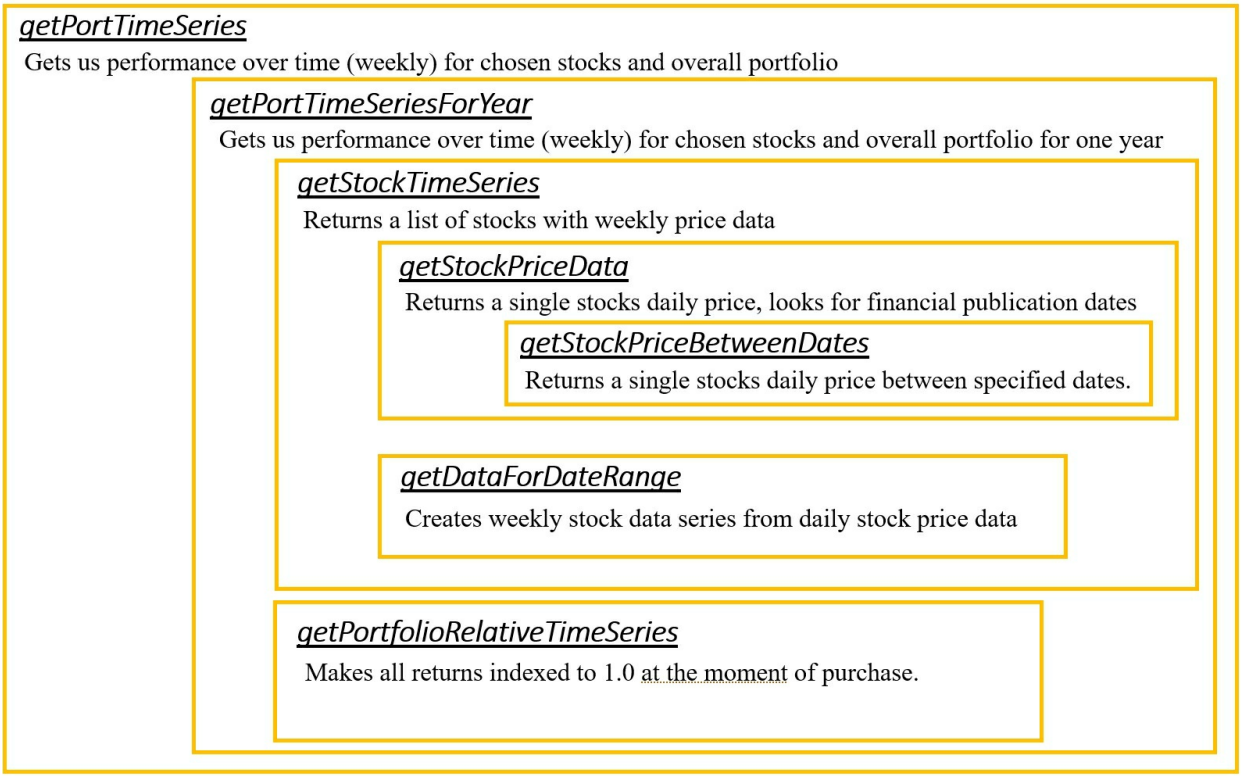
```
def getStockTimeSeries(dateTimeIndex, y_withData,
                       tickers, mask, daily_stock_prices):
    '''
    Get the stock price as a time series DataFrame
    for a list of tickers.
    A mask is used to only consider stocks for a certain period.
    dateTimeIndex is typically a weekly index,
    so we know what days to fetch the price for.
    '''
    stockRet = pd.DataFrame(index=dateTimeIndex)
    dTI_new = dateTimeIndex.strftime('%Y-%m-%d') # Change Date Format
    rows = pd.DataFrame()
    for tick in tickers:
        # Here "rows" is stock price time series data
        # for individual stock
        rows = getStockPriceData(tick,
                                 y_withData,
                                 mask,
```

```
                                daily_stock_prices,
                                rows)
        rows.index = pd.DatetimeIndex(rows["Date"])
        WeeklyStockDataRows = getDataForDateRange(dTI_new,
                                                  rows)
        # Here can use Open, Close, Adj. Close, etc. price
        stockRet[tick] = WeeklyStockDataRows["Close"].values
    return stockRet
```

In our loop over the stock tickers, *getStockPriceData* returns the rows of the *daily_stock_prices* DataFrame which only contains the stock we want, at the year we want.

The other function in our loop, *getDataForDateRange* gets our stock price data to be in a weekly format, using the closest day if the data for a day we want is missing.

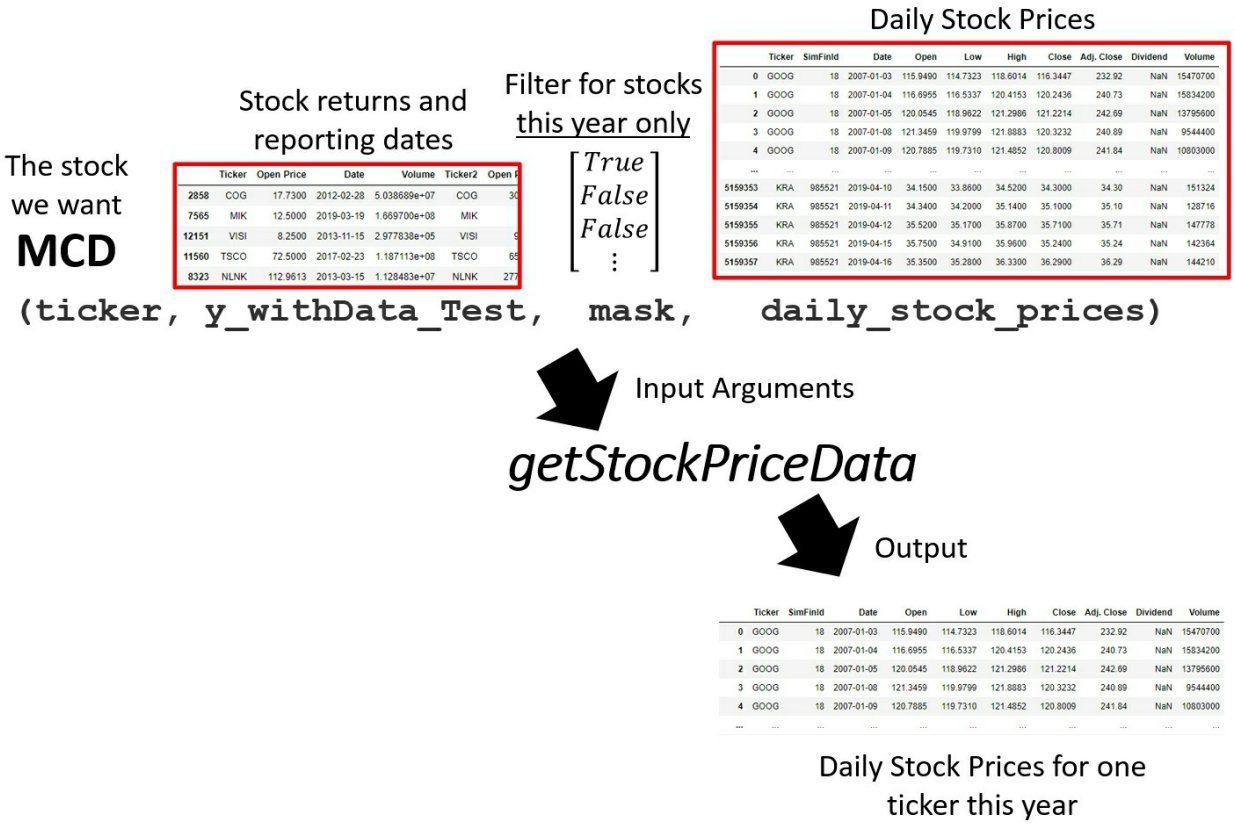
At this point you can see how most of the functions fit together, here's a map of the Russian doll-like series of function making up the backtesting program to help get your head around the structure:



Backtest Data for an Individual Stock

getStockPriceData

getStockPriceData performs a filter on our *daily_stock_prices* data to get the stock price we want on the days that are relevant to us:



This is done by first getting the financial data report date from the *y* data from the report date at the beginning of the year (around when we might buy a stock) and the report date 1 year later. These are *Date1* and *Date2* columns in the *y* DataFrame respectively.

In[10]:

```
def getStockPriceData(ticker, y_, mask, daily_stock_prices, rows):
    date1 = y_[mask][y_[mask]["Ticker"] == ticker]["Date"].values[0]
    date2 = y_[mask][y_[mask]["Ticker"] == ticker]["Date2"].values[0]
    rows = getStockPriceBetweenDates(date1, date2,\
                                     ticker, daily_stock_prices, rows)
    return rows
```

We get the rows of the daily stock data, from the day the annual report is publicly available to the day we sell the stock, by filtering the *daily_stock_prices* DataFrame inside the *getStockPriceBetweenDates* function, using a bit of logic.

getStockPriceBetweenDates

The logic filter is essentially saying "give me *daily_stock_prices* where the date column is between *date1* and *date2*, and the ticker is the stock ticker I care about". Inside this function *daily_stock_prices* is called *d* for short. You can try this function out on its own if you want to get the stock price history between two dates.

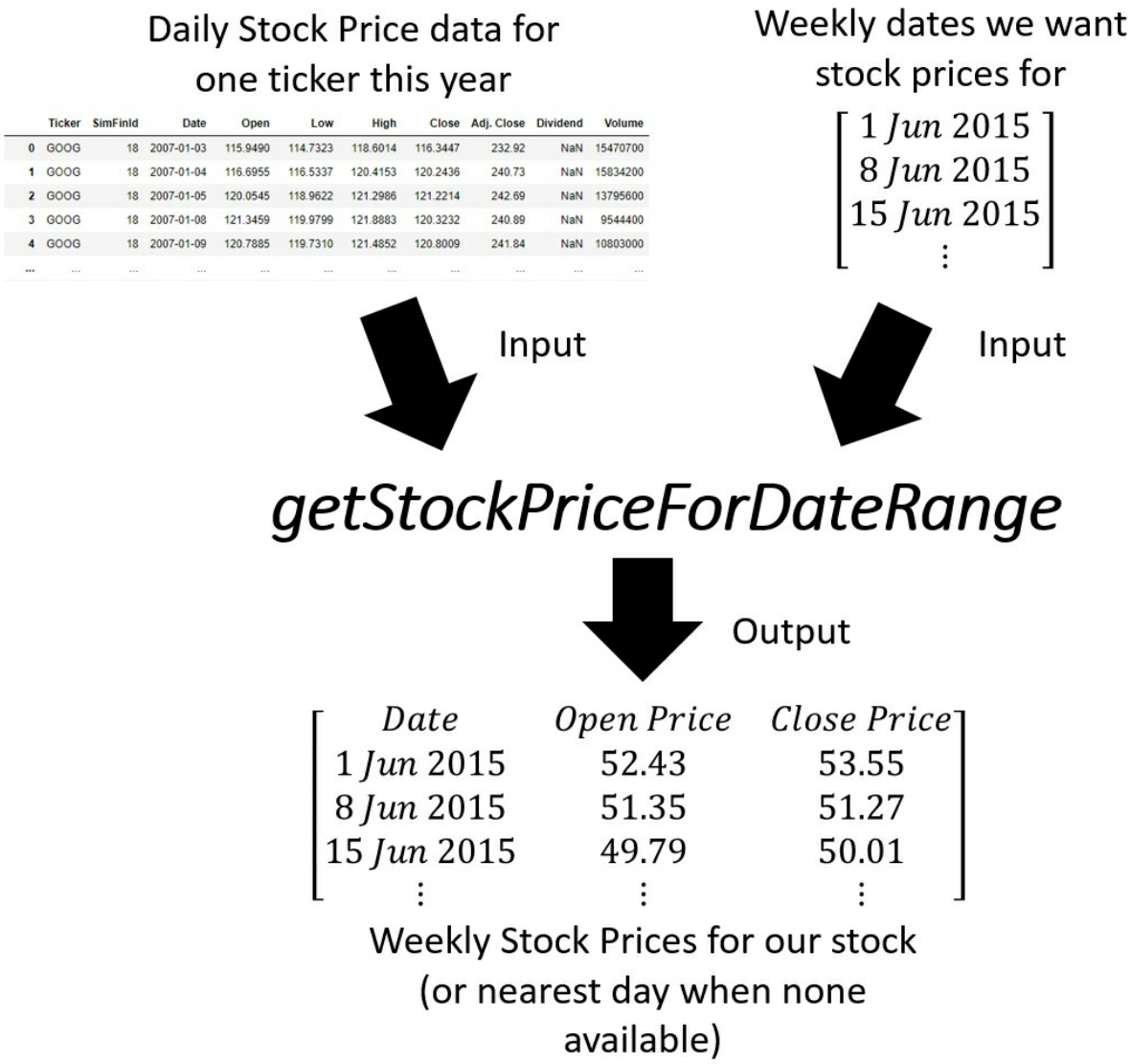
In[11]:

```
def getStockPriceBetweenDates(date1, date2, ticker, d, rows):
    rows = d.loc[(d["Date"].values>date1) &\
                 (d["Date"].values<date2) &\
                 (d["Ticker"]==ticker)]
    return rows
```

Thankfully getting the price data for a single stock is the lowest level function, this is as deep as the rabbit hole goes.

getStockPriceForDateRange

The other function in *getStockTimeSeries* is *getStockPriceForDateRange*. Here *rows* is the daily stock prices for our chosen stock, and *date_Index_New* is the series of weekly dates we want our stock price data for:



	Ticker	SimFinId	Date	Open	Low	High	Close	Adj. Close	Dividend	Volume
0	GOOG	18	2007-01-03	115.9490	114.7323	118.6014	116.3447	232.92	NaN	15470700
1	GOOG	18	2007-01-04	116.6955	116.5337	120.4153	120.2436	240.73	NaN	15834200
2	GOOG	18	2007-01-05	120.0545	118.9622	121.2986	121.2214	242.69	NaN	13795600
3	GOOG	18	2007-01-08	121.3459	119.9799	121.8883	120.3232	240.89	NaN	9544400
4	GOOG	18	2007-01-09	120.7885	119.7310	121.4852	120.8009	241.84	NaN	10803000
...	...	...	...	...	...	...	...	...	...	...

In[12]:

```
def getDataForDateRange(date_Index_New, rows):
    '''
    Given a date range(index), and a series of rows,
    that may not correspond exactly,
    return a DataFrame that gets rows data,
    for each period in the date range(index)
    '''
    WeeklyStockDataRows = pd.DataFrame()
    for I in date_Index_New:
        WeeklyStockDataRows = WeeklyStockDataRows.append(\
          rows.iloc[rows.index.get_loc(I, method="nearest")])
    return WeeklyStockDataRows
```

Notice that we are using the function *iloc*, getting the nearest value in the rows DataFrame.

getPortfolioRelativeTimeSeries

The final function *getPortfolioRelativeTimeSeries* will take in the DataFrame from *getStockTimeSeries* and normalise the stock data so that they start the year at 1.0. This way we can get the portfolio performance by averaging the stock returns.

In[13]:

```
def getPortfolioRelativeTimeSeries(stockRet):
    '''
    Takes DataFrame of stock returns, one column per stock
    Normalises all the numbers so the price at the start is 1.
    Adds a column for the portfolio value.
    '''
    for key in stockRet.keys():
        stockRet[key]=stockRet[key]/stockRet[key][0]
  stockRet["Portfolio"]=stockRet.sum(axis=1)/\
                        (stockRet.keys().shape[0])
    return stockRet
```

First Backtest

Excellent, congratulations if you have made it this far. Now we can get some nice backtesting results and see how our investing AI would do!

Running our master function *getPortTimeSeries* will print the performance for each stock selected for each year as well as the portfolio performance. The output below is with our Linear Regressor training with 50% of the stocks (back testing with the other 50%). It seems to offer us reasonable results.

In[14]:

```
trained_model_pipeline = pickle.load(open("pl_linear.p", "rb" ))
#trained_model_pipeline = pickle.load(open("rfregressor.p", "rb" ))

backTest = getPortTimeSeries(y_withData_Test, X_test,
                             daily_stock_prices_data,
                             trained_model_pipeline)

print('Performance is: ',
      100 * (backTest["Indexed Performance"][-1]-1),
      '%')
```

Out[14]:

```
Backtest performance for year starting  2009-12-31 00:00:00  is: 21.95 %
With stocks: ['ALNY' 'SBH' 'GRA' 'EV' 'TMUS' 'WEN' 'SBAC']
ALNY Performance was: -42.39 %
```

SBH Performance was: 57.52 %
GRA Performance was: 23.31 %
EV Performance was: 1.98 %
TMUS Performance was: 102.73 %
WEN Performance was: -3.55 %
SBAC Performance was: 14.09 %

Backtest performance for year starting 2010-12-31 00:00:00 is: 0.05 %
With stocks: ['REV' 'INVA' 'PCRX' 'AAL' 'KATE' 'AXL' 'RGC']
REV Performance was: 0.96 %
INVA Performance was: -1.77 %
PCRX Performance was: 25.18 %
AAL Performance was: -45.56 %
KATE Performance was: 68.94 %
AXL Performance was: -29.93 %
RGC Performance was: -17.49 %

Backtest performance for year starting 2011-12-31 00:00:00 is: -12.05 %
With stocks: ['NURO' 'DPZ' 'NCMI' 'MTOR' 'NAV' 'CALL' 'CBB']
NURO Performance was: -40.52 %
DPZ Performance was: 11.08 %
NCMI Performance was: -6.83 %
MTOR Performance was: -30.66 %
NAV Performance was: -39.99 %
CALL Performance was: -18.5 %
CBB Performance was: 41.07 %

Backtest performance for year starting 2012-12-31 00:00:00 is: 95.03 %
With stocks: ['NURO' 'OTEL' 'DISH' 'NCMI' 'REV' 'AKS' 'CBB']
NURO Performance was: 7.48 %
OTEL Performance was: 400.0 %
DISH Performance was: 64.75 %
NCMI Performance was: 32.96 %
REV Performance was: 18.13 %
AKS Performance was: 126.59 %
CBB Performance was: 15.28 %

Backtest performance for year starting 2013-12-31 00:00:00 is: -8.95 %
With stocks: ['WYNN' 'NCMI' 'SSNI' 'BXC' 'NKTR' 'AAL' 'LEE']
WYNN Performance was: -37.01 %
NCMI Performance was: -6.67 %
SSNI Performance was: -53.98 %
BXC Performance was: -21.23 %
NKTR Performance was: 21.43 %
AAL Performance was: 46.4 %
LEE Performance was: -11.62 %

Backtest performance for year starting 2014-12-31 00:00:00 is: 63.7 %
With stocks: ['VAPE' 'NURO' 'ALLE' 'CNXR' 'AVXL' 'INVA' 'CLF']
VAPE Performance was: -98.91 %
NURO Performance was: -64.94 %
ALLE Performance was: 12.73 %
CNXR Performance was: -63.2 %
AVXL Performance was: 778.75 %
INVA Performance was: -43.44 %
CLF Performance was: -75.07 %

Backtest performance for year starting 2015-12-31 00:00:00 is: 45.83 %
With stocks: ['VAPE' 'CHRS' 'AC' 'WYNN' 'SGY' 'UIS' 'EXEL']
VAPE Performance was: nan %
CHRS Performance was: 85.75 %

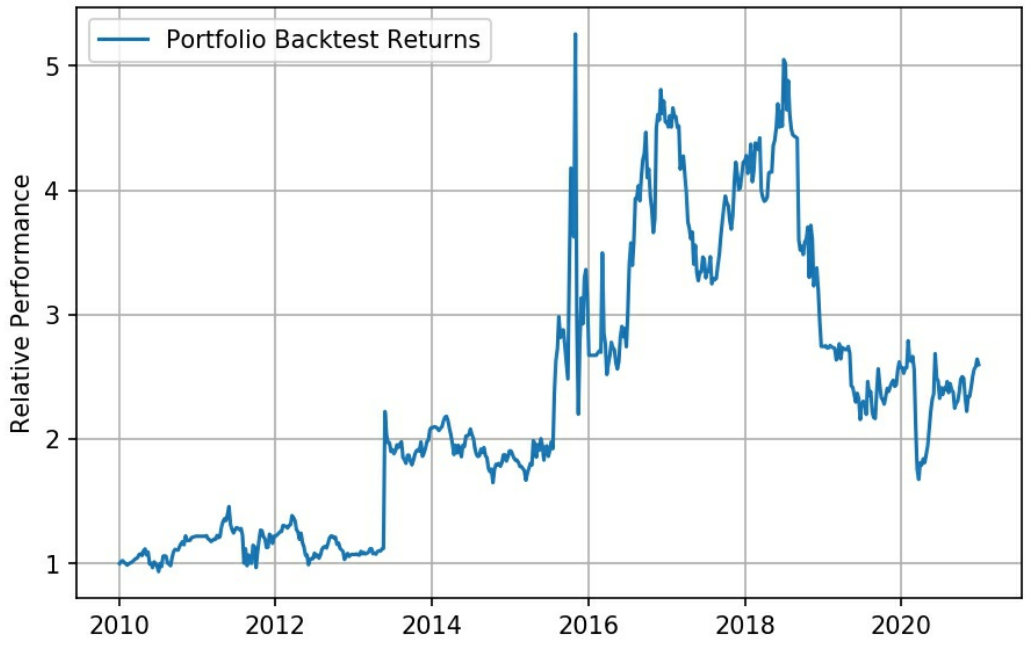
```
AC Performance was: 30.36 %
WYNN Performance was: 4.41 %
SGY Performance was: -48.78 %
UIS Performance was: 39.11 %
EXEL Performance was: 309.92 %
-----------------------------------------------
Backtest performance for year starting  2016-12-31 00:00:00  is: -7.06 %
With stocks: ['HALO' 'NVAX' 'INVA' 'ONS' 'SGY' 'LEE' 'OPHT']
HALO Performance was: 57.05 %
NVAX Performance was: -21.19 %
INVA Performance was: 21.69 %
ONS Performance was: -60.7 %
SGY Performance was: -13.16 %
LEE Performance was: -19.3 %
OPHT Performance was: -13.83 %
-----------------------------------------------
Backtest performance for year starting  2017-12-31 00:00:00  is: -35.06 %
With stocks: ['VTVT' 'LEE' 'WNDW' 'SGMS' 'DIN' 'NAV' 'EGAN']
VTVT Performance was: -79.47 %
LEE Performance was: -5.22 %
WNDW Performance was: -65.0 %
SGMS Performance was: -61.94 %
DIN Performance was: 1.16 %
NAV Performance was: -36.81 %
EGAN Performance was: 1.87 %
-----------------------------------------------
Backtest performance for year starting  2018-12-31 00:00:00  is: -6.21 %
With stocks: ['DBD' 'MNKD' 'WK' 'PZZA' 'SGMS' 'ENR' 'ENDP']
DBD Performance was: 18.67 %
MNKD Performance was: -29.95 %
WK Performance was: -12.19 %
PZZA Performance was: 45.85 %
SGMS Performance was: -8.46 %
ENR Performance was: 4.21 %
ENDP Performance was: -61.57 %
-----------------------------------------------
Backtest performance for year starting  2019-12-31 00:00:00  is: 0.9 %
With stocks: ['TDG' 'LEE' 'DPZ' 'IO' 'PHX' 'KMB' 'DDOG']
TDG Performance was: -1.17 %
LEE Performance was: -21.13 %
DPZ Performance was: 38.52 %
IO Performance was: -43.16 %
PHX Performance was: -67.79 %
KMB Performance was: -7.98 %
DDOG Performance was: 109.01 %
-----------------------------------------------
Performance is:  159.81371725599084 %
```

Investigating Back Test Results

The returned *port_perf_all_years* DataFrame when plotted out gives us the stock portfolio performance visually over the whole backtest.

In[15]:

```
plt.plot(test)
plt.grid()
plt.legend(['Portfolio Backtest Returns'])
plt.ylabel('Relative Performance')
```



Notice that the returns are quite volatile, and the returns are below the market most of the time when compared with the S&P500 index. Don't worry about the underperformance for now, we have many more regressors we can choose from, and we won't know much until we rigorously backtest the strategies.

We can take a closer look at results from the backtest to gain insight into what the investing AI is doing. The process is relatively simple since our functions we wrote just now can be used outside of the backtest function.

Let's take a look at the 2018 year of this backtest, as it delivered a negative return there. We can create a boolean mask to filter our data down to near the start of that year for company financial reporting.

In[16]:

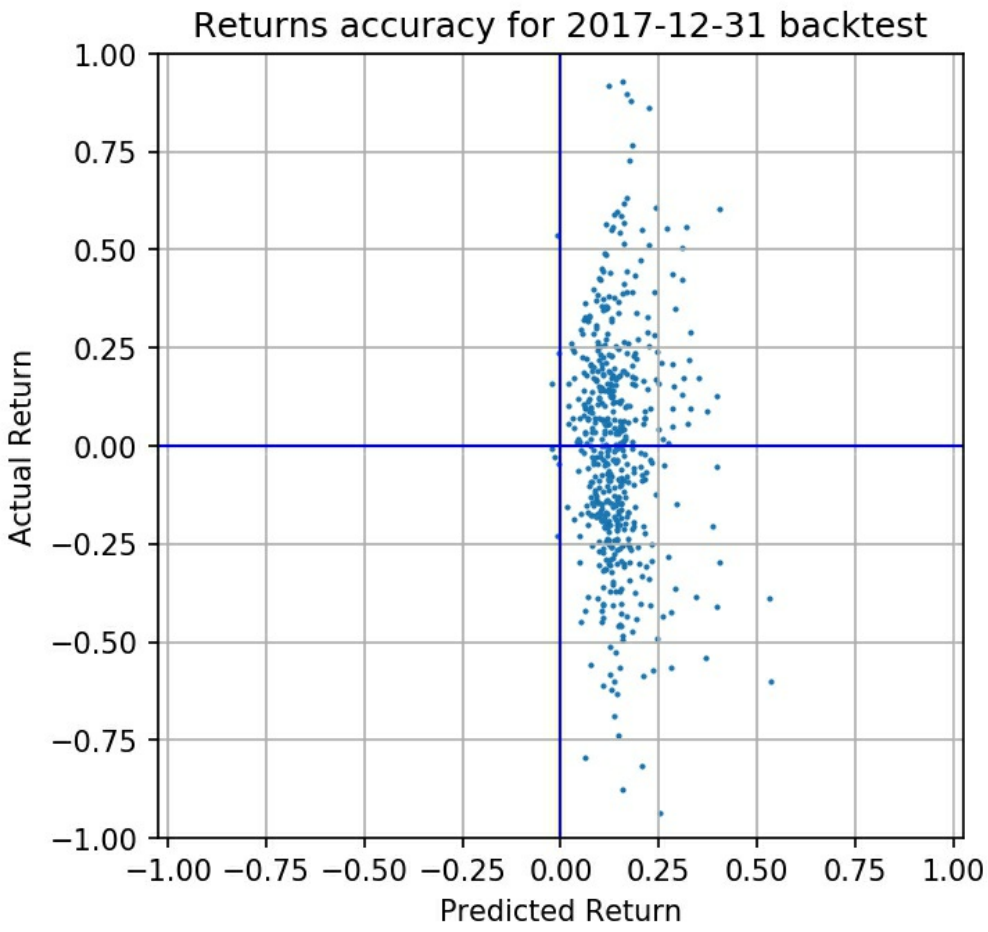
```
y_small=getYPerf(y_withData_Test)
# y_small is cut down version of y with stock returns only

# Create a boolean mask for the backtest year we are interested in
myDate = pd.to_datetime('2017-12-31')
mask2015 = y_withData_Test["Date"].between(pd.to_datetime(myDate)
                                           -pd.Timedelta(days=60),
                                           pd.to_datetime(myDate)
                                           +pd.Timedelta(days=60))
```

Loading our model and plotting the scatter as we have done before gives us a scatter plot with fewer points, as to be expected as these predictions are for

only one year.

In[17]:

```
# Load the model pipeline
ml_model_pipeline = pickle.load(open("pl_linear.p", "rb" ))

# Get stock performance predictions
y_pred = ml_model_pipeline.predict(X_test[mask2015])
y_pred = pd.DataFrame(y_pred) # Turn into DataFrame


# Now output scatter graph to see prediction/actual
# for that year only.
plt.figure(figsize=(5,5))
from matplotlib import pyplot as plt
import matplotlib.lines as mlines
plt.scatter(y_pred[0], y_small[mask2015]["Perf"], s=1)
# Formatting
plt.grid()
plt.axis('equal')
plt.title('Returns accuracy for {}-{}-{} backtest'.format(myDate.year, myDate.month, myDate.day))
plt.xlabel('Predicted Return')
plt.ylabel('Actual Return')
plt.axvline(c='blue', lw=1)
plt.axhline(c='blue', lw=1)
plt.savefig('result.png')
plt.axis([-1,1,-1,1]);
```



We can see that the predictions weren't good for 2015 as there are points approaching the bottom right of the scatter plot, and the distribution has an slant to the left. Remember the AI just picks the stocks it predicts will give the greatest return, so it will pick the 7 rightmost stocks, so if the plot is slanting to the left we are selecting bas stocks. The overall return was negative this time around. We know that one of the worst-performing stock selections was NAV from our printout:

Backtest performance for year starting 2017-12-31 00:00:00 is: -35.06 %
With stocks: ['VTVT' 'LEE' 'WNDW' 'SGMS' 'DIN' 'NAV' 'EGAN']
VTVT Performance was: -79.47 %
LEE Performance was: -5.22 %
WNDW Performance was: -65.0 %
SGMS Performance was: -61.94 %
DIN Performance was: 1.16 %
NAV Performance was: -36.81 %
EGAN Performance was: 1.87 %

We can plot the stock price graph for just this one stock in that year by using the *getStockPriceData* function individually with the boolean mask we made earlier.

In[18]:

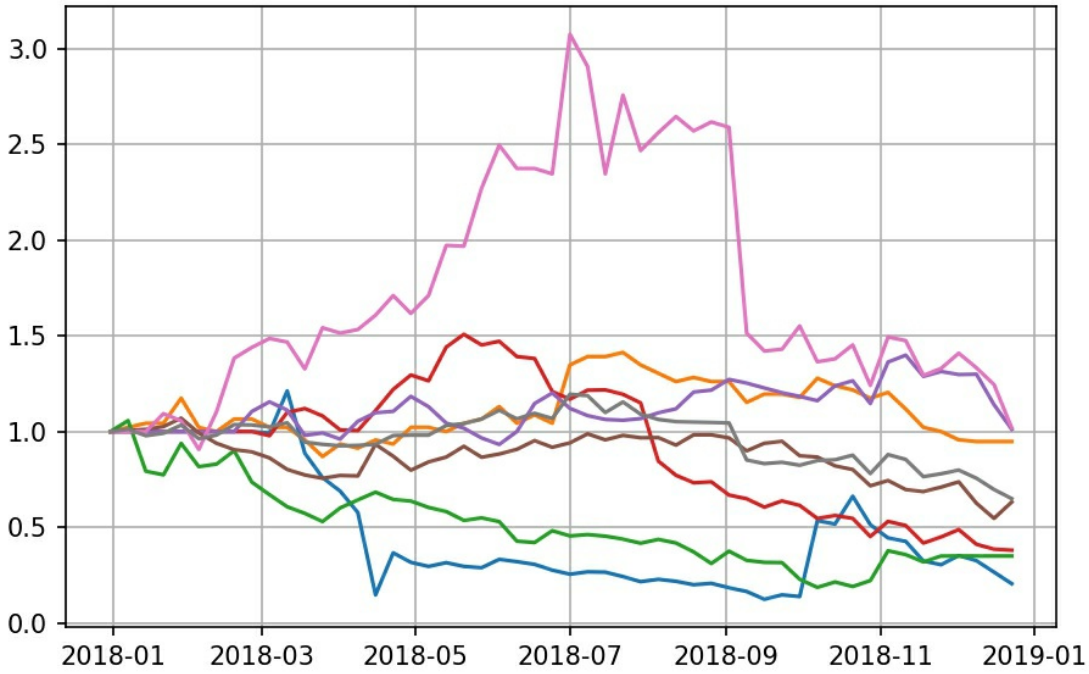
```
rows = getStockPriceData("BXC",
                         y_withData_Test,
                         mask2015,
                         daily_stock_prices_data,
                         rows=pd.DataFrame())

plt.plot(rows["Date"], rows["Close"]) # Adj. Close
plt.grid(True)
```

It seems that stock did not do too well. We know that overall any AI outperformance is only going to be a slight one (remember the potato density shape), we are almost certainly going to have some duds. It seems Navistar International Corp (NAV) doubled in the year before, and roas in the year after our AI purchased it, so the AI timed the investment perfectly for a loss. Plotting all the stocks out, we can see why the portfolio had a loss for the year:

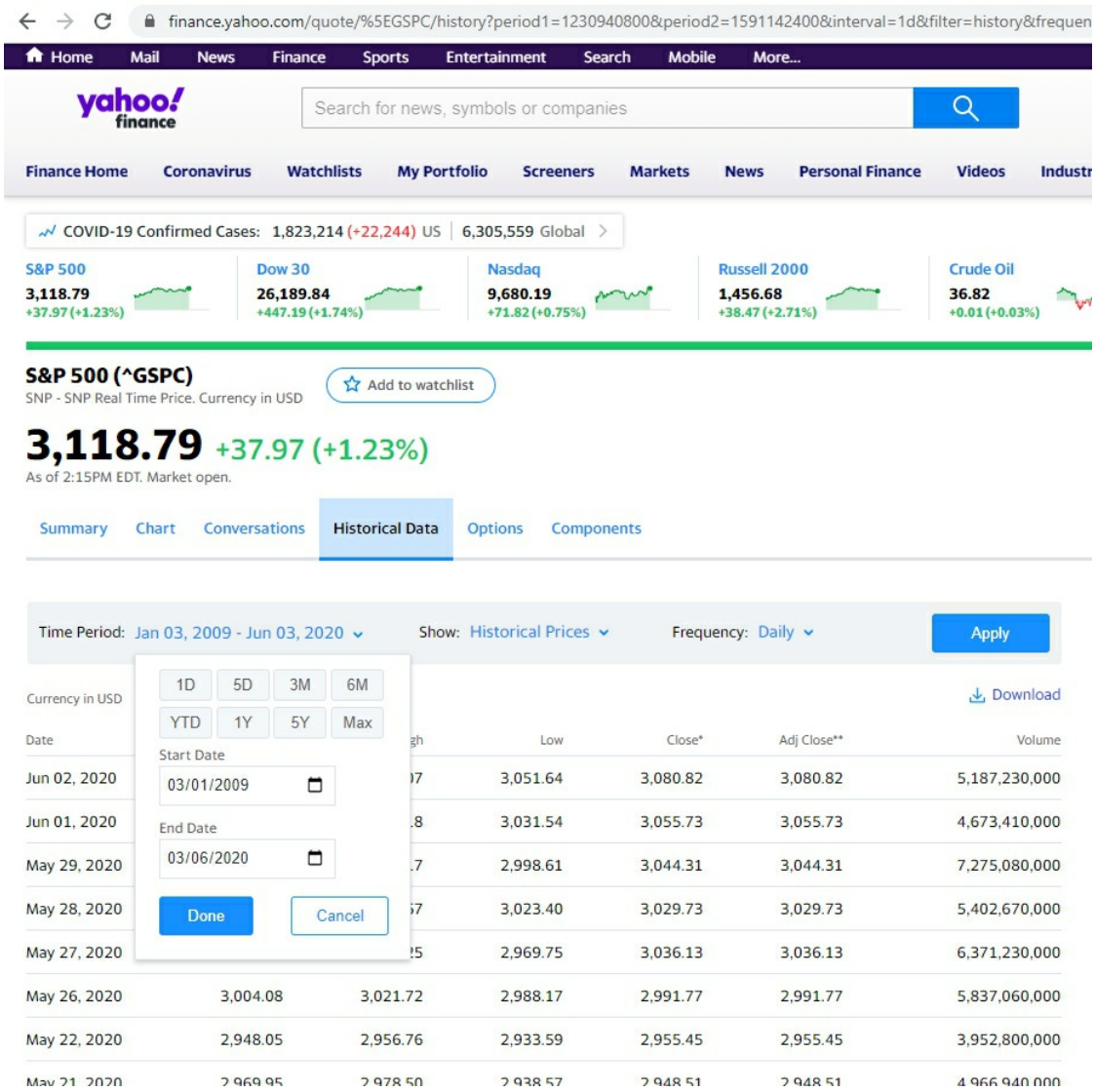
In[19]:

```
# Make X ticks standard, and grab stock prices as close to
# those points as possible for each stock (To track performance)
```

```
#DatetimeIndex
date_range = pd.date_range(start=myDate, periods=52, freq='W')
bl_bestStocks = (y_pred[0] > y_pred.nlargest(8,0).tail(1)[0].values[0])

# 7 greatest performance stocks of y_pred
ticker_list = y_small[mask2015].reset_index(drop=True)[bl_bestStocks]\
                                                ["Ticker"].values

print(ticker_list)

stockRet = getStockTimeSeries(date_range, y_withData_Test,
                              ticker_list,
                              mask2015,
                              daily_stock_prices_data)

stockRetRel = getPortfolioRelativeTimeSeries(stockRet)

stockRetRel.head()

plt.plot(stockRetRel);
plt.grid()
```



We can hope that the other regressors give us better returns. For now let's compare our returns to the S&P500 index. S&P500 index data can be found on *Yahoo.com* as a csv file. You can also get the S&P500 index data using the Pandas datareader function directly, as in the Pandas tutorial in this book (This is an extension of the Pandas library that doesn't look to be well maintained, so the steps here are for downloading the .csv file and reading it).



Be sure to download it with a weekly frequency to match our backtest. We can use this to compare the volatility of returns as well as the returns themselves. The volatility is the standard deviation of the returns, which has the equation:

$$\sigma = \sqrt{\frac{\sum_{t=1}^{T} (r(t) - \mu)^2}{T - 1}}$$

Where *r(t)* is the returns every week (in our case), μ is the average of all the weekly returns, *T* is the total number of time periods and *t* is the current time period. We can convert this to annual volatility by multiplying the weekly volatility by *sqrt(52)*. If our return data is in a DataFrame this can be done

with this one-liner if the S&P500 returns are in a DataFrame:

```
spy["SP500"].diff().std()*np.sqrt(52)
```

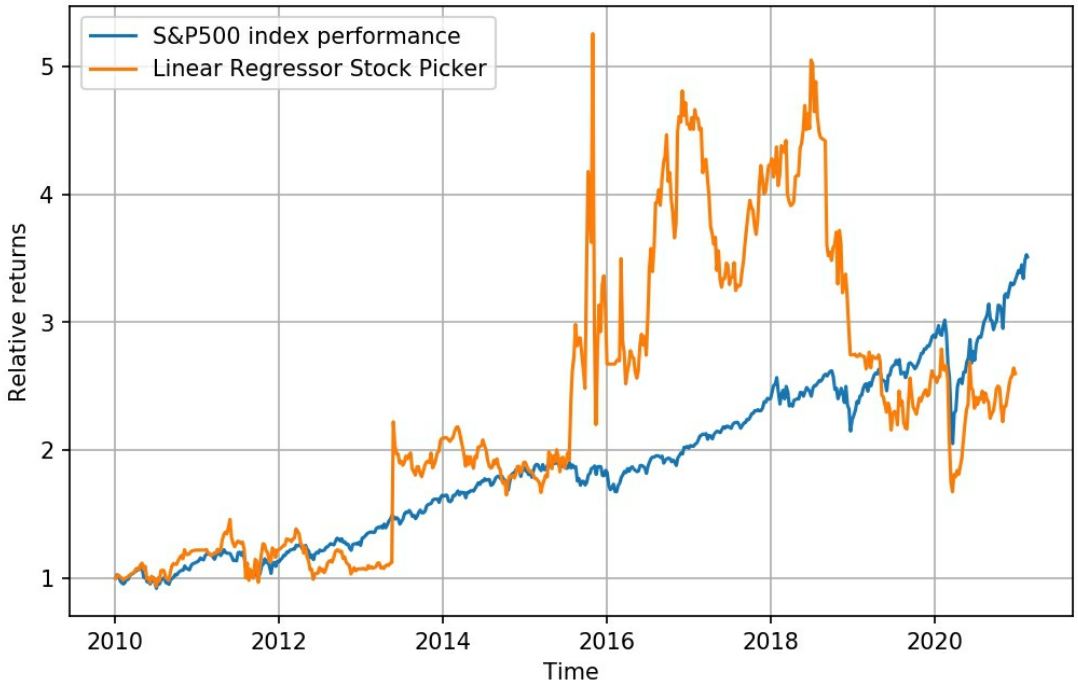
Let’s compare our AI investor to the S&P500 from 2010 to 2019:

In[20]:

```
spy=pd.read_csv("GSPC.csv", index_col='Date', parse_dates=True)
spy['Relative'] = spy["Open"]/spy["Open"][0]

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(8,5))
plt.plot(spy['Relative'])
plt.plot(backTest)
plt.grid()
plt.xlabel('Time')
plt.ylabel('Relative returns')
plt.legend(['S&P500 index performance', \
            'Linear Regressor Stock Picker'])
#plt.savefig('spy.png')
print('volatility of AI investor was: ', \
      backTest['Indexed Performance'].diff().std()*np.sqrt(52))
print('volatility of S&P 500 was: ', \
      spy["Relative"].diff().std()*np.sqrt(52))
```

Out[20]:

```
volatility of AI investor was:  1.3759320717225751
volatility of S&P 500 was:  0.325059094152396
```



In this backtest our volatility is much higher than the S&P 500, with performance that doesn't adequately compensate us for the swings. Bear in mind that so far we have only tried our Linear Regressor in the backtest, with it choosing from half of the available rows to invest in and trained using the other half of the rows as training data.

It should also be noted that this is only a single backtest, and the train/test split will be different if we make another random sample, changing the resulting return and volatility.

An issue we have is that our data doesn't contain stock information for companies that went bankrupt. This is survivorship bias. Rather than take free stock data from SimFin.com you could pay for better data that does contain stocks that go to 0, here we are going to use Altman Z scores to minimise our chances of picking stocks that are about to go bankrupt.

Accounting for Chance of Default

Edward Altman is an academic who found some correlation formulas linking company ratios to the probability of default. He has published several formulas in his career, a quick google will usually give you his first formula published in 1968. We will use his newer formula published in 1995 in our AI investor, which is as follows:

$$Z'' = \ 3.25 + 6.56X_1 + 3.26X_2 + 6.72X_3 + 1.05X_4$$

Where:

$$X_1 = \frac{Current\ Assets - Current\ Liabilities}{Total\ Assets}$$

$$X_2 = \frac{Retained\ Earnings}{Total\ Assets}$$

$$X_3 = \frac{Earnings\ Before\ Interest\ and\ Taxes}{Total\ Assets}$$

$$X_4 = \frac{Equity\ Book\ Value}{Total\ Liabilities}$$

The lower the *Z* score, the higher the risk of default. A *Z* score of 4 is roughly equivalent to a B credit rating, which is roughly equivalent to a 5% risk of default, though the range is quite large. The chance of default is published in S&P and Moodys ratings data.

A look at the Z scores of the stocks selected in the backtest reveals that a few of them were close to bankruptcy. For our Linear Regressor in the 2015 backtest three of the stocks have a negative Z score.

In[21]:

```
# Top stocks picked, and predicted performance.
bl_bestStocks = (y_pred[0] > y_pred.nlargest(8,0).tail(1)[0].values[0])

print("\nTop predicted perf. stocks picked and predicted performance is:")
print(y_small[mask2015].reset_index(drop=True)[bl_bestStocks]["Ticker"])
print(y_pred[bl_bestStocks])

print("\nActual performance was: ")
print(y_small[mask2015].reset_index(drop=True)[bl_bestStocks])

# Calc Altman Z score.
# To check if these companies were going to go bankrupt
Z = 3.25 \
+ 6.51 *X_test[mask2015].reset_index(drop=True)[bl_bestStocks]\
        ['(CA-CL)/TA']\
+ 3.26 *X_test[mask2015].reset_index(drop=True)[bl_bestStocks]\
        ['RE/TA']\
+ 6.72 *X_test[mask2015].reset_index(drop=True)[bl_bestStocks]\
        ['EBIT/TA']\
+ 1.05 *X_test[mask2015].reset_index(drop=True)[bl_bestStocks]\
        ['Book Equity/TL']
```

```
print('\nZ scores:\n',Z)
```

Out[21]:

```
Top predicted perf. stocks picked and predicted performance is:
44     VTVT
180     LEE
257    WNDW
269    SGMS
341     DIN
377     NAV
473    EGAN
Name: Ticker, dtype: object
                0
44   0.534452
180  0.399055
257  0.532018
269  0.405163
341  0.403704
377  0.397023
473  0.398858

Actual performance was:
     Ticker      Perf
44    VTVT -0.599374
180    LEE -0.052174
257   WNDW -0.386965
269   SGMS -0.296420
341    DIN  0.605108
377    NAV -0.408809
473   EGAN  0.125424

Z scores:
44     -1.691952
180     2.118304
257   -71.950523
269     3.284198
341     1.908852
377     1.155005
473   -27.662049
dtype: float64
```

We can make our investing AI only choose from companies that have a Z score above a certain value, you can change this value depending on how conservative you want your AI to be. There are in fact a few Z scores in the literature, you can take your pick of which you want to use. The original formula from the 1968 paper published in The Journal of Finance works fine even with modern companies, which is as follows:

$$Z = 0.012X_1 + 0.014X_2 + 0.033X_3 + 0.006X_4 + X_5$$

Where:

$$X_1 = \frac{Current\ Assets - Current\ Liabilities}{Total\ Assets}$$

$$X_2 = \frac{Retained\ Earnings}{Total\ Assets}$$

$$X_3 = \frac{Earnings\ Before\ Interest\ and\ Taxes}{Total\ Assets}$$

$$X_4 = \frac{Market\ Cap}{Total\ Liabilities}$$

$$X_5 = \frac{Revenue}{Total\ Assets}$$

The ratios present do make intuitive sense, a company that has an immense quantity of earnings relative to total assets is going to have a lower risk of default than one that does not. If you think that this kind of formula looks like our Linear Regressor you would be right. In fact, our Linear Regressor for predicting stock performance is just a more complicated version of this as we have a lot more ratios and a lot more data points to work with.

The equivalent credit ratings for Z scores of 8 and 4 are AAA and B respectively. We can make our AI as conservative as we want it to be, though it will have a smaller universe of stocks to pick from if you limit it too much. A good splitting of companies by Z scores from Altman's original paper (with different labels) is as follows:

Z>2.99 – Non-Bankrupt Zone

2.99>Z>1.81 – Grey Zone

Z<1.81 – Distress Zone

To limit Z scores in the stock selection we will first generate a DataFrame containing Z scores for all our x test data near the beginning of our notebook.

In[22]:

```
def calcZScores(X):
    '''
    Calculate Altman Z" scores 1995
    '''
    Z = pd.DataFrame()
    Z['Z score'] = 3.25 \
    + 6.51 * X['(CA-CL)/TA']\
```

```
    + 3.26 * X['RE/TA']\
    + 6.72 * X['EBIT/TA']\
    + 1.05 * X['Book Equity/TL']
    return Z

z = calcZScores(X)
```

Then we will edit our *getPortTimeSeriesForYear* function to select the 7 highest predicted return stocks only from stocks which have a Z score above 2 (above the distress zone. We create a new boolean mask *bl_safeStocks* which is the bool mask of the *z* DataFrame items greater than 2.

We then reduce *y_pred* (the predicted stock performances) with this mask and find the *nlargest* from that reduced DataFrame. When it comes to selecting our tickers we use a combined boolean filter (*bl_bestStocks & bl_safeStocks*). Here's what our *getPortTimeSeriesForYear* function looks like with the Z scores integrated:

In[23]:

```
### Proper function ###
def getPortTimeSeriesForYear(date_starting, y_withData, X,
                             daily_stock_prices, ml_model_pipeline):
    '''
    Function runs a backtest.
    Returns DataFrames of selected stocks/portfolio performance,
    for 1 year.
    y_withData is annual stock performances (all backtest years)
    date_starting e.g. '2010-01-01'
    daily_stock_prices is daily(mostly) stock price time series for
    all stocks
    '''

    # get y dataframe with ticker performance only
    y = getYPerf(y_withData)

    # Get performance only for time frame we care about,
    # mask original data using the start date
    thisYearMask = y_withData["Date"].between(\
                pd.to_datetime(date_starting) - pd.Timedelta(days=60),\
                pd.to_datetime(date_starting) + pd.Timedelta(days=60) )

    # Get return prediction from model
    y_pred = ml_model_pipeline.predict(X[thisYearMask])

    # Make it a DataFrame to select the top picks
    y_pred = pd.DataFrame(y_pred)
```

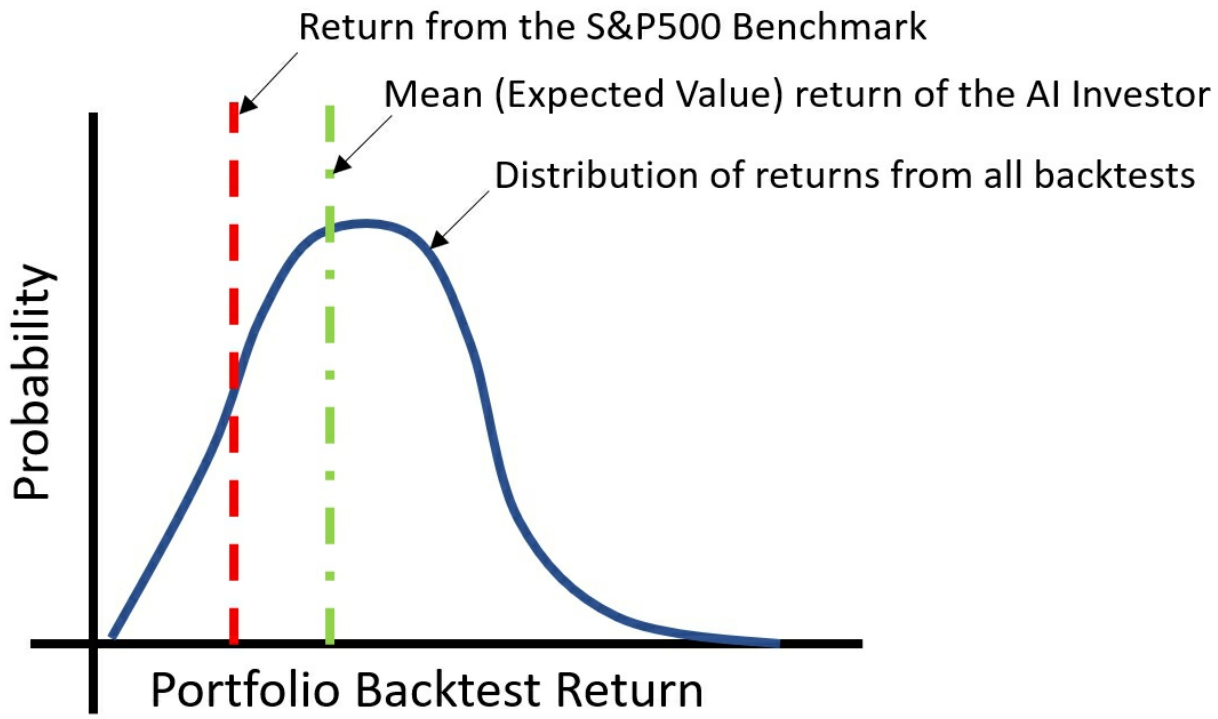
```
    ##### Change in code for Z score filtering #####
    # Separate out stocks with low Z scores
    z = calcZScores(X)

    # 3.75 is approx. B- rating
    bl_safeStocks=(z['Z score'][thisYearMask].reset_index(drop=True)>2)
    y_pred_z = y_pred[bl_safeStocks]

    # Get bool list of top stocks
    bl_bestStocks=(
        y_pred_z[0]>y_pred_z.nlargest(8,0).tail(1)[0].values[0])

    dateTimeIndex = pd.date_range(\
                          start=date_starting, periods=52, freq='W')

    # 7 greatest performance stocks of y_pred
    ticker_list = \
    y[thisYearMask].reset_index(drop=True)\
                      [bl_bestStocks&bl_safeStocks]["Ticker"].values
    ##### Change in code for Z score filtering #####

    # After we know our stock picks, we get the stock performance
    # Get DataFrame index of time stamp, series of stock prices,
    # keys=tickers
    stockRet = getStockTimeSeries(dateTimeIndex, y_withData,
                                  ticker_list, thisYearMask,
                                  daily_stock_prices)

    # Get DataFrame of relative stock prices from 1st day(or close)
    # and whole portfolio
    stockRetRel = getPortfolioRelativeTimeSeries(stockRet)
    return [stockRetRel, stockRetRel["Portfolio"], ticker_list]
```

Using Z scores in our investing AI yields a slightly lower backtest returns. We cannot come to any definitive conclusions from this with only a single backtest, we would have to run many backtests with and without Z score filtering to really know the difference.

We can try any of the regressors we tested in the last chapter here, just replace the Linear Regressor pipeline with the regressor you want to try out. All the code is in the code repository. In the next section we will put our Investing AIs through a battery of backtests and see if we can really trust the returns we are seeing.

Chapter 6 - Backtesting - Statistical Likelihood of Returns

To overcome the uncertainty of our investing AI we can run many backtests, each splitting the train/test data differently. From doing this we will have a large number of backtest returns and volatilities, which we can plot the distributions of to understand the abilities of our investing AIs:



We want the mean result to give us the highest return possible (by selecting the right model), and we would also like the return distribution to be as far to the right of our S&P500 benchmark return as possible.

Keep in mind that all these algorithms are selecting from a reduced universe of stocks, that is, only the test data, and that Altman Z scores don't perfectly protect our results from survivorship bias, so the backtesting performance comparison to the S&P500 is not a perfect one.

Each backtest will be from the beginning of 2010 to the end of 2020, comparing performance to our benchmark in the same period.

Backtesting Loop

We need a way to train our models on a new training set each iteration, so we will train models in functions which return the pipeline itself, using model hyperparameters we found to be favourable earlier:

In[1]:

```
# Linear model pipeline
def trainLinearModel(X_train, y_train):
    pl_linear = Pipeline([('Power Transformer', PowerTransformer()),
        ('linear', LinearRegression())])
    pl_linear.fit(X_train, y_train)
    return pl_linear

# ElasticNet model pipeline
def trainElasticNetModel(X_train, y_train):
    pl_ElasticNet = Pipeline([('Power Transformer', PowerTransformer()),
        ('ElasticNet', ElasticNet(l1_ratio=0.00001))])
    pl_ElasticNet.fit(X_train, y_train)
    return pl_ElasticNet

# KNeighbors regressor
def trainKNeighborsModel(X_train, y_train):
    pl_KNeighbors = Pipeline([('Power Transformer', PowerTransformer()),
        ('KNeighborsRegressor', KNeighborsRegressor(n_neighbors=40))])
    pl_KNeighbors.fit(X_train, y_train)
    return pl_KNeighbors

# DecisionTreeRegressor
def traindecTreeModel(X_train, y_train):
    pl_decTree = Pipeline([
        ('DecisionTreeRegressor',\
         DecisionTreeRegressor(max_depth=20, random_state=42))
    ])
    pl_decTree.fit(X_train, y_train)
    return pl_decTree

# RandomForestRegressor
def trainrfregressorModel(X_train, y_train):
    pl_rfregressor = Pipeline([
        ('RandomForestRegressor',\
         RandomForestRegressor(max_depth=10, random_state=42))
    ])
    pl_rfregressor.fit(X_train, y_train)

    return pl_rfregressor

# GradientBoostingRegressor
def traingbregressorModel(X_train, y_train):
    pl_GradBregressor = Pipeline([
```

```
        ('GradBoostRegressor',\
         GradientBoostingRegressor(n_estimators=100,\
                                   learning_rate=0.1,\
                                   max_depth=10,\
                                   random_state=42,\
                                   loss='ls'))  ])
    pl_GradBregressor.fit(X_train, y_train)

    return pl_GradBregressor

# SVM
def trainsvmModel(X_train, y_train):
    pl_svm = Pipeline([('Power Transformer', PowerTransformer()),
        ('SVR', SVR(kernel='rbf', C=100, gamma=0.1, epsilon=.1))])
    pl_svm.fit(X_train, y_train)
    return pl_svm
```

Before we get out backtesting statistics proper we need a rough idea of what a good train/test split is, as we want to fix this number for all backtest runs. Of course, you could do a grid search, however, we want to keep computing requirements reasonable.

We would like a one size fits all, and we don't need to be accurate in finding an optimal split here, we need a split that will tell us enough information about the investing AI for us to gauge the probable risk/reward it would subject us to if it managed our money, bear in mind that different models will have different optimal splits.

We will do a quick loop through all the models, and for each model run a range of *test_size* values in a backtest. We do not need to run multiple backtests for each model/*test_size* as we just need a rough value for us to fix this value at.

In[2]:

```
model_pipeline_list = ['pl_linear', 'pl_KNeighbors', 'pl_decTree',\
                       'pl_GradBregressor', 'pl_rfregressor', 'pl_svm']

for model in model_pipeline_list:
    for myTstSze in [0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]:
        X_train,X_test,y_train,y_test=train_test_split(X, y_pec,
                                                       test_size=myTstSze)

        if (model =='pl_ElasticNet'):
            model_pl = trainElasticNetModel(X_train, y_train)
        if (model =='pl_KNeighbors'):
            model_pl = trainKNeighborsModel(X_train, y_train)
        if (model =='pl_decTree'):
```

```
        model_pl = traindecTreeModel(X_train, y_train)
    if (model =='pl_rfregressor'):
        model_pl = trainrfregressorModel(X_train, y_train)
    if (model =='pl_GradBregressor'):
        model_pl = traingbregressorModel(X_train, y_train)
    if (model =='pl_svm'):
        model_pl = trainsvmModel(X_train, y_train)
    else:
        # Linear model default
        model_pl = trainLinearModel(X_train, y_train)

    y_withData_Test=y_withData.loc[X_test.index]

    # Here is our backtesting code
    test = getPortTimeSeries(y_withData_Test, X_test,
                             daily_stock_prices,
                             model_pl,
                             verbose=False)

    perf = test['Indexed Performance'][-1]
    vol = test['Indexed Performance'].diff().std()*np.sqrt(52)
    print('Performance:', round(perf, 2),
          'Volatility:', round(vol,2),
          'Model', model, 'Test size: ', myTstSze)
```

Out[2]:

```
Performance: 4.69 Volatility: 0.6 Model pl_linear Test size:  0.02
Performance: 5.08 Volatility: 0.8 Model pl_linear Test size:  0.1
Performance: 2.55 Volatility: 0.42 Model pl_linear Test size:  0.25
Performance: 1.36 Volatility: 0.29 Model pl_linear Test size:  0.5
Performance: 3.69 Volatility: 0.57 Model pl_linear Test size:  0.75
Performance: 4.69 Volatility: 0.73 Model pl_linear Test size:  0.9
Performance: 9.1 Volatility: 1.29 Model pl_linear Test size:  0.98
Performance: 2.5 Volatility: 0.37 Model pl_KNeighbors Test size:  0.02
Performance: 2.09 Volatility: 0.43 Model pl_KNeighbors Test size:  0.1
Performance: 2.48 Volatility: 0.37 Model pl_KNeighbors Test size:  0.25
Performance: 3.06 Volatility: 0.62 Model pl_KNeighbors Test size:  0.5
Performance: 2.93 Volatility: 0.77 Model pl_KNeighbors Test size:  0.75
Performance: 2.13 Volatility: 0.26 Model pl_KNeighbors Test size:  0.9
Performance: 3.75 Volatility: 0.68 Model pl_KNeighbors Test size:  0.98
Performance: 5.99 Volatility: 0.73 Model pl_decTree Test size:  0.02
Performance: 2.54 Volatility: 0.42 Model pl_decTree Test size:  0.1
Performance: 2.2 Volatility: 0.36 Model pl_decTree Test size:  0.25
Performance: 2.58 Volatility: 0.34 Model pl_decTree Test size:  0.5
Performance: 1.18 Volatility: 0.21 Model pl_decTree Test size:  0.75
Performance: 3.67 Volatility: 0.54 Model pl_decTree Test size:  0.9
Performance: 2.58 Volatility: 0.38 Model pl_decTree Test size:  0.98
Performance: 2.64 Volatility: 0.41 Model pl_GradBregressor Test size:  0.02
Performance: 2.13 Volatility: 0.51 Model pl_GradBregressor Test size:  0.1
Performance: 2.71 Volatility: 0.61 Model pl_GradBregressor Test size:  0.25
Performance: 1.89 Volatility: 0.31 Model pl_GradBregressor Test size:  0.5
Performance: 3.86 Volatility: 0.73 Model pl_GradBregressor Test size:  0.75
Performance: 2.08 Volatility: 0.5 Model pl_GradBregressor Test size:  0.9
Performance: 1.57 Volatility: 0.25 Model pl_GradBregressor Test size:  0.98
Performance: 2.34 Volatility: 0.43 Model pl_rfregressor Test size:  0.02
Performance: 4.13 Volatility: 0.98 Model pl_rfregressor Test size:  0.1
```

```
Performance: 2.76 Volatility: 0.46 Model pl_rfregressor Test size:  0.25
Performance: 3.28 Volatility: 0.53 Model pl_rfregressor Test size:  0.5
Performance: 1.88 Volatility: 0.44 Model pl_rfregressor Test size:  0.75
Performance: 2.13 Volatility: 0.29 Model pl_rfregressor Test size:  0.9
Performance: 2.32 Volatility: 0.47 Model pl_rfregressor Test size:  0.98
Performance: 3.35 Volatility: 0.46 Model pl_svm Test size:  0.02
Performance: 3.0 Volatility: 0.59 Model pl_svm Test size:  0.1
Performance: 2.5 Volatility: 0.42 Model pl_svm Test size:  0.25
Performance: 8.34 Volatility: 0.88 Model pl_svm Test size:  0.5
Performance: 6.96 Volatility: 0.76 Model pl_svm Test size:  0.75
Performance: 1.21 Volatility: 0.37 Model pl_svm Test size:  0.9
Performance: 2.62 Volatility: 0.64 Model pl_svm Test size:  0.98
```

The performance is quite varied, and it seems that the optimal test set sizes for the models certainly differ too. This is not all that surprising given how different all the Machine Learning algorithms are. A *test_size* split of 0.5 is reasonable, so we will use this for the backtesting, we only want to separate the wheat from the chaff at this stage.

Almost all our backtest code so far is used within the *getPortTimeSeries* function. We pass our *y* and *X* vectors, daily stock price data and the model object to the function, and we get back the portfolio backtest price history, which we use to get our return (the last number) and the volatility of the portfolio given those arguments.

To make our statistical back tester we need to run *getPortTimeSeries* in a loop and record the return and volatility each time, passing a new train/test split to our function each iteration. Depending on how many backtests we run this can take a long time. This is a good place to use it if you have a multicore CPU or access to a computing cluster, rudimentary multithreaded code is provided here as no doubt many readers have multicore CPUs and want to speed up this stage.

As there are many regression models we want to run the backtests for, we will loop through the list of model pipeline file names, running a loop of tests for each model. As this will take a while and we will want the data at the end we will save the performance for each backtest in a CSV file to plot later, appending to the file as results are gathered. In parallel there is an extreme remote chance that two cores will try and access the same file, though for code simplicity this is an adequate compromise.

Our backtest loop is placed in a function and can be called normally to use 1 CPU core or run in parallel with more CPU cores. The backtesting results you see here took a weekend of processing to obtain with our code.

In[3]:

```
def getResultsForModel(model_pipeline_list, runs_per_model=1,
                       verbose=True):
    '''
    getResultsForModel
    Choose the model pipelines to run loop for.
    '''
    i, results = 0, []
    for model in model_pipeline_list:
        for test_num in range(0, runs_per_model):
            X_train,X_test,y_train,y_test=train_test_split(X, y_pec,
                                                           test_size=0.5
                                                          )
            #Train different models
            if (model =='pl_linear'):
                model_pl = trainLinearModel(X_train, y_train)
            if (model =='pl_ElasticNet'):
                model_pl = trainElasticNetModel(X_train, y_train)
            if (model =='pl_KNeighbors'):
                model_pl = trainKNeighborsModel(X_train, y_train)
            if (model =='pl_rfregressor'):
                model_pl = trainrfregressorModel(X_train, y_train)
            if (model =='pl_decTree'):
                model_pl = traindecTreeModel(X_train, y_train)
            if (model =='pl_GradBregressor'):
                model_pl = traingbregressorModel(X_train, y_train)
            if (model =='pl_svm'):
                model_pl = trainsvmModel(X_train, y_train)
            y_withData_Test=y_withData.loc[X_test.index]

            # Here is our backtesting code
            test = getPortTimeSeries(y_withData_Test, X_test,
                                     daily_stock_prices, model_pl,
                                         verbose=False)

            perf = test['Indexed Performance'][-1]
            vol = test['Indexed Performance'].diff().std()*np.sqrt(52)
            if verbose:
                print('Performed test ',i , [i, model, perf, vol])
            results.append([i, model, perf, vol])
            i=i+1

    # Save our results for plotting
    results_df = pd.DataFrame(results, columns=["Test Number",\
                                                "Model Used",\
                                                "Indexed Return",\
                                                "Annual Volatility"])

    # Append to an existing results file if available,
    # else make new results file.
    # In parallel there is an extremely remote chance
```

```
    # two cores try and access file at same time.
    # To keep code simple this is OK.
    import os
    if os.path.isfile("Backtest_statistics.csv"):
        results_df.to_csv("Backtest_statistics.csv",
                          mode='a',
                          header=False)
    else:
        results_df.to_csv("Backtest_statistics.csv")
```

To run in series use:

In[4]:

```
# Run with 1 core
#model_pipeline_list = ['pl_KNeighbors']
model_pipeline_list = ['pl_rfregressor',
                       'pl_decTree',
                       'pl_svm',
                       'pl_linear',
                       'pl_GradBregressor',
                       'pl_KNeighbors',
                       'pl_ElasticNet']

getResultsForModel(model_pipeline_list, 3)
```

To run in parallel (this is a quick and dirty implementation, but it is all we need) use:

In[5]:

```
#Run in Parallel
# Specific models on each core
# Can run this many times, will just keep appending to results file.
model_pipeline_list_list = [['pl_GradBregressor', 'pl_ElasticNet'],
                            ['pl_decTree', 'pl_rfregressor'],
                            ['pl_svm'],
                            ['pl_linear', 'pl_KNeighbors']]

import threading
l = len(model_pipeline_list_list)

thread_list = []
for i in range(l):
    thread = threading.Thread(target=getResultsForModel, args=(model_pipeline_list_list[i], 30,
False))
    thread_list.append(thread)
    thread.start()
    print('Thread '+str(i)+' started.')
```

Out[5]:

```
Thread 0 started.
```

Thread 1 started.
Thread 2 started.
Thread 3 started.

Here the *model_pipeline_list_list* is iterated through, with one thread starting for each item in the list. If you have an 8 core CPU you may split the model pipelines to run with one on each thread, for a total of 7 threads. The threads run in the background, to see if they are still running use:

In[6]:

```
for i in thread_list:
    print(i.is_alive())
```

Out[6]:
True
True
True
True

The code can be executed as and when desired, results just get added to the end of the .csv file. This however can't happen if the .csv file is currently open in a program.

Once the simulations are complete, the resulting DataFrame will contain the model used, the test number for that model, the return and volatility.

In[7]:

```
# Backtesting results
results_df = pd.read_csv('Backtest_statistics.csv', index_col=0)
```

```
In [28]: results_df.head()
```

Out[28]:

	Test Number	Model Used	Indexed Return	Annual Volatility
0	0	pl_linear.p	1.822730	0.862156
1	1	pl_linear.p	2.739422	1.549202
2	2	pl_linear.p	10.608832	4.030092
3	3	pl_linear.p	4.893971	1.864813
4	4	pl_linear.p	1.915634	0.758215

Now that we have the statistics we worked so hard for, we can plot a histogram of the data and see how well each of the investing AIs perform.

Is the AI any good at picking stocks?

Let us put the plotting code in a function to make life easier. We will take advantage of a kernel density estimation (KDE) function again to give us a smooth curve from our histogram data, as our histogram data may be noisy depending on how many simulations you ran. Here is our histogram plotting function:

In[8]:

```
def plotBacktestDist(results_df, model_file, col):
    ax=results_df[results_df['Model Used']==model_file][col].hist(\
                                    bins=50, density=True, alpha=0.7)

    ax2=results_df[results_df['Model Used']==model_file][col].plot.kde(\
                                    alpha=0.9)

    max_val=results_df[results_df['Model Used']==model_file][col].max()
    ax.set_xlabel(col)
    ax.set_ylabel('Normalised Frequency')
    ax.set_title('{} Backtest Distribution for {}, {} Runs'.format(\
          col,\
          model_file,\
          results_df[results_df['Model Used']==model_file][col].size))
    ax.grid()
    mean=results_df[results_df['Model Used']==model_file][col].mean()
    ymin, ymax = ax.get_ylim()

    if (col=='Indexed Return'):
        # Plot S&P 500 returns
        # begin 2010 -> end 2020
        ax.plot([3.511822, 3.511822], [ymin, ymax],\
                color='r', linestyle='-', linewidth=1.5, alpha=1)
        ax.plot([mean, mean], [ymin, ymax],\
                color='lime', linestyle='--', linewidth=1.5, alpha=1)
        plt.xlim(0, 10)

    if (col=='Annual Volatility'):
        # Plot S&P 500 volatility
        # begin 2010 -> end 2020
        ax.plot([0.325059094152396, 0.325059094152396], [ymin, ymax],\
                color='r', linestyle='-', linewidth=2)
        ax.plot([mean, mean], [ymin, ymax],\
                color='lime', linestyle='--', linewidth=2)
        plt.xlim(0, 1.5)
    ax.legend(['Fitted Smooth Kernel','S&P500 Benchmark', \
               'Simulation Mean {}'.format(round(mean,2)),\
               'Simulation Backtests'])
```

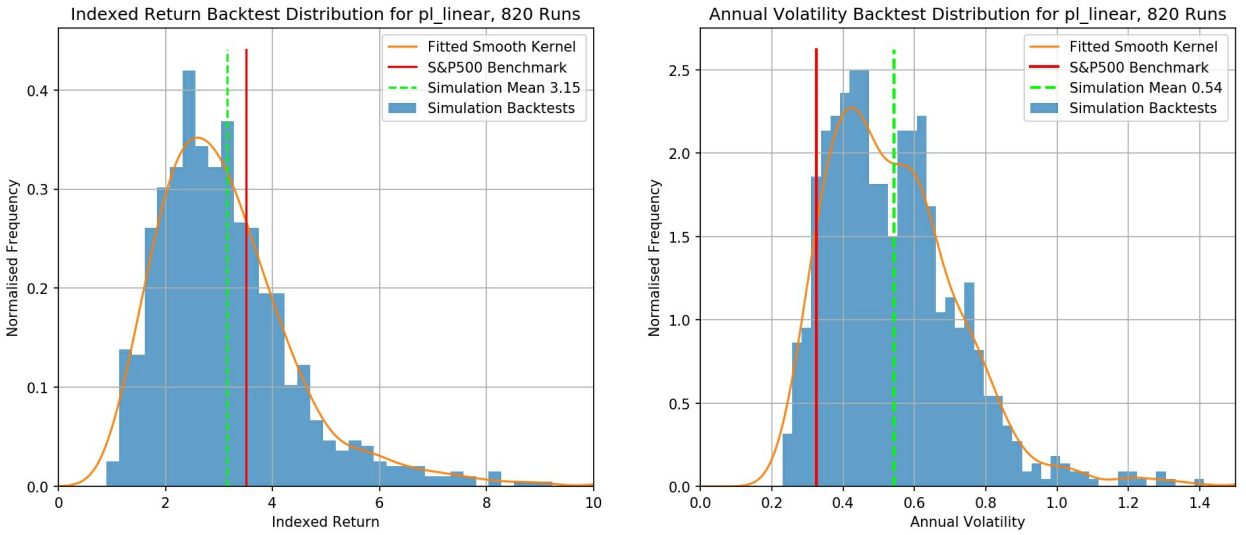
The S&P 500 benchmark we are comparing our investing AI to has an

indexed return of about 3.51 and a volatility of 0.325. Plotting these on our histograms gives a good idea of how well our Investing AIs are doing.

Let's plot out our simplest model (the Linear Regressor) and see if our promising results from earlier carry over to the backtest performance:

In[9]:

```
model_file = 'pl_linear'#'pl_linear', 'pl_ElasticNet', 'pl_rfregressor'
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plotBacktestDist(results_df, model_file, 'Indexed Return')
plt.subplot(1,2,2)
plotBacktestDist(results_df, model_file, 'Annual Volatility')
```



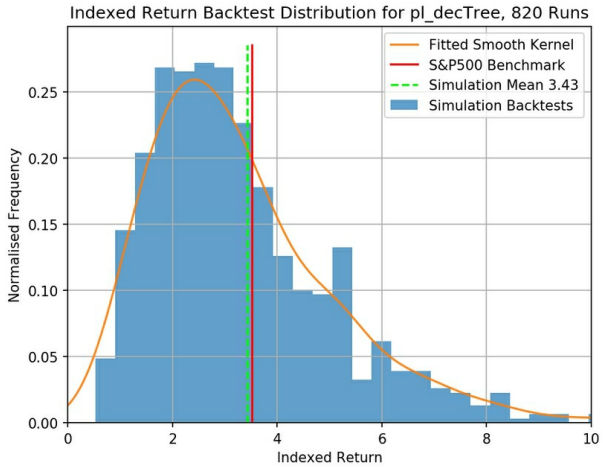
Our Linear Regressor based investing AI isn't particularly good. Not only is the backtest performance lower than the S&P500 index, but the linear regression selected portfolios were also a lot more volatile. In fact, almost all the backtest portfolios were much more volatile (see next figures), the whole distribution is well to the right of the S&P500 volatility.

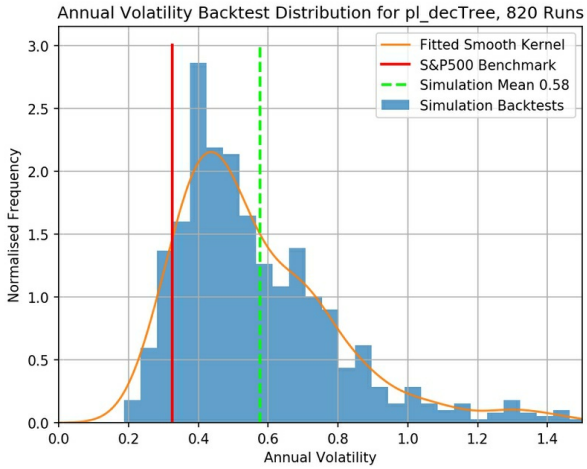
It's true that the AI was only trained on half of the company data, and that it could only select stocks from the other half with Altman Z scores outside of the danger zone. Even so, we would like to have high standards for our Investing AI.

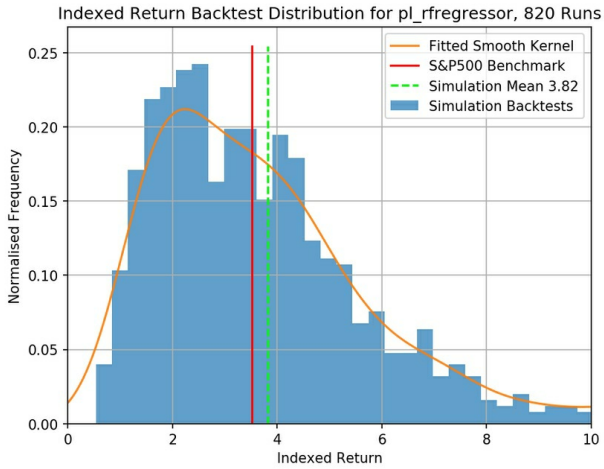
Let's plot all of our stock selection algorithms out and see if they work any better. Hopefully our hard work has paid off and at least one will be promising. This is the moment of truth!

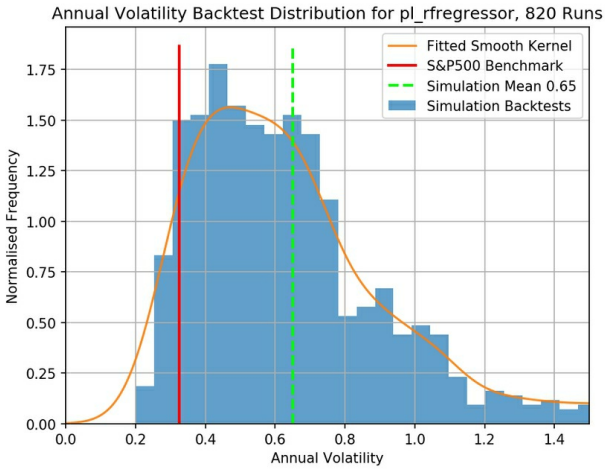
In[10]:

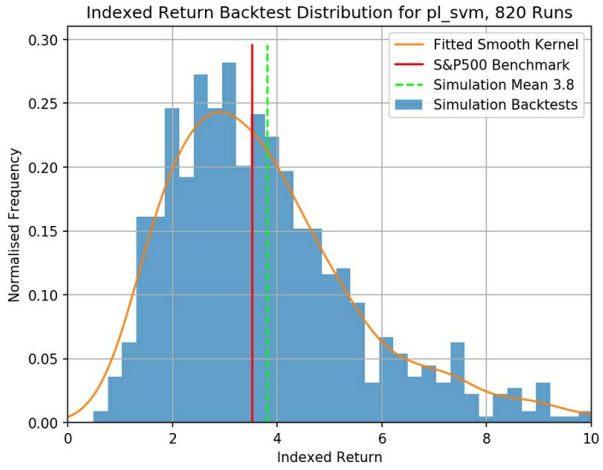
```
model_file = ['pl_decTree', 'pl_rfregressor', 'pl_svm', 'pl_KNeighbors']
numPlots = len(model_file)
plt.figure(figsize=(15,5*numPlots))
for I in range(0, numPlots):
    plt.subplot(numPlots,2,I*2+1)
    plotBacktestDist(results_df, model_file[I], 'Indexed Return')
    plt.subplot(numPlots,2,I*2+2)
    plotBacktestDist(results_df, model_file[I], 'Annual Volatility')
```

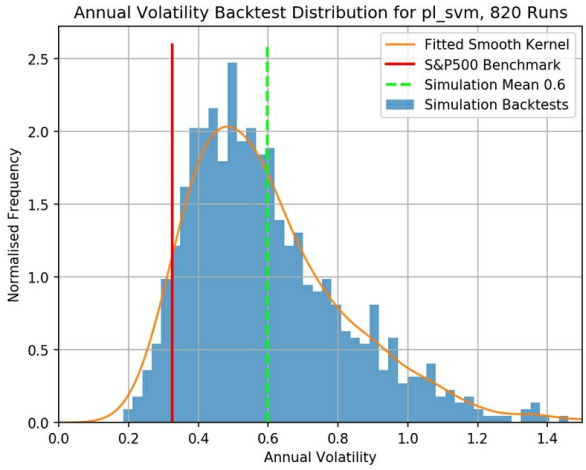






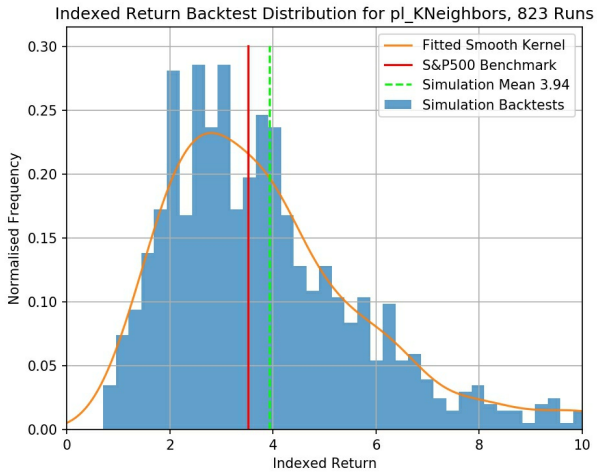


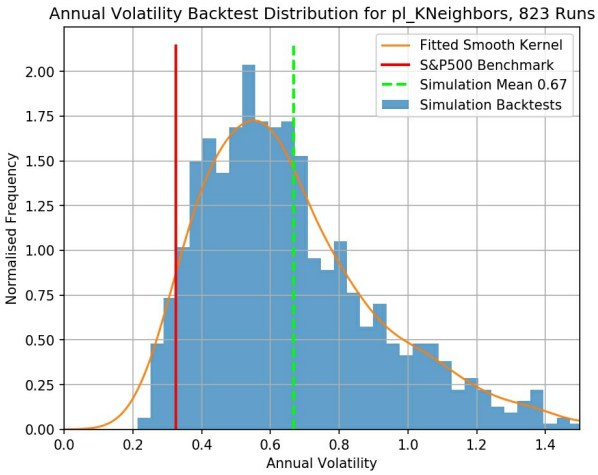


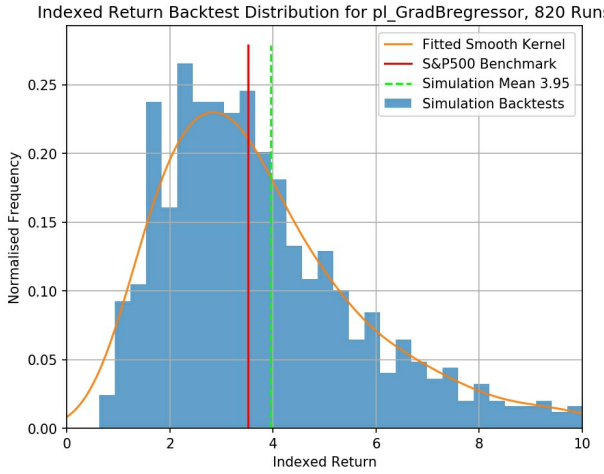


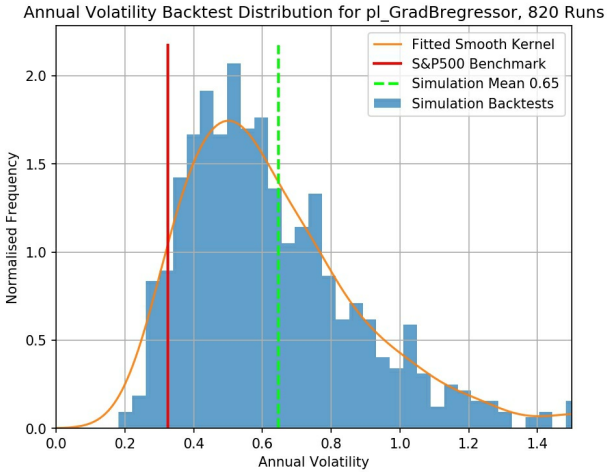
Some of our investing AIs start to outperform the S&P500 benchmark with training/testing data split half-half, but only just. We see that the Decision Tree regressor does just as badly as the Linear Regressor with similar volatility. The Random Forest does slightly better, though we are still at the almost the same performance of the S&P500 with greater volatility. Our

Support Vector Machine stock picker produced a similar return to the Random Forest.









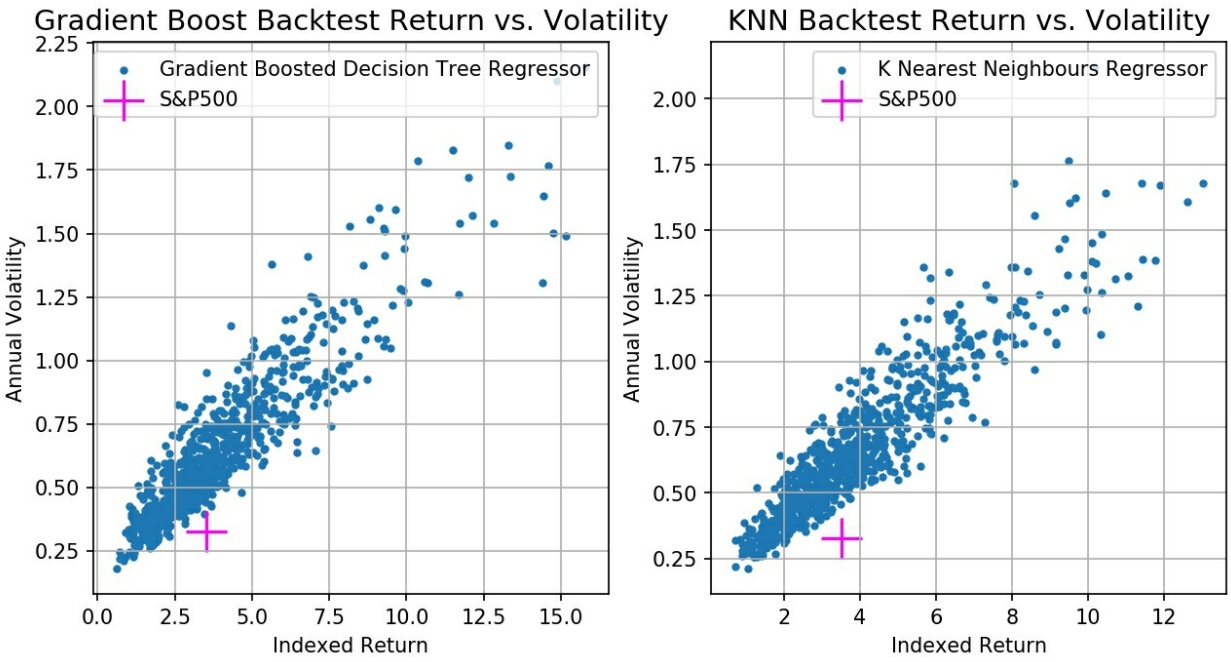
Thankfully our last two regressors produced the best returns, slightly above our benchmark, with the K-Nearest Neighbours regressor giving us an expected return of 3.94× and the Gradient Boosted Decision Tree regressor giving us a 3.95× return. Unfortunately, the K-Nearest Neighbour regressor also gives us the highest expected volatility.

It appears that in general the investing AI does well, but the volatility is much higher. If you are a seasoned investor the list of stocks your AIs recommends might be a good hunting ground, think of it as a more complex kind of machine-learned filter than a Graham number or Joel Greenblatt's Magic Formula.

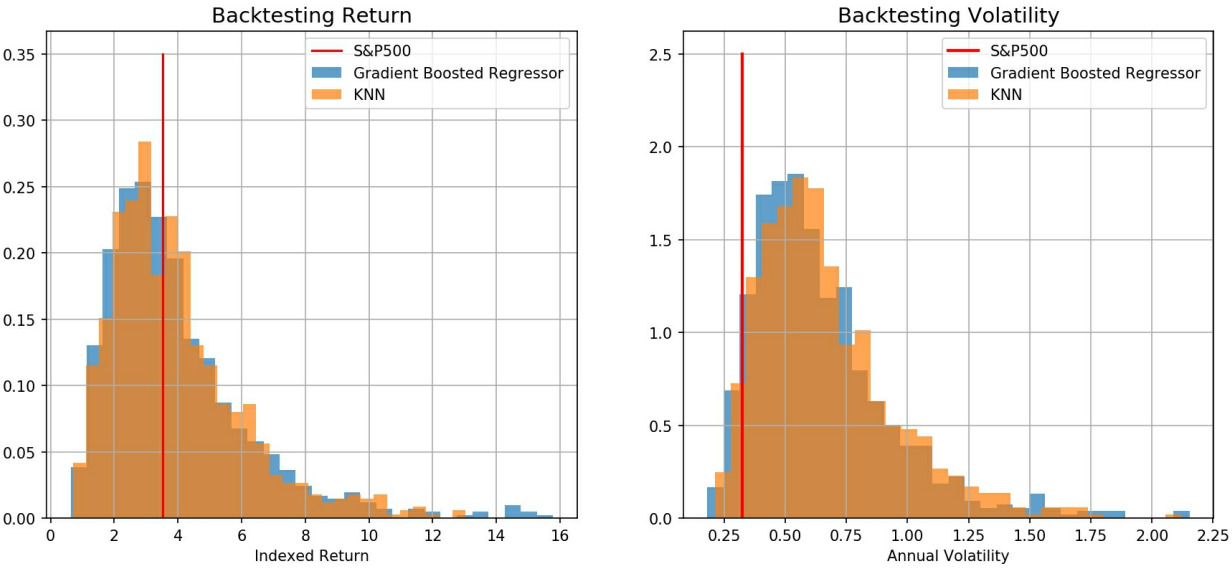
We can plot out the expected volatility vs. return from this data. When plotted, it appears that the expected return is almost linearly related to the volatility, and is quite bad from a “risk adjusted return” basis, in industry jargon:

In[11]:

```
# GRAPH OF RETURN VS VOLATILITY
plt.figure(figsize=(10,5))

plt.subplot(1,2,1)
returns = results_df[results_df['Model Used']=='pl_GradBregressor']['Indexed Return']
vols = results_df[results_df['Model Used']=='pl_GradBregressor']['Annual Volatility']
plt.scatter(returns, vols)
plt.scatter(2.340448, 0.309318)
plt.xlabel('Indexed Return')
plt.ylabel('Annual Volatility')
plt.grid()
plt.legend(['Gradient Boosted Decision Tree Regressor', 'S&P500'])
plt.title('Gradient Boost Backtest Return vs. Volatility', fontsize=14)

plt.subplot(1,2,2)
returns = results_df[results_df['Model Used']=='pl_KNeighbors']['Indexed Return']
vols = results_df[results_df['Model Used']=='pl_KNeighbors']['Annual Volatility']
plt.scatter(returns, vols)
plt.scatter(2.340448, 0.309318)
plt.xlabel('Indexed Return')
plt.ylabel('Annual Volatility')
plt.grid()
plt.legend(['K Nearest Neighbours Regressor', 'S&P500'])
plt.title('KNN Backtest Return vs. Volatility', fontsize=14)
```



Plotting the distributions of returns and volatility of our best regressors on top of each other, we see that the K-Nearest Neighbours regressor just inches ahead of the gradient boosted regressor by having a fatter tail to its distribution, however, the cost is quite a large increase in volatility:



The final regressor to use will depend on what trade-off is best between volatility and return is from what we have available, which is mostly a subjective trade-off. Our best two regressors as candidates for the investing AI are the K-Nearest Neighbours regressors and the Gradient Boosted

Decision Tree regressor-just like last year.

AI Performance Projection

the K-Nearest Neighbours regressors and the Gradient Boosted Decision Tree regressor returned 3.95× in our backtesting. Although this averaged return was above the S&P500, we still don't have an idea of how well these will perform when they can select from all the stocks available, whilst being trained with all of our historical stock market data.

The performance is likely to be higher than in our backtest, as having them train on one half and then select stocks from the other half limits the investing AIs in a big way. If Warren Buffett could only invest in stock stocks beginning A through to L, he probably wouldn't perform as well as he did.

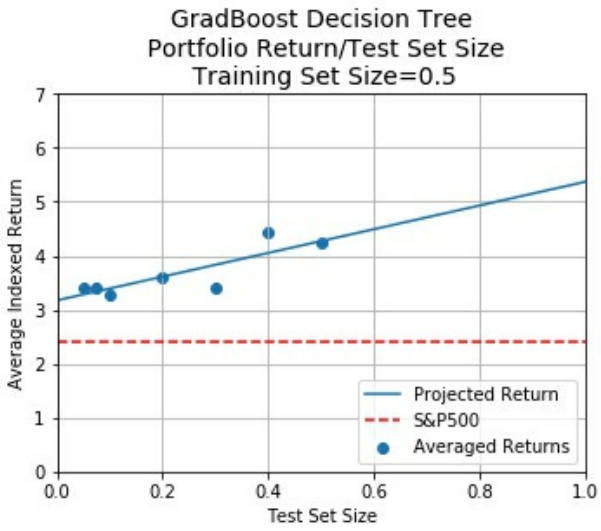
We can get some insight into the expected performance by finding the average performance we obtain with increasing testing set size (the size of the universe of stocks our AIs can choose from):

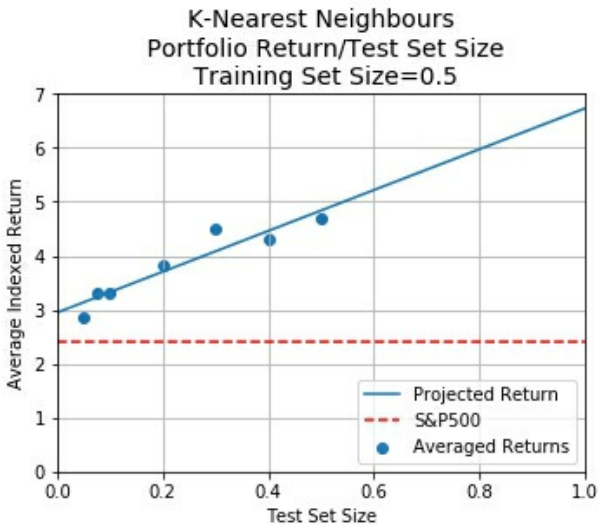
In[12]:

```
# Choose the model pipelines to run loop for
model_pipeline_list = ['pl_GradBregressor', 'pl_KNeighbors']
tests_per_model=20

# Reduce test size and check performance to try and see a trend
# Is effectively *0.5 as test size is 1/2 of total data.
test_size_modifier_list = [0.8, 0.6, 0.4, 0.2, 0.1]

# Results DataFrame
results_df2 = pd.DataFrame( columns=["Test Number",\
                                     "Model Used",\
                                     "Test Set Size",\
                                     "Indexed Return",\
                                     "Annual Volatility"])

i, results = 0, []
for model in model_pipeline_list:
    for test_size_mod in test_size_modifier_list:
        for test_num in range(0, tests_per_model):

            X_train, X_test, y_train, y_test = \
            train_test_split(X, y_pec, test_size=0.5)

            new_test_size = round(len(X_test)//(1/test_size_mod))

            X_test, y_test = \
```

```
            X_test[:new_test_size], y_test[:new_test_size]

            if (model =='pl_KNeighbors'):
                model_pl = trainKNeighborsModel(X_train, y_train)
            if (model =='pl_GradBregressor'):
                model_pl = traingbregressorModel(X_train, y_train)
            y_withData_Test=y_withData.loc[X_test.index]

            # Here is our backtesting code
            test = getPortTimeSeries(y_withData_Test, X_test,
                       daily_stock_prices, model_pl,
                           verbose=False)

            perf = test['Indexed Performance'][-1]

            vol = test['Indexed Performance'].diff().std()*np.sqrt(52)

            #print('Performed test ',i ,
                    [i, model, new_test_size, perf, vol])

            results_df2=results_df2.append(\
                        pd.DataFrame(\
                             [[i, model, new_test_size, perf, vol]],\
                                    columns=results_df2.keys()))
            i=i+1

        print('For {} test size {}, Performance: {}, Vol: {}.'.format(\
                model, new_test_size, \
            results_df2['Indexed Return'].tail(tests_per_model).mean(),\
            results_df2['Annual Volatility'].tail(tests_per_model).mean()))

# Save our results for plotting
results_df2.to_csv("Backtest_statistics_Modified_test_size.csv")
```

Plotting the return results with increasing test set size shows us that the average backtesting portfolio performance increases with increasing test set size:





If we extrapolate these plots we can get an idea of what the performance should be, however the extrapolating line in these plots can't be used with certainty. On the one hand the performance might be higher than projected because the training set size is fixed at 0.5 for these plots, on the other hand we do have survivorship bias to contend with. The plots points don't appear to go up linearly like our best fit line, to be safe we may assume that the average indexed return will be about 5, which is well below where the linear fit lines crosses the end of the graph.

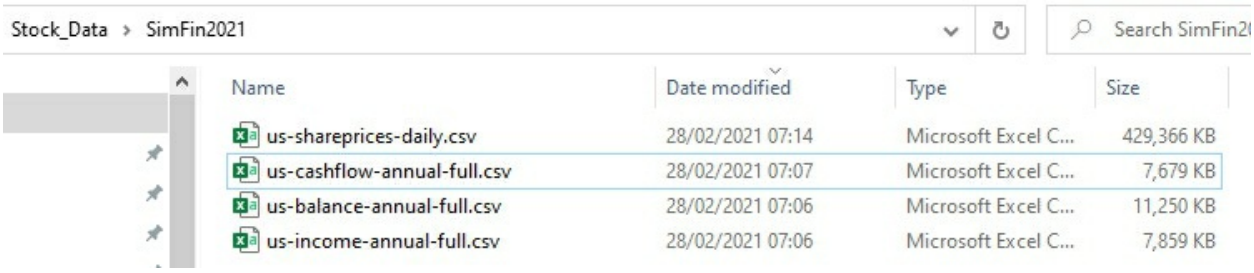
The final variable we need to consider is the cut-off Z-score. Changing this value to be above 3 (inside the "safe zone") barely changed the returns, though we would need a very large number of tests to discern the difference in performance.

If the projection can be depended on, the highest return we are getting per year is 5^(1/10), which is about 17% per annum. Let's see what stocks our investing AIs pick for 2021.

Chapter 7 - AI Investor Stock Picks 2021

Getting the Data Together

Now we know what regressors our investing AI will use. Let's put them to the test and pick a portfolio for us. At the time of writing (Early 2021) the 10-K report data for 2020 was available, these are generally available around February to March every year. To obtain the latest information in the same format as the training data, you will need to purchase a SimFin+ account from *SimFin.com* (it's about $20 at time of writing), the bulk download is done from the same place as for the free data we saw earlier in this book. The code this time uses the "Full" details file from SimFin+.



The fundamental stock data will need to go through similar cleaning and feature selection steps that we performed for the backtesting data, though you will need to change the file address in the code to look for the 2021 report data.

The code for this final chapter is at the end of the respective notebooks which contain the corresponding data processing steps which we used for our data so far. These are the AI Investor Notebook files 1, 2 and 5.

Looking back at the notebook from Chapter 4, (Jupyter Notebook 1), a simple version of the function *getYPricesReportDateAndTargetDate* is desired as we only want the stock ticker, publication data etc. for a single point in time, so we write a stripped-down function: *getYPricesReportDate*.

In[1]:

```
#Get the stock prices for X matrix to create Market Cap. column later.
def getYPricesReportDate(X, d, modifier=365):
    i=0

    # Preallocation list of list of 2
```

```
    y = [[None]*8 for i in range(len(X))]

    whichDateCol='Publish Date'
    # or 'Report Date', is the performance date from->to.
    # Want this to be publish date.
    # Because of time lag between report date
    #(which can't be actioned on) and publish date
    # (data we can trade with)

    for index in range(len(X)):
        y[i]=getYPriceDataNearDate(\
                    X['Ticker'].iloc[index],\
                   X[whichDateCol].iloc[index], 0, d)
        i=i+1
    return y
```

We will combine the Income Statement, Balance Sheet and Cash Flow Statement data into a single DataFrame with the *getXDataMerged* function as before. This time the file from SimFin+ will have a filename "-full" suffix added, e.g. "us-cashflow-annual-full.csv". The full version has a lot more data.

In[2]:

```
X = getFullXDataMerged()
```

Out[2]:

```
Income Statement CSV data is(rows, columns):  (19827, 69)
Balance Sheet CSV data is:  (19827, 96)
Cash Flow CSV data is:  (19827, 63)
Merged X data matrix shape is:  (19827, 216)
```

Now we need to take out the data that we want, data from the 10-K reports released early in 2021.

In[3]:

```
# Get data only for 2021
PublishDateStart = "2021-01-01"
PublishDateEnd = "2021-03-01"
bool_list = X['Publish Date'].between(\
                pd.to_datetime(PublishDateStart),\
                pd.to_datetime(PublishDateEnd) )
X=X[bool_list]
```

You can view the *X* DataFrame in a cell for a sense check, it should be much smaller and only contain recent financial data (about 1000 companies to select from). We can now use our *getYPricesReportDate* function to get our *y* data

In[4]:

```
y = getYPricesReportDate(X, d)
y = pd.DataFrame(y, columns=['Ticker', 'Open Price', 'Date', 'Volume'])
y
```

Out[4]:

	Ticker	Open Price	Date	Volume
0	AA	16.01	2020-02-21	8.850677e+07
1	AAL	28.84	2020-02-19	2.422856e+08
2	AAMC	22.30	2020-02-28	1.181677e+05
3	AAN	49.14	2020-02-20	2.520094e+08
4	AAOI	8.95	2020-02-28	1.327681e+07
...	...	...	...	...
1216	ZUMZ	13.13	2020-03-16	1.293007e+07
1217	ZVO	1.94	2020-02-20	1.750210e+05
1218	ZYNE	3.61	2020-03-10	2.898415e+06
1219	ZYXI	12.38	2020-02-27	4.285139e+06
1220	low	65.22	2020-03-23	6.254391e+08

We need to eliminate strange stocks that are in our list, like ones with no volume or no shares outstanding (somehow) as before. We would also like to create the column in our *X* DataFrame for market capitalisation:

In[5]:

```
y=y.reset_index(drop=True)
X=X.reset_index(drop=True)

# Issue where no share price
bool_list1 = ~y["Open Price"].isnull()
# Issue where there is low/no volume
bool_list2 = ~(y['Volume']<1e4)

y=y[bool_list1 & bool_list2]
X=X[bool_list1 & bool_list2]

# Issues where no listed number of shares
bool_list4 = ~X["Shares (Diluted)_x"].isnull()
y=y[bool_list4]
X=X[bool_list4]

y=y.reset_index(drop=True)
X=X.reset_index(drop=True)

X["Market Cap"] = y["Open Price"]*X["Shares (Diluted)_x"]
```

You can check the *X* and *y* DataFrames to see that the list has been trimmed down and should contain about a thousand stocks for 2021. Let's save this information ready for further cleaning and feature engineering in Jupyter Notebook 2:

In[6]:

```
X.to_csv("Annual_Stock_Price_Fundamentals_Filtered_2021.csv")
y.to_csv("Tickers_Dates_2021.csv")
```

The code in Jupyter Notebook 2 is perfectly usable for this step with no modification, just run the same steps to filter down the *X* DataFrame and save the .csv file to get all our accounting ratios for the stocks reporting financials at the beginning of this year.

In[7]:

```
X=pd.read_csv("Annual_Stock_Price_Fundamentals_Filtered_2021.csv",
              index_col=0)

# Net Income fix, checked annual reports.
X['Net Income'] = X['Net Income_x']

fixNansInX(X)
addColsToX(X)
X=getXRatios(X)
fixXRatios(X)
X.to_csv("Annual_Stock_Price_Fundamentals_Ratios_2021.csv")
```

That's all that is needed data-wise, now we just need to train the AI and make it pick stocks for us in Jupyter Notebook 5.

Making the AI Investor Pick Stocks

Our conclusion to Chapter 6 is that the Random Forest and K-nearest neighbour regressors were the best predictors for our AI to use. It is a simple matter to get our AI picking stocks.

As we aren't doing traditional data science, there is no need for a train/test split here. Because of this, we can let the regressors be trained on all data before 2020, before making the predictions on our 2021 financial data we just obtained. First, let's load in our historical and 2021 data:

In[1]:

```
# Training X and Y of all previous year data
X=pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios.csv",
              index_col=0)
```

```
# annual stock performances
yperf=pd.read_csv("Annual_Stock_Price_Performance_Percentage.csv",
                  index_col=0)

yperf=yperf["Perf"]

# Stock selection ratios for 2021 X
X_2021=pd.read_csv("Annual_Stock_Price_Fundamentals_Ratios_2021.csv"  ,
index_col=0)

# And the row tickers
tickers=pd.read_csv("Tickers_Dates_2021.csv", index_col=0)
```

Load the model pipeline and train it will all the historical data. You can choose which regressor to use, remember the KNN regressor had very high volatility but produced the highest backtest return of the two. Now we can make performance predictions with our *X_2021* DataFrame:

```
#Gradient Boosted tree
model_pl = traingbregressorModel(X, yperf)
y_pred=model_pl.predict(X_2021)
y_pred=pd.DataFrame(y_pred)

#KNN
model_pl = trainKNeighborsModel(X, yperf)
y_pred=model_pl.predict(X_2021)
y_pred=pd.DataFrame(y_pred)
```

Now that we have our predictions, we will extract out the stocks with bad Z scores and create a final DataFrame table sorted by the predicted performance. Remember we invest in the top 7 stocks.

In[2]:

```
# FINAL STOCK PICKS
# Separate out stocks with low Z scores
# 3.75 is approx. B- rating
z = calcZScores(X_2021)
zbl = (z['Z score'].reset_index(drop=True) > 2)

Final_Predictions = pd.DataFrame()
Final_Predictions[['Ticker','Report Date']] = \
tickers[['Ticker','Date']].reset_index(drop=True)\
                          [zbl].reset_index(drop=True)

Final_Predictions['Perf. Score'] = y_pred.reset_index(drop=True)\
                                   [zbl].reset_index(drop=True)

return Final_Predictions.sort_values(by='Perf. Score',
                                     ascending=False)\
```

```
.reset_index(drop=True).head(20)
```

Let’s put the code in this section into a function, called *pickStocksForMe*. Now we can ask our investing AI what stocks to invest in:

In[3]:

```
pickStockForMe()
```

AI Investor
Final Stock Picks

Gradient Boosted Decision Tree

	Ticker	Report Date	Perf. Score
0	NWL	2021-02-19	3.563221
1	GLYC	2021-03-02	1.225311
2	KCAC	2021-02-23	1.057181
3	MAR	2021-02-18	0.983623
4	TRUP	2021-02-12	0.887129
5	BKD	2021-02-25	0.845757
6	AFI	2021-03-01	0.826474
7	CMLS	2021-02-23	0.825346
8	AWRE	2021-02-17	0.713920
9	YELP	2021-02-26	0.707375
10	PI	2021-02-17	0.685698
11	MGNX	2021-02-25	0.684336
12	BGNE	2021-02-25	0.677278
13	FET	2021-03-02	0.652172
14	TNDM	2021-02-24	0.637915
15	TWTR	2021-02-17	0.616184
16	RDFN	2021-02-24	0.610589
17	RGNX	2021-03-01	0.574157
18	CLRB	2021-03-02	0.572009
19	RFP	2021-03-01	0.523544

K-Nearest Neighbours

	Ticker	Report Date	Perf. Score
0	TBIO	2021-03-01	0.973940
1	AXL	2021-02-12	0.594791
2	SBGI	2021-03-01	0.571145
3	YELP	2021-02-26	0.467473
4	RCUS	2021-02-25	0.460436
5	EB	2021-03-01	0.447316
6	COMM	2021-02-17	0.422411
7	CMLS	2021-02-23	0.414910
8	CTMX	2021-02-24	0.412434
9	BGNE	2021-02-25	0.410689
10	FET	2021-03-02	0.403149
11	WDAY	2021-03-02	0.392513
12	PI	2021-02-17	0.392427
13	IONS	2021-02-24	0.391844
14	BL	2021-02-25	0.363219
15	CCXI	2021-03-01	0.362704
16	NVTA	2021-02-26	0.362503
17	ALRM	2021-02-25	0.361941
18	CAR	2021-02-17	0.359378
19	FSLY	2021-03-01	0.349854

These are our final picks! (with runners up). Though we should make sure there are no strange companies that have wondered into our list, or any financials that should have been removed by our data provider.

Unfortunately we have KCAC (Kensington Capital Acquisition Corp.) stock

in the first list, which is a special acquisition corporation (A.K.A. a SPAC). A SPAC is rather different from an ordinary company providing goods or services, which is what we are looking to select. SPACs have no commercial operations, so we know that its operation doesn't match the underlying reasoning of our AI, so we will ignore this company from the final pick.

Interestingly, the Gradient Boosting and K-Nearest Neighbour regressors seem to want to pick different kinds of companies. Last year, the KNN regressor was drawn to pharma stocks, this year 2/7 of the top stock picks are pharma stocks, compared with 1/7 for the gradient boosting regressor, so this trend still seems present.

Our Gradient Boosted Decision Tree seems to be acting like a modern value investor, with picks just as boring as last year. In our list we have CMLS (Cumulus Media Inc.) is a radio company, AFI (Armstrong Flooring Inc.) which does flooring, a pet insurance company and a hotel chain are present, and let's not speak about BKD (Brookdale Senior Living Inc.). These companies aren't exactly as cool as a TSLA or AMZN.

Final Results, Stock Selection for 2021

If you have not done much coding before, congratulations! You have come all this way and built your own investing AI which is likely to beat the market over time. You have also gained knowledge on how to use Python for any other Machine Learning problems you might face too, certainly enough to tune and tweak your investing AI to make it personal to you.

You could make it invest as conservatively as you want by changing the minimum Altman Z ratio and the number of stocks your AI holds, correlate performance with other company ratios or whatever else your imagination can come up with to make your code suit your needs.

Bear in mind we can expect the portfolio to fluctuate more than the S&P500, and that in some years it will underperform. We should also remember that in backtesting these models were hamstrung because the universe of stocks they could choose from is significantly reduced due to training/testing data separation.

This book was written in March 2021. Given the annual report data for end 2020 being available by the great data provider *SimFin.com*, this is a good

opportunity to unleash the investing AI. Let’s use the Gradient Boosted Decision Tree as the final investing AI, as the K-Nearest Neighbours regressor appears to expose us to a lot more volatility for little extra gain.

The select few stocks our final AI chose for 2021, excluding the special acquisition company, made with the code you have seen in this book are:

NWL

GLYC

MAR

TRUP

BKD

AFI

CMLS

The AI model portfolio is shown on the website, *www.valueinvestingai.com* with its performance since March 2020 displayed, if you are curious to see how the AI portfolio performs.

The stock selections are recorded on the Bitcoin blockchain, to be sure that the performance is genuine, even if it performs poorly (which it might!). Sadly the 2020 portfolio was only recorded there from Jan 2021, though new stock picks will be put on the blockchain every March going forward.

The transaction messages can be seen on any website that provide Bitcoin cryptocurrency transaction data, for example *blockchain.com* or *blockchair.com*. A direct link is provided on the website *valueinvestingai.com*, alternatively, if you are technically inclined, the Bitcoin transaction hash where you may see the 2020 stock selection is:

4234bd073d7c261b490789ddb59bf2adc3192b89cfc22ca84ed04d353cf69396

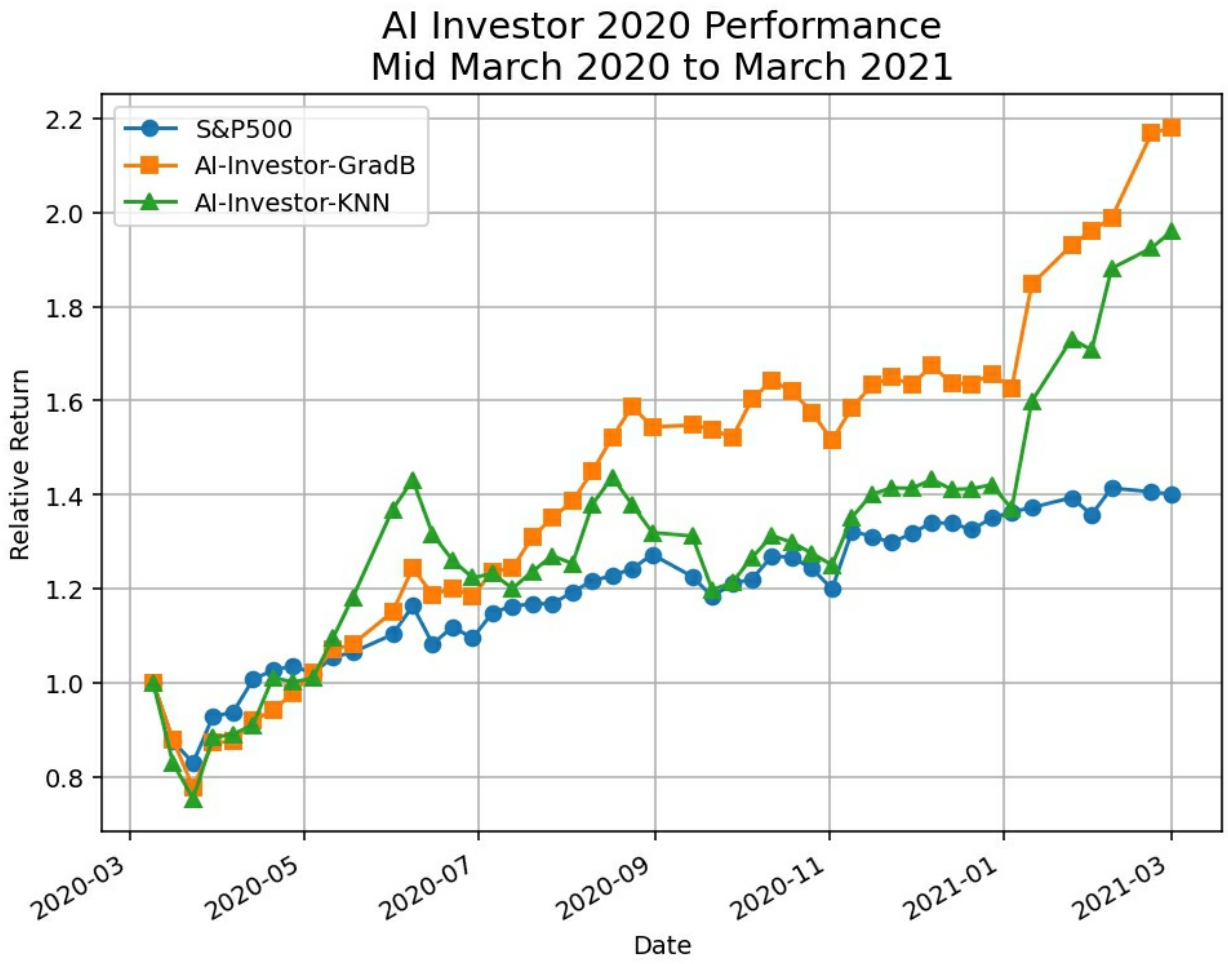
And the Bitcoin transaction hash for 2021 stock selection is:

e01ce40222389e3af2a8356961f096033f2568d72202097f4aa779fb0ea60589

Messages were placed on the blockchain using *btcmessage.com*.

Discussion of 2020 Performance

From March 2020 to March 2021 the AI Investor stock selection and portfolio performance is shown below (With the alternative K-Nearest Neighbours model performance too):



AI Investor 2020 Performance (March 2020 – March 2021)		
	Relative Return	Weekly Volatility
AI Investor (GradBoostRegressor)	2.16	0.45
AI Investor (KNNRegressor)	1.92	0.57
S&P500	1.4	0.25

It is an unfortunate co-incidence that the AI Investor started in March 2020, as this was during a market crash, so the performance measure will vary widely depending on exactly which day it was measured from. It is also

important to temper our future expectations given the supposed high returns:

THE AI INVESTOR WAS LUCKY IN 2020, THIS KIND OF RETURN WAS NOT EXPECTED AND IS UNLIKELY TO CONTINUE.

Furthermore, on a “risk-adjusted return” basis, the performance isn’t very good anyway.

The performance in 2020 was in some ways in line with what we expected from our backtests, in general there was much higher volatility, with the K-NN regressor selecting stocks that gave us a higher volatility than the Gradient Boosted Decision Tree. We expected the KNN regressor to produce a higher return than the Gradient Boosted Tree, though this did not materialise.

Readers who have followed through this far can easily calculate performance metrics used in the industry like Sharpe ratios, Sortino ratios etc. with a bit of Python, however, as this is just stock selection of 7 stocks, and the strategy holds them for one entire year, it probably isn’t worth measuring performance that way.

As the stock selection occurs every March, when the annual reports of corporate America become available, the stocks were selected right when the Covid-19 pandemic was spooking the market. This might be partly responsible for the outperformance, as out of favour stocks tend to fall further in a market crash, and if they recover to reasonable valuations afterwards, they will be rising from a lower level.

Individually, the performance of the 2020 chosen stocks is worth looking at:

	Relative Return
JBHT	1.74
EPZM	0.49
NOW	1.91
LNT	0.87
RFP	4.39
GPRE	4.48
MA	1.38
Portfolio	2.18

Not all of the picks performed well, one even lost more than 50% of its value. The portfolio performance was dragged upwards with two stocks in particular, Resolute Forest Products Inc. (RFP) and Green Plains Inc. (GPRE). The first company is a wood, pulp and paper company, the second is primarily an Ethanol producer.

One might say these are (literally) run-of-the-mill companies, yet these kinds of stocks can give outsize returns if purchased at a low enough price relative to the perceived business quality in the future, especially when purchased during a market panic (March 2020).

There will be down years for the AI Investor, our backtests show an extremely high probability of this, and with stock selection purely based on the financial numbers only, we run the risk of buying things we don't understand on even a basic level, for example KCAC stock, which a mindless follower might have purchased, which is a special purpose acquisition company that has no commercial operations.

Another risk is that accounting changes could undermine the AI entirely. For example, in 2017 Berkshire Hathaway had to account for the change in value of their securities as profit. If too much of this kind of accounting change occurs, the underlying reasoning of our model falls apart and we will no longer be able to rely on it, furthermore, as with all investing strategies, this strategy will not work if too many people follow it (assuming of course that this book is correct, and that the model does deliver some return beyond

random selection).

From a “risk-adjusted return” perspective, this strategy performs badly.