AI STOCK PICKER

A Project Report Submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

K UDAY KIRAN (190330245)

N PRANATHI (190330158)

A SHIVANI(190330018)

E SAI SRIKAR(190330064)

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
K L DEEMED TO BE UNIVERSITY
AZIZNAGAR, MOINABAD , HYDERABAD-500 075

MARCH 2023

BONAFIDE CERTIFICATE

This is to certify that the project titled **AI STOCK PICKER** is a bonafide record of the work done by

K UDAY KIRAN (190330245)

N PRANATHI (190330158)

A SHIVANI(190330018)

E SAI SRIKAR(190330064)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **COMPUTER SCIENCE AND ENGINEERING** of the **K L DEEMED TO BE UNIVERSITY, AZIZNAGAR, MOINABAD , HYDERABAD-500 075**, during the year 2022-2023.

DR K SRINIVAS, DR SUMIT HAZRA	**DR BABU RAO K**
Project Guides	Head of the Department

Project Viva-voce held on _____

Internal Examiner	**External Examiner**

ABSTRACT

While traditional investment in stocks is a laborious, arduous, confusing, demanding, and time-consuming task, we wish to optimize, fasten, and smoothen the process. With this project, we aim to create an investing Artificial Intelligence which could pick stocks for the user. The stocks for the user are picked based on his/her financial power. We plan to use core Python, a plethora of machine learning algorithms, and libraries. This is a data-driven approach to the world of stock markets. The consumer gets to choose how aggressive or defensive one wants to be when investing. The end user needs to worry least about losses while celebrating assured profits with each investment and trade. We choose to devise value investing which has been proven to work over time, notably by prominent investors like Warren Buffett and Charlie Munger.

ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

DR K SRINIVAS, DR SUMIT HAZRA, our project guides, for helping us and guiding us in the course of this project .

DR BABU RAO K, the Head of the Department, Department of DEPARTMENT NAME.

Our internal reviewers, **DR K SRINIVAS** , **DR SUMIT HAZRA** , **DR SUMIT HAZRA** for their insight and advice provided during the review sessions.

We would also like to thank our parents and friends for their constant support.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Background of the Project

“An investment in knowledge pays the best interest”. -Benjamin Franklin

What are stocks and why do they have value? Why do they keep changing in value? 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 one can directly run the place, one cannot enter the office and tell people to do things if one has 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 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 we get to vote on who goes on the board and sometimes vote on a meaningful company event like a big acquisition. Our vote is proportional to the number of shares we 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 we want to increase our stake in the company, we 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.

Now, 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 does not change all that much over time. Reasonable people would say that the value of a company is not going to change much from one month to the next. Are people reasonable all the time? Mostly no. Things happen. When the world is pessimistic people might say the sky is falling. Consider May 2020, when there were mass riots across USA, a worldwide pandemic, a trade war between superpowers, and so on. Stock prices were at a real low. The Ie Is possible too: the ”dot ’om’ bubble, which burst 23 years ago. Look at the wild swings in the NASDAQ index around that time:

Figure 1.1: NASDAQ wild swings in 2000

1.2 Problem Statement

• While traditional investment in stocks is a laborious, arduous, confusing, demanding, and time-consuming task, we wish to optimize, fasten, and smoothen the process.

• With this project we aim to create an investing Artificial Intelligence which could pick stocks for the user.

• The stocks for the user are picked based on his/her financial power.

• We plan to use core Python, a plethora of machine learning algorithms, and libraries.

• This is a data-driven approach to the world of stock markets. • The consumer gets to choose how aggressive or defensive one wants to be when investing.

• The end user need to worry least about losses while celebrating assured profits with each investment and trade.

• We choose to devise value investing which has been proven to work overtime, notably by prominent investors like Warren Buffett and Charlie Munger.

1.3 Objectives

1. Obtain Financial Data and Stock Performance Data of the previous years. This is the historical data.

2. 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'.

3. 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.

4. If the models are found to have predictive power, we will use the models, we will use models to create value-investing AIs.

5. Next, we backtest the investing methods with our historical data to identify the best-performing algorithms.

6. Pick the winner and tweak it further for performance improvements.

1.4 Scope of the Project

Resources:

• Coding team (2 people), 15 hours of work a week for 3 months

• Engineering manager (1 person), 10 hours of work a week for 3 months

• Documenting testing (1 person), 10 hours of work a week for 3 months

Deliverables:1

• Understanding the problem and empathizing in 1st half of December 2022

- Defining the problem in 2nd half of December 2022
- Ideating and experimenting in 1st half of January 2023
- Prototyping in 2nd half of January 2023
- Testing in February 2023

Chapter 2

Literature Review

The best AI investing software and apps use machine learning to determine the best investment picks for us. It does this by analyzing data from a variety of sources, including market movements, news events, economic cycles, and social media sentiment.
It uses algorithms to analyze more data than any human ever could. AI is also better at spotting patterns and correlations in data than humans can be.
This means that these programs can potentially spot trends and make decisions faster than humans can, which is an important advantage in today's fast-moving markets.

2.1 Overview of related works

- Trade Ideas
  - Scanning for stocks bases on technical analysis, volume, and price action.
  - Professional real-time data feeds to help make decisions.
  - Claims to use proven algorithms to make trades.

Figure 2.1: Trade Ideas

- Cons of Trade Ideas:
  - Essentially a scanning software alone
  - Cannot perform actual investments
- Trend Spider
  - This software engine is designed to work with automated drawing tools.
  - Algorithms generate Fibonacci retracements, resistance lines, and flags.
  - Makes use of previous time stocks.

Figure 2.2: Trend Spider

- Cons of Trend Spider:
  - Gives way too much technical data.
  - Cannot be easily understood by new investors.
- Black Box Stocks
  - Provides alerts for daily penny stocks and large-cap stocks.
  - Has a subscription service and is reviewed to be good.

Figure 2.3: Black Box Stocks

- Cons of Black Box Stocks:
  - Since it requires subscription, one has to pay for not just buying stocks, but to also use their application.
- Tickeron
  - Identifies the sectors and industries that are poised to perform well over the next few days.
  - Get ideas from other members of the application
  - Can analyze the views of fellow investors regarding a particular stock.

Figure 2.4: Tickeron

- Cons of Tickeron:
  - The user's profile, stock purchase patterns and insights are available to every other user.
  - Can lead to potential data breach and stock theft.
- Vector Vest
  - Gives investment recommendations.
  - Real-time stock quotes

Figure 2.5: Vector Vest

- Cons of Vector Vest:
  - Expensive subscription compared to other stock scanners.
  - Does not include real-time data.

2.2 Advantages and Limitations of Existing Systems

- The pros of AI Investing:
  - 1.Improved Customer Experience
    - Artificial intelligence can significantly improve the customer experience. Take chatbots, for example. Suppose you have questions or are having trouble with your investments. In that case, you can connect with a chatbot, which is often much quicker and more efficient than contacting human support.
    - AI can also take on many time-consuming tasks in investing and finance, saving everyone more time in a very time-sensitive industry.
  - 2.Fraud Detection
    - AI algorithms can be programmed to identify fraud and suspicious activity much faster than humans. Having reliable fraud detection will prevent financial losses and faulty dealings. Also, unlike humans, AI doesn't have to take breaks. Thus, it can monitor investments and economic activity to keep your money safe.
  - 3. Automated Decision Making
    - Saving time and streamlining investment decisions can be the difference between making money or missing out on a great deal.
    - AI investing algorithms constantly review data and determine the best-case scenario for our funds. When the AI comes across an opportunity, there's no waiting around. It will automate our investments based on programmed criteria determined by us, making it so that we don't have to worry about it.

- 4.Increased Accuracy
    - With AI investing, there's no risk of human error. When computers are correctly programmed, we don't have to stress the mistakes a team of humans could make. By compiling data with the help of intricately designed algorithms, there's more accuracy, fewer errors, and better precision.
- 5.Around-the-clock Service
    - Machines don't need to take vacations. When we are done for the day, AI investing applications continue working for us, preventing missed opportunities during our restful hours.
    - AI algorithms and even Robo-advisors can work around the clock, thus increasing productivity to levels standard human workflows can't achieve. When our productivity increases, our revenue often does, too.
- 6.Takes the emotions out of Investing
    - The investment process can be very emotional. As soon as the market dips, many investors panic, sell and regret it later. When investing, human emotions often get in the way and lead us to make poor decisions.
    - However, AI doesn't have human emotions. With AI investing, all our decisions are based on market conditions, predictive analyses, and the criteria set from the beginning to identify our goals. AI uses data to make decisions rather than subjective emotions. Ultimately, it means more effective, stable, and rewarding investments.
- The Cons of AI Investing:
    - 1.Poses a Risk to Human Jobs
        - Artificial intelligence doesn't require much — if any — human interference. Therefore, it could pose a risk to human employment. This is something that those who are hesitant or against AI worry about in the future. However, the current level of AI we often work

with isn't capable of much more than handling basic tasks. Still, future AI will likely be far more advanced, resulting in increased job losses.

* 2.Lacks and Emotions to Ethics
    - AI's lack of emotions is often seen as an advantage in preventing poor emotional judgment when handling investments. However, lacking emotions can become a disadvantage in some industries, like marketing, where human connections are vital to increasing profits.
    - Similarly, AI can't incorporate ethics into its decisions. Ethics and morality are essential parts of what makes a human and often drive a lot of our decision-making. AI, on the other hand, only makes decisions based on data. It can't consider our ethical stances, which can limit its abilities on a larger scale
* 3.Often Limited Flexibility
    - At the end of the day, AI is a machine with programmed capabilities. Many AI cannot pivot or be more flexible with their operations, which can sometimes hinder investors. For example, Robo-advisors have a specific set of functions they can carry out to help us manage our finances. However, beyond those set functionalities, they may not be as helpful as working with a human who can offer more adaptability.
* 4.Further Development is Expensive
    - While AI may be more affordable than human labor, the initial set-up is expensive. Further development can get pricey as technology evolves and our needs change. It's also not just the AI that's costly, though.
    - There's also the need for the latest hardware and software to support it, along with regular updates. It's a significant investment to

make at the start that will continue costing us money as we use it. Integrating AI investing is also a costly process for the developers who improve upon the tech for the future.

* 5.Lacks Creativity
    - Creativity is another human-specific feature that AI can't quite master yet. AI learns over time through data and experience. Still, it can't think outside the box and present creative approaches to problems. Many predict that AI will become more intelligent in the coming years as further developments are made. However, the tech is still lacking without the ability to think creatively.

Chapter 3

Proposed System

3.1 System Requirements

To be used efficiently, all computer software needs certain hardware components or other software resources to be present on a computer. These prerequisites are known as (computer) system requirements and are often used as a guideline as opposed to an absolute rule. Most software defines two sets of system requirements: minimum and recommended. With increasing demand for higher processing power and resources in newer versions of software, system requirements tend to increase over time. Industry analysts suggest that this trend plays a bigger part in driving upgrades to existing computer systems than technological advancements.

3.1.1 SOFTWARE SPECIFICATIONS

Software specification deal with defining software resources, requirements, and prerequisites that need to be installed on a computer to provide optimal functioning of an application.

These requirements or prerequisites are generally not included in the software installation package and need to be installed separately before the software is installed.

- Software Requirements:
  - Operating System : Windows 10
  - Platforms: Jupyter Notebook, Google Collab
  - Language : Python

3.1.2 HARDWARE SPECIFICATIONS

The most common set of requirements defined by any operating system or software application are the physical computer resources, also known as hardware. A hardware requirements list is often accomplished by a hardware compatibility list, especially in case of operating systems.

All computer operating systems are designed for a particular computer architecture. Most software applications are limited to operating systems running on particular architectures. Although architecture-independent operating systems and applications exist, most need to be recompiled to run on a new architecture.

The power of the central processing unit (CPU) is a fundamental system requirement for any software. Most software running on x86 architecture define processing power as the model land for the clock speed of the CPU. Many other features of a CPU that influence its speed and power, like bus speed, cache, and MIPS are often ignored. This definition of power is often erroneous, as AMDAthlon and Intel Pentium CPUs at similar clock speed often have different throughput speeds.

- RAM: 12 GB
- Processor : Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz
- Hard Disk : 1 TB
- Monitor: 15.6” color monitor
- Keyboard : 122 keys

3.2 Design of the System

We approached 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 cannot directly go about value investing as our machine learning tools do not 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 we have designed, we made our lives easier by setting the algorithms to predict performance over the course of a year, between annual reports.

The first goal, then, was 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 was 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.

Since our models found to have predictive power, we used the models to create value investing AIs, after which we backtested the investing methods with our historical data to identify the best performing algorithms. We then picked the winner and tweaked it further for performance improvements.

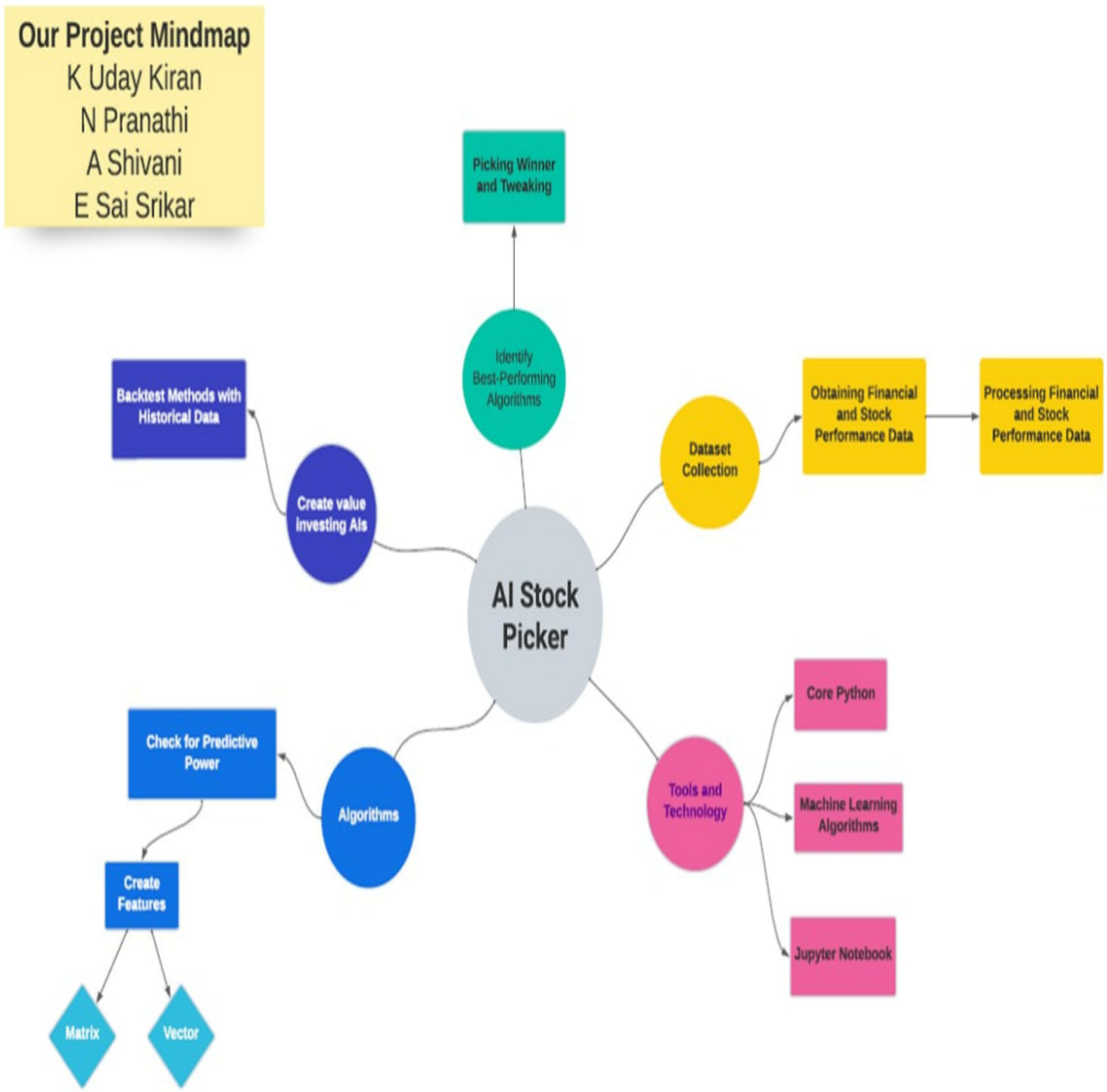


Figure 3.1: Design of The System - Our Mindmap

3.3 Algorithms and Techniques used

We explored several classification algorithms and regression algorithms for our project. Some algorithms, technical terms and concepts have been highlighted here:

3.3.1 Classification Algorithms

- Decision Trees
  - As the name suggests, decision trees are trees of decisions. The decisions to be made to classify our data are binary (yes/no, greater or less than) in the standard Scikit-learn library implementation.

- CART Algorithm
  - The measure being used in this algorithm is the Gini Impurity. The Gini Impurity is essentially the probability of mislabelling a random element in our set, given that we know the distribution of all classes in the set. Our classes usually being True and False.

3.3.2 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 and a training set. We train our models 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.

3.3.3 Measuring Success in Classification

There are many ways of evaluating performance in Machine Learning, for classification algorithms asking for something like mean squared error does not make sense, whilst it would make sense for regression algorithms.
If we have a table with horizontal columns being the actual outcomes (True, False) and the rows being predictive outcomes, recording the number of cases of True Positives, False Positives, True Negatives, False Negatives in the table, we would have a Confusion Matrix.

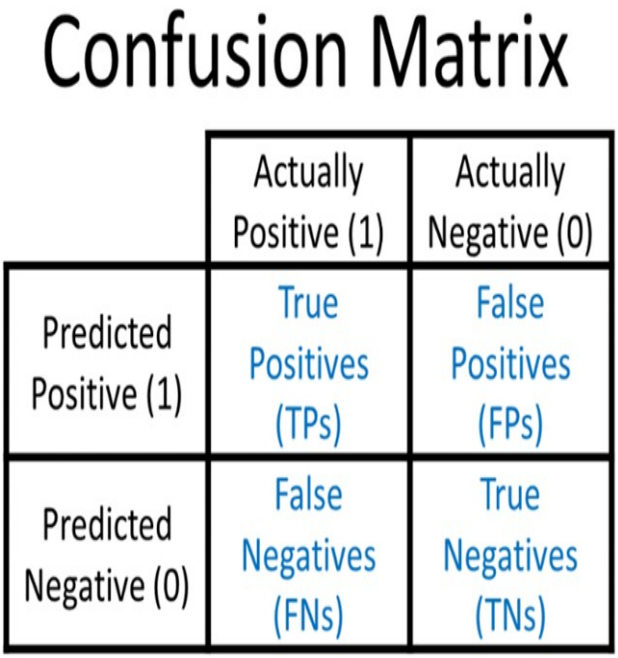


Figure 3.2: Confusion Matrix

3.3.4 Regression Algorithms

Instead of classifying things, these algorithms predict numbers, such as temperature, pressure, price of something, etc.

We used these in our project to predict stock performance. For the investing AI, we tried out a group of regressors and picked 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 it has learnt the broad rules underlying the data, it hopefully can make predictions:

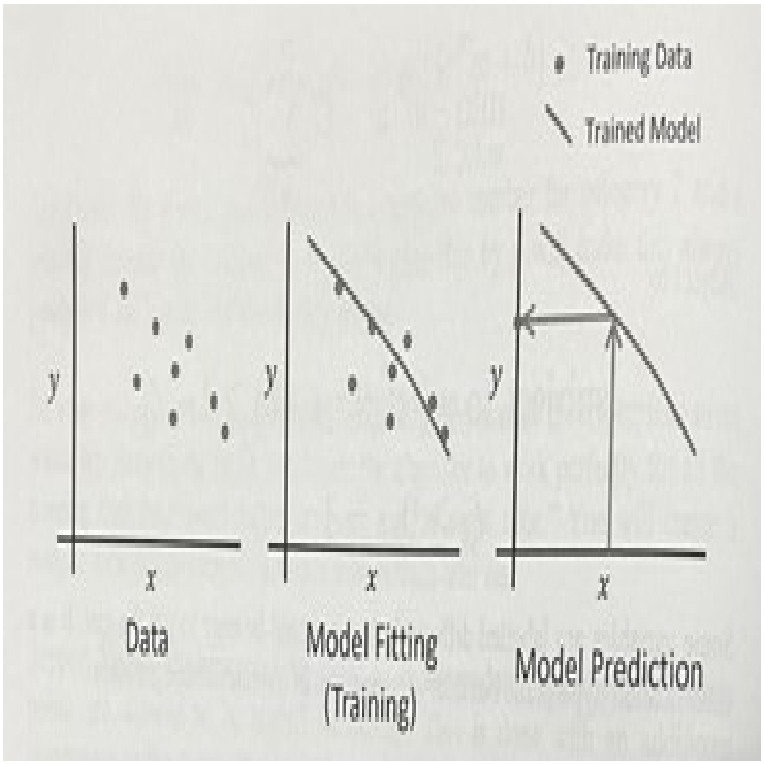


Figure 3.3: Regression

Of course, the above graphs look simple with one feature dimension, but it can be very powerful where there are a lot of features that we want our model to make a prediction from.

When a model is fitted to the data, just like with the 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 our model overfits and accounts too much for the variations in the data, it cannot 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 represen-

tative, 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.

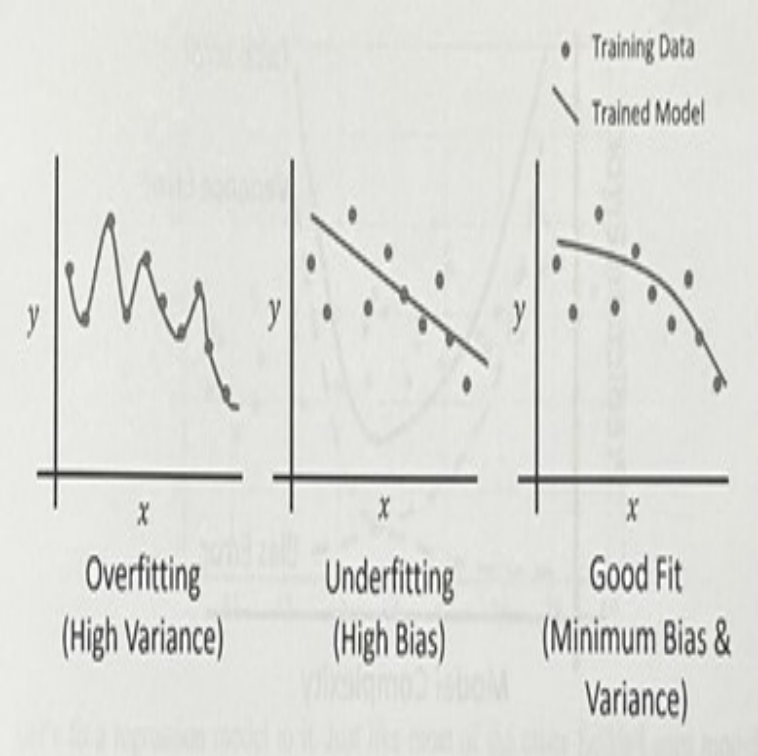


Figure 3.4: Curve Fitting

There is a sweet spot between bias and variance that will give the best prediction accuracy of our 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:

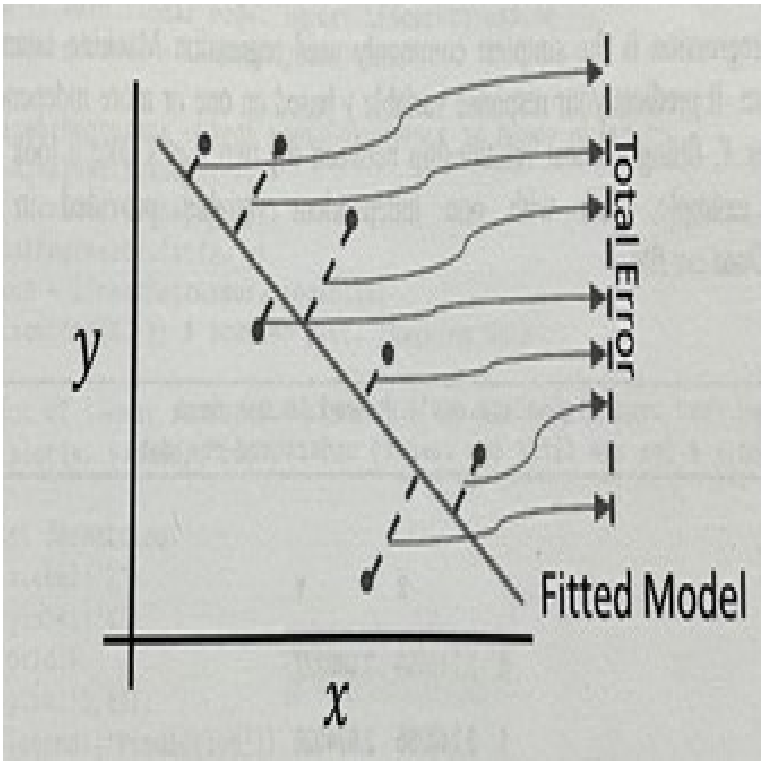


Figure 3.5: Fitted Model

The above diagram is not exactly what is going on in finding the error, but 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:

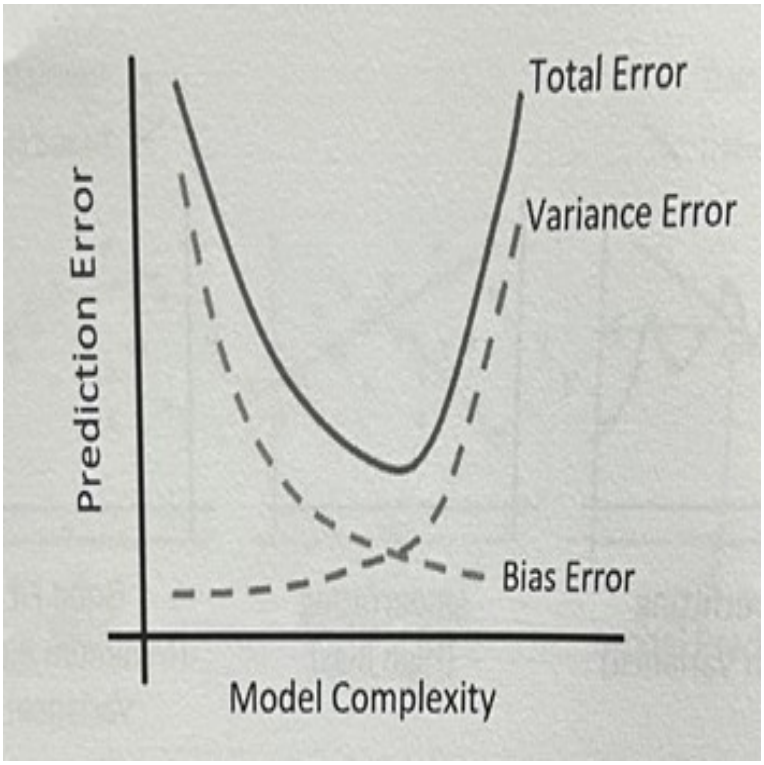


Figure 3.6: Curve Fitted

3.3.5 Linear Regression

It is the simplest and most 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.

3.3.6 Decision Tree Regression

Decision Trees can also be used for regression predictions. The DecisionTreeRegressor object from Scikit-Learn can create a regression Decisionj Tree for us out of the box.

3.3.7 Support Vector Machine Regression

Classification SVM 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, and we want to maximize the street width.

This problem can be flipped on its head and used as a regression algorithm by incentivizing 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:

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

Subject to:

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

Figure 3.7: SVM Math

3.3.8 K-Nearest Neighbour Regressor

The K-Nearest Neighbours algorithm is perhaps the simples ML 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.

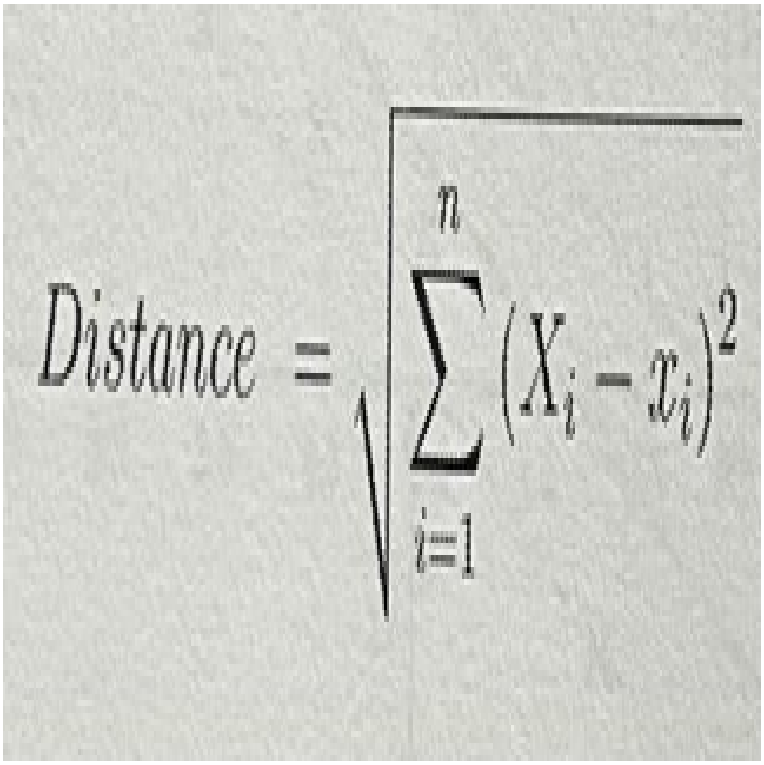


Figure 3.8: Distance Formula

3.3.9 Random Forest Regressor

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 is 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:

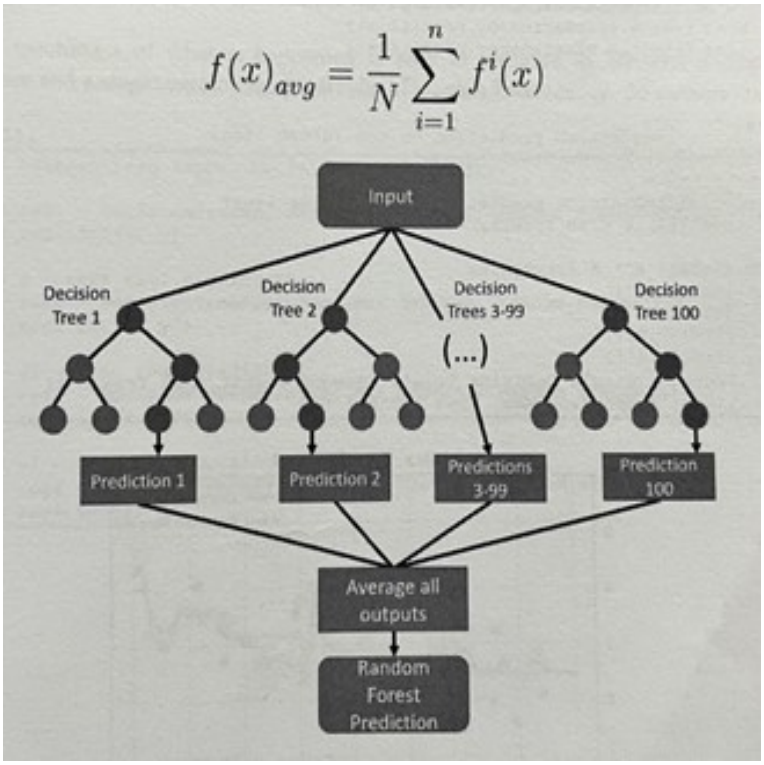


Figure 3.9: Random Forest

Chapter 4

Implementation

4.1 Tools and Technologies used

- Python
- Jupyter Notebook
- Machine Learning
- Python Libraries:
  - Numpy
  - Pandas
  - Keras
  - CV2
  - Matplotlib
  - Scipy
  - Scikit-learn

Figure 4.1: Libraries Used

4.1.1 Modules and their Descriptions

- Module 1 - Obtaining and Processing Fundamental Financial Stock Data
  - Getting Data and Initial Filtering
  - Feature Engineering
- Module 2 - Using Machine Learning to Predict Stock Performance
  - Using Machine Learning to Predict Stock Performance
  - Linear Regression
  - Elastic-Net Regression
  - K-Nearest Neighbours
  - Support Vector Machine Regressor
  - Decision Tree Regressor
  - Random Forest Regressor
- Module 3 - AI Backtesting
  - Using an Example Regressor

  - Building the First Bits
  - getPortTimeSries function
  - Looping Through Each Year of a Backtest
  - Backtest Data for an Individual Stock
  - First Backtest
  - Investing Backtest Results
  - Accounting for Chance of Default

- Module 4 - Backtesting(Statistical Likelihood of Returns)
  - Backtesting Loop
  - Is the AI any good at picking stocks?
  - AI Performance Projection

- Module 5 - AI Investor Stock Picks 2021
  - Getting the Data Together
  - Making the AI Investor Pick Stocks
  - Final Results, Stock Selection for 2021
  - Discussion of 2022 Performance

4.2 Flow of the System

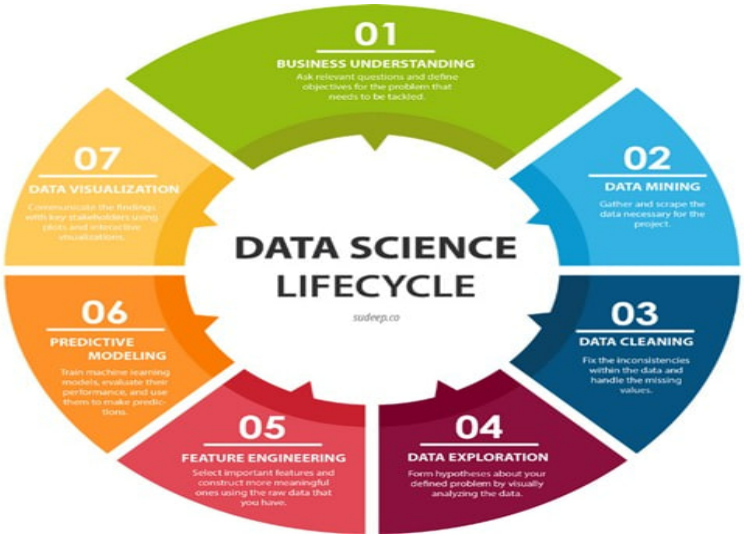


Figure 4.2: Flow of the System

Chapter 5

Making the AI Stock Picker

5.1 Obtaining and Processing Fundamental Financial Stock Data

5.1.1 Getting Data and Initial Filtering

We got our data set from SimFin.com as they have done the hard work for us and cleaned up the data. The data comes in .csv format.

We made a function to get this data loaded and merge the income statement, balance sheet, and cash flow into a single Data Frame. The columns that contain date information got converted from strings to a date format for our code to easily refer to later.

```
In [8]: def getXDataMerged(myLocalPath='C:/Users/saisr/OneDrive/Desktop/AI Stock Picker/Stock Data/'):

            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()
```

```
Income Statement CSV data is(rows, columns):  (11910, 28)
Balance Sheet CSV data is:  (11910, 30)
Cash Flow CSV data is:  (11910, 28)
Merged X data matrix shape is:  (11910, 74)
```

We can see the columns to the data at our disposal by looking at the DataFrame keys

```
In [10]: X.keys()

Out[10]: Index(['Ticker', 'SimFinId', 'Currency', 'Fiscal Year', 'Fiscal Period_x',
       'Report Date', 'Publish Date', 'Restated Date_x', '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', 'Restated Date_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', 'Restated Date', '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.

```
In [11]: def getYRawData(my_local_path='C:/Users/saisr/OneDrive/Desktop/AI Stock Picker/Stock Data/'):

             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
```

There is a lot of data here, 3.2 million rows!

```
In [14]: d = getYRawData()
         d[d['Ticker'] == "GOOD"]
```

```
Stock Price data matrix is:  (3231095, 11)
```

Out[14]:

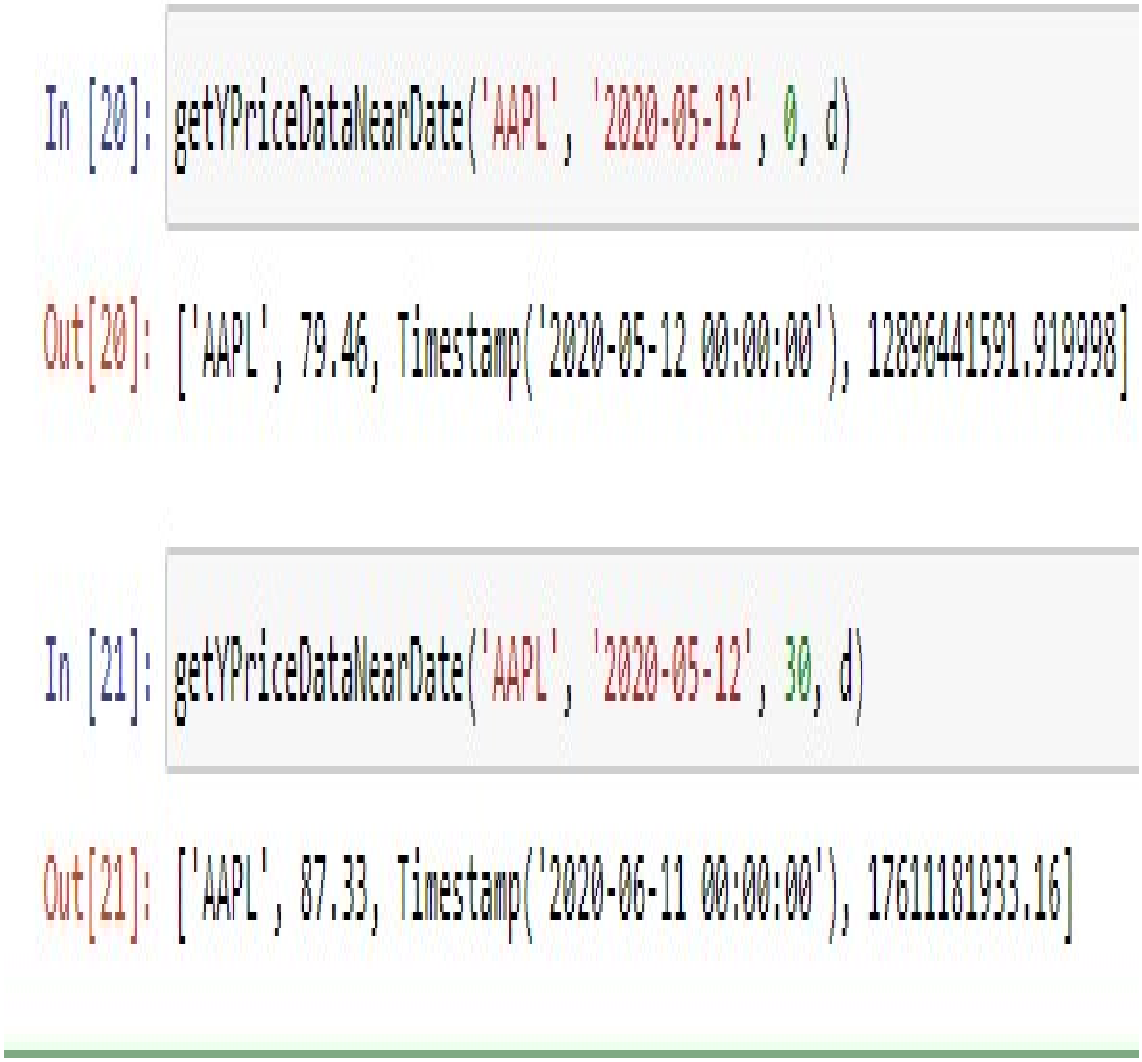
	Ticker	SimFinId	Date	Open	Low	High	Close	Adj. Close	Dividend	Volume	Shares Outstanding
314474	GOOD	106972	2017-02-09	19.48	19.48	19.71	19.67	12.46	NaN	75459	24882758.0
314475	GOOD	106972	2017-02-10	19.68	19.68	20.03	20.01	12.68	NaN	91779	24882758.0
314476	GOOD	106972	2017-02-13	20.03	19.95	20.51	20.46	12.96	NaN	171451	24882758.0
314477	GOOD	106972	2017-02-14	20.23	19.86	20.23	20.17	12.86	0.12	119891	24882758.0
314478	GOOD	106972	2017-02-15	19.99	19.70	20.03	19.99	12.74	NaN	123587	24882758.0
...	...	...	...	...	...	...	...	...	...	...	...
315729	GOOD	106972	2022-02-03	22.99	22.52	22.99	22.57	20.88	NaN	195123	36768779.0
315730	GOOD	106972	2022-02-04	22.58	21.91	22.65	22.34	20.66	NaN	286830	36768779.0
315731	GOOD	106972	2022-02-07	22.34	21.99	22.49	22.08	20.42	NaN	264547	36768779.0
315732	GOOD	106972	2022-02-08	22.05	21.92	22.23	22.00	20.35	NaN	280789	36768779.0
315733	GOOD	106972	2022-02-09	22.17	22.05	22.33	22.33	20.65	NaN	284025	36768779.0

1260 rows × 11 columns

Now, in the code below, we are saying ‘Give me rows that are between the date I want and that date plus a little bit, whilst having the ticker I want”:

```
In [15]: def getYPriceDataNearDate(ticker, date, modifier, dailySharePrices):

             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"]]

In [16]: d=getYRawData()

         Stock Price data matrix is:  (3231095, 11)
```

Now, let's try out our function on some stocks. Here is AAPL (Apple) at some point in 2020, and a month later using the modifier argument as 30.



The results match historical prices online for a bunch of stocks.

Now we can use that function to get our y data. There will be as many y tows 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.

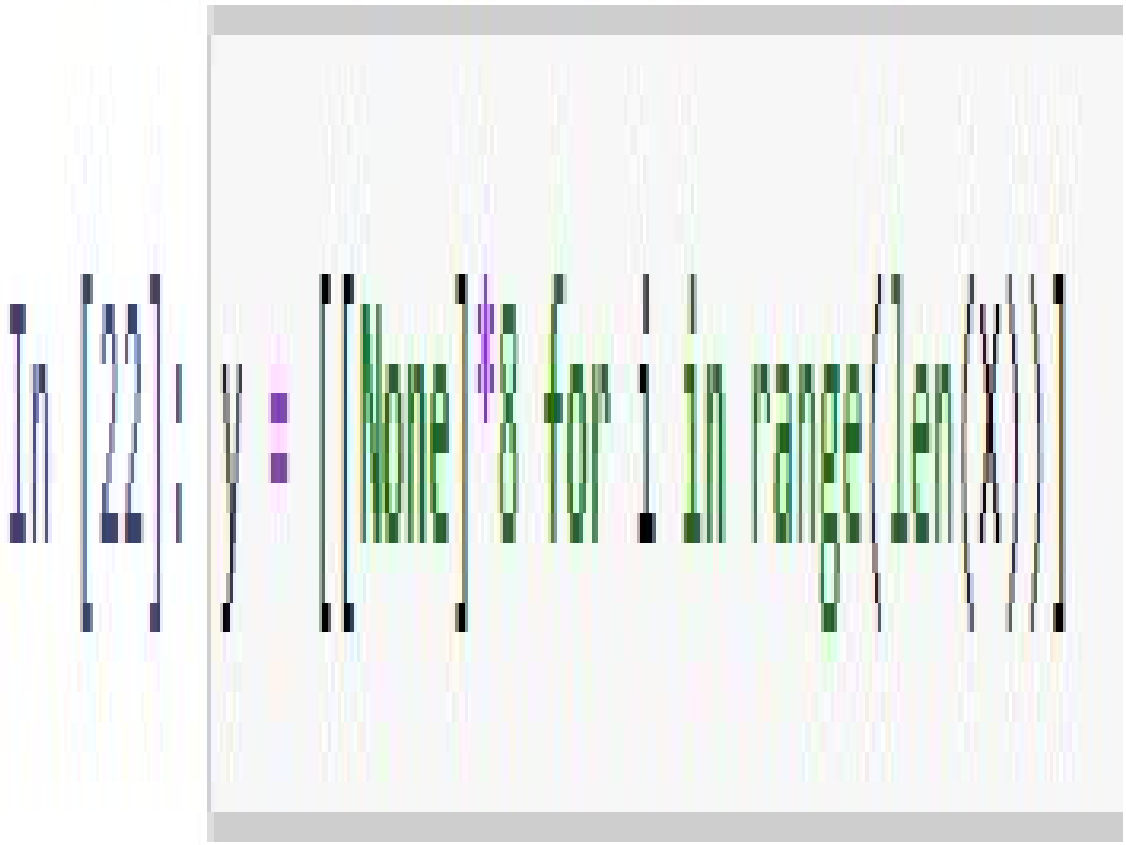
We'll pass the whole of X and the y raw data as arguments, as well as the modifier which we will use as the time in days between start price and end price (In Python all arguments are passed by reference so if you're not used to this don't worry).

The function will return the y data we want for each row of X which will be the returned result from get Y PriceNear Date 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 reallocate 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:



Next is the getYPricesReportDateAndTargetDate() functiomn:

In [23]:
```
def getYPricesReportDateAndTargetDate(x, d, 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)

    # In the end decided this instead of iterating through index.
    # Iterate through a range rather than index, as X might not have
    # monotonic increasing index 1, 2, 3, etc.
    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
```

Next for easy access, we do the following:

```
In [24]: X = getXDataMerged()
         X.to_csv("Annual_Stock_Price_Fundamentals.csv")
```

```
Income Statement CSV data is(rows, columns):  (11910, 28)
Balance Sheet CSV data is:  (11910, 30)
Cash Flow CSV data is:  (11910, 28)
Merged X data matrix shape is:  (11910, 74)
```

Now, this next step takes a very long time as the code is running through our entire 3 million line matrix to find the stock price data within a time window for every stock:

In [27]:
```
# We want to know the performance for each stock, each year, between 10-K report dates.
# takes VERY long time, several hours,
y = getYPricesReportDatesAndTargetDates(X, d, 365) # because of lookups in this function.
```

In [29]:
```
y = pd.DataFrame(y, columns=['Ticker', 'Open Price', 'Date', 'Volume',\
                             'Ticker2', 'Open Price2', 'Date2', 'Volume2'
                            ])
y.to_csv("Annual_Stock_Price_Performance.csv")
```

In [30]:
```
y
```

Out[30]:

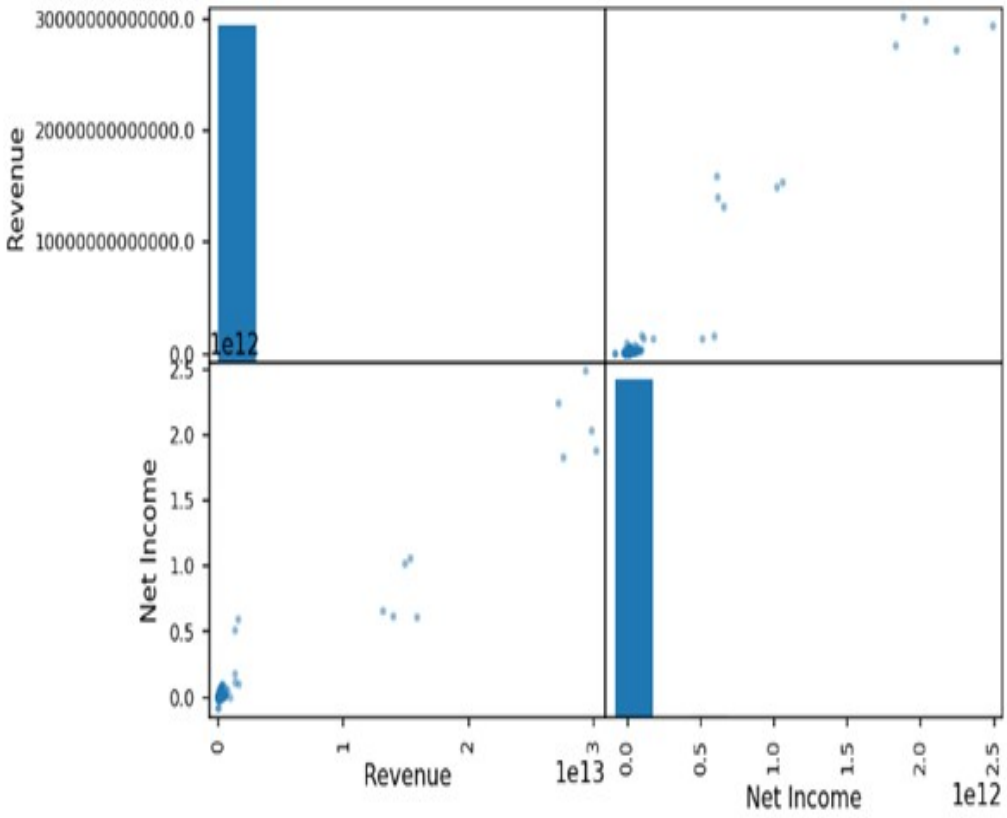
	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
0	A	67.51	2017-12-21	1.009366e+08	A	65.22	2018-12-21	3.054318e+08
1	A	66.33	2018-12-20	2.353265e+08	A	84.94	2019-12-20	1.943251e+08
2	A	83.95	2019-12-19	1.423797e+08	A	119.23	2020-12-18	4.664016e+08
3	A	119.23	2020-12-18	4.664016e+08	A	150.76	2021-12-20	3.056217e+08
4	A	150.35	2021-12-17	3.952235e+08	A	NaN	NaT	NaN
...	...	...	...	...	...	...	...	...
11905	ZYXI	4.55	2018-02-28	5.852392e+05	ZYXI	4.51	2019-02-28	2.961627e+05
11906	ZYXI	5.09	2019-02-26	2.683601e+05	ZYXI	11.90	2020-02-26	3.442361e+06
11907	ZYXI	11.25	2020-02-27	4.283404e+06	ZYXI	14.55	2021-02-26	3.741924e+07
11908	ZYXI	16.78	2021-02-25	7.177796e+06	ZYXI	NaN	NaT	NaN
11909	ZYXI	NaN	NaT	NaN	ZYXI	NaN	NaT	NaN

11910 rows × 8 columns

Now, removing rows with issues

In [31]:
```
X=pd.read_csv("Annual_Stock_Price_Fundamentals.csv", index_col=0)
y=pd.read_csv("Annual_Stock_Price_Performance.csv", index_col=0)
```

In [32]:
```
from pandas.plotting import scatter_matrix
attributes=["Revenue","Net Income"]
scatter_matrix(X[attributes]);
```



```
In [33]: # Find out things about Y data
         print("y Shape:", y.shape)
         print("X Shape:", X.shape)

         y Shape: (11910, 8)
         X Shape: (11910, 74)
```

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 om the data with less than 10,000 shares traded om a day:

```
In [34]: y[(y['Volume']<1e4) | (y['Volume2']<1e4)]# rows with volume issues
```

Out[34]:

	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
103	ABMC	0.09	2018-04-12	0.00	ABMC	0.07	2019-04-12	0.00
104	ABMC	0.07	2019-04-16	0.28	ABMC	0.21	2020-04-15	237501.60
105	ABMC	0.72	2020-07-24	291122.64	ABMC	0.05	2021-07-26	4023.00
156	ACAN	0.64	2021-12-06	4406.40	ACAN	NaN	NaN	NaN
190	ACIA	114.99	2021-03-01	0.00	ACIA	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...
11791	YGYI	0.28	2021-06-25	5021.24	YGYI	NaN	NaN	NaN
11798	YOGAQ	1.02	2019-03-27	116936.88	YOGAQ	0.13	2020-03-26	2455.70
11832	ZDGE	2.46	2017-10-30	64385.58	ZDGE	1.86	2018-10-30	4268.70
11833	ZDGE	1.72	2018-10-29	1735.48	ZDGE	1.69	2019-10-29	1855.62
11834	ZDGE	1.67	2019-10-28	1075.48	ZDGE	2.60	2020-10-27	2466289.80

272 rows × 8 columns

Looks like there are 6800 rows now after filtering :

```
In [39]: y
```

Out[39]:

	Ticker	Open Price	Date	Volume	Ticker2	Open Price2	Date2	Volume2
0	A	67.51	2017-12-21	1.009366e+08	A	65.22	2018-12-21	3.054318e+08
1	A	66.33	2018-12-20	2.353265e+08	A	84.94	2019-12-20	1.943251e+08
2	A	83.95	2019-12-19	1.423797e+08	A	119.23	2020-12-18	4.664016e+08
3	A	119.23	2020-12-18	4.664016e+08	A	150.76	2021-12-20	3.056217e+08
4	AA	47.42	2018-02-26	1.685191e+08	AA	30.70	2019-02-26	1.143155e+08
...	...	...	...	...	...	...	...	...
6804	ZYNE	4.80	2019-03-11	2.979552e+06	ZYNE	3.61	2020-03-10	2.898415e+06
6805	ZYNE	3.61	2020-03-10	2.898415e+06	ZYNE	4.58	2021-03-10	7.940090e+06
6806	ZYXI	4.55	2018-02-28	5.852392e+05	ZYXI	4.51	2019-02-28	2.961627e+05
6807	ZYXI	5.09	2019-02-26	2.683601e+05	ZYXI	11.90	2020-02-26	3.442361e+06
6808	ZYXI	11.25	2020-02-27	4.283404e+06	ZYXI	14.55	2021-02-26	3.741924e+07

6809 rows × 8 columns

```
In [40]: X.shape
Out[40]: (6809, 74)

In [41]: y.shape
Out[41]: (6809, 8)
```

5.1.2 Feature Engineering

Our data will need to be cleaned, as there are null values in the company financial data. 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.

In [6]:
```
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",\
            "Total Current Liabilities",\
            "Property, Plant & Equipment, Net"]
    x[keyCheckNullList]=x[keyCheckNullList].fillna(0)

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"]
```

In [6]:
```
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",\
            "Total Current Liabilities",\
            "Property, Plant & Equipment, Net"]
    x[keyCheckNullList]=x[keyCheckNullList].fillna(0)

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 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 [7]:

```
# 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"]
```

Now, we create our final vector y by finding the percentage change between the open price at the start and end period y-.

In [12]:
```
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_)
```

```
In [13]: y
```

Out[13]:

	Ticker	Perf
0	A	-0.033921
1	A	0.280567
2	A	0.420250
3	A	0.264447
4	AA	-0.352594
...	...	...
6804	ZYNE	-0.247917
6805	ZYNE	0.268698
6806	ZYXI	-0.008791
6807	ZYXI	1.337917
6808	ZYXI	0.293333

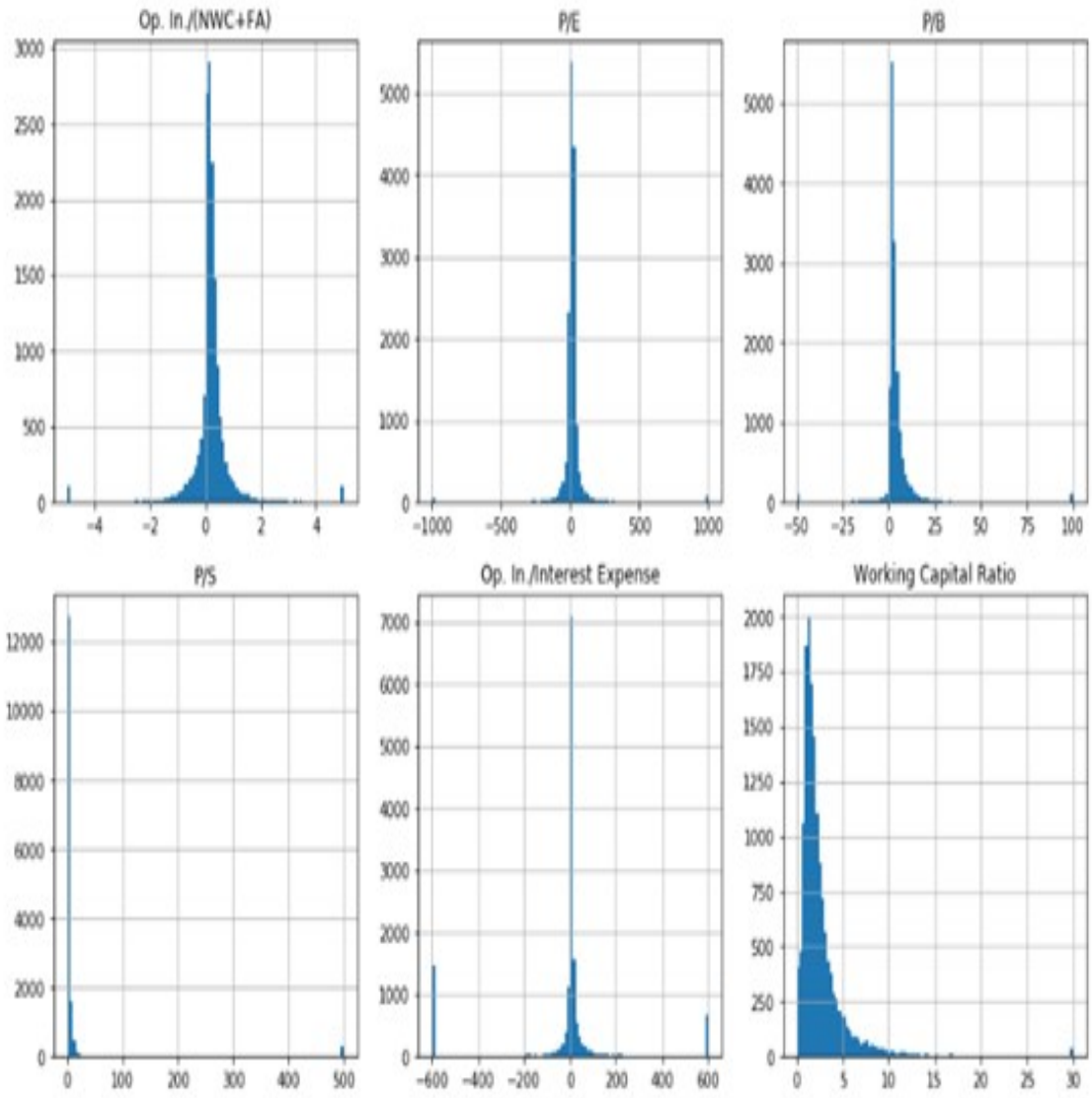
6809 rows × 2 columns

Now we have real market data and stock features all cleaned up ready to train and test our Machine Learning algorithms with.

We can view the distributions:

In [17]:

```
# 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);
```



5.1.3 Using Machine Learning to Predict Stock Performance

Here we see if the Scikit-Learn ML models can give us any predictive power with out stock data. We will try out with a few models, take notes of any models that show predictive ability, save them for later for rigorous backtesting.

Firstly reading our X and y data:

In [4]:
```
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
```

Linear Regression

This is a simple model. We do a single train/test split and train the regressor, just to see if our data works. We chose test-size of 0.1, training with 14000 rows and testing with 1500 rows.

In [16]:

```
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=4
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)
```

```
X training matrix dimensions:  (14029, 18)
X testing matrix dimensions:  (1559, 18)
y training matrix dimensions:  (14029,)
y testing matrix dimensions:  (1559,)
```

We put Powertransformer in a pipeline with our Linear Regressor

In [17]:

```
# Try out linear regression
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))

#import pickle # To save the fitted model
#pickle.dump(pl_linear, open("pl_linear.p", "wb" ))
```

```
Train MSE:  0.3945880050592681
Test MSE:  0.2361519130480995
```

The errors are not that good, and they can vary a lot depending on train/test splits. We then try many runs and see if the regressor is actually learning anything.

In [22]:

```
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, 0.8]
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)

results_df = pd.DataFrame(index=train_sizes) #Create a DataFrame of results
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)
```

In [23]:

```
results_df # see results
```

Out[23]:

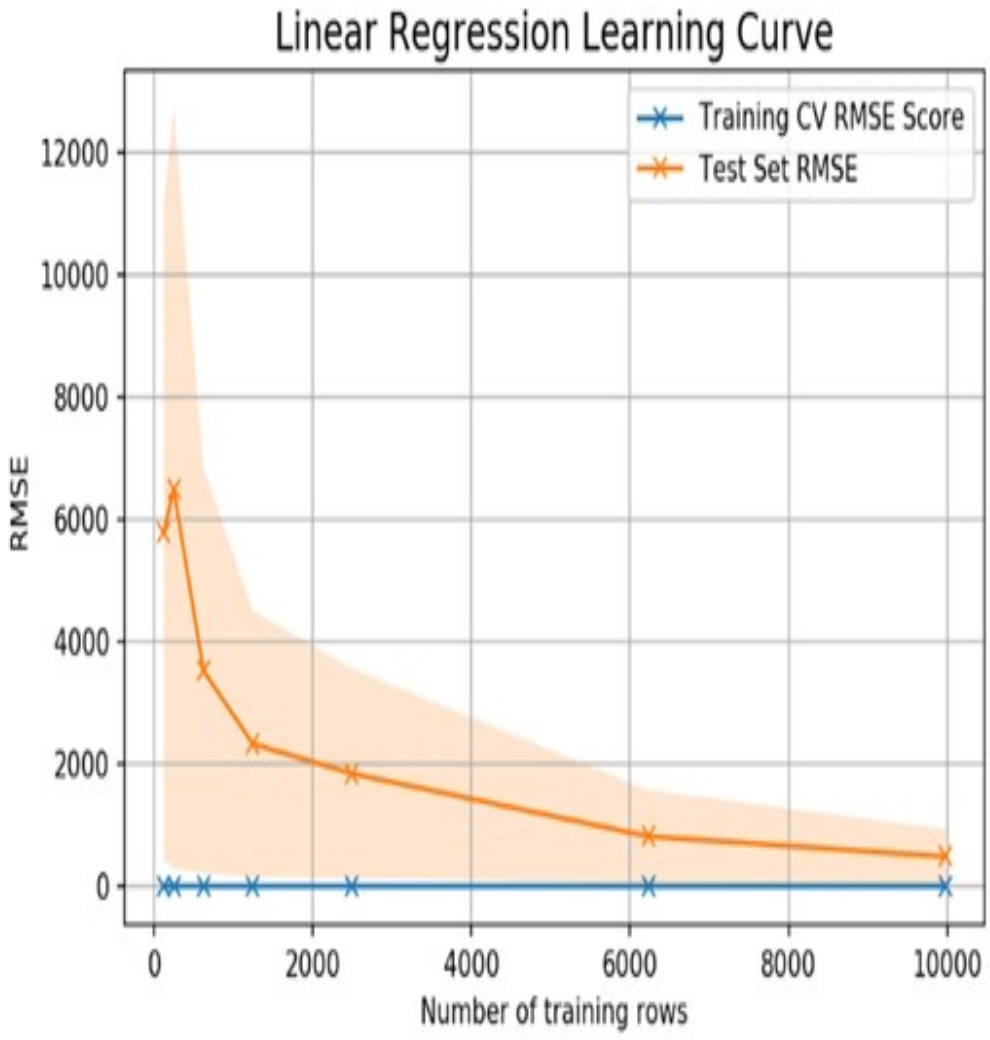
	train_scores_mean	train_scores_std	test_scores_mean	test_scores_std	fit_times_mean	fit_times_std
124	0.469482	0.125088	5792.721898	5371.911839	0.044464	0.009262
249	0.513085	0.128952	6498.078872	6209.732892	0.049661	0.009391
623	0.527395	0.088025	3530.542594	3320.070351	0.064168	0.010495
1247	0.555333	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

In [25]:

```
results_df['train_scores_mean'].plot(style='-x')
results_df['test_scores_mean'].plot(style='-x')

plt.fill_between(results_df.index,\
                 results_df['train_scores_mean']-results_df['train_scores_std'],\
                 results_df['train_scores_mean']+results_df['train_scores_std'], alpha=
plt.fill_between(results_df.index,\
                 results_df['test_scores_mean']-results_df['test_scores_std'],\
                 results_df['test_scores_mean']+results_df['test_scores_std'], alpha=0.
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]);
```

Out[25]: Text(0.5, 1.0, 'Linear Regression Learning Curve')

```
Out[25]: Text(0.5, 1.0, 'Linear Regression Learning Curve')
```



Linear Regression Prediction Analysis

Let us see how good our predictions are in a bit more depth. Our AI will be depending on these predictions so we need to be sure stocks can be picked well in this chapter.

-Plotting Function

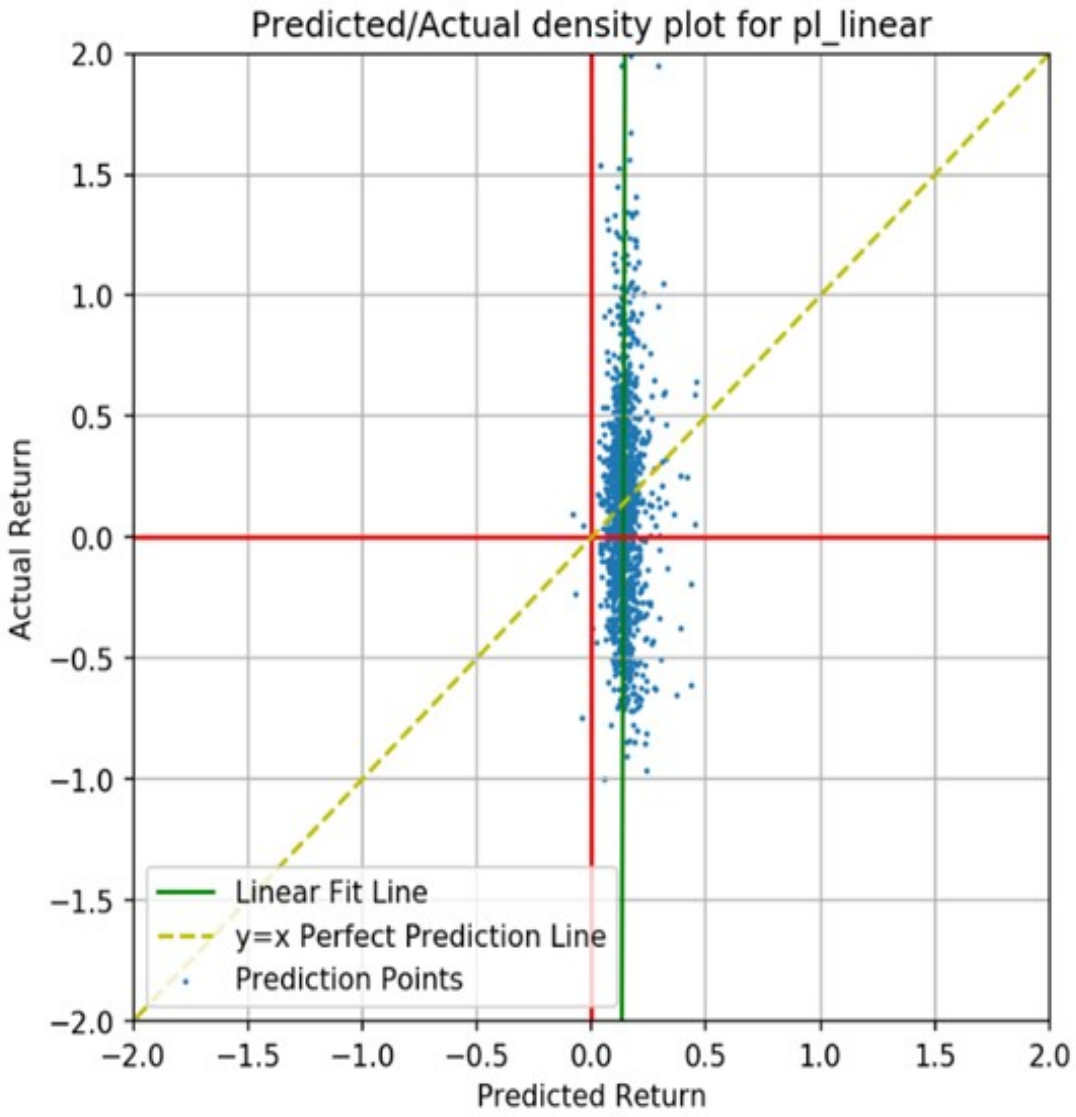
To get a better view of how good the predictions are visually without depending on mean squared error.

In [26]:

```
# Output scatter plot and contour plot of density of points to see if prediciton matche
# Line of x=y is provided, perfect prediction would have all density on this line
# Also plot linear regression of the scatter

# Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

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'])
    # Save Figure
    #plt.figure(figsize=(5,5))
    plt.savefig('result.png')
```

In [27]:

```
plt.figure(figsize=(6,6))
plotDensityContourPredVsReal('pl_linear', y_pred, y_test.to_numpy(), 2)
```



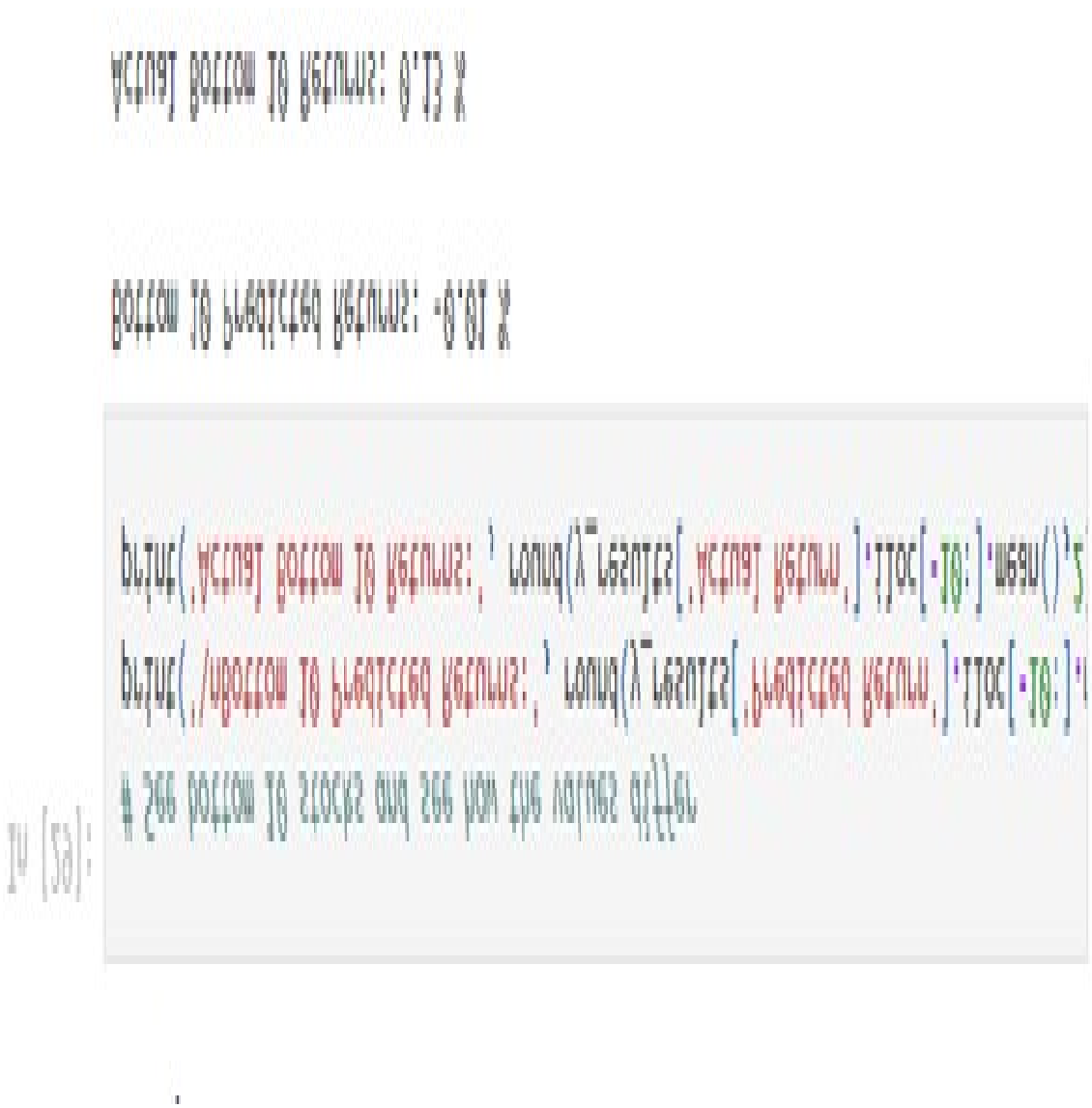
It does not look so good visually, but there seems to be some ability there. Let us take a closer look by taking our predictors top few stock return predictions and comparing them to reality.

In [28]:

```
# 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
print('Actual Top 10 Returns:', round(y_results['Actual Return'].iloc[:10].mean(),2) ,
```

```
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 %
```

Tring the bottom 10:



There is some predictive ability here, it is definitely worthwhile using linear regression in the backtest under greater scrutiny later.

Let's try the model with a few more train/test samplings by changing the random-state and see if the predictive ability stays.

In [30]:

```
def observePredictionAbility(my_pipeline, X, y, returnSomething=False, verbose=True):
    '''
    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

    The arguments returnSomething=False, verbose=True,
    will be used at the notebook end to get results.
    '''
    Top10PredRtrns, Top10ActRtrns=[], []
    Bottom10PredRtrns, Bottom10ActRtrns=[], []

    for i in range (0, 10): # Can try 100
        # 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))

    if verbose:
        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')
```

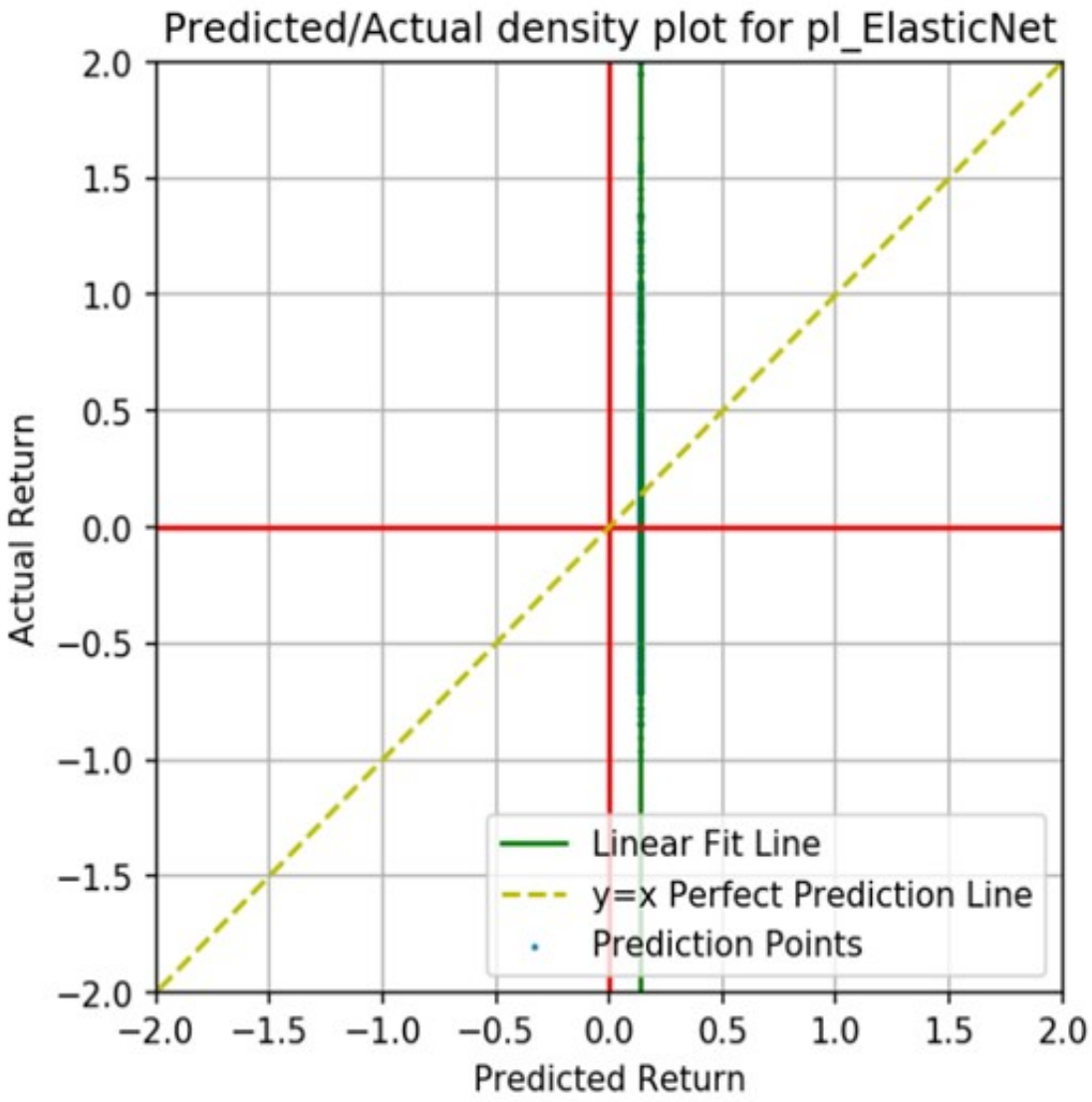
```
        #PERFORMANCE MEASURES HERE
        print(\
        '\033[4mMean Predicted Performance of Top 10 Return Portfolios:\033[0m',\
              round(np.mean(Top10PredRtrns), 2))
        print(\
        '\t\033[4mMean Actual Performance of Top 10 Return Portfolios:\033[0m',\
              round(np.mean(Top10ActRtrns), 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')

    if returnSomething:
        # Return the top10 and bottom 10 predicted stock return portfolios
        # (the actual performance)
        return Top10ActRtrns, Bottom10ActRtrns

    pass
```

```
In [31]: observePredictionAbility(pl_linear, X, y)
```

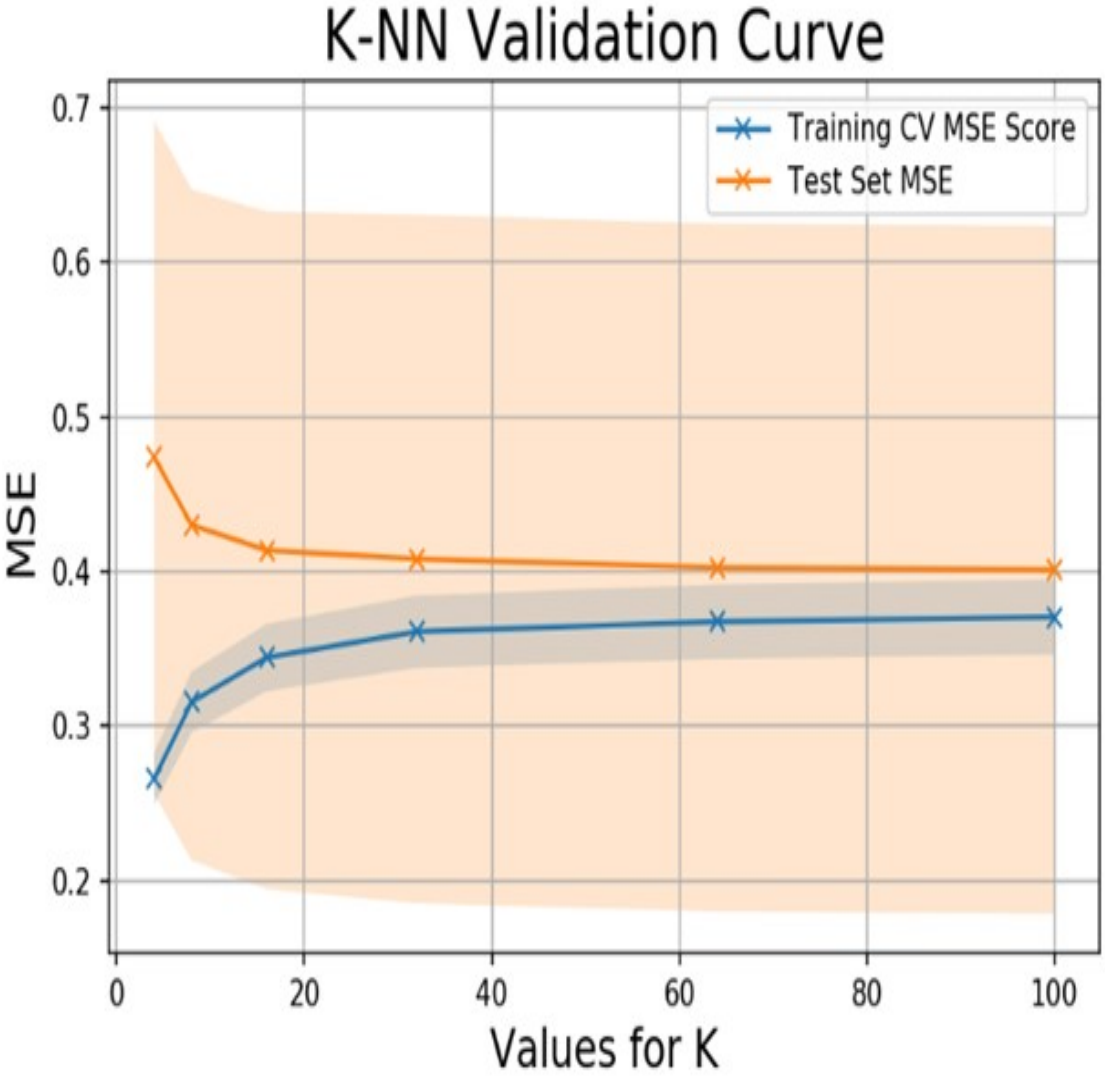
```
Predicted Performance of Top 10 Return Portfolios: [41.59, 46.95, 43.69, 48.56, 37.84, 4
6.13, 40.99, 49.67, 37.02, 38.76]
Actual Performance of Top 10 Return Portfolios: [0.52, 10.32, 0.22, -38.5, 71.35, 2.98,
-20.28, -31.45, 240.97, 134.4]

Predicted Performance of Bottom 10 Return Portfolios: [-1.27, 0.95, 1.67, -2.54, 3.27,
1.76, -0.26, -3.99, 1.14, -163866.64]
Actual Performance of Bottom 10 Return Portfolios: [12.98, -9.32, 19.09, 6.05, -7.1, 14.
42, 56.16, 15.36, 26.5, 1.59]
-------------

Mean Predicted Std. Dev. of Top 10 Return Portfolios: 4.33
Mean Actual Std. Dev. of Top 10 Return Portfolios: 83.96
Mean Predicted Std. Dev. of Bottom 10 Return Portfolios: 49160.02
Mean Actual Std. Dev. of Bottom 10 Return Portfolios: 17.8
-------------

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
-------------
```

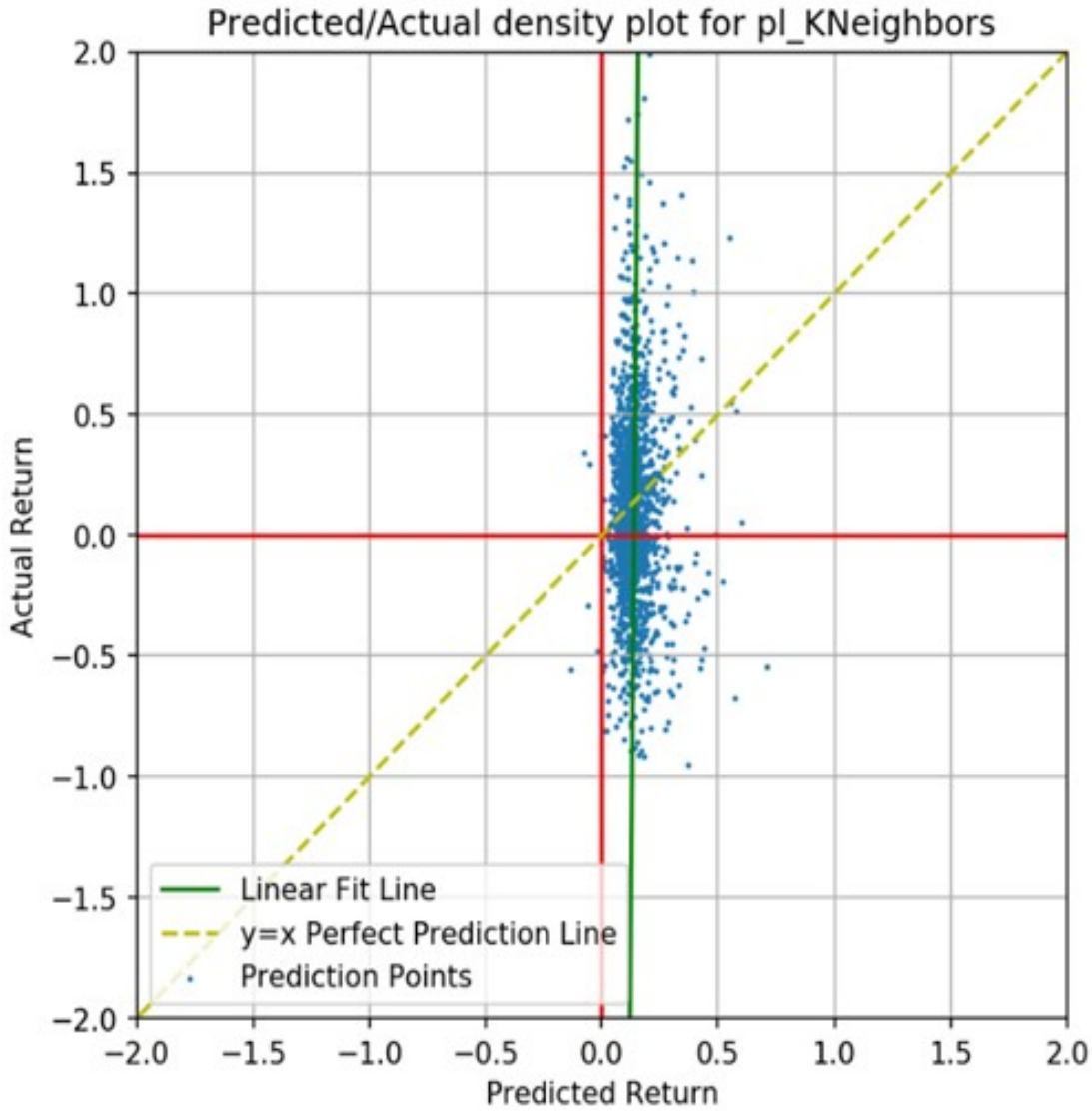
Same above process was done using:

Elastic Net Regression:



K-Nearest Neighbours Regression

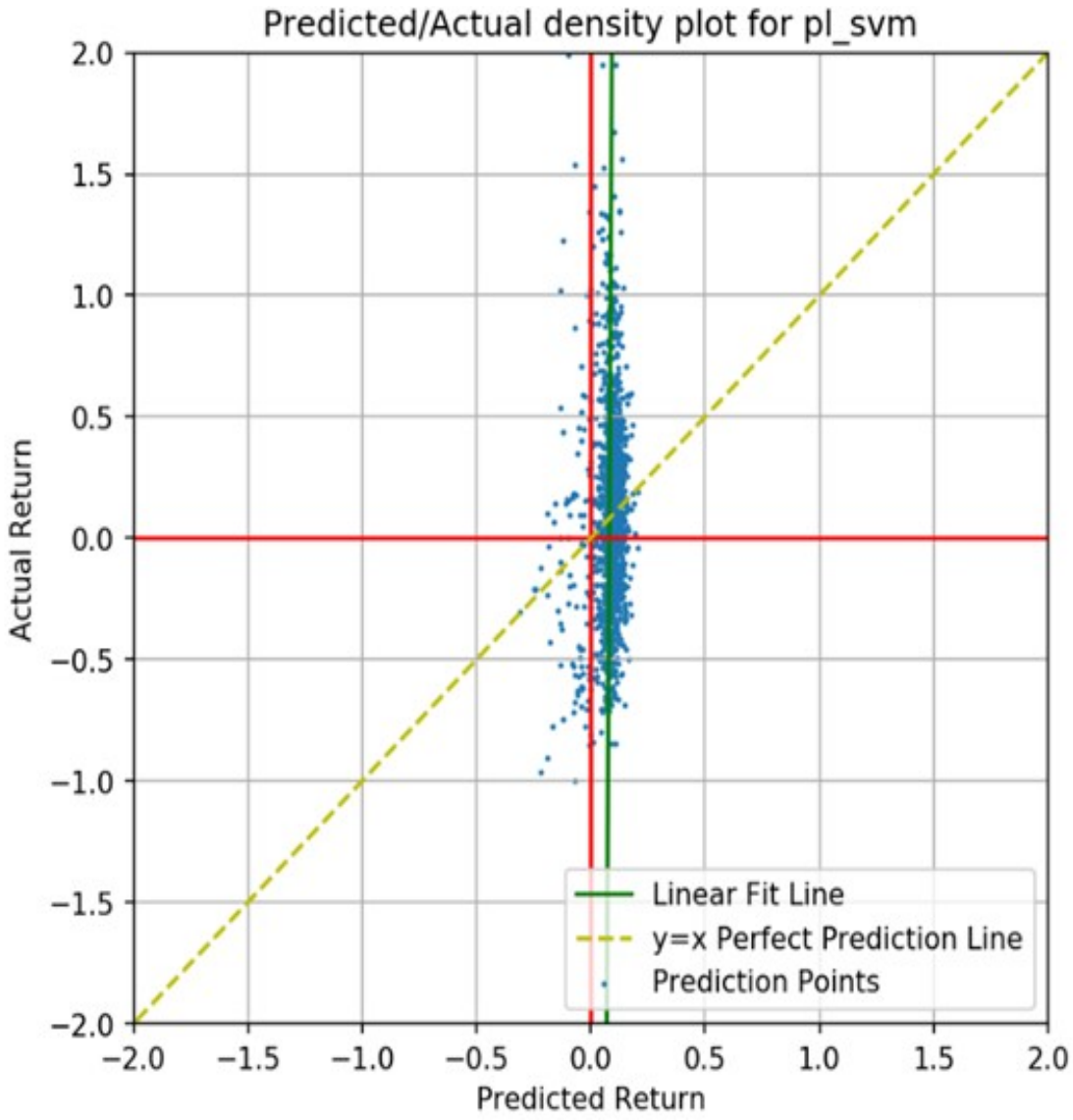

K-NN Validation Curve
Training CV MSE Score
Test Set MSE
MSE
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100
Values for K


Predicted/Actual density plot for pl_KNeighbors
Actual Return
Predicted Return
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Linear Fit Line
y=x Perfect Prediction Line
Prediction Points

In [48]:
```
observePredictionAbility(pl_KNeighbors, X, y)
```

```
Predicted Performance of Top 10 Return Portfolios: [54.64, 62.13, 52.65, 60.98, 51.77, 5
7.02, 55.27, 62.51, 47.62, 51.84]
Actual Performance of Top 10 Return Portfolios: [-12.85, 9.31, 15.65, 2.61, 88.3, -9.27,
-15.23, -21.86, 166.27, 103.65]

Predicted Performance of Bottom 10 Return Portfolios: [-4.32, -5.46, -4.87, -7.87, -8.5
3, -3.26, -4.4, -5.15, -6.93, -2.32]
Actual Performance of Bottom 10 Return Portfolios: [9.38, -3.32, 6.25, 6.78, -34.88, 5.6
2, -12.06, 1.74, -21.91, 17.08]
-------------

Mean Predicted Std. Dev. of Top 10 Return Portfolios: 4.73
Mean Actual Std. Dev. of Top 10 Return Portfolios: 60.68
Mean Predicted Std. Dev. of Bottom 10 Return Portfolios: 1.86
Mean Actual Std. Dev. of Bottom 10 Return Portfolios: 15.13
-------------

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
-------------
```

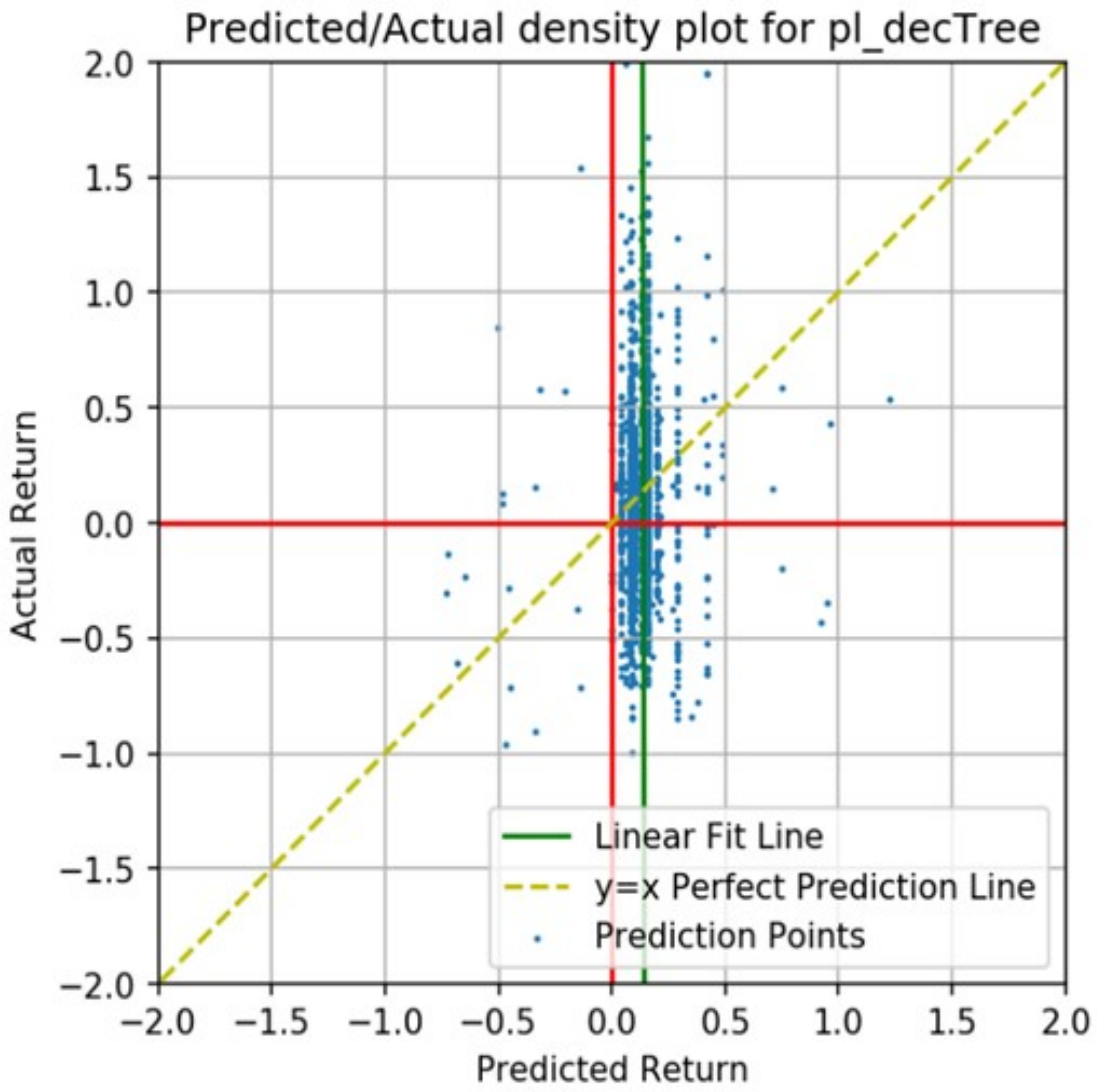
Support Vector Machine Regression


Predicted/Actual density plot for pl_svm
Actual Return
Predicted Return
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Linear Fit Line
y=x Perfect Prediction Line
Prediction Points

In [57]:
```
observePredictionAbility(pl_svm, X, y)
```

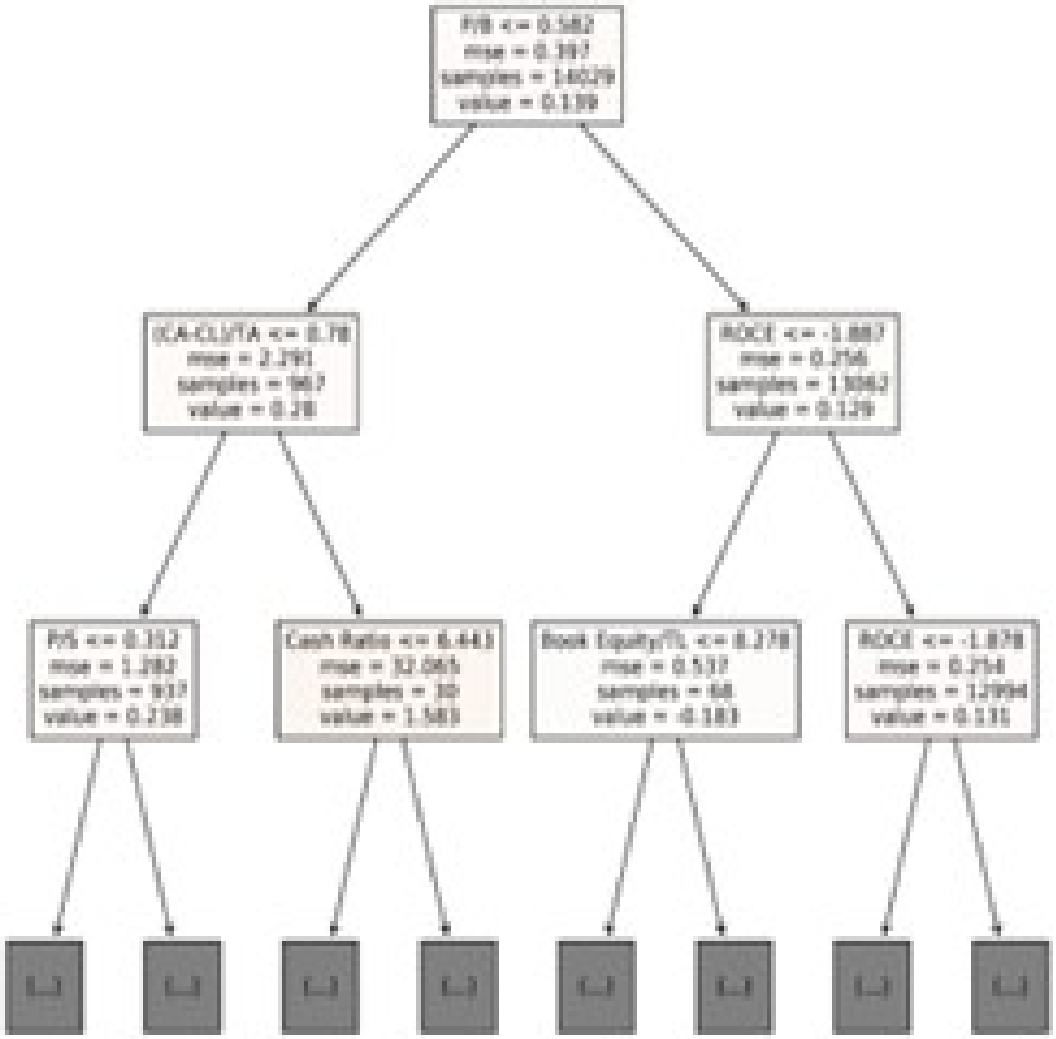
```
Predicted Performance of Top 10 Return Portfolios: [18.67, 20.34, 19.24, 18.48, 19.98, 1
9.47, 18.44, 19.66, 19.27, 19.29]
Actual Performance of Top 10 Return Portfolios: [12.42, 9.19, -6.12, -4.19, 19.75, 5.13,
-1.35, 10.58, 8.87, 26.87]

Predicted Performance of Bottom 10 Return Portfolios: [-22.58, -32.14, -26.75, -20.91, -
25.05, -22.3, -24.29, -30.21, -30.01, -26.54]
Actual Performance of Bottom 10 Return Portfolios: [2.15, -15.87, -5.49, -24.14, -35.9,
-26.53, -29.74, 32.46, 0.87, 9.43]
-------------

Mean Predicted Std. Dev. of Top 10 Return Portfolios: 0.59
Mean Actual Std. Dev. of Top 10 Return Portfolios: 9.83
Mean Predicted Std. Dev. of Bottom 10 Return Portfolios: 3.57
Mean Actual Std. Dev. of Bottom 10 Return Portfolios: 20.07
-------------

Mean Predicted Performance of Top 10 Return Portfolios: 19.28
       Mean Actual Performance of Top 10 Return Portfolios: 8.11
Mean Predicted Performance of Bottom 10 Return Portfolios: -26.08
       Mean Actual Performance of Bottom 10 Return Portfolios: -9.28
-------------
```

Decision Tree Regression

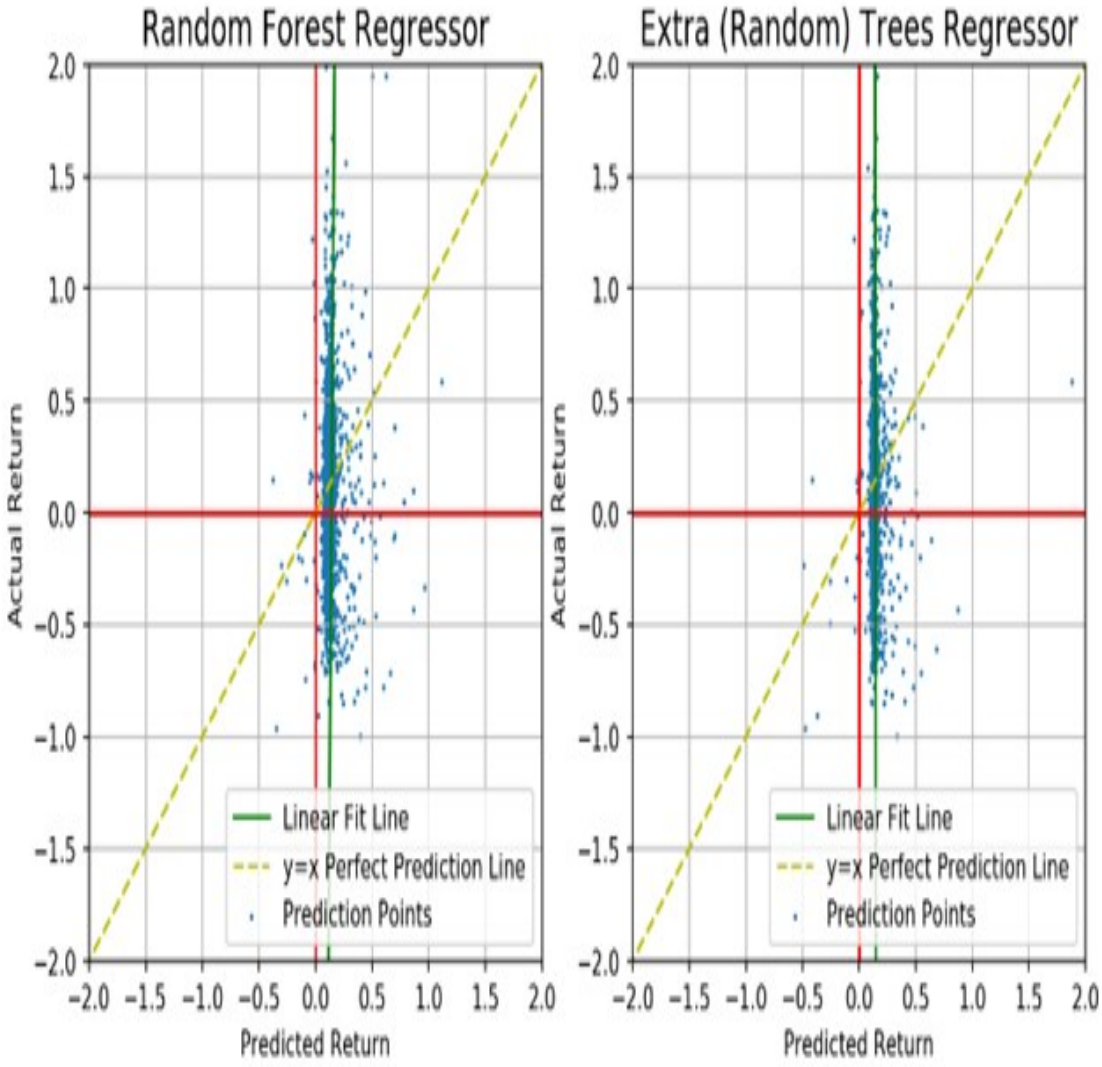

Predicted/Actual density plot for pl_decTree
Actual Return
Predicted Return
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Linear Fit Line
y=x Perfect Prediction Line
Prediction Points

In [75]:
```
observePredictionAbility(pl_decTree, X, y)
```

```
Predicted Performance of Top 10 Return Portfolios: [370.2, 333.28, 351.2, 399.45, 692.7,
702.31, 259.02, 734.34, 656.18, 365.56]
Actual Performance of Top 10 Return Portfolios: [-18.52, 1.76, 3.96, 25.2, 97.11, 59.76,
-8.53, 39.01, 41.0, 8.95]

Predicted Performance of Bottom 10 Return Portfolios: [-72.13, -76.11, -85.97, -71.96, -
82.47, -76.96, -86.46, -81.6, -88.93, -84.73]
Actual Performance of Bottom 10 Return Portfolios: [-6.55, 22.4, 17.35, -29.47, 71.63, -
13.61, -23.36, -8.56, 23.37, 18.95]
-------------

Mean Predicted Std. Dev. of Top 10 Return Portfolios: 175.66
Mean Actual Std. Dev. of Top 10 Return Portfolios: 33.35
Mean Predicted Std. Dev. of Bottom 10 Return Portfolios: 5.78
Mean Actual Std. Dev. of Bottom 10 Return Portfolios: 28.34
-------------

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
-------------
```



Random Forest Regression


Random Forest Regressor
Extra (Random) Trees Regressor
Actual Return
Predicted Return
Linear Fit Line
y=x Perfect Prediction Line
Prediction Points
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0

In [82]:
```
observePredictionAbility(rfregressor, X, y)
```

```
Predicted Performance of Top 10 Return Portfolios: [157.08, 177.49, 80.24, 75.78, 110.5,
220.22, 115.46, 161.29, 99.9, 109.18]
Actual Performance of Top 10 Return Portfolios: [11.95, -6.73, 52.44, 6.65, 86.8, 58.2,
-3.98, -17.18, 192.04, 23.25]

Predicted Performance of Bottom 10 Return Portfolios: [-22.06, -18.51, -14.23, -17.81, -
17.63, -11.43, -23.19, -24.85, -21.98, -11.29]
Actual Performance of Bottom 10 Return Portfolios: [9.39, -8.67, -24.27, -43.3, -45.55,
5.55, -11.19, 4.85, -28.39, 36.98]
-------------

Mean Predicted Std. Dev. of Top 10 Return Portfolios: 44.1
Mean Actual Std. Dev. of Top 10 Return Portfolios: 59.46
Mean Predicted Std. Dev. of Bottom 10 Return Portfolios: 4.56
Mean Actual Std. Dev. of Bottom 10 Return Portfolios: 24.42
-------------

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
```

Now plotting results from all regression models

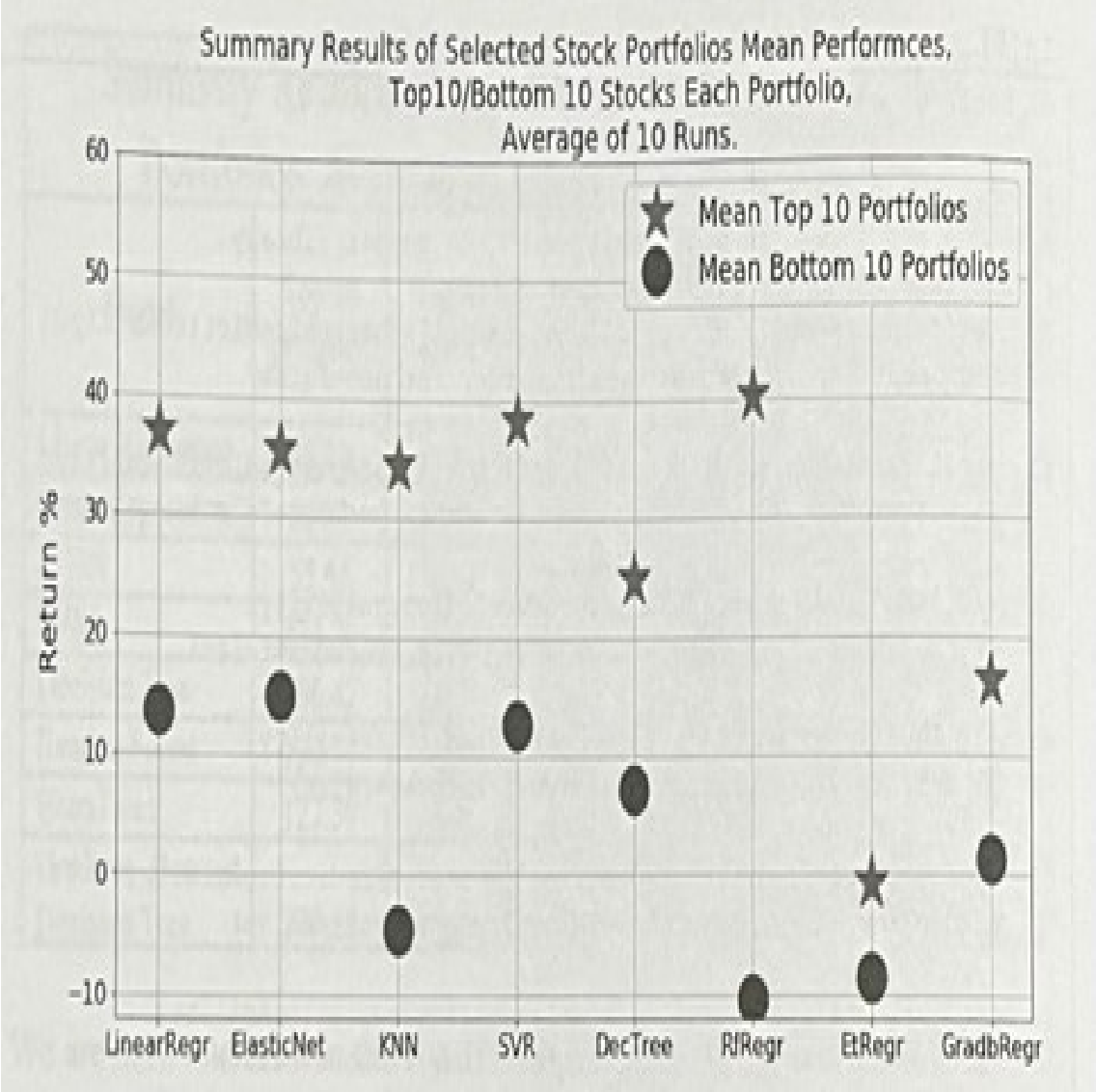
In [93]:

```
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)
```

In [94]:

```
# 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_best.mean()+df_best.std()).plot(linewidth=0, marker='*', markersize=10, markerface
#(df_best.mean()-df_best.std()).plot(linewidth=0, marker='*', markersize=10, markerface
df_worst.mean().plot(linewidth=0,
                     marker='o',
                     markersize=25,
                     markerfacecolor='b',
                     markeredgecolor='b')
#(df_worst.mean()+df_worst.std()).plot(linewidth=0, marker='o', markersize=10, markerfa
#(df_worst.mean()-df_worst.std()).plot(linewidth=0, marker='o', markersize=10, markerfa

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()
```

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



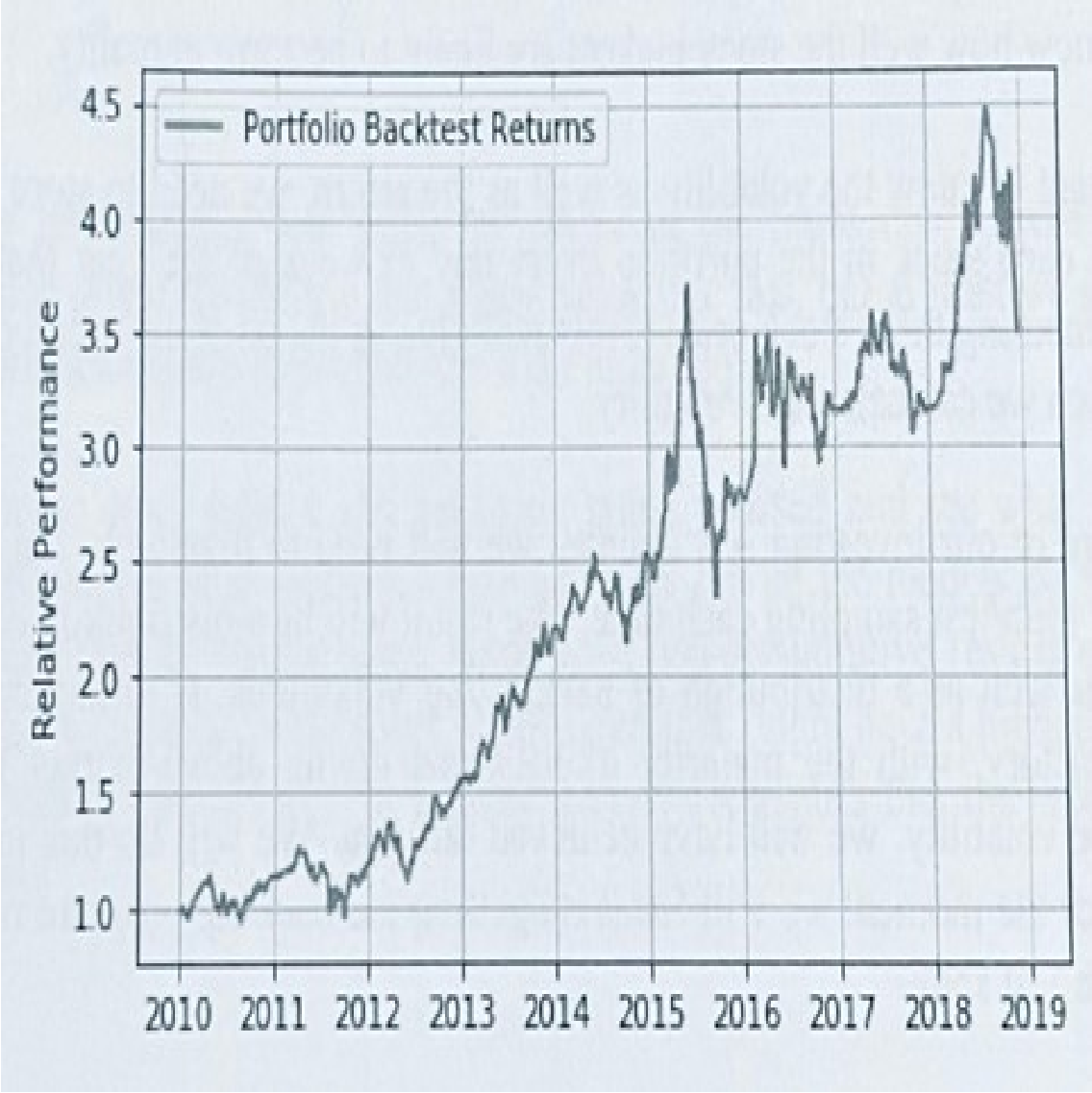
5.1.4 AI Backtesting

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 portfolio.
- Top 7 highest return predictions from the test set to create an equally weighted 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:



In [4]:

```
def loadXandyAgain(randRows=False):
    '''
    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

    if randRows:
        # 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
```

Training a model for backtest:

We select stocks in a backtest with a picked, pretrained mode.

Then the train set trains the model; the test set is sent to the backtester.

So using linear regression:

In [5]:

```
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" ))
```

```
train mse:  0.3668897622925844
test mse:  462.67364066367236
```

Next Read in train/test:

In [6]:

```
# 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=0)
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"])
```

In [7]:

```
y_test.head() # y targets
```

Out[7]:

	Perf
7217	0.616757
12508	0.184577
14996	0.524234
5852	0.153191
8857	0.039030

Z Score to account for default chance:

In [9]:

```
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
```

n [10]:

```
z = calcZScores(X)
z.head()
```

ut[10]:

	Z score
0	7.664439
1	7.435224
2	7.538172
3	9.311402
4	8.385200

Then we write the backtest program: (below each represent a function/method):

- We trace out daily stock price time series for all stock, 3 million rows, some days missing
- Get stock prices between dates
- Given a data range(index), and a series of rows, that may not correspond exactly, return a DataFrame that gets rows data for each period in the data range(index)
- 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. dateTimeIn-dex is typically a weekly index, so we know what days to fetch the price for.

- 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 port-folio value.
- Function runs a backtest. Returns DataFrames of selected stocks/portfolio performance, for 1 year.

Run a Backtest: Training a model pipeline and backtesting with get-PortTimeSeries function:

In [21]:
```
daily_stock_prices_data=getYRawData()
```

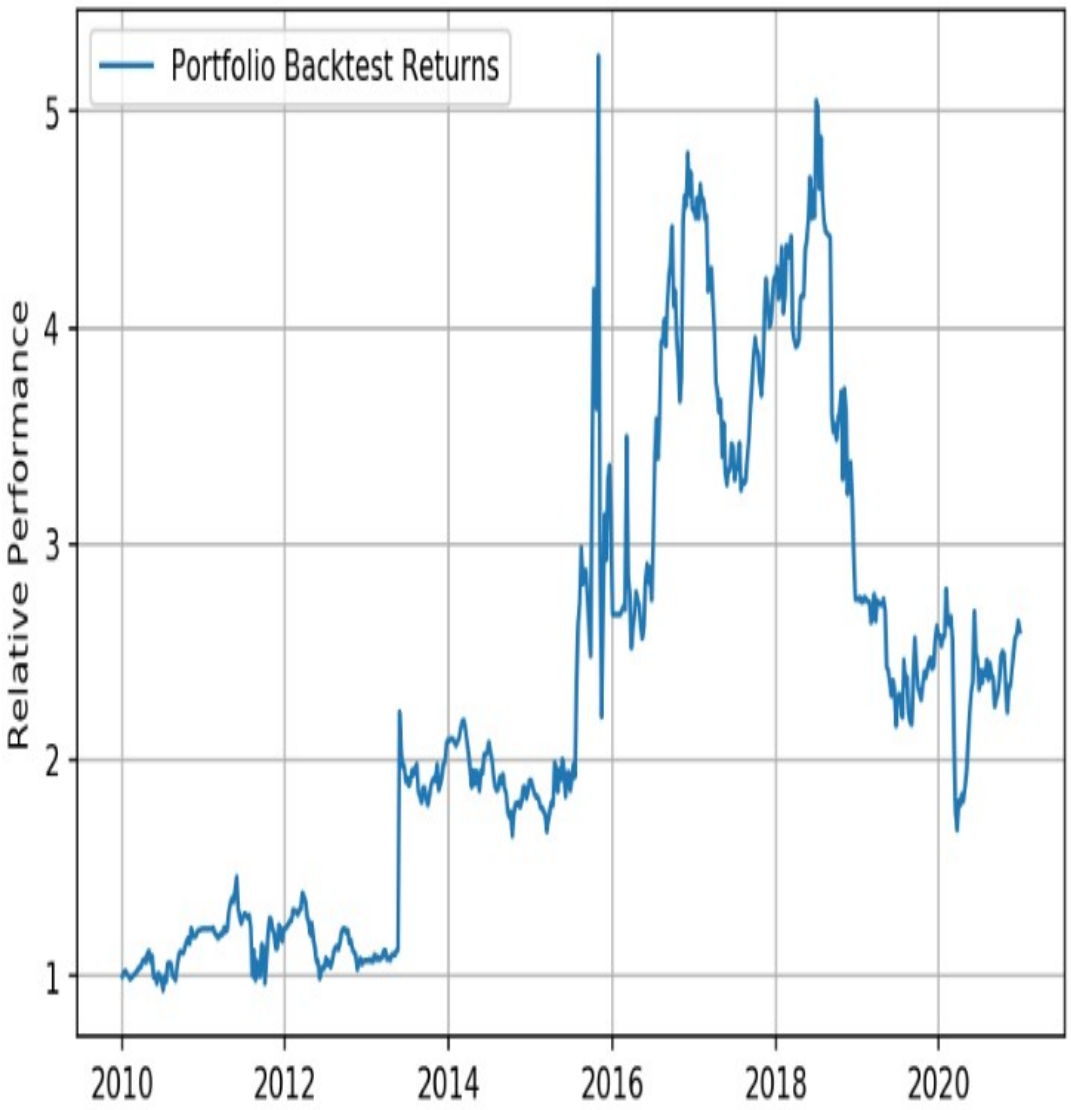
```
Reading historical time series stock data, matrix size is:  (6482985, 11)
```

In [48]:
```
#pl_linear.p, pl_ElasticNet.p, pl_KNeighbors.p, pl_rfregressor.p, pl_decTree.p, pl_svm.p
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),
      '%')
```

For all the historical data: years 2009-2019

```
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  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 %
```



Seeing how Individual Stocks Performed

In [81]:

```
# Make X ticks standard, and grab stock prices as close to those points as possible for ea

#DatetimeIndex
date_range = pd.date_range(start=myDate, periods=52, freq='W')

# 7 greatest performance stocks of y_pred
ticker_list = y_withData_Test[mask2015].reset_index(drop=True)[bl_bestStocks]["Ticker"].va
stockRet = getStockTimeSeries(date_range, y_withData_Test, ticker_list , mask2015, daily_s
```

In [82]:

```
y_small
```

Out[82]:

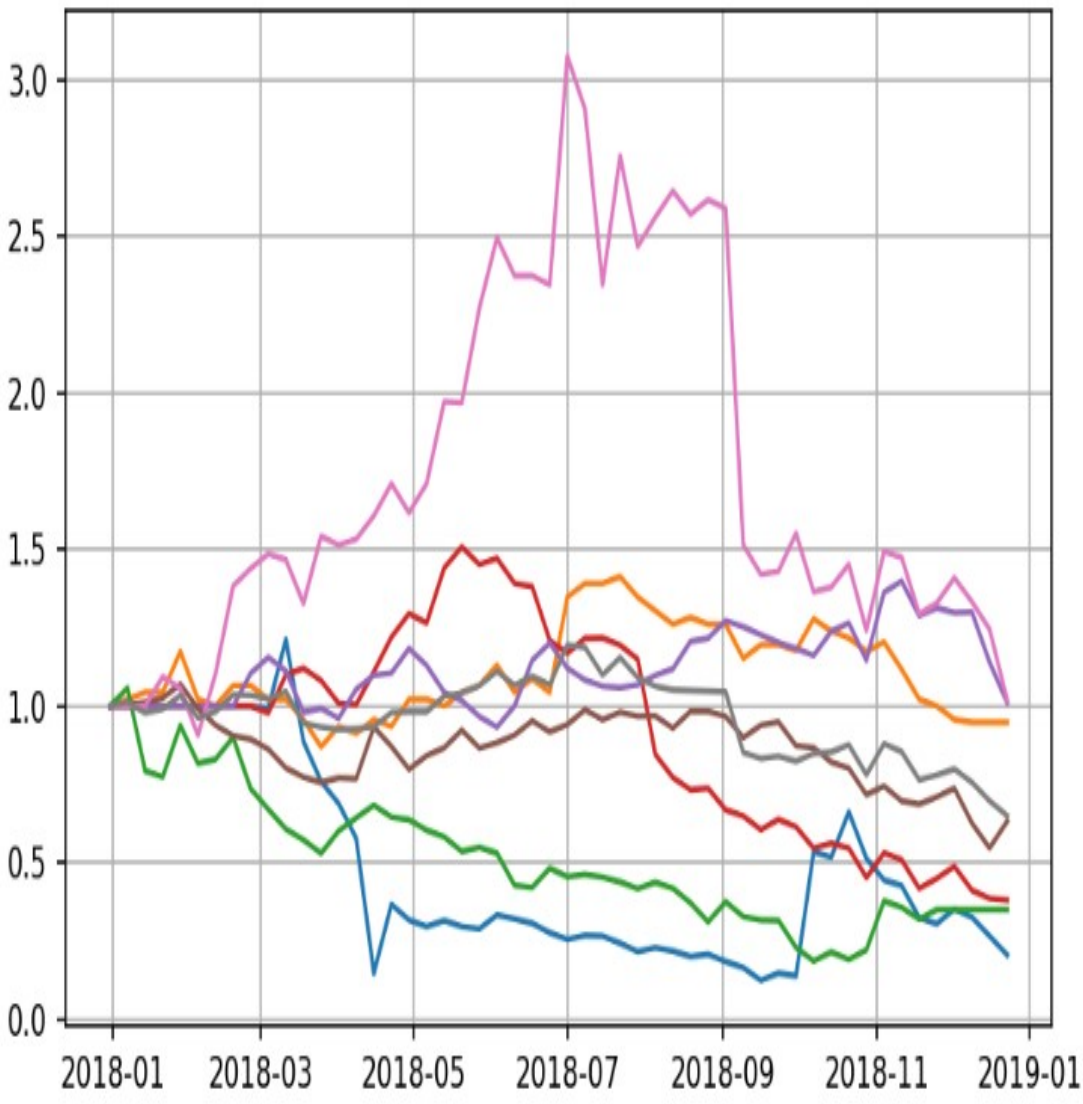
	Ticker	Perf
7217	INTC	0.616757
12508	SIX	0.184577
14996	WHR	0.524234
5852	FUEL	0.153191
8857	MCHP	0.039030
...	...	...
8845	MCD	0.232620
12718	SNDK	0.090199
2379	CA	0.152164
11142	POST	0.077282
9881	NFLX	0.443002

7794 rows × 2 columns

In [58]:

```
#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()
```

```
['VTVT' 'LEE' 'WNDW' 'SGMS' 'DIN' 'NAV' 'EGAN']
```

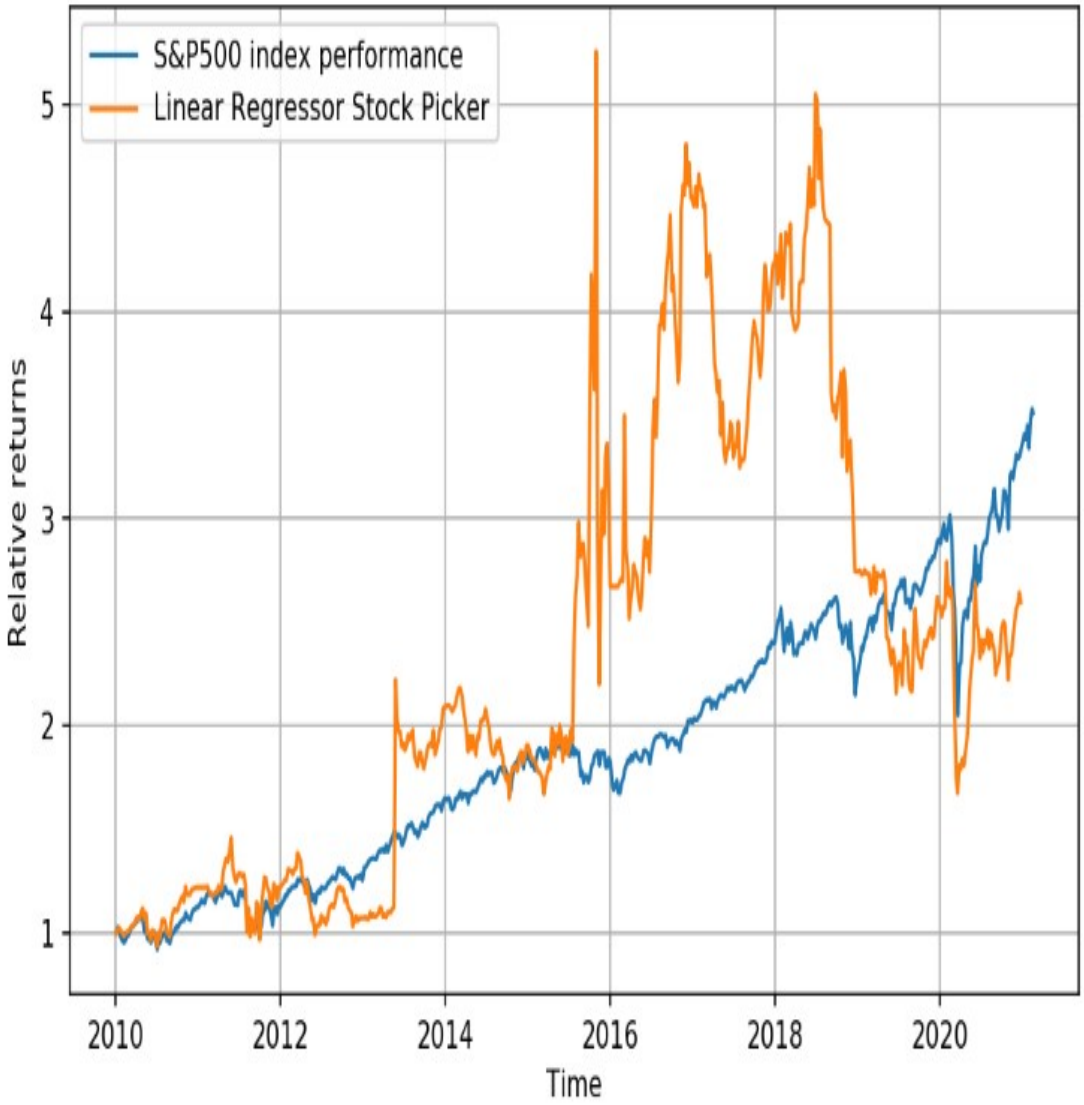


PLOT BACKTEST WITH SP500

In [63]:

```
# GSPC.csv taken directly from Yahoo.com is the S&P500.
# https://finance.yahoo.com/quote/%5EGSPC/history?period1=1235174400&period2=1613865600&in
spy=pd.read_csv("GSPC.csv", index_col='Date', parse_dates=True)
spy = spy[spy.index > pd.to_datetime('2010-01-01')]
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.s
print('volatility of S&P 500 was: ', spy["Relative"].diff().std()*np.sqrt(52))
```

```
volatility of AI investor was:  1.3759320717225751
volatility of S&P 500 was:  0.325059094152396
```


S&P500 index performance
Linear Regressor Stock Picker
Relative returns
1
2
3
4
5
2010
2012
2014
2016
2018
2020
Time

Chapter 6

Conclusion and Shortcomings

6.1 Summary of the Project

It works! The AI Stock Picker which is trained using Machine Learning and which runs on Python actually works!

AI Performance

The K-Nearest Neighbours regressor and the Gradient Boosted Decision tree regressor returned 3.95x in our backtesting. This return was above the SP500 index. These two regressors have outperformed when they can select from the stocks available, whilst being trained with all our historical stock market data.

This performance was high in backtesting, as having them train on one half and then select stocks from the other half limits the investing AIs in a big way which is done in the classical regression testing.

Making the AI Investor Pick Stocks

Our conclusion is that the 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 are not 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 making the predictions on our 2021 financial data.

Final Stock Selection for 2020 with Gradient Boosted Decision Tree and K Nearest Neighbours

In [57]:

```
def pickStockForMe(model='GBoost'):
    '''
    Pick stocks.
    Reads Annual_Stock_Price_Fundamentals_Ratios.csv,
    and Annual_Stock_Price_Performance_Percentage.csv,
    trains the AI with the best model/parameters,
    Then picks stocks using outputs from Notebooks 1 and 2:
    Annual_Stock_Price_Fundamentals_Ratios_2021.csv,
    and Tickers_Dates_2021.csv.
    Outputs a DataFrame of best picks.
    '''
    # 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)

    if model == 'GBoost':
        # Gradient Boosted tree
        model_pl = traingbregressorModel(X, yperf)
        y_pred=model_pl.predict(X_2021)
        y_pred=pd.DataFrame(y_pred)

    elif model == 'KNN':
        # KNN
        model_pl = trainKNeighborsModel(X, yperf)
        y_pred=model_pl.predict(X_2021)
        y_pred=pd.DataFrame(y_pred)

    else:
        return None
```

```
# 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)
```

In [60]:

```
# What it's all about.
pickStockForMe()
```

Out[60]:

	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

```
In [61]: pickStockForMe('KNN')
```

Out[61]:

	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

6.2 Shortcomings

• We can expect the portfolio to fluctuate more than the SP500, and that in some years it will underperform.

• It is to be remembered that in backtesting these models were restricted because the universe of stocks they could choose from is significantly reduced due to training/testing data separation.

• While it shows good performance, the K-Nearest Neighbours regressor appears to expose us to a lot more volatility for little extra gain.

• 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-favor stocks tend to fall further in a market crash, and if they recover to reasonable valuations afterward, they will be rising from a lower level.

• 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 financial numbers only, we run the risk of buying things we do not understand on even a basic level.

• Another risk is that accounting changes could undermine the AI completely. For example, in 2017 Berkshire Hathaway had to account for the change in the value of their securities as profit. If too much of this accounting change occurs, the underlying reasoning of our model falls apart and we will no longer be able to rely on it.

6.3 Conclusion - AI Investor Final Stock Picks

	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

Figure 6.1: Gradient Boosted Decision Tree

	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

Figure 6.2: K-Nearest Neighbours

Interestingly, the Gradient Boosting and K-Nearest Neighbour regressors seem to want to pick different kinds of companies.

Our Gradient Boosted Decision Tree seems to be acting like a modern value investor, with picks boring stock. 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(Tesla) or AMZN(Amazon) or AAPL(Apple).

The advantage we have with this AI Stock Picker is we could make it to invest as conservatively as we want by changing the minimum Altman Z Ratio and the number of stocks our AI holds, correlate performance with other company ratios or whatever else our imagination can come up with to make to make our code suit our needs.

Bibliography

[1] Pike, William H., and Patrick C. Gregory. Why Stocks Go Up (and Down). Thomson Executive Press, 1996.

[2] Graham, Benjamin, and Luke Daniels. The Intelligent Investor Rev Ed. HarperCollins, 2015.

[3] Seth, Klarman. ”Margin of Safety.” (1991).

[4] Greenblatt, Joel. The little book that still beats the market. John Wiley Sons, 2010.

Online Resources

- www.magicformulainvesting.com
- www.valueinvestingai.com
- www.blockchain.com
- www.blockchair.com
- www.wikipedia.org
- www.linkedin.com
- www.google.com
- www.lucidchart.com

Dataset Respositories

- SimFin: https://simfin.com/data/bulk
- Github: https://github.com/trending/python
- Kaggle : https://www.kaggle.com/datasets