Machine Learning Algorithms: A Horse Race

Module 10 Practical Exercises in Research Methods Case Study

at

University of Applied Sciences Lucerne

2nd Semester, M.Sc. Banking and Finance

Submitted by:

Jean-Philipp Brühwiler
Yannick Arfaoui
Jawahar Vimalendran

Lucerne, May 2023

University of Applied Sciences Lucerne

Master of Science in Banking and Finance

Case Study:

Machine Learning Algorithms: A Horse Race

May 2023, Module 10

Authors:

Jean-Philipp Brühwiler, Reckenbühlstrasse 14 6005 Luzern, jean-philipp.bruehwiler@stud.hslu.ch
Yannick Arfaoui, Maihofstrasse 42 6004 Luzern, yannick.arfaoui@stud.hslu.ch
Jawahar Vimalendran, Jupiterstrasse 1/1143 3015 Bern, jawahar.vimalendran@stud.hslu.ch

Supervisors:

Mr. Moreno Frigg, Suurstoffi 1 6343 Rotkreuz, moreno.frigg@hslu.ch
Dr. Jürg Fausch, Suurstoffi 1 6343 Rotkreuz, juerg.fausch@hslu.ch

Table of Content

1. ABSTRACT **2**
2. INTRODUCTION **2**
3. LITERATURE REVIEW **4**
4. OVERVIEW OF EXPERIMENT **10**
4.1 METRICS 10
4.2 DATA CLEANING 12
4.3 DATA PREPARATION 15
4.4 MACHINE LEARNING MODELS 18
4.4.1 Linear Regression Model (OLS) *18*
4.4.2 Linear Regression Model Adjusted (OLS) *19*
4.4.3 Linear Regression Model Adjusted 2 (OLS) *20*
4.4.4 Gradient boosting model *21*
4.4.5 Random Forest Model *22*
4.4.6 Random Forest Model with Grid Search *23*
4.4.7 Random Forest Model with Randomized Grid Search *25*
5. RESULTS **26**
6. DISCUSSION **36**
7. CONCLUSION **37**
8. BIBLIOGRAPHY ..... **I**
9. APPENDIXES ..... **VIII**

List of appendixes

APPENDIX 1: T_ALPHA_FF5MOM TARGET VARIABLE PLOTS VIII
APPENDIX 2: ALPHA_FF5MOM TARGET VARIABLE PLOTS X

List of Figure

Figure 1, (code) import packages and data set. 12
Figure 2, (code) dealing with NaN in Fund Name. 12
Figure 3, (code) dealing with Rear and Front load. 12
Figure 4, (code) dealing with share classes' age. 13
Figure 5, (code) dealing with Equity size. 13
Figure 6, (code) dealing with U.S equity. 13
Figure 7, (code) dealing with actively managed funds (1). 13
Figure 8, (code) dealing with actively managed funds (2). 14
Figure 9, (code) dealing with Total Net Asset. 14
Figure 10, (code) creating dummy variables for team of managers vs. single manager. 15
Figure 11, (code) dropping unnecessary columns. 16
Figure 12, (code) shifting target variable for training the models. 16
Figure 13, (code) drop NaN for the models. 16
Figure 14, (code) splitting the data frame into training and testing data. 17
Figure 15, (code) defining plot functions. 17
Figure 16, (code) defining an indicator function. 17
Figure 17, (code) defining independent & dependent variables for the training and the testing of the models. 17
Figure 18, (code) Linear Regression Model (OLS). 18
Figure 19, (code) Linear Regression Model Adjusted (OLS). 19
Figure 20, (code) Linear Regression Model Adjusted 2 (OLS). 20
Figure 21, (code) Gradient Boosting Model. 21
Figure 22, (code) Random Forest Model. 22
Figure 23, (code) Random Forest Model with Grid Search. 23
Figure 24, (code) Random Forest Model with Randomized Grid Search. 25
Figure 25, ML Models' results with "alpha_FF5MOM". Own Elaboration. 26
Figure 26, ML Models' results with "t_alpha_FF5MOM". Own Elaboration. 27
Figure 27, Gradient Boosting Residuals Plot (alpha_FF5MOM as target). Own Elaboration. 28
Figure 28, Wilcoxon Signed-Rank Python Code (alpha_FF5MOM target). Own Elaboration. 29
Figure 29, Wilcoxon Signed-Rank Outputs. Own Elaboration. 30
Figure 30, Wilcoxon Signed-Rank Outputs (t_alpha_FF5MOM target variable). Own Elaboration. 30
Figure 31, Friedman Test Python Code alpha_FF5MOM. Own Elaboration. 31
Figure 32, Friedman Test Output (alpha_FF5MOM target variable). Own Elaboration. 31
Figure 33, Friedman Test Output (t_alpha_FF5MOM target variable). Own Elaboration. 32
Figure 34, Nemenyi Test Code. Own Elaboration. 32
Figure 35, Nemenyi Test Code. Own Elaboration. 33
Figure 36, Bonferroni-Dunn Test Code. Own Elaboration. 34
Figure 37, Bonferroni-Dunn Test Code. Own Elaboration. 34
Figure 38, Effect Size Python Code. Own Elaboration. 35
Figure 39, Effect Size Output Table. Own Elaboration. 35

1. Abstract

Beginning in the late 1990s and early 2000s, the study of machine learning algorithms' ability to predict financial returns came into focus. Many years later, research today now shows that various machine learning algorithms can accurately model the return of actively managed mutual funds. Much of this research uses the CRSP survivor bias free mutual fund database to train and test these models. We analyze a large dataset primarily made of up of CRSP data to show both random forest and gradient boosting models are effective at predicting the outperformance of actively managed mutual funds. Using common evaluation metrics, we also find there to be no statistically significant difference in predictive effectiveness between the two models on our dataset. We validate our findings through training and testing with multiple measurements of return.

2. Introduction

Every morning, millions of people drive to work in any given country. Every one of those commutes has one thing in common: an end destination. However, every commute probably has many ways to get to that end destination. How do you decide which route to take every morning? Well, it probably depends on many factors such as the weather forecast, if there is construction on the route, how much time you have to get to that destination, and likely many other factors we don't consciously think about.

In this way, millions of people all over the world conduct their own analysis every morning similar to what a machine learning algorithm would do. Through years of travelling to the same end destination, commuters are pretty good at judging the best way to get to work given all these factors. Well, what if we could make this process better? What if there was a way to take even more variables into account to predict the best way to get to work every morning?

With modern modelling techniques, we can do just that. In fact, there are probably several different techniques that can model this with varying degrees of accuracy. That's great to know, but which model should one use if there are several options? Which one is most likely to get me to work the most efficiently? Think about how you could apply this line of thinking to more impactful areas of your life. Your morning commute likely doesn't affect your life very much in the grand scheme of things (those who are prone to road rage and/or have been on the receiving end of road rage likely disagree).

What certainly does affect your life in a meaningful way are the returns your investments generate. In fact, whoever is reading this likely has some of their money invested in an actively managed mutual fund and with this same line of thinking, we can attempt to build models that can predict these returns. As we outline in our literature review, it has been shown that machine learning models can accurately predict which funds are likely to outperform ex post. However, just like with your morning commute, there are multiple ways to get there.

The applications for machine learning in finance are quite vast. There are few industries where the use cases for machine learning are clearer than in finance since it is a data-driven field. However, being able to implement a model is very different than knowing whether the outputs are useful or whether there is possibly a better model than what you are using. Just like with our example of a morning commute, the use case for modelling is clear, but identifying the inputs and which model to use is still a real challenge.

At this point, it is common knowledge that most actively managed mutual funds underperform their benchmark in the long term (Carhart, 1997). However, this is not the case for every fund

and there certainly exist funds which are capable of outperforming. Additionally, while a fund may underperform long term, there are often stretches of time where a fund can outperform in the short to medium term. These stretches of outperformance are generally long enough for investors to theoretically generate index beating returns.

With thousands of mutual funds to choose from, identifying which fund is likely to outperform in the future is like finding a needle in a haystack. With machine learning, finding the needle is now much easier (in theory). Herein lies the significance of this study and the field of research more generally. If machine learning algorithms can be applied in real time, for which there is growing evidence, it could fundamentally change how investors allocate to mutual funds.

Given all this, our paper aims to examine which algorithms are most effective in predicting ex post outperformance. We also examine whether there is a statistically significant difference in predictive power between the various algorithms we examine.

In our analysis, we run two separate analyses with different definitions of outperformance. We first trained and tested our models using the t-statistic of the five factor Fama-French model plus momentum alpha (t_alpha_FF5MOM). We chose this measure of outperformance because it is a measure of the statistical significance of outperformance, not just the raw number. Also, this model accounts for the momentum effect, which is the well documented effect of past winners/losers remaining winners/losers short term (Zoghlami, 2015).

Our other definition of outperformance is the raw five factor Fama-French model plus momentum (alpha_FF5MOM). We chose this metric as it is a strict measure of outperformance, much more so than just raw outperformance which is heavily influenced by noise. If the models are effective, they should perform well on both measures of outperformance.

The goal of our study is not to build portfolio and or create trading strategies, rather our paper seeks to validate the findings of current literature and examine if there is a difference in effectiveness between models. Our study therefore contributes to the existing literature by evaluating these algorithms in depth and drawing conclusions about their overall efficacy.

The findings of this study also have the potential to provide valuable insights into the applicability of machine learning algorithms in finance and investment research more generally. Our study contributes to the ongoing debate surrounding the usefulness of these models for making investment decisions.

Our findings empower investors and researchers to draw more informed conclusions with respect to their models and have the confidence that their analysis is done on a sound basis. By examining the effectiveness of popular machine learning models, we further the growing body of research dedicated to machine learning its applications in predicting performance.

3. Literature Review

The applications of machine learning techniques in finance have been researched for decades. It is hard to pinpoint when the first academic studies about machine learning began to emerge. It is commonly accepted that machine learning applications in finance first began to be seriously researched in the 1990s (Gogas et al., 2021). It should be noted that terms such as machine learning and artificial intelligence don't necessarily have one definition.

A commonly cited catalyst for research into machine learning was in 2003 when Robert F. Engle and Clive W.J. Granger won the Nobel Prize in Economics (Diebold, 2004). Engle and Granger revolutionized time series analysis and econometrics. For example, Engle is credited with having developed both the Autoregressive Conditional Heteroscedasticity (ARCH) model and its extension, the Generalized Autoregressive Conditional Heteroscedasticity (GARCH) model.

These concepts and models have served for much of the basis for machine learning research in the financial world that is taking place today. By the mid-2000s, machine learning gained ever more popularity in the financial space as its use cases became clearer. However, in order to understand how we got here today, we must first take a step back to understand the problem this paper addresses.

One of the first long-term studies that was published on the performance of actively managed mutual funds was published in 1968 called *The Performance of Mutual Funds in the Period 1945-1964*. Jensen empirically showed that the vast majority of actively managed mutual funds underperform a passively managed index fund in the long term.

This casts doubt on the notion that a fund manager's skill has any impact on a fund's returns. It also implies that there isn't any value in allocating to actively managed funds at all. These findings give credence to various forms of the efficient market hypothesis.

Much to the chagrin of active managers, Jensen's findings have been consistently backed up by subsequent studies. Carhart also backed up Jensen's findings in 1997 that outperformance doesn't hold up over time. Carhart also demonstrates that common market influencing factors outside the control of managers is what truly drives stock returns.

Carhart's findings are also in line with those of Kaniel, R., Lin, Z., Pelger, M., & Van Nieuwerburgh, S., who suggest the two key factors affecting mutual fund outperformance are positive fund flows and momentum of the fund's holdings. In fact, the only aspect not explainable by factors out of the managers' control is the persistent *underperformance* of certain funds.

Also, indicators of activeness as measured by turnover are negatively correlated with returns. According to Carhart's findings, for every 1% increase in fund turnover, performance drops by about 0.95%. Additionally, a 1% increase in fees is correlated with a drop in performance of about 1.54% (Carhart, 1997).

Additionally, Carhart also came to some other interesting conclusions. Namely, if a fund does outperform its index, this usually lasts between one to three years. He also found that for the funds which do outperform for longer periods of time, the outperformance is not statistically different from zero. Of the funds Carhart analyzed, the funds which generate alpha essentially "break even" on the outperformance net fees.

However, a crucial limitation of the study is that funds often close, so the later data are potentially subject to survivorship bias. In Carhart's original dataset, about a third of all funds ceased operations since the inception date of the dataset. Another takeaway from this study is that funds which reach the top decile of performance will often fall into the "loser" category the following year.

On a more positive note, one of the most interesting findings from Carhart's study is that funds which generated high returns in one year, are likely to have higher than average returns the next year, but not after. This suggests an investor could theoretically use ex post return data to identify funds which will outperform in the near-term ex ante.

This notion is further explored by Fama and French in *A Five-Factor Asset Pricing Model* originally published in 2015. In this study, Fama and French expand on their famous three-factor model to include factors for profitability (RMW) and investment (CMA). The RMW factor measures the difference in returns between companies with high and low profitability. Rather intuitively, the study shows companies with high profitability tend to have higher returns than firms with low profitability. The CMA factor captures the difference in return behind companies with high and low levels of investment. Perhaps counterintuitively, companies that invest conservatively tend to have higher returns than those that invest aggressively.

However, a caveat of this model is that the factors are based on accounting heuristics such as book-to-market ratio and profitability. These calculations can often be subject to measurement error and can be greatly affected by the accounting practices employed by the individual companies. The model also assumes that the exposure to risk is constant over time. Lastly, the model is static and there are clearly other factors which influence market returns which are not captures in these factors.

Despite these drawbacks, the model is better at explaining return than the three-factor model and is a very comprehensive metric. For this reason, this model is often used to predict the out/underperformance of mutual funds as well. While not perfect, it serves as a useful benchmark which is why it is often used as a target variable for training machine learning models.

Alongside continued developments in modelling came an ever-increasing capacity of computer power and sophisticated software. Additionally, we are currently experiencing a boom in the sheer amount of data available to researchers (Greenemeier, 2017). With more computing power and data than ever before, researchers are now able to train models faster and more accurately than ever before.

More recently (as of writing) study *Machine Learning and Fund Characteristics Help to Select Mutual Funds with Positive Alpha* was published (Gil-Bazo et al., 2022). The researchers used a vast dataset of CRSP mutual fund data to train various machine learning models and evaluate their respective performance. The researchers structured their dataset by using risk-adjusted excess return of a mutual fund as the target variable. The goal of this study was to use machine learning models to construct long-only portfolios of actively managed mutual funds that were able to generate statistically significant alpha net all transaction costs.

They selected various fund characteristics, some available in the CRSP database, and some calculated independently to use as inputs. The researchers selected metrics such as manager tenure, expense ratio, and realized alpha. In total there were over ten variables they used to train various machine learning models to predict returns.

They used three different machine learning models: elastic nets, random forests, and gradient boosting. They also used traditional OLS models as a benchmark to compare their models against. This makes sense since a rule of thumb in modelling is to not use a complex model for the sake of complexity if a simpler model performs just as well.

Briefly, elastic nets are linear regression models which combines both L1 and L2 regularization penalties. The purpose of these penalties is to avoid overfitting, which is a common issue with linear regression models (Babyak, 2004). This allows the regression to conduct feature selection and better deal with correlated values. This type of model is commonly used when there are many inputs, but it is not immediately clear which ones are the most important.

On the other hand, gradient boosting is an ensemble learning method. Ensemble learning is a modelling method which combines multiple weakly predictive models to then create a stronger predictive model. In the case of gradient boosting, this is usually done with decision trees, which is also the methodology Gil-Bazo et al. employed. The final model is constructed in an iterative way where each subsequent model is used to correct the failings of the previous model. Finally, weights are added to the collection of weaker models to build a final model. If implemented correctly, the model is known for having very high accuracy.

Finally, random forest algorithms are also ensemble learning methods in a similar vain to gradient boosting. Random forests also employ decision trees but do so differently than gradient boosting algorithms. Every tree is trained on a random subset of the data and a random subset of the features for making decisions on each so-called split. Random forest models are generally known for being able to deal with overfitting on datasets with a plethora of features. However, if not tuned properly, random forest models can also be subject to overfitting.

The researchers used the popular R programming language to run these models. The models were largely successful in their stated goals. They were able to show that these machine learning models were able to select long only portfolios that generated alpha. Their findings contradict the notion that skill plays no role in mutual fund performance. Since the models were consistently able to select portfolios which did outperform, this suggests there is some skill aspect that affects mutual fund returns.

The researchers also found that biannual rebalancing improved the performance of the portfolio, lending credence to the notion that outperformance is mostly short term but can be captured. All in all, the main take ways from this study are three-fold. Firstly, machine learning algorithms can be trained to accurately predict fund returns. Secondly, these algorithms can potentially be used to make alpha generating investment decisions. Thirdly, fund characteristics alone are sufficient measures to predict fund outperformance.

There is further growing evidence that Gil-Bazo and his cohorts are onto something with their findings. Researchers Kaniel, R., Lin, Z., Pelger, M., & Van Nieuwerburgh, S. also demonstrate in their study *Machine-learning the Skill of Mutual Fund Managers* that machine learning can be used to select positive alpha funds and predict return.

These researchers used a neural network to construct their portfolios. Gil-Bazo et al. also built a neural network model in their study but focused more on the other three mentioned models. The researchers also used CRSP data to train and test their model but also incorporated a factor to serve as a barometer for overall economic activity. The benchmark they chose is the Chicago Fed National Activity Index (CFNAI).

Broadly, it is an economic indicator that measures overall market activity and inflationary pressures in the United States. Since the funds they examined are US-based funds, it makes

sense to use a barometer which focuses on that jurisdiction. The default value of this index is zero, which indicates the economy is growing at its historical trend rate of growth. A value above one indicates the economy is growing at a rate above this historical trend, and a value below zero indicates the opposite.

In addition to their indicator of overall economic activity, the researchers also included a barometer for sentiment. Unlike the CFNAI, there are no standardized measurements for investor sentiment, and it is something which is notoriously difficult to accurately capture. The authors based their sentiment indicator on Baker and Wurgler's 2006 study *Investor Sentiment and the Cross Section of Stock Returns.*

Briefly touching on their model, a neural network is a machine learning model which is loosely modelled on the human brain, hence the name. The mechanics of these algorithms are very complicated. However, it is not necessary to fully grasp its mechanics to understand their usefulness and applications. Briefly, a so-called "neuron" takes in an input and uses mathematical transformation to produce an output. There are many connections between multiple neurons which are weighted, which then defines the strength of the inputs.

More specifically, the neural network they used was a feedforward neural network. In this type of neural network, the information moves in a single direction. The term feedforward refers to the fact that there are no feedback connections or loops in the network. A feedforward neural network consists of three parts: an input layer, a hidden layer, and an output layer. The input layer receives the input data, and each so-called node represents a feature or attribute. The hidden layer resides between the input and output layers. The input data is then mathematically transformed, and a weighted sum of the inputs is applied to each node followed by an activation function which introduces non-linearity. Finally, the output layer is what the model then produces as an estimate.

One of the advantages of feedforward neural networks is their ability to automatically learn and extract features from the data. Additionally, feedforward neural networks are often used when analyzing financial data since they are good at identifying relationships between non-linear data. Such algorithms are also very good at feature extraction, which reduces the need for manual tuning. The models are especially effective on large datasets with properly labeled data. If set up correctly, they are known for having a high degree of predictive accuracy, especially with large datasets and many inputs.

Kaniel, Pelger, and Van Nieuwerburgh identified fund flow and fund momentum as the two most important predictors of risk-adjusted performance. They also found that the characteristics of the stocks that funds hold are not inherently predictive. This is in line with the findings of Gil-Bazo et al., who indicated that fund characteristics alone are enough to predict performance. Another finding of their study is the interaction between sentiment and both fund flow and fund momentum.

This is an interesting finding that has several potential interpretations. This arguably suggests that manager skill is not a relevant factor in mutual fund returns. Since the main factors affecting returns are fund flow and fund momentum, which are influenced by sentiment, it suggests the factors affecting return are out of the manager's control. However, they did caveat this by stating there is not much evidence that sentiment on its own has an outsized impact on return, rather it is useful as an input for a larger model.

Using their model, they were able to accurately identify outperformers and predict returns. They also found that fund flow, fund momentum, and sentiment alone were nearly as good at

predicting returns as the full model. This suggests a rather interesting dichotomy. What makes these findings so interesting is there are multiple ways one could interpret this. On the one hand, you could argue that since momentum and flow are related to the overall economy, it is simply a good economy and the positive sentiment associated with it that drives returns. On the other hand, you could argue that positive sentiment leads investors to want to invest, and they put their money in the funds they feel will perform best. In this sense, investors are potentially recognizing the skill of the manager, at least during that specific time period, as being superior and therefore driving flow which then in turn reinforces momentum.

Another interesting conclusion which highlights the effectiveness of their model was the return that it generated. If an investor were to purchase a weighted portfolio of the top decile of funds the model predicted monthly, the investor could generate an abnormal return of 72% over their sample period. This is of course an interesting finding which suggests that these models are not only highly accurate, but also have theoretical usability in practice.

Certainly, this is not a reasonable strategy for the average investor, but this is also not what the researchers set out to do. The goal of the demonstration was to show the potential that this type of modelling has. This is another key takeaway from this study, namely that there is massive potential for the application of these algorithms in investment decision making. They further found that the algorithms are predictable in real time for a period of three years. Essentially, they can accurately forecast three years into the future given past data up to the point where the prediction is made.

Additionally, they found that the top performing funds had similar fee structures as the worst performing funds. Oftentimes, proponents of actively managed funds will justify higher expense structures by arguing their superior abilities that demand higher fees. This study clearly finds this notion should at the very least be taken with a grain of salt.

Another interesting finding from this study is that much of the benefits from using such a model come in the form of avoiding the worst performing funds. This presents a very interesting use case for advisors and asset managers. If they are interested in a fund, they can use a negative screening approach to test whether they should analyze the fund further. Knowing what to avoid is often just as important as knowing what to include in your portfolio.

There are several key takeaways from both the Gil-Bazo et al. and Kaniel et al. studies. What both studies empirically prove is that machine learning algorithms can accurately predict mutual fund returns. They also prove that funds recommended by these various algorithms can generate market beating returns, net of all fees. Not only that, but that there are several different algorithms and approaches which are able to make accurate predictions. Both studies also find that outperformance is relatively short lived. However, despite the outperformance being relatively short-lived, there is also evidence investors can take advantage of these alpha generating funds to earn abnormal returns.

Both studies focus heavily on ex post return. Researcher Yu Xia in 2021 published the study *Real-Time Predictability of Mutual Fund Performance Indicators*. Xia notes that ex ante predictability is lower than in sample predictability, but still useful for portfolio construction. Xia also notes that many precious studies assume shorting as an option, which is unrealistic for normal investors and many mutual funds' shares simply cannot be shorted in practice. While this study was published in 2021, prior to Gil-Bazo et al. and Kaniel et al., it further reinforces their findings.

Xia used two approaches for building their models, namely rule-based portfolio sorting and regression-based machine learning methods. Their goal was to build as robust a model as possible while mitigating the potential of model error or building a model which is more complex than it needs to be. For the portfolio sorting approach, Xia adapts the recursive two-way approach outlined in Cooper et al. (2005).

A good way to think about the recursive two-way approach is to imagine several funnels. Specific criteria are then applied as the funds filter through the funnels to build more specific groups based on certain characteristics. This is an iterative process which is done multiple time to create several subgroups among the data. It is essentially what the name implies: it sorts the funds based on rules, not dissimilar to a decision tree.

In the case of Xia, they used this sorting methodology to build portfolios of the best performing funds in a sample period to create a portfolio in "real time" for the following year. To train this model, a six-year expanding window approach is used. The model is then expanded one year at a time after the initial six-year period. Another, albeit very similar, approach is also used by considering fractions of rules instead of using one rule for portfolio selection.

Since mutual fund are themselves diversified, using multiple rules can be more informative to capture a fund manager's skill over time. For the second model with multiple rules, the initial training sample is only five years. These two methods are not necessarily what some would classify as strictly machine learning but follow similar principles.

For the named machine learning models, Xia uses six different regression-based machine learning methods which can be split between penalized linear and dimension reduction. Briefly, penalized linear models are designed to address the limitations of traditional linear regression, such as overfitting and high variance by imposing a penalty term. Dimension reduction is a similar concept which attempts to reduce the number of variables while preserving as much of the relevant information as possible. It is a way to address the common "curse of dimensionality," which refers to many of the challenges faced when working with high-dimensional data.

The raw data used for training, much like the other studies on this topic, was gathered from the CRSP survivor-bias-free mutual fund database. The sample ranged from 1994 to 2016. The training methodologies were also like those of Gil-Bazo et al. and Kaniel et al. Xia uses 12 input variables such as momentum, fund flow, Carhart alpha etc.

Xia finds that the best predictor of next month's return is the prior month's return. They also find that both the regression-based machine learning models as well as the rules-based portfolio construction approach are effective at predicting return. The study shows that these models can be theoretically implemented ex ante to generate alpha. Furthermore, Xia argues this alpha is the compensation for the investors which create these models, since these models are not easy to implement or maintain. At the time of writing, these modelling techniques are still very new, and many asset/fund managers lack the expertise to implement these models themselves.

As the literature shows, machine learning's applications in finance are clearly vast. It will be incredibly interesting to see the different directions the research into these topics goes. Thanks to the existing research on this topic, several things are now much clearer. Firstly, machine learning algorithms are effective tools for return prediction and portfolio construction. Secondly, some actively managed mutual funds can achieve statistically significant outperformance, but this outperformance is mostly short lived. Thirdly, CRSP data and the inputs this database provides can be effectively used to train and validate models. Fourthly,

there are multiple machine learning algorithms that can be effectively used to accurately predict performance ex post. Fifthly, there is at least some evidence skill plays some role in the stretches of mutual fund outperformance. Finally, there is growing evidence that these algorithms can also be applied in real time to predict performance.

4. Overview of Experiment

The goal of our experiment was to test the effectiveness of the machine learning algorithms' ability to predict the future return of actively managed mutual funds. We define effectiveness as the model which minimizes error terms as much as possible. We focused our research on traditional OLS regression models, gradient boosting models, and random forest models. Unlike much of the previous research, our goal is not to attempt portfolio construction. Our sole focus is return prediction. To achieve this, we test whether there is a statistically significant difference in predictive capability between the algorithms. If there is, we reject the null hypothesis that there is not a statistically significant difference in predictive power.

Since our goal is to glean insights into the overall predictive significance of these models, we collected performance metrics with two separate methodologies for measuring return. We trained and tested our models on data where the target variable was the alpha_FF5MOM. Additionally, we also trained and tested our models on the same data but used t_alpha_FF5MOM as the target variable. By using two strict measures of outperformance, we can better evaluate the effectiveness of the models.

4.1 Metrics

To conduct our analysis, we first had to gather various metrics to compare the algorithms with. We began by collecting statistics on the R-squared, adjusted R-squared, mean squared error (MSE), mean absolute error (MAE), and the mean absolute percentage error (MAPE). Briefly, the MSE is a common metric used to evaluate the performance of regression models. It measures the average squared difference between the predicted values and the actual values of the target variable. It is calculated by taking the sum of the squared residuals and dividing it by the number of data points. We chose this metric because it is generally very good for comparing regression-based models.

However, one of the disadvantages of MSE is its sensitivity to large outliers. Since we are dealing with a massive dataset of financial returns, there will naturally be many outliers which need to be adjusted for.

We also include the MAPE as an evaluation metric as well. The main advantage of the MAPE is its interpretability. As the name suggests, it measures the average error as a percentage. This makes it easier to understand the relative magnitude of a model's accuracy. It also serves as a good barometer for standardizing comparisons between different models, since the result is quoted in percentage terms. However, it suffers from a similar drawback as MSE since it is susceptible to the disproportionate impact of outliers.

In a similar vein, we included the MAE as well. This metric involves taking the average of the absolute differences between the predicted values and the actual values. It directly represents the average absolute deviation between predicted and actual values. Additionally, MAE is less susceptible to the disproportionate influence of outliers since it considers the absolute differences. Additionally, we are not interested in the direction of the errors, only their magnitude. For this, the MAE shines as a prediction metric.

Finally, we also included the R-squared and adjusted R-squared. Both models are measures of goodness of fit. They indicate the proportion of the variance in the dependent variable which is explained by the independent variable. It is measured on a scale from zero to one, with a higher number indicating better accuracy. It is calculated by first finding the total sum of squares, which involves taking the squared differences between each observed value of the dependent variable and the mean of the dependent variable. The regression model is fit, usually with statistical software, but could also theoretically be done by hand for small datasets. The residual sum of squares is calculated by taking the squared difference between the actual and the predicted value. Finally, the R-squared is calculated by subtracting one from the residual sum of squares divided by the total sum of squares.

The adjusted R-squared is a modified version of the traditional R-squared metric which also takes the number of predictors and the sample size of the regression model into account. The idea behind the adjusted R-squared is to essentially penalize the inclusion of additional predictors in a model that do not significantly contribute to improving the model's overall goodness of fit.

Both metrics are some of the most commonly used for comparing machine learning models. They are great for understanding the goodness of fit but are prone to errors with large and complex datasets. This is why we have decided to incorporate a myriad of different metrics for comparing our algorithms.

We subsequently used these metrics as datapoint to conduct statistical tests to evaluate whether there is a difference in predictive ability. To do this, we first must assess whether the residuals of each model are normally distributed. For large sample sizes, the assumptions of normality are not as important as with smaller sample sizes, but it is still relevant for post-hoc analysis.

For example, two of the most used tests for evaluating model performance are t-tests and analysis of variance (ANOVA) tests. Both tests assume normality in the residuals and a lack thereof can decrease the accuracy of the tests. Additionally, violations of normality in some tests can lead to distorted p-values in certain tests, leading to highly inflated/deflated p-values which give inaccurate results.

To test for normality in the residuals, we took two approaches. Firstly, we created histograms of the residuals. Histograms are a very useful first step since they are easy to analyze and don't require much computing power to generate. One drawback of histograms is their potential for false interpretations. If the number of bins on a histogram are not properly tuned, this can lead to false assumptions of normality. For this reason, we started with 60 bins, then continually reduced them over several iterations to see how the plots developed.

Another test we used to evaluate normality are quantile-quantile plots (Q-Q plots). The idea behind a Q-Q plot is relatively straightforward. The plot is meant to show the degree to which a dataset follows a theoretical distribution. In our case, this is a normal distribution. It divides the data into quantiles and plots the observed data against the theoretical distribution. In a normally distributed dataset, the observed data would follow a straight line along the theoretical data. Deviations from the theoretical line indicate that the residuals do not follow a normal distribution.

Once we have determined the metrics and assessed the normality, we then move on to conducting the tests which check for differences in predictive ability. Before we discuss our findings, we first discuss our approach to cleaning and preparing our dataset.

4.2 Data cleaning

The first step to performing any sort of data analysis is cleaning and preparing the dataset. The original dataset we were provided was a 1.8 GB behemoth with close to five million rows and about 50 columns. The dataset contains various information about mutual funds from the CRSP financial database. The data stretches back from 1980, all the way through 2022. To get to a point where we could run our tests, several data preparation steps needed to be employed.

```
#import .txt and packages to python
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
from tqdm import tqdm
import sklearn
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
import scipy.stats as stats
from sklearn.metrics import mean_absolute_error


df = pd.read_csv("/Users/yannickarfaoui/dataset_module_10.txt", delimiter=" ", low_memory=False)
```

Figure 1, (code) import packages and data set.

The first step of the data cleaning process was to import the different packages that are used to deal with data frames in python. Additionally, we imported the HSLU dataset to the Spyder Python environment as a data frame to begin the formatting. The dataset we were provided is in a tabular format and was saved as a *.txt* file.

```
#Drop blanks from FUND_NAME
df = df.dropna(axis=0, subset=['FUND_NAME'])
```

Figure 2, (code) dealing with NaN in Fund Name.

The next step was to begin removing incoherent data from the data frame. We started by dropping the empty cells (NaN) in the column "FUND_NAME," using the function *.dropna*. If the name of the fund is empty, we cannot reliably assume that the other inputs are accurate and useable.

```
#Set FRONT and REAR Loads NaN entries to zero
values = {'REAR_LOAD': 0, 'FRONT_LOAD': 0}
df = df.fillna(value=values)

#Only funds with FRONT_LOAD & REAR_LOAD of zero
df = df[df.FRONT_LOAD == 0]
df = df[df.REAR_LOAD == 0]

#Drop FRONT_LOAD and REAR_LOAD Columns
df = df.drop(['FRONT_LOAD', 'REAR_LOAD'], axis=1)
```

Figure 3, (code) dealing with Rear and Front load.

We were also only interested in analyzing funds which had no front or rear load fees. We also assume an entry of NaN to be zero. Therefore, the first step was to set the NaN values in the "FRONT_LOAD" and "REAR_LOAD" columns to zero. We did this using the *.fillna* function which allowed us to apply a condition to only keep the share classes charging no fees. Once all the rows containing a value other than zero were dropped, we deleted the columns "FRONT_LOAD" and "REAR_LOAD" using the function *.drop* as the columns were no longer necessary.

```
#Only funds over 35 months in age
df = df[df.Age > 35]
```

Figure 4, (code) dealing with share classes' age.

Additionally, our analysis only focused on share classes that had existed for more than 35 months. Therefore, we used a simple conditional statement to filter out these funds. We applied the filter to our data frame, keeping only the rows containing a number bigger than 35 in the column "Age".

```
#Only funds over 70% in Equities
df = df[df.ALL_EQ > 70]
```

Figure 5, (code) dealing with Equity size.

An additional condition for our analysis was to only evaluate funds that invested over 70% of its capital in equities. With the use of a conditional statement, we filtered out the irrelevant rows from our data frame to keep only the rows containing a value higher than 70 in the column "ALL_EQ".

```
#US funds filter step 01 according to CRSP documentation
df = df[df['CRSP_OBJ_CD'].str.startswith('ED', na = False)]
#Filter out funds with hedge fund strategies according to CRSP documentation
hedge_strategies = ['EDYH', 'EDYS']
df = df[~df['CRSP_OBJ_CD'].isin(hedge_strategies)]
```

Figure 6, (code) dealing with U.S equity.

Much like the studies outlined in our literature review, our analysis only focused on share classes which invest exclusively in the U.S. equity market. The CRSP documentation shows that in the column "CRSP_OBJ_CD," the strings starting with the letters ED represent share classes focusing on domestic equity. To get rid of the rows not starting with those two letters, we used the function *.str.startswith* to only keep the corresponding rows.

Furthermore, two additional styles of equity funds needed to be filtered out, even though they start with the right letters. Those were the share classes using hedging or short investment styles. To drop the share classes using these styles, we created a variable "hedge_strategies" which contained the undesired classifiers ("EDYH" & "EDYS"). We then use the function *~df … .isin* to only keep the rows where the variable were not present.

```
#Create a copy to avoid potential errors with original dataframe
df1 = df.copy()
#Filter out all index funds
#----------------------------------------#
#Convert the values in the CLADT to datetime
df1.loc[:, 'CALDT'] = pd.to_datetime(df['CALDT'])
#Now create the two dataframes before and after the 'D' index was introduced
df1_1 = df1[df1['CALDT'] < pd.to_datetime('2003-06-30')]
df1_2 = df1[df1['CALDT'] >= pd.to_datetime('2003-06-30')]
```

Figure 7, (code) dealing with actively managed funds (1).

Our analysis solely focuses on funds which are actively managed. CRSP provides a convenient identifier to denote the type of fund, i.e. whether the fund is an index fund or actively managed in some way. This data is found in the "INDEX_FUND_FLAG" column. However, this demarcation has only been available since June 30th, 2003.

We therefore had to first split our data into two separate data frames: one containing the information prior to June 30th, 2003, and one containing all the data afterwards. To properly

separate the data, we first must convert the "CALDT" column from a string to a datetime format. We then use the *.to_datetime* function which allows us to make the conversion. Once we do this, we can then easily separate the data frames for further filtering.

```
#Filter out index funds from dataframe which has index fund flag
df1_3 = df1_2[df1_2.INDEX_FUND_FLAG != 'D']
#Filer out index funds prior to index fund flag
dfx = df1_1.drop(df1_1[df1_1.FUND_NAME.str.contains('Index')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Idx')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Ix')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Indx')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('NASDAQ')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Nasdaq')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Dow')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('Mkt')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('DJ')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('S&P 500')].index)
dfx = dfx.drop(dfx[dfx.FUND_NAME.str.contains('BARRA')].index)
#Concatenate the two dataframes back together
df1 = pd.concat([dfx, df1_3], axis=0)
```

Figure 8, (code) dealing with actively managed funds (2).

After creating the two subset data frames, we can now filter out the index funds. According to CRSP documentation, index funds are denoted with the classifier of "D" in "INDEX_FUND_FLAG" data. Therefore, we can easily filter out the index funds by only including rows that do not contain "D".

For the "younger" subset, the process is slightly more complex. Luckily, we are not the first to face this same conundrum. To further filter the data, we employed the methods of Gil-Bazo et al. from their 2009 paper *The relation between Price and Performance in the Mutual Fund Industry*. To filter out index funds, they deleted all the rows containing one of these terms in the "FUND_NAME" column: 'index', 'Idx', 'Ix', 'Indx', 'NASDAQ', 'Nasdaq', 'Dow', 'Mkt', 'DJ', 'S&P 500', 'BARRA'. We did this using the functions *.drop* and *.str.contains.*

After successfully filtering out the index funds from the two subset data frames, we can concatenate them back together. We set the axis value of the *.concat* function to '0', indicating we want to concatenate by rows. We are able to do this because both subset data frames have the same columns.

```
#Filter out fund rows prior to them reaching 5 million in assets under management
#----------------------------------------#
#Create a list with dataframes for each fund
portfolio_NO = df1["CRSP_FUNDNO"].unique()
list_filtering = [df1[df1["CRSP_FUNDNO"] == i].reset_index(drop = True) for i in tqdm(portfolio_NO)]

#Delete observations
portfolios_sample = []

for df_ in tqdm(list_filtering):
    tmp = df_
    if tmp["MTNA"].max() >= 5:
        mask = tmp[tmp["MTNA"] >= 5]
        portfolios_sample.append(tmp.iloc[mask.index[0]:])

#Combine sample again
sample = pd.concat(portfolios_sample, axis = 0)
#Make a copy of sample as df15
df2 = sample.copy()
```

Figure 9, (code) dealing with Total Net Asset.

We are also only interested in share classes from the point they reach $5 million in total net asset. Any data relating to a share class prior to reaching this point can therefore be excluded. However, a fund can conceivably reach a net asset value of $5 million, then fall back below

that threshold. Therefore, we had to filter funds from the point they first reach the $5 million mark.

We decided to filter by fund number. The first step of this process was to create a variable called "portfolio_NO" which contains a unique series of every fund number. Using the function *for i in tqdm*, we created a comprehension list iterating over each element *i* contained in the variable "portfolio_NO." This allows us to create a list ("list_filtering") that contains one data frame for each unique fund number and its corresponding rows. We also reset the index of each list to have chronological data allowing us to find the first time a share class reached $5 million in total net assets.

In the next step we create a *for* loop which iterates over every value of the "list_filtering" variable. It then applies an *if* function which checks if the maximum value of the current data frame is bigger or equal to 5. The data is denoted in millions. The next line masks values bigger than 5.

The subsequent line allows us to select a subset of rows starting from the index of the first row that satisfies the condition until the end of the data frame. This is then appended to the previously created empty array "portfolios_sample." We can then create the complete data frame by concatenating all the values of the "portfolios_sample" array into a new variable called "sample" which we then rename as "df2."

4.3 Data preparation

Once we had cleaned the data sufficiently, we then had to further modify our dataset to be suitable for training and testing our machine learning models. Some additional data needed to be added, and further inputs needed to be removed. We also had to split the data into a training and a testing subset. We detail the different steps followed to create our models.

```
#Create boolean columns which return whether fund is managed by team or single manager
df1 = df1.reindex(df1.columns.tolist() + ['IS_TEAM','NOT_TEAM'], axis=1)
#Create dummy variable for IS_TEAM
df1['IS_TEAM'] = np.where(df1['MANAGER_MSTAR'].str.contains('/', na = False) == True, 1, 0)
#Create dummy variable for NOT_TEAM
df1['NOT_TEAM'] = np.where(df1['IS_TEAM'] == 1, 0, 1)
#Drop MANAGER_MSTAR column
df1 = df1.drop('MANAGER_MSTAR', axis=1)
```

Figure 10, (code) creating dummy variables for team of managers vs. single manager.

The first step consisted in adding a new column to differentiate between the share classes managed by a single manager versus a team. Even though this process was shown in earlier excerpts of our code, it belongs to this chapter and does not have any impact on previous steps. To achieve this, we first create a list containing all the column names and then add two new names to the list to differentiate between team and single manager.

By default, these two columns are initially filled with NaN values. We then assign the value "True" in the column "IS_TEAM" if the column "MANAGER_MSTAR" contains the string "/" using the functions *np.where* and *.str.contains.* In the "MANAGER_MSTAR" column, the names of managers are separated by a backslash. Therefore, if a row contains a backslash, the fund is managed by more than one manager.

Next, we assign the value 1 when the value of "IS_TEAM" is equal to True and 0 if not. We are then able to assign a value of 0 in the column "NOT_TEAM" when the value of "IS_TEAM" is 1 and vice versa. We can then remove the column "MANAGER_MSTAR." This step allows

us to create dummy variables to assess if a single or team management structure has an impact on return.

```
#Data preparation for ML models
#Drop unnecessary columns
drop_columns = ['FUND_NAME', 'INDEX_FUND_FLAG', 'ET_FLAG', 'CRSP_OBJ_CD', 'flows', 'FIRST_OFFER_DT',
'TERMINATION_DT']
df2 = df2.drop(columns=drop_columns)

#Sort the dates column from lowest to highest
df2 = df2.sort_values(by = 'CALDT')
```

Figure 11, (code) dropping unnecessary columns.

Next, we created a variable containing an array of the columns we deemed unnecessary for our modelling purposes and wanted to drop. In the next line we drop all the columns contained in the previously mentioned list using the function *.drop*.

To create our training and testing subsets, we wanted to split the data based on dates. Our first step was therefore to chronologically sort our data using the function *.sort_values* on the column "CALDT" previously converted to datetime values.

```
#Shift the t_alpha_FF5MOM back by one month for prediciton
df2['alpha_FF5MOM_shifted'] = df2.groupby('CRSP_FUNDNO')['alpha_FF5MOM'].shift(-1)
```

Figure 12, (code) shifting target variable for training the models.

For our training and testing purposes, we first defined our target variable as the raw "alpha_FF5MOM." As mentioned, we also did another analysis where the target variable was the "t_alpha_FF5MOM." Since including another overview of this code is redundant, we show the initial versions with "alpha_FF5MOM" as the target variable. For the second model, these metrics were simply switched, no other code was changed.

The next step was to shift all the data present in the column "alpha_FF5MOM" by -1 for training. To do so, we first started by grouping the data frame by their fund number, then using the function .shift(-1), we shifted all the "alpha_FF5MOM" values back by one period (one month).

This process is crucial since with our machine learning programs, we try to build a model that predicts the alpha at time *t*. We want to be able to predict the alpha at *t* also using the inputs at time *t*. In order to do this, we shift the target variable back to *t-1* because the models are all used to predict the value at *t+1*, which in our case would be *t*. Assuming that past data has an effect on the current values, we can train a model with a lagged target variable and then we compare the predicted outputs to the actual outputs at time *t*.

```
#Drop NaN values from dataframe for linear regression
df2 = df2.dropna()
```

Figure 13, (code) drop NaN for the models.

We need to also ensure the data is as robust as possible. To do this, only include rows where the data is complete for all the input variables. Including any NaN values would ultimately confuse the training of our models and make them less precise. Now that we have made all the necessary adjustments, we simply drop the NaN values from the data frame.

```
#Split data into training and test data
df2_1 = df2[df2['CALDT'] < pd.to_datetime('2015-01-01')]
df2_2 = df2[df2['CALDT'] >= pd.to_datetime('2015-01-01')]

#Drop CLADT columns from the new dataframes
training_data = df2_1.drop(['CALDT'], axis = 1)
testing_data = df2_2.drop(['CALDT'], axis = 1)
```

Figure 14, (code) splitting the data frame into training and testing data.

Machine learning models need to be trained and tested, convention suggests that 70% to 80% of available data be used to train and the remaining to test the models. To split our data into one training subset and one testing subset we used the column "CALDT" and assigned the data before January 1st 2015 to a new data frame "df2_1" and the data after or equal to January 1st 2015 do a new data frame "df2_2". Those two new data frames were then renamed to "training_data" and "testing_data" respectively.

```
#Create the funcion to test for normality in the residuals
def plot_residuals(y_true, y_pred):
    residuals = y_true - y_pred

    #Plot histogram of residuals
    plt.hist(residuals, bins=20)
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.title('Histogram of Residuals')
    plt.show()

    #Plot Q-Q plot of residuals
    stats.probplot(residuals, dist="norm", plot=plt)
    plt.title('Q-Q plot of Residuals')
    plt.show()
```

Figure 15, (code) defining plot functions.

As previously mentioned, we want to assess the normality of the residuals. To this, we created a call back function which created both the histogram and Q-Q plots. After several iterations, we decided on 20 bins as sufficient for our purposes. We simply set up and define the parameters of the plots according to the documentation.

```
#mean absolute percentage error function
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true))
```

Figure 16, (code) defining an indicator function.

Similarly, we created a second call back function "mape" to calculate the absolute percentage error between the true alpha (y_true) and the predicted alpha (y_pred). The function is specified using the MAPE formula.

```
#Define independent and dependent variables
x_train = training_data.drop('alpha_FF5MOM_shifted', axis = 1)
y_train = training_data['alpha_FF5MOM_shifted']
x_test = testing_data.drop('alpha_FF5MOM_shifted', axis = 1)
y_test = testing_data["alpha_FF5MOM"]
```

Figure 17, (code) defining independent & dependent variables for the training and the testing of the models.

Finally, four variables were created to be used in the different models. The "x_train" and "y_train" data will be used to train the models and the "x_test" and "y_test" will be used to test the models for their accuracy.

4.4 Machine Learning Models

In this section, we give a brief overview of the different models we selected, and the code used to set them up. We used one gradient boosting model, three separate OLS regression models, and three separate random forest models. We chose these models as they were often used in previous research and shown to perform well.

4.4.1 Linear Regression Model (OLS)

```
#Linear Regression Model
#----------------------------------------#
#Create regression object
reg_OLS = LinearRegression()

#Fit reg to the data
reg_OLS.fit(x_train, y_train)

#Print model coefficients
print('Coefficients: ', reg_OLS.coef_)
print('Intercept: ', reg_OLS.intercept_)

#Predict the target variable on the testing data
y_pred_OLS = reg_OLS.predict(x_test)

#Compute the mean squared error between the true and predicted values
mse_linear_regression_unadjusted = mean_squared_error(y_test, y_pred_OLS)

#Calculate R-Squared value for testing data
r2_linear_regression_unadjusted = reg_OLS.score(x_test, y_test)

#Calculate adjusted R-squared value for testing data
n = len(y_test)
k = x_test.shape[1]
adj_r2_linear_regression_unadjusted = 1 - (1 - r2_linear_regression_unadjusted ) * ((n - 1) / (n - k - 1))

#Plot residuals for Linear Regression Model
plot_residuals(y_test, y_pred_OLS)

#Return mape score
mape_linear_regression_unadjusted = mape(y_test, y_pred_OLS)

#mean absolute error
mae_linear_regression_unadjusted = mean_absolute_error(y_test, y_pred_OLS)
```

Figure 18, (code) Linear Regression Model (OLS).

The first model used was a linear regression model using the ordinary least squares (OLS) method with all the input variables. The first step to implement this model was to create a "reg_OLS" variable assigning it the *LinearRegression()* function. The next step was to fit the training data to the OLS model, which was achieved using the built-in function *.fit*. To assess the value of the coefficients and the intercept, we print them using the function *print* as well as *.coef_* and *.intercept_*. We then test the model on our variable called "x_test" and using the function *.predict.*

From this point call the functions to collect the evaluation metrics. The first used was the mean squared error ("mse_linear_regression_unadjusted") using the *mean_squared_error* function on y_test and y_pred_OLS. The second the coefficient of determination or R^2 ("r2_ linear_regression_unadjusted") using the function *.score* on x_test and y_test.

The third was the adjusted coefficient of determination or adjusted R^2 ("adj_r2_linear_regression_unadjusted"). To be able to calculate this indicator, the size of the sample (*n*) had to be determined using the function *len()* on y_test. As well as the number of independent variables used in the OLS model (*k*), using the function *.shape[1]* on x_test. Once *n* and *k* are calculated, we can use *1 - (1 - r2_linear_regression_unadjusted) * ((n - 1) / (n - k - 1))* to calculate the adjusted R^2.

We can then estimate use the previously defined function *plot_residuals* to produce the histogram of residuals and the Q-Q plot. Next, the average percentage error is also calculated using the previously defined function *mape*.

Finally, the mean absolute error is computed using the *mean_absolute_error* function on "y_test" and 2y_pred_OLS".

4.4.2 Linear Regression Model Adjusted (OLS)

```
#Linear Regression Model - Adjusted
#----------------------------------------#

#independent variables for linear regression
drop_columns2 = ["alpha_FF5MOM_shifted", "ALL_EQ", "flows_wins", "flows_vola", "excess_return",
"vola_return","alpha_FFC", "t_alpha_FFC", "value_added", "FUND_TNA"]
x_train_OLS = training_data.drop(columns=drop_columns2)
x_test_OLS = testing_data.drop(columns=drop_columns2)

#Create regression object
reg_OLS_2 = LinearRegression()

#Fit reg to the data
reg_OLS_2.fit(x_train_OLS, y_train)

#Print model coefficients
print('Coefficients: ', reg_OLS_2.coef_)
print('Intercept: ', reg_OLS_2.intercept_)

#Predict the target variable on the testing data
y_pred_OLS_2 = reg_OLS_2.predict(x_test_OLS)

#Compute the mean squared error between the true and predicted values
mse_linear_regression_adjusted_01 = mean_squared_error(y_test, y_pred_OLS_2)

#Calculate R-Squared value for testing data
r2_linear_regression_adjusted_01  = reg_OLS_2.score(x_test_OLS, y_test)

#Calculate adjusted R-squared value for testing data
n = len(y_test)
k_OLS = x_test_OLS.shape[1]
r2_adj_linear_regression_adjusted_01 = 1 - (1 - r2_linear_regression_adjusted_01) * ((n - 1) / (n - k_OLS - 1))

#Plot residuals for Linear Regression Model
plot_residuals(y_test, y_pred_OLS_2)

#Return mape score
mape_linear_regression_adjusted = mape(y_test, y_pred_OLS_2)

#mean absolute error
mae_linear_regression_adjusted = mean_absolute_error(y_test, y_pred_OLS_2)
```

Figure 19, (code) Linear Regression Model Adjusted (OLS).

The set up of the second OLS regression model was done the same as in the first model. In this second model, we attempted to address the issue of overfitting by testing a new combination of independent variables. To do this, we assigned it a new training data set removing the columns "alpha_FF5MOM_shifted", "ALL_EQ", "flows_wins", "flows_vola", "excess_return", "vola_return","alpha_FFC", "t_alpha_FFC", "value_added", "FUND_TNA", on the the independent variables used to train and test the model (x_train_OLS and x_test_ols respectively).

The steps used to calculate the relevant metrics are all the same as with the previous OLS model.

4.4.3 Linear Regression Model Adjusted 2 (OLS)

```
#Linear Regression Model - Adjusted 02
#----------------------------------------#

#independent variables for linear regression
keep_columns = ["alpha_FF5MOM_shifted", "ALL_EQ", "flows_wins", "alpha_FF5MOM", "flows_vola", "excess_return",
"vola_return","alpha_FFC", "t_alpha_FFC", "value_added", "FUND_TNA"]
#for new regression
df_new_reg_train = training_data.loc[:, keep_columns]
df_new_reg_test = testing_data.loc[:, keep_columns]

x_train_OLS_adj_02 = df_new_reg_train.drop("alpha_FF5MOM_shifted", axis = 1)
y_train_OLS_adj_02 = df_new_reg_train["alpha_FF5MOM_shifted"]
x_test_OLS_adj_02 = df_new_reg_test.drop('alpha_FF5MOM_shifted', axis = 1)
y_test_new_reg = df_new_reg_test["alpha_FF5MOM"]

#Create regression object
reg_OLS3 = LinearRegression()

#Fit reg to the data
reg_OLS3.fit(x_train_OLS_adj_02, y_train_OLS_adj_02)
#Print model coefficients
print('Coefficients: ', reg_OLS3.coef_)
print('Intercept: ', reg_OLS3.intercept_)

#Predict the target variable on the testing data
y_pred_OLS3 = reg_OLS3.predict(x_test_OLS_adj_02)

#Compute the mean squared error between the true and predicted values
mse_lin_regression_adjusted_02 = mean_squared_error(y_test_new_reg, y_pred_OLS3)

#Calculate R-Squared value for testing data
r2_lin_regression_adjusted_02 = reg_OLS3.score(x_test_OLS_adj_02, y_test_new_reg)

#Calculate adjusted R-squared value for testing data
n = len(y_test_new_reg)
k = x_test_OLS_adj_02.shape[1]
adj_r2_tlin_regression_adjusted_02 = 1 - (1 - r2_lin_regression_adjusted_02) * ((n - 1) / (n - k - 1))

#Plot residuals for Linear Regression Model
plot_residuals(y_test_new_reg, y_pred_OLS3)

#Return mape score
mape_lin_regression_adjusted_02 = mape(y_test_new_reg, y_pred_OLS3)

#mean absolute error
mae_lin_regression_adjusted_02 = mean_absolute_error(y_test_new_reg, y_pred_OLS3)
```

Figure 20, (code) Linear Regression Model Adjusted 2 (OLS).

Lastly, we created a third OLS regression model using the same basic set up as the previous two. We again tried to address the issues of overfitting by cutting down on the number of independent variables. As opposed to the previous model, this model uses only the columns "alpha_FF5MOM_shifted", "ALL_EQ", "flows_wins", "alpha_FF5MOM", "flows_vola", "excess_return", "vola_return","alpha_FFC", "t_alpha_FFC", "value_added", "FUND_TNA" to train and test the data.

The steps followed to achieve this model are the same as the previous models, the only difference are that the variables "x_train_OLS_adj_02", "y_train_OLS_adj_02", "x_test_OLS_adj_02", "_test_new_reg" instead of "x_train", "y_train", "x_test", and "y_train". This was done to not "confuse" the call back functions and prevent the previous variables from being overwritten.

4.4.4 Gradient boosting model

```python
#----------------------------------------#
# Gradient Boosting with hyperparameters tunning:
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.model_selection import GridSearchCV

# Define the parameter grid to search over
param_grid = {
    'learning_rate': [0.01, 0.1, 1],
    'max_iter': [100, 500, 1000],
    'max_depth': [5, 10, 50, 100]
}

# Create HistGradientBoostingRegressor object
reg_GradientBoosting = HistGradientBoostingRegressor(random_state=42)

# Use GridSearchCV to search over the parameter grid
grid_search_GradientBoosting = GridSearchCV(reg_GradientBoosting, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search_GradientBoosting.fit(x_train, y_train)

# Get the best model
best_model_GradientBoosting = grid_search_GradientBoosting.best_estimator_

# Fit the best model to the data
best_model_GradientBoosting.fit(x_train, y_train)

# Predict the target variable on testing data
y_pred_GradientBoosting = best_model_GradientBoosting.predict(x_test)

# Compute the mean squared error between the true and predicted values
mse_gradient_boosting = mean_squared_error(y_test, y_pred_GradientBoosting)

# Calculate R-Squared value for testing data
r2_gradient_boosting = best_model_GradientBoosting.score(x_test, y_test)

# Calculate adjusted R-squared value for testing data
n = len(y_test)
k = x_test.shape[1]
adj_r2_gradient_boosting = 1 - ((1 - r2_gradient_boosting) * (n - 1) / (n - k - 1))

#Plot residuals for Gradient Boosting Model
plot_residuals(y_test, y_pred_GradientBoosting)

#Return mape score
mape_gradient_boosting = mape(y_test, y_pred_GradientBoosting)

#mean absolute error
mae_gradient_boosting = mean_absolute_error(y_test, y_pred_GradientBoosting)
```

Figure 21, (code) Gradient Boosting Model.

The fourth model we used was the gradient boosting model using a grid search to tune the hyperparameters. We used this hyperparameter tuning technique to try and create a model which was as accurate as possible.

The first step to create this model was to import two packages from the Scikit-learn library to enable the functions *HistGradientBoostingRegressor* and *GridSearchCV.* Then, the second step was to create a dictionary "param_grid" defining the parameters that will later be used to proceed with the grid search. Lastly, we then created the "reg_GradientBoosting" variable and assign it the function *HistGradientBoostig.*

Once those steps are complete, we can start the grid search process. This process consists of a five-fold cross-validation procedure to try and avoid overfitting. Through this process, the function *GridSearchCV* considers all the hyperparameters defined in the dictionary "param_grid" ("learning_rate", "max_iter", "max_depth") in the function HistGradientBoosting. The training data are then fitted to the model to proceed with the cross-validation to provide us with the hyperparameters with the optimum negative mean square error.

We then want to continue the rest of the code with the model tuned most optimally. We then select the model with the best hyperparameters, storing it in a variable called "best_model_GradientBoosting." Now we can move on to training our best performing model using the variable and fitting it the training data "x_train" and "y_train".

From that point the testing of the model can be initiated. The "y_pred_GradientBoosting" variable is created returning the predicted value of alpha using the "best_model_GradientBoosting" variable and the function *.predict* with the "x_test" data.

To be able to compare the different models that we work with, we then collect the same indicators as the previous models. The first, the mean squared error between the variable "y_pred_GradientBoosting" and the "y_test". Then the coefficient of determination using "x_test" and "y_test" and finally the adjusted coefficient of determination using *n* and *k* defined in the first model.

We can then use the previously defined function *plot_residuals* to draw the histogram of residuals and the Q-Q plot. Using the function *mape* we compute the average percentage error between the variables "y_test" and "y_pred_GradientBoosing". Lastly, we use the built-in function *mean_absolute_error* to compute the mean absolute error between "y_test" and "y_pred_GradientBoosting".

4.4.5 Random Forest Model

```
#----------------------------------------#
#Random Forest
from sklearn.ensemble import RandomForestRegressor

RandFor = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1,
verbose=3)

RandFor.fit(x_train, y_train)

y_pred_RandFor = RandFor.predict(x_test)

mse_random_forest_base_model = mean_squared_error(y_test, y_pred_RandFor)
r2_random_forest_base_model = r2_score(y_test, y_pred_RandFor)
adj_r2_random_forest_base_model = 1- ((1-r2_random_forest_base_model) * (n-1) / (n-
k-1))

print("Mean Squarred Error: " ,mse_random_forest_base_model)
print("R squarred: " , r2_random_forest_base_model)

#Plot residuals for adjusted Random Forest model 01
plot_residuals(y_test, y_pred_RandFor)

#Return mape score
mape_random_forest_base_model = mape(y_test, y_pred_RandFor)

#mean absolute error
mae_random_forest_base_model = mean_absolute_error(y_test, y_pred_RandFor)
```

Figure 22, (code) Random Forest Model.

The fifth model used was the random forest model with no hyperparameters tuning. This is our so-called base case random forest model. The first step to this model was to import the *RandomForestRegressor* function from the Scikit-learn library.

We can then define a new variable "RandFor" and assign it the function *RandomForestRegressor*. In this first Random Forest, we tried a model with 100 trees and a "random_state" of 42 (the 42nd state of shuffled data). The parameter "n_jobs" assigns as many threads available as possible to the computation of the model and the parameter "verbose" allows us to have an overview of the progress of the model computation.

We can then fit the training data to the model using the *.fit* function. The variable "y_pred_RandFor" is created to predict the value of alpha using the "x_test" variable.

From this point, we can proceed with the calculation of the three indicators calculated in every other model (mse, R^2 and adjusted R^2). We are finally able to produce the histogram of residuals and the Q-Q plot as well as the average percentage error and the mean absolute error between the "y_pred_RandFor2" and the "y_test".

However, this model is not as accurate as we would like. Not tuning the hyperparameters creates a generic model that isn't optimised for our data and is prone to serious overfitting. The next models use hyperparameter tuning through grid search and randomized grid search to address this.

4.4.6 Random Forest Model with Grid Search

```
#----------------------------------------#
#Hyperparameters tunning Random Forest with GridSearchCV:

# Define the random forest model with default hyperparameters
RandFor2 = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1,
verbose=3)

# Define the hyperparameter grid to search
param_grid = {
    'max_depth': [10, 30, None],
    'min_samples_split': [2, 10],
    'min_samples_leaf': [1, 4],
    'max_features': [1.0],
    'bootstrap': [False]
}

# Define the grid search object
grid_search_RandFor = GridSearchCV(estimator=RandFor2, param_grid=param_grid, cv=5,
n_jobs=-1, verbose=3)

# Fit the grid search object to the training data
grid_search_RandFor.fit(x_train, y_train)

# Retrieve the best model and its hyperparameters
best_model_RandFor = grid_search_RandFor.best_estimator_
best_params_RandFor = grid_search_RandFor.best_params_

# Use the best model to make predictions on the test data
y_pred_RandFor2 = best_model_RandFor.predict(x_test)

# Calculate the evaluation metrics
mse_random_forest_grid_search = mean_squared_error(y_test, y_pred_RandFor2)
r2_random_forest_grid_search = r2_score(y_test, y_pred_RandFor2)
adj_r2_random_forest_grid_search = 1 - ((1 - r2_random_forest_grid_search) * (n - 1)
/ (n - k - 1))

#Plot residuals for adjusted Random Forest model 02
plot_residuals(y_test, y_pred_RandFor2)

#Return mape score
mape_random_forest_grid_search = mape(y_test, y_pred_RandFor2)

#mean absolute error
mae_random_forest_grid_search = mean_absolute_error(y_test, y_pred_RandFor2)
```

Figure 23, (code) Random Forest Model with Grid Search.

Our second random forest model uses the hyperparameter tuning GridSearchCV method. This is to address the common pitfall of overfit random forest models and allows us to generate a more accurate model that minimizes errors.

To initiate the model, we created a variable "RandFor2" taking into consideration the same hyperparameters as the previous model. We then defined a dictionary "param_grid" containing the hyperparameters to be tunned ("max_depth", "min_samples_split", "min_samples_leaf", "max_features" and "bootstrap"). From that point, the grid search can be initiated, we created a variable "grid_search_RandFor" and assigned it the *GridSearchCV* function with the necessary parameters. The model is trained with the "x_train" and "y_train" data using the *.fit* function.

In the next variable, “best_model_RandFor,” we collect the best model found through the grid search using the *.best_estimator_* function and collect the best hyperparameters in the variable “best_params_RandFor” using the function *best_params_* function.

Once the grid search process is complete, we can on and test our model. We produce a prediction of alpha in the variable “y_pred_RandFor2” using the function *.predict* on the “x_test” data. The final step is then to compute the different indicators (mse, R^2 and adjusted R^2), draw the histogram of residuals and the Q-Q plot, then compute the average percentage error as well as the mean absolute error using the variables “y_pred_RandFor2” and “y_test”.

The problem with this mode is the computing power necessary and the large amount of time needed to perform the grid search as well as the prediction with the random forest. Unlike our gradient boosting model which ran in about 30 minutes, this random forest model took close to four hours to complete and crashed several times.

To counter these issues, a randomized search can be used. This allows us to diminish the computing power needed and the time needed to compute the model by randomizing the grid search instead of going through each possible combination of hyperparameters. The tuning might be less accurate than a normal grid search, however the trade-off is often worth it when working with standard consumer grade machines. Our final model used this very approach.

4.4.7 Random Forest Model with Randomized Grid Search

```
#----------------------------------------#
#Hyperparameters tunning Random Forest with RandomizedSearchCV:

from sklearn.model_selection import RandomizedSearchCV

# Define the random forest model with default hyperparameters
RandFor3 = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1,
verbose=3)

# Define the random search space
random_grid = {
    'max_depth': [10, 30, None],
    'min_samples_split': [2, 10],
    'min_samples_leaf': [1, 4],
    'max_features': [1.0],
    'bootstrap': [False]
}

# Define the random search object
random_search = RandomizedSearchCV(estimator=RandFor3,
param_distributions=random_grid, n_iter=36, cv=5, n_jobs=-1, verbose=3,
random_state=42)

# Fit the random search object to the training data
random_search.fit(x_train, y_train)

# Retrieve the best model and its hyperparameters
best_model = random_search.best_estimator_
best_params = random_search.best_params_

# Use the best model to make predictions on the test data
y_pred_RandFor3 = best_model.predict(x_test)

# Calculate the evaluation metrics
mse_random_forest_ran_grid_search = mean_squared_error(y_test, y_pred_RandFor3)
r2_random_forest_rand_grid_search = r2_score(y_test, y_pred_RandFor3)
adj_r2_random_forest_rand_grid_search = 1 - ((1 - r2_random_forest_rand_grid_search)
* (n - 1) / (n - k - 1))

#Plot residuals for adjusted Random Forest model with RandomizedSearchCV
plot_residuals(y_test, y_pred_RandFor3)

#Return mape score
mape_random_forest_ran_grid_search = mape(y_test, y_pred_RandFor3)

#mean absolute error
mae_random_forest_ran_grid_search = mean_absolute_error(y_test, y_pred_RandFor3)
```

Figure 24, (code) Random Forest Model with Randomized Grid Search.

The seventh and final model used is a random forest model with a randomised grid search. The first step for this mode was to import the function *RandomizedSearchCV* from the Scikit-learn library.

The following steps are essentially the same as those of the previous model. we created a variable "RandFor3," then using the same hyperparameters in the dictionary as the previous model, we proceeded to a randomized grid search using the function *RandomizedSearchCV* where in the previous model the function *GridSearchCV* was used. After fitting the model with the training data, we produced an alpha prediction in the variable "y_pred_RandFor3". We finally computed the different indicators and drew the histogram of residuals and the Q-Q plot.

5. Results

At first glance, there seems to be little difference in predictive power between the random forest and gradient boosting methods for both target variables. However, what is abundantly clear is that all our OLS regression methods suffer from a high degree of overfitting. This isn't necessarily surprising. The data we are using is highly complex and has dozens of variables. Our unadjusted model incorporated all the input variables and was clearly overfit. The adjusted OLS models had much fewer variables, 32 and ten respectively. Despite these adjustments, both models also exhibited a high degree of overfitting as represented by the excessively high R-squared and adjusted R-squared values.

The overfitting is likely caused by the complexity of the data as well as a lack of regularization. We also observe much higher errors than the machine learning models. We likely still included too many irrelevant variables for the OLS models to be useful and not suffer from overfitting.

This demonstrates that these simpler models are not necessary nor effective at capturing the complex relationships between our data. It therefore makes sense to use other available models which can better capture these complex relationships. There was certainly enough training data for the models to be accurate, but our initial analysis suggests there are better options than traditional OLS regression. An overview of our metrics is summarized in the tables below.

Model	Accuray Metric - alpha_FF5MOM				
--	R-Squared	Adjusted R-Squared	Mean Squared Error	Mean Absolute Percentage Error	Mean Absolute Error
Linear Regression - Unadjusted	0.996409328976956	0.996408793073465	2.89563041514652E-08	0.303920420653305	0.000117177700537559
Linear Regression - Adjusted 01	0.996890648816338	0.996890535642021	2.50747890881719E-08	0.303920420653305	0.000107728795889211
Linear Regression - Adjusted 02	0.998835879435462	0.998835837063745	9.38783556593825E-09	0.132528495377751	0.000063274229149925
Gradient Boosting	0.776859274597358	0.776825971102600	1.70129339425417E-06	0.777226228626114	0.000107728795889211
Random Forest - Base Model	0.901961797440210	0.901947165352055	7.90611000998292E-07	0.378233268979794	0.000182758449606535
Random Forest - GridSearchCV	0.776859274597358	0.776825971102600	1.79947722079540E-06	0.287829699824572	0.000122642680432675
Random Forest - RandomizedSearchCV	0.776859274597358	0.776851152710286	1.79947722079540E-06	0.287829699824573	0.000122642680432675

Figure 25, ML Models' results with "alpha_FF5MOM". Own Elaboration.

As the summary statistics demonstrate, the models are generally accurate for models trained for alpha_FF5MOM. We see some very interesting initial findings when evaluating the outputs of our random forest models. Our base case random forest model, which doesn't use any hyperparameter tuning, has the highest R-squared of the machine learning models. However, it performs worse than the tuned models on every other metric. While interesting, it is not inherently surprising and there are several reasons why this could be occurring.

This is most likely a result of overfitting. Random forest models which are not properly tuned can overfit the training data, but then fails to generalize and thus perform worse on the test data. Additionally, both the grid search and randomized search models' hyperparameters are optimized, while the base model's hyperparameters aren't. This is likely the reason for the better generalized performance as measured by the MAE, MAPE, and MSE. This optimization leads to more accurate capturing of the underlying relationships between variables. Lastly, there is likely a bias-variance tradeoff occurring between the tuned and untuned models. Since the base model is not tuned, it has a higher variance on the test set because it is more sensitive to the specific training data. On the other hand, the random search and grid search models are tuned better, which means the bias-variance tradeoff is closer to an optimal level. This helps explain why the R-squared for the tuned models is lower, but they both perform significantly better on the other metrics.

In summary, it is not necessarily unusual to see such a finding and it suggests the untuned model models suffer from overfitting. Since it is likely overfit, the other two random forest models are more effective at predicting return. Additionally, we see very similar outputs for the

tuned random forest models and the gradient boosting model. This is not necessarily surprising and is likely a result of the similar way the models are calculated.

Both gradient boosting and random forests are ensemble learning methods but use different approaches to build their respective models. Random forests essentially combine the predictions of multiple decision trees, while gradient boosting builds its ensemble with weak learners in a sequential manner. Each subsequent model then focuses on correcting the mistakes made by the previous model. These subtle differences can have an impact on their respective performances, even though the calculation methodologies overall results are similar.

At first glance, this suggests that the tuned random forest algorithms perform slightly better than the gradient boosting model on our dataset. However, having only one metric be noticeably different from each other is not enough to come to a definitive conclusion. We see that the R-squared values across the three models are essentially the same. This is not necessarily unusual, since both modelling types were given the same inputs.

An R-squared of about 0.78 might be considered low in many statistical applications. However, in a financial context, a good rule of thumb is an R-squared above 0.7 shows a high level of accuracy. This indicates that all three models are equally good at explaining the relationship between the variables and alpha_FF5MOM. Another interesting finding is the similarity between the R-squared and adjusted R-squared values.

We see that the adjusted R-squared values are more or less the same as the regular R-squared values for the gradient boosting and two tuned random forest models. This suggests that the addition of variables does not significantly contribute to the explanatory power of the model. This is in line with the findings from previous studies which suggest a limited number of variables can substantially explain outperformance.

We can see from these initial results that the machine learning models we tested are good at predicting outperformance as measured by the alpha_FF5MOM. Now we move on and review the results for a different measure of outperformance, namely the t_alpha_FF5MOM. We have now seen that the models are good at predicting raw outperformance, but we now want to use this even stricter measure to evaluate statistically significant outperformance. The results are summarized in the table below.

Model	**Accuray Metric - t_alpha_FF5MOM**				
--	R-Squared	Adjusted R-Squared	Mean Squared Error	Mean Absolute Percentage Error	Mean Absolute Error
Linear Regression - Unadjusted	0.998015694323973	0.998015398168650	4.08180470492956E-03	0.174447812358168	0.049273097891015700
Linear Regression - Adjusted 01	0.998141786901246	0.998141570450852	3.82242668601486E-03	0.157501367653974	0.047775957679704100
Linear Regression - Adjusted 02	0.998665451632392	0.998665398199745	2.74522512920655E-03	0.107987982189782	0.040308717018548000
Gradient Boosting	0.996287341234720	0.996286787124700	7.63710359699315E-03	0.152183911071124	0.053821072028438900
Random Forest - Base Model	0.989530613745129	0.989529051201359	2.15359914498804E-02	0.322597882470064	0.093271323723161500
Random Forest - GridSearchCV	0.993575346558572	0.993574387686489	1.32157968208173E-02	0.155805293699349	0.058738450136528100
Random Forest - RandomizedSearchCV	0.993575346558572	0.993574387686489	1.32157968208173E-02	0.155805293699349	0.058738450136528100

Figure 26, ML Models' results with "t_alpha_FF5MOM". Own Elaboration.

At first glance, we can see some notable differences. Most strikingly, the R-squared values for the machine learning models are now all close to one. This indicates a high degree of overfitting; however, it is not very surprising to get such high R-squared values. Even for tuned machine learning models, the R-squared values can often be misleading. Complex data can lead the metric to become artificially inflated, even for models with hyperparameter tuning.

However, it is noteworthy that the R-squared values (both raw and adjusted) increased with the new target variable. Setting aside the magnitude, the fact there was an increase suggests the models are better at predicting statistically significant outperformance than the raw outperformance. A possible explanation for this is the noise reduction the t-statistic of the

alpha_FF5MOM provides. The t-statistic considers certain elements of variability and adjusts for them, leaving less noise to be filtered out by the models themselves.

Another possible explanation is a high signal strength of t_alpha_FF5MOM as a target variable. It intuitively makes sense that the independent variables are capturing relevant patterns which contribute to the alpha's statistical properties. The features are likely better able to capture relevant patters which contribute to this statistical significance more so than the raw outperformance.

We also see improvements in the error measurements. The measurements of the t-statistic vs the raw alpha_FF5MOM are obviously different, so MSE and MAE are not a good barometer for this. However, when evaluating the MAPE, we see significant reductions across the board for all models. The largest of which being for the gradient boosting model. This bolsters the case that these machine learning models are better at predicting the significance of the outperformance than the raw outperformance.

Taking a step back though, one thing is certainly clear. The models are good at predicting outperformance. The machine learning models generally perform well for multiple measures of outperformance. We decided to include these two comparisons because our goal is to assess the general useability of these models. If they perform well on multiple measures of return, then one can conclude they are useful in their application. The results themselves differ when the target is different, which is not at all surprising, but generally perform well on both metrics.

Now that we have established the effectiveness of the models, we can test for statistical significance between them. The approach we used was to run two sets of tests, one for each target variable. For each set of metrics, our goal is to establish whether there is a statistically significant difference in performance between the models based on the target. Our null hypothesis is that there is no statistically significant difference between the models for either measurement of outperformance.

However, before we check statistical significance, we examine our histograms and Q-Q plots (full collection in appendixes 1 and 2). The plots suggest the residuals are not normally distributed. We can see this by their left-skewed distributions. Since the residuals are not normally distributed, we can use evaluation formulas which don't rely on this assumption. Now that we have established this, we can decide on which tests to perform.

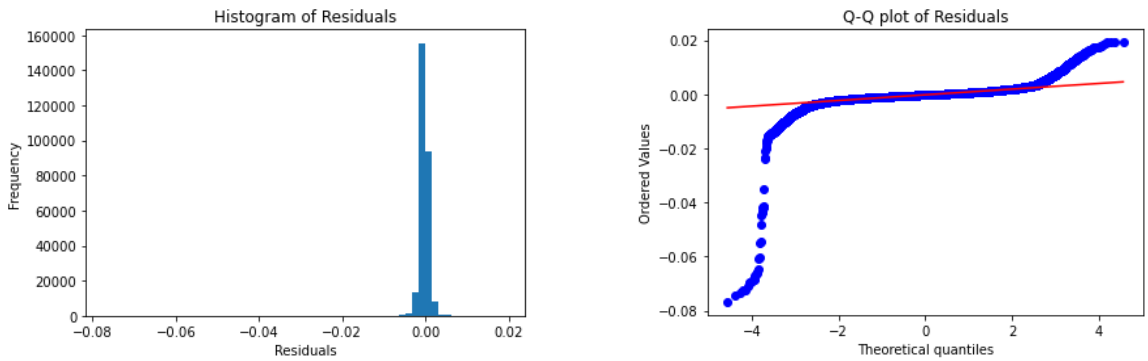


Figure 27, Gradient Boosting Residuals Plot (alpha_FF5MOM as target). Own Elaboration.

Before we moved on to test the various algorithms, we first had to decide which algorithms to compare. Our OLS models clearly suffer from overfitting, which makes them less useful for our purposes. Additionally, our base case random forest model also appeared to suffer from overfitting. The tuned random forest models as well as the gradient boosting model are

therefore the best candidates for comparison. Such tuned models are in line with those used by academic researchers as well, so it makes sense to compare these. For those reasons, we chose these three algorithms to statistically compare.

The metrics we decided to evaluate are the MSE, MAE, and MAPE. We decided to exclusively focus on these three metrics as they are the best measurements of accuracy in our context. Since the R-squared values are prone to giving misleading results for gradient boosting and random forest models, we decided to exclude them.

Since the residuals are not normally distributed, we decided on using the Wilcoxon signed-rank and Friedman tests. Additionally, we can use each of these formulae to test multiple models at once using a collection of metrics. For these reasons, we decided to use these two tests to evaluate statistical significance.

Briefly, the Wilcoxon signed-rank test is a non-parametric statistical test which is used to compare related samples. It basically compares the sample medians of the metrics we used and compares them against a hypothetical median. In the case of this test, the null hypothesis is that the medians are the same. If we reject the null, then there is likely a statistically significant difference between the metrics. This would in turn indicate a statistically significant difference in predictive power between at least two of the models.

The test works by first finding the differences between the metrics of the two models being compared. The results are then ranked from lowest to highest, ignoring sign. Then, either a '-' or a '+' is assigned to the specific difference depending on whether it is a positive or negative value. The absolute differences are then summed together based on whether their value was assigned a '-' or a '+'. Finally, the smaller of the two values is taken and then compared to the table of critical values. If the calculated value is higher than the critical value, we reject the null that there is no significant difference in means.

The test can be set up relatively easily in Python with common packages. To run these tests, we used the following code:

```python
#Import packages to run the tests
import itertools
import scipy.stats as stats

#Create array of evaluation metrics for three models
model_criteria = [
    [1.701293394254 17E-06, 0.777226228626114, 0.000107728795889211], #gradient_boosting
    [1.799477220795 40E-06, 0.287829699824572, 0.000122642680432675], #random_forest_grid_search_cv
    [1.799477220795 40E-06, 0.287829699824573, 0.000122642680432675] #random_forest_rand_search_cv
]

#Perform the pairwise comparison using Wilcoxon signed-rank test
alpha = 0.05
total_models = len(model_criteria)
comp = list(itertools.combinations(range(total_models), 2))

#Create for loop
for i, j in comp:
    model_1 = model_criteria[i]
    model_2 = model_criteria[j]

    statistic, p_value = stats.wilcoxon(model_1, model_2, zero_method="wilcox")

    #Print results
    print(f"Comparing Model {i+1} and Model {j+1}")
    print("Wilcoxon Signed-Rank Test Results")
    print("Statistic:", statistic)
    print("P-Value:", p_value)

    #Interpret results
    if p_value < alpha:
        print('statistically significant difference in predictive power between at least two models')
    else:
        print('there is no statistically significant difference in predictive power between the models')

    #Add separation line for clarity
    print("----------------------------------------------")
```

Figure 28, Wilcoxon Signed-Rank Python Code (alpha_FF5MOM target). Own Elaboration.

The Wilcoxon signed-rank test can only compare two models at once. For this reason, we had to first create an array which held the metrics for each of the models. We then defined the

significance level of 5% and created a variable which holds the length of the model criteria array. The "comp" variable then holds all the possible pairwise combinations of the three models for comparison purposes. We then create a for loop which iterates through these pairwise combinations and runs the statistical test.

You will notice the *'zero_method'* is set to the string value *'wilcox.'* This is the default method for dealing with zero values when evaluating and ranking differences. There are three different ways to deal with zero values in Wilcoxon signed-rank tests: *'wilcox'*, *'pratt'*, and *'zsplit'*. The *'wilcox'* method assumes the signs are exchangeable between '-' and '+'. The *'pratt'* method assigns a rank of 0.5 to all zero differences. It also assumes the zero differences are exchangeable between both groups but treats them as a separate group. This is generally considered a more conservative approach to running the test. Lastly, *'zsplit'* divides the differences into the positive and negative ranks. This model assumes the zero differences are not interchangeable between positive and negative signs. It is theoretically a more nuanced approach.

For our purposes, we decided to use each *'zero_method'* to see whether the outcomes would be different. The results are summarized in this table grouped by the zero method:

Wilcoxon Signed-Rank Summary Outputs - alpha_FF5MOM

wilcox	pratt	zsplit
Comparing Model 1 and Model 2 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 1 and Model 2 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 1 and Model 2 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------
Comparing Model 1 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 1 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 1 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------
Comparing Model 2 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 0.0 P-Value: 0.3173105078629141 5 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 2 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 0.0 P-Value: 0.3173105078629141 5 there is no statistically significant difference in predictive power between the models ----------	Comparing Model 2 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 1.5 P-Value: 0.4142161782425252 there is no statistically significant difference in predictive power between the models ----------

Figure 29, Wilcoxon Signed-Rank Outputs. Own Elaboration.

We can see from this output that there are no statistically significant differences between the models, regardless of the zero_method chosen. From the p-values, we clearly fail to reject the null that there is a difference between the medians in these models.

We then ran the exact same test with the outputs for our second model as well, however we could only perform the *'zsplit'* method. The metrics for the t_alpha_FF5MOM model are close to each other, the model asses the differences as zero. When this is the case, only the *'zslipt'* methodology can be used. The outputs from the test are shown here:

Wilcoxon Signed-Rank Summary Outputs - t_alpha_FF5MOM

zsplit
Comparing Model 1 and Model 2 Wilcoxon Signed-Rank Test Results Statistic: 0.0 P-Value: 0.25 there is no statistically significant difference in predictive power between the models ---------- Comparing Model 1 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 0.0 P-Value: 0.25 there is no statistically significant difference in predictive power between the models ---------- Comparing Model 2 and Model 3 Wilcoxon Signed-Rank Test Results Statistic: 3.0 P-Value: 1.0 there is no statistically significant difference in predictive power between the models ----------

Figure 30, Wilcoxon Signed-Rank Outputs (t_alpha_FF5MOM target variable). Own Elaboration.

We can see from these outputs there is no statistically significant difference between the models for this measure of return either. We also see here that the p-values are clearly above the alpha of 0.05. So far, we fail to reject the null for either measure of return.

However, for such a complicated dataset, one cannot rely on one test alone, which is why we also conducted the Freidman test. The Friedman test is a non-parametric alternative to the analysis of variance (ANOVA) test. The test works by ranking the metrics from best to worst. If there are ties in the data, their ranks are averaged. Once each point has been assigned a

ranking, the test statistic is calculated based on the differences between the ranks of each observation. The test then considers the average rank for each metric and its deviation from the overall overage. These are then summed to find the test statistic. The larger the test statistic, the greater the likelihood we can reject the null that there is no difference.

This type of test is useful because it can evaluate models which were created using the same data. There isn't a need for the models to be independently trained for the test to be effective. Another advantage is the ability to apply this test to multiple models at once. This makes it easy to interpret and highly useful as a comparison metric. Additionally, the test considers the relative performance of the models and not the specific values. For our purposes, an absolute value representing the models' effectiveness is not important. We are mainly interested in the relative performance of the three models we are comparing.

Thanks to Python, such tests are relatively easy to conduct. Since all the models are compared at once, there is also no need for looping or other complexities. This reduces the risk of error and makes it easier to implement. An outline of our code is found here:

```
#Import packages to run the tests
import scipy.stats as stats

#Create array of evaluation metrics for three models
gradient_boosting = [1.70129339425417E-06, 0.777226228626114, 0.000107728795889211]
random_forest_grid_search_cv = [1.79947722079540E-06, 0.287829699824572, 0.000122642680432675]
random_forest_rand_search_cv = [1.79947722079540E-06, 0.287829699824573, 0.000122642680432675]

#Execute the Friedman test
f_value, p_value = stats.friedmanchisquare(gradient_boosting, random_forest_grid_search_cv,
random_forest_rand_search_cv)

#Set alpha
alpha = 0.05

#Print results
print("Friedman Test Results")
print("Statistic", f_value)
print("P-Value", p_value)

#Interpret results
if p_value < alpha:
    print('statistically significant difference in predictive power between at least two models')
else:
    print('there is no statistically significant difference in predictive power between the models')
```

Figure 31, Friedman Test Python Code alpha_FF5MOM. Own Elaboration.

The code in this case is self-explanatory compared to the Wilcoxon signed-rank test. We imported the necessary packages, then created arrays holding the evaluation metrics for each of the models. We then implement the Friedman test according to the package's specifications. As with the signed-rank test, we set the significance level to 5% and then execute the code. A summary of the outputs can be found here:

Friedman Test Results - alpha_FF5MOM	
Statistic:	0.7999999999999972
P-Value	0.6703200460356403
Result:	Fail to reject null

Figure 32, Friedman Test Output (alpha_FF5MOM target variable). Own Elaboration.

As we can see from this table, we also fail to reject the null using the Friedman test. The results are in line with our findings from the Wilcoxon signed-rank test for the alpha_FF5MOM

trained model. We therefore fail to find evidence that there is a statistically significant difference in predictive ability between our random forest and gradient boosting models for this measure of outperformance.

However, for our models trained to predict t_alpha_FF5MOM, the results are different. According to the Friedman test, there is a statistically significant difference between the predictive accuracy of the models. This contradicts our findings from the Wilcoxon signed-rank test, but isn't entirely surprising. Since the Wilcoxon signed-rank test is a pairwise comparison method, it is not inconceivable this test would indicate no difference between the pairs, but the Friedman test indicates an overall difference between at least two of the models.

Friedman Test Results - t_alpha_FF5MOM	
Statistic:	6.0
P-Value	0.04978706836786395
Result:	Evidence for Rejecting Null

Figure 33, Friedman Test Output (t_alpha_FF5MOM target variable). Own Elaboration.

We see here from these outputs a p-value ever so slightly below the 0.05 significance level. While this is certainly an indication that there is a statistically significant difference, the fact that it presents conflicting results warrants post-hoc investigation. One of the most commonly performed post-hoc tests done in conjunction with the Friedman test is the Nemenyi test. Once the null is rejected, this test is done to determine which groups differ significantly from each other.

The Nemnyi test works by first ranking the models based on their performance metrics. A so-called critical difference is then calculated which is based on the number of models and observations in each model. Next, the pairwise differences are computed between the mean ranks of each model and these results are then compared with the critical differences. If the absolute difference is bigger than the critical difference, then those models are considered statistically significant.

With the help of some Python libraries, we were able to easily set this is up.

```
#Import Packages to run ad-hoc tests
import numpy as np
import pandas as pd
import scipy.stats as stats
from scikit_posthocs import posthoc_nemenyi
from statsmodels.stats.multitest import multipletests

#Create a dataframe of evaluation metrics for the models
model_metrics = pd.DataFrame({
    'Model': ['gradient_boosting', 'random_forest_grid_search_cv', 'random_forest_rand_search_cv'],
    'MSE': [7.63710359699315E-03, 1.32157968208173E-02, 1.32157968208173E-02],
    'MAPE': [0.152183911071124, 0.155805293699349, 0.155805293699349],
    'MAE': [0.053821072028438900, 0.058738450136528100, 0.058738450136528100]
    })

#Convert df to NumPy array
data = model_metrics.iloc[:, 1:].to_numpy()

#Perform the Nemenyi post-hoc test
posthoc_results = posthoc_nemenyi(data)

#Print results
print("Nemenyi Post-Hoc Test Results:")
print(posthoc_results)
```

Figure 34, Nemenyi Test Code. Own Elaboration.

Luckily for our purposes, this test can easily be set up according to the specifications of the Python library. We simply used their guide and set the test up accordingly. The output is a matrix array of pairwise comparisons for each model. The model assigns a p-value for each comparison which can then be compared to the significance level. If the pairwise p-value is below the significance level, we reject the null that there is no difference.

Nemenyi Post-Hoc Test Results			
--	1	2	3
1	1.000000	0.793923	0.793923
2	0.793923	1.000000	0.793923
3	0.793923	0.793923	1.000000

Figure 35, Nemenyi Test Code. Own Elaboration.

We can clearly see from this matrix that the p-values are well above the significance level of 0.05. Therefore, this test casts doubt on the result of the Friedman test. However, since we now have further conflicting results, it makes sense to perform further post-hoc tests to back up or reject this finding. To do this, we now perform the Bonferroni-Dunn test.

The Bonferroni-Dunn test is also often done in conjunction with the Friedman test. This test also evaluates whether there are statistically significant differences between multiple models. The first step is to calculate the adjusted alpha value based on the number of models. Then, pairwise p-values are calculated and adjusted using the so-called Bonferroni correction. This correction is meant to account for the increased likelihood of obtaining a false positive (Type 1 error) when comparing multiple models. Finally, we compare the adjusted p-values to the adjusted alpha and reject if they are smaller.

To run this test, we set up the code using readily available Python packages.

```python
#Import Packages to run ad-hoc tests
import numpy as np
import pandas as pd
import scipy.stats as stats
from scikit_posthocs import posthoc_nemenyi
from statsmodels.stats.multitest import multipletests

#Create a dataframe of evaluation metrics for the models
model_metrics = pd.DataFrame({
    'Model': ['gradient_boosting', 'random_forest_grid_search_cv', 'random_forest_rand_search_cv'],
    'MSE': [7.637103596993 15E-03, 1.32157968208173E-02, 1.32157968208173E-02],
    'MAPE': [0.152183911071124, 0.155805293699349, 0.155805293699349],
    'MAE': [0.053821072028438900, 0.058738450136528100, 0.058738450136528100]
    })

#Convert df to NumPy array
data = model_metrics.iloc[:, 1:].to_numpy()

#Set alpha
alpha = 0.05

#Perform the Bonferroni-Dunn test
num_models = len(model_metrics)
adjusted_alpha = alpha / (num_models * (num_models - 1) / 2)

#Create an array to store the test statistics and p-values
test_stats = np.zeros((num_models, num_models))
p_values = np.zeros((num_models, num_models))

#Perform pairwise comparisons and calculate test statistics and p-values
for i in range(num_models):
    for j in range(i + 1, num_models):
        test_stat, p_val = stats.ranksums(data[:, i], data[:, j])
        test_stats[i, j] = test_stat
        p_values[i, j] = p_val

#Adjust the p-values using the Bonferroni correction
reject, adjusted_p_values, _, _ = multipletests(p_values[np.triu_indices(num_models, k=1)],
alpha=adjusted_alpha)

#Print the adjusted p-values for each model comparison
print("Bonferroni-Dunn Test Results:")
index = 0
for i in range(num_models):
    for j in range(i + 1, num_models):
        print(f"Comparison: Model {i+1} vs Model {j+1}")
        print("Test Statistic:", test_stats[i, j])
        print("Adjusted p-value:", adjusted_p_values[index])
        print("----------------------------------------------")
        index += 1
```

Figure 36, Bonferroni-Dunn Test Code. Own Elaboration.

We see from this code that the setup is somewhat complicated compared to our other tests. To create this code, we followed the Python library's specifications to build the loops which can perform the pairwise comparisons. Once we ran this code, we got an adjusted alpha of 0.0167. The outputs for the pairwise comparisons are as follows:

Bonferroni-Dunn Test Results

```
Comparison: Model 1 vs Model 2
Test Statistic: -1.9639610121239315
Adjusted p-value: 0.14136434850989898
----------------------------------------------
Comparison: Model 1 vs Model 3
Test Statistic: -1.9639610121239315
Adjusted p-value: 0.14136434850989898
----------------------------------------------
Comparison: Model 2 vs Model 3
Test Statistic: 1.9639610121239315
Adjusted p-value: 0.14136434850989898
----------------------------------------------
```

Figure 37, Bonferroni-Dunn Test Code. Own Elaboration.

We can clearly see from these outputs that we fail to reject the null for any of the pairwise comparisons. This is further evidence against our findings of the Friedman test which indicate statistically significant differences between the models. However, just to be sure, we conducted a Cohen's d analysis on the metrics to evaluate the effect size of the differences. What this does is signal whether there is a *practical* difference between the models, i.e. if the differences are relevant in practice or not. The methodology provides a standardized way of quantifying the differences between two models. It is calculated by taking the difference between two means and dividing it by the pooled standard deviation. The absolute value is then compared to different thresholds, typically 0.2, 0.5, and 0.8 to test for small, medium, or large differences respectively.

We then do the comparison tests between the gradient boosting and random forest models for the MSE, MAE, and MAPE. To do this, we set up the following Python code:

```python
#Calculate MSE effect size gradient_boosting vs. random_forest models
mse_gradient_boosting = 7.63710359699315E-03
mse_random_forest_models = 1.32157968208173E-02
pooled_std_dev = np.sqrt((mse_gradient_boosting + mse_random_forest_models) / 2)
effect_size_gradient_boosting_vs_random_forest_models = (mse_gradient_boosting - mse_random_forest_models) / pooled_std_dev

#Print results
print("MSE Effect Size:")
print(abs(effect_size_gradient_boosting_vs_random_forest_models))

#Check for significance
if abs(effect_size_gradient_boosting_vs_random_forest_models) >= 0.2:
    print("There is a small practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models) >= 0.5:
    print("There is a medium practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models) >= 0.8:
    print("There is a large practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models) <= 0.2:
    print("There is no practical effect size")

#Insert line break
print('----------------------------------------')

#Calculate MAPE effect size gradient_boosting vs. random_forest models
mape_gradient_boosting = 0.152183911071124
mape_random_forest_models = 0.155805293699349
pooled_std_dev = np.sqrt((mape_gradient_boosting + mape_random_forest_models) / 2)
effect_size_gradient_boosting_vs_random_forest_models2 = (mape_gradient_boosting - mape_random_forest_models) / pooled_std_dev

#Print results
print("MAPE Effect Size:")
print(abs(effect_size_gradient_boosting_vs_random_forest_models2))

#Check for significance
if abs(effect_size_gradient_boosting_vs_random_forest_models2) >= 0.2:
    print("There is a small practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models2) >= 0.5:
    print("There is a medium practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models2) >= 0.8:
    print("There is a large practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models2) <= 0.2:
    print("There is no practical effect size")

#Insert line break
print('----------------------------------------')

#Calculate MAE effect size gradient_boosting vs. random_forest models
mae_gradient_boosting = 0.053821072028438900
mae_forest_models = 0.058738450136528100
pooled_std_dev = np.sqrt((mae_gradient_boosting + mae_forest_models) / 2)
effect_size_gradient_boosting_vs_random_forest_models3 = ((mae_gradient_boosting - mae_forest_models) / 2)

#Print results
print("MAE Effect Size:")
print(abs(effect_size_gradient_boosting_vs_random_forest_models3))

#Check for significance
if abs(effect_size_gradient_boosting_vs_random_forest_models3) >= 0.2:
    print("There is a small practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models3) >= 0.5:
    print("There is a medium practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models3) >= 0.8:
    print("There is a large practical difference between gradient_boosting and random_forest models")
if abs(effect_size_gradient_boosting_vs_random_forest_models3) <= 0.2:
    print("There is no practical effect size")
```

Figure 38, Effect Size Python Code. Own Elaboration.

Using this simple Python code, we can calculate the statistic and print the results. Since the effect size is always reported as a positive value, we take the absolute value when making the comparisons. Based on our metrics, we find no practical size effect between the gradient boosting and random forest models.

Cohen's d Effect Size Tests

```
MSE Effect Size:
0.05463415659329316
There is no practical effect size
----------------------------------------
MAPE Effect Size:
0.009228299152785306
There is no practical effect size
----------------------------------------
MAE Effect Size:
0.0024586890540445987
There is no practical effect size
```

Figure 39, Effect Size Output Table. Own Elaboration.

We can clearly see from these outputs that there is no effect size between any of the metrics. This further bolsters the case that the results from the Friedman test indicating a statistically

significant difference between models is false. Of course, we cannot fully mitigate the chance we are making a mistake, but it seems much more likely that there is no statistically significant difference in predictive power between models than the latter case. We therefore conclude that there is very likely no statistically significant difference between the predictive power of the gradient boosting and tuned random forest models. This applies for both measures of outperformance, defined as alpha_FF5MOM and t_alpha_FF5MOM.

6. Discussion

Our findings suggest there is no statistically significant difference in ex post predictive accuracy between gradient boosting and random forest machine learning models when trained and tested on our expanded CRSP dataset. This finding holds true for multiple measures of outperformance such as alpha_FF5MOM and t_alpha_FF5MOM. We also find the popular Python hyperparameter tuning packages GridSearchCV and RandomizedSearchCV from scikit-learn are equally effective at tuning random forest models. Both tuning methods produces virtually identical errors. Our findings also demonstrate the overall effectiveness of these machine learning models to predict the outperformance of actively managed mutual funds. Lastly, we find that traditional OLS regression models are prone to overfitting and not useful in the face of better alternatives for the purpose of predicting outperformance.

These findings are in line with previous research which suggests these machine learning models are effective at accurately identifying outperformance and portfolio selection. Additionally, it backs up the findings that fund characteristics and metrics which can be derived from said characteristics are sufficient to predict outperformance. Much like other studies on the matter, our findings demonstrate the usefulness of the CRSP survivor bias free database as a training and testing dataset for machine learning models. Using this training data, our models were also able to perform well on out-of-sample testing data. Our models' stellar performance on the testing data also suggests Xia's findings of ex ante return prediction are likely valid and merit further research.

An unexpected insight we gleamed from our research was just how prone certain models are to overfitting if not tuned properly. The astronomical R-squared values of the OLS models demonstrate the pitfalls of using linear models on such complex datasets. Additionally, our base case random forest model also exhibited signs of overfitting. This further demonstrates that hyperparameter tuning is a necessary step. Even if the metrics might not demonstrate overfitting at first glance, this should still be done whenever possible.

Another unexpected insight is the limitations of the R-squared metric for gradient boosting and random forest models. This was exemplified by the dramatic increase in both normal and adjusted R-squared when we ran our second tests with t_alpha_FF5MOM as a target variable. It highlights the importance of not relying solely on one or two metrics and the value of performing post-hoc tests.

However, we cannot exclude the possibility that neural network models perform better than gradient boosting and random forest models in a statistically significant way because we did not evaluate them. This is a clear limitation of our research as many of the prevailing studies included such model and it merits further consideration. Additionally, we did not include any indicators intended to measure sentiment and/or general economic activity like Kaniel et al. did. There is a possibility the inclusion of such metrics could have affected our results.

Another important limitation was the computer hardware we used to conduct this research. The computers we have access to are in no way built to handle such a large dataset and perform

complex modelling. In the process of training and testing these models, our computers would constantly crash. This forced us to shrink down the tuning parameters for our random forest models. It is not inconceivable that this led to our models being less robust than if the code had been executed on a more powerful machine.

Despite these limitations, we stand behind our findings which contribute to the growing body of research on this subject. What our findings reinforce in an abundantly clear manner is the need for further research into these topics. We suggest further research be done in incorporating factors more specific to the fund managers themselves and not just fund characteristics. This would help better explain the components of outperformance which are more robustly explained by manager skill.

In conclusion, our findings reinforce and contribute the growing body of research which shows that machine learning algorithms are effective at predicating the return and outperformance of actively managed mutual funds. We also find there is no statistically significant difference in predictive power between tuned random forest and gradient boosting models. Lastly, we also identify the ease with which such models can be overfit to training data and the need for post-hoc validation of one's results. These findings further bolster the vast potential of machine learning algorithms and confirm the validity of their effectiveness in the world of financial modelling.

7. Conclusion

While the effectiveness of a machine learning model's ability to predict the returns of mutual funds depends on many factors, our findings suggest tuned random forest and gradient boosting models are similarly effective. Like our fictional colleague in the introduction trying to figure out the best way to get to work, our findings show there is more than one way to accurately reach your destination. It also shows there is no need to overcomplicate a model when it doesn't need to be. For example, our gradient boosting model was able to run in about 30 minutes, while our random forest models took several days.

For decades, being able to predict return has been the proverbial holy grail of financial modelling. Many have staked their reputations and careers on the notion that return can or can't be predicted and that fund managers do or don't have superior stock picking skills. Any argument made in one direction or the other had an equally plausible counterargument. In today's world of quantum computing and big data, we may be coming closer to answering these fundamental questions. However, a few things are still not necessarily clear. Firstly, how feasible it is to actually implement such algorithms to make real time investment decisions outside of the theoretical realm. Secondly, it is still up in the air to what degree skill plays a role in the return of actively managed mutual funds.

On the other hand, many things are now clearer than ever before. Firstly, machine learning algorithms can both predict return and outperformance with a high degree of accuracy in an ex-post environment. Secondly, these insights can be used to construct alpha generating portfolios of actively managed mutual funds. We have officially entered a new era in the world of finance where one thing is abundantly clear: change is coming, and big data is the driving force behind it. There is more financial data available now than at any prior point in history and computers have never been more powerful. While some prefer to remain skeptical about machine learning, the world keeps turning whether they like it or not. The application for these powerful algorithms is multi-faceted and have the potential to disrupt almost all aspects of the financial industry. This all begs the question: would you invest your money in a fund if a model told you to?

8. Bibliography

Babyak, M. A. (2004). What You See May Not Be What You Get: A Brief, Nontechnical Introduction to Overfitting in Regression-Type Models. *Psychosomatic Medicine*.

Baker, M., & Wurgler, J. (2006). Investor Sentiment and the Cross-Section of Stock Returns. *The Journal of Finance*, *61*(4), 1645–1680. https://doi.org/10.1111/j.1540-6261.2006.00885.x

Banerjee, A. (2022, November 2). Hyperparameter Tuning Using Randomized Search. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2022/11/hyperparameter-tuning-using-randomized-search/

Banerjee, A., Chitnis, U. B., Jadhav, S. L., Bhawalkar, J. S., & Chaudhury, S. (2009). Hypothesis testing, type I and type II errors. *Industrial Psychiatry Journal*, *18*(2), 127–131. https://doi.org/10.4103/0972-6748.62274

Bradburn, S. & PhD. (2019, February 12). The Bonferroni Correction Method Explained. *Top Tip Bio*. https://toptipbio.com/bonferroni-correction-method/

Brownlee, J. (2016, September 8). A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning. *MachineLearningMastery.Com*. https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

Carhart, M. M. (1997). On Persistence in Mutual Fund Performance. *The Journal of Finance*, *52*(1), 57–82. https://doi.org/10.1111/j.1540-6261.1997.tb03808.x

Center Fore Research In Securty Prices (CRSP). (n.d.). *Survivor-Bias-Free US Mutual Fund Guide*. https://www.crsp.org/files/MFDB_Guide.pdf

Cross Validation Explained: Evaluating estimator performance. | by Rahil Shaikh | Towards Data Science. (n.d.). Retrieved May 20, 2023, from https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85

Cross-validation: Evaluating estimator performance. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/cross_validation.html

Cureton, E. E. (1967). The Normal Approximation to the Signed-Rank Sampling Distribution When Zero Differences are Present. *Journal of the American Statistical Association*, *62*(319), 1068–1069. https://doi.org/10.1080/01621459.1967.10500917

DataTechNotes: Regression Accuracy Check in Python (MAE, MSE, RMSE, R-Squared). (n.d.). Retrieved May 23, 2023, from https://www.datatechnotes.com/2019/10/accuracy-check-in-python-mae-mse-rmse-r.html

Decision Trees. (2018, November 22). Quantitative Finance & Algo Trading Blog by QuantInsti. https://blog.quantinsti.com/decision-tree/

DeMiguel, V., Gil-Bazo, J., Nogales, F. J., & Santos, A. A. P. (n.d.). *Machine Learning and Fund Characteristics Help to Select Mutual Funds with Positive Alpha*.

Didur, K. (2018, July 11). *Machine learning in finance: Why, what & how*. Medium. https://towardsdatascience.com/machine-learning-in-finance-why-what-how-d524a2357b56

Diebold, F. X. (2004). The Nobel Memorial Prize for Robert F. Engle. *Scandinavian Journal of Economics*, *106*(2), 165–185. https://doi.org/10.1111/j.0347-0520.2004.00360.x

Dsanr. (2022, August 5). Statistical Effect Size and Python Implementation. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2022/08/statistical-effect-size-and-python-implementation/

Explained: Neural networks. (2017, April 14). MIT News | Massachusetts Institute of Technology. https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

Fama, E. F., & French, K. R. (1993). Common risk factors in the returns on stocks and bonds. *Journal of Financial Economics*, *33*(1), 3–56. https://doi.org/10.1016/0304-405X(93)90023-5

Fama, E. F., & French, K. R. (2015). A five-factor asset pricing model. *Journal of Financial Economics*, *116*(1), 1–22. https://doi.org/10.1016/j.jfineco.2014.10.010

Frigg, M. (2023). *Machine Learning Algorithms: A Horse Race*.

Frost, J. (2022, June 27). Cohens D: Definition, Using & Examples. *Statistics By Jim*. https://statisticsbyjim.com/basics/cohens-d/

Gil-Bazo, J., & Ruiz-Verdú, P. (2009). The Relation between Price and Performance in the Mutual Fund Industry. *The Journal of Finance*, *64*(5), 2153–2183. https://doi.org/10.1111/j.1540-6261.2009.01497.x

Gogas, P., & Papadimitriou, T. (2021). Machine Learning in Economics and Finance. *Computational Economics*, *57*(1), 1–4. https://doi.org/10.1007/s10614-021-10094-w

Greenemeier, L. (n.d.). *20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess*. Scientific American. Retrieved May 23, 2023, from https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess/

How to tune Hyper parameters using Grid Search in Python? -. (n.d.). ProjectPro. Retrieved May 23, 2023, from https://www.projectpro.io/recipes/tune-hyper-parameters-using-grid-search-in-python

Is R-squared Useless? | University of Virginia Library Research Data Services + Sciences. (n.d.). Retrieved May 23, 2023, from https://data.library.virginia.edu/is-r-squared-useless/

Kaniel, R., Lin, Z., Pelger, M., & Nieuwerburgh, S. V. (n.d.). *Machine-Learning the Skill of Mutual Fund Managers*.

Modasiya, K. (2022, November 28). What the heck is random_state? *MLearning.Ai*. https://medium.com/mlearning-ai/what-the-heck-is-random-state-24a7a8389f3d

Normal Probability Plot of Residuals | STAT 462. (n.d.). Retrieved May 23, 2023, from https://online.stat.psu.edu/stat462/node/122/

Pratt, J. W. (1959). Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures. *Journal of the American Statistical Association*, *54*(287), 655–667. https://doi.org/10.1080/01621459.1959.10501526

Random Forest Algorithms—Comprehensive Guide With Examples. (n.d.). Retrieved May 23, 2023, from https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

Random Forest Hyperparameter Tuning in Python | Machine learning. (n.d.). Retrieved May 20, 2023, from https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/

Rowe, W. (n.d.). *Mean Square Error & R2 Score Clearly Explained*. BMC Blogs. Retrieved May 23, 2023, from https://www.bmc.com/blogs/mean-squared-error-r2-and-variance-in-regression-analysis/

R-Squared: Definition, Calculation Formula, Uses, and Limitations. (n.d.). Investopedia. Retrieved May 23, 2023, from https://www.investopedia.com/terms/r/r-squared.asp

Sarr, M. (2021, June 4). Hyperparameter Tuning with Random Search in Python. *Malick Sarr*. https://www.malicksarr.com/hyperparameter-tuning-with-random-search/

Saxena, S. (2020, March 12). A Beginner's Guide to Random Forest Hyperparameter Tuning. *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/

scipy.stats.friedmanchisquare—SciPy v1.10.1 Manual. (n.d.). Retrieved May 23, 2023, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html

Shaikh, R. (2018, November 26). *Cross Validation Explained: Evaluating estimator performance.* Medium. https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85

Sklearn.ensemble.HistGradientBoostingRegressor. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/generated/sklearn.ensemble.HistGradientBoostingRegressor.html

Sklearn.ensemble.RandomForestRegressor. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

Sklearn.model_selection.GridSearchCV. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Sklearn.model_selection.RandomizedSearchCV. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Stephanie. (2014, April 9). *Friedman's Test / Two Way Analysis of Variance by Ranks*. Statistics How To. https://www.statisticshowto.com/friedmans-test/

Team, G. L. (2020, September 24). *What is Cross Validation in Machine learning? Types of Cross Validation*. Great Learning Blog: Free Resources What Matters to Shape Your Career! https://www.mygreatlearning.com/blog/cross-validation/

Tuning the hyper-parameters of an estimator. (n.d.). Scikit-Learn. Retrieved May 20, 2023, from https://scikit-learn/stable/modules/grid_search.html

Understanding QQ Plots | University of Virginia Library Research Data Services + Sciences. (n.d.). Retrieved May 23, 2023, from https://data.library.virginia.edu/understanding-q-q-plots/

Using Machine Learning to Predict Mutual Fund Performance. (n.d.). NBER. Retrieved May 23, 2023, from https://www.nber.org/digest/202205/using-machine-learning-predict-mutual-fund-performance

What Is Gradient Descent? | Built In. (n.d.). Retrieved May 23, 2023, from https://builtin.com/data-science/gradient-descent

What Is the Bonferroni Test (Correction) and How Is It Used? (n.d.). Investopedia. Retrieved May 23, 2023, from https://www.investopedia.com/terms/b/bonferroni-test.asp

Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, *1*(6), 80. https://doi.org/10.2307/3001968

Wilcoxon Signed Rank Test: Definition, How to Run, SPSS. (n.d.). Statistics How To. Retrieved May 23, 2023, from https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/wilcoxon-signed-rank-test/

Xia, Y. (2021). Real-Time Predictability of Mutual Fund Performance Predictors. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3876180

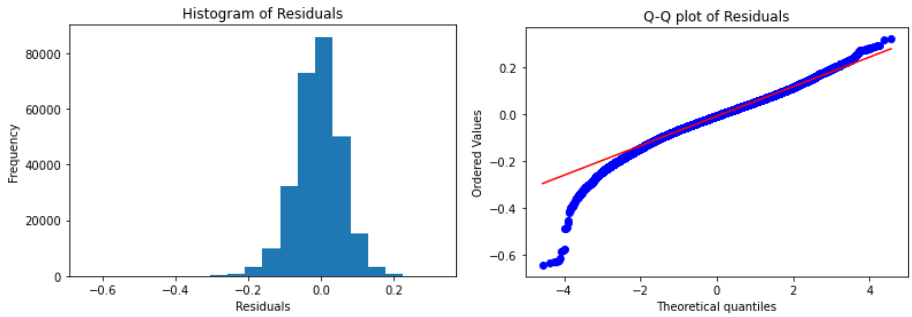
Zach. (2020a, July 10). How to Conduct a Wilcoxon Signed-Rank Test in Python. *Statology*. https://www.statology.org/wilcoxon-signed-rank-test-python/

Zach. (2020b, July 13). How to Perform the Friedman Test in Python. *Statology*. https://www.statology.org/friedman-test-python/

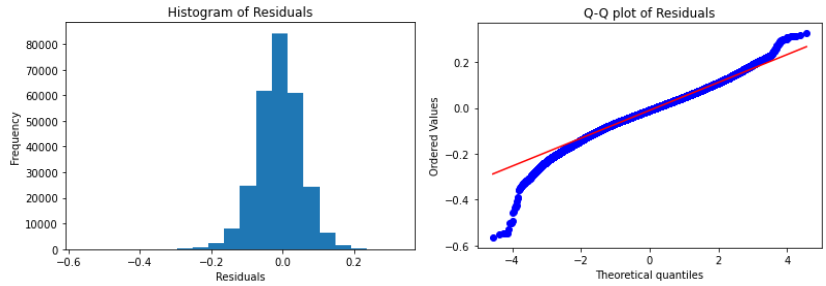
Zoghlami, F. (2013). *Momentum effect in stocks' returns between the rational and the behavioural financial theories: Proposition of the progressive rationality*. *2*(1).

9. Appendixes

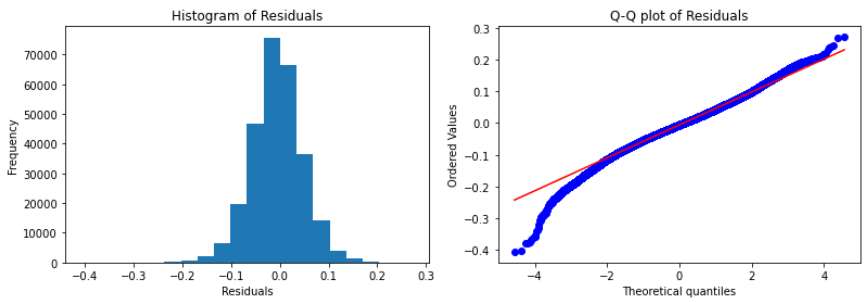
Appendix 1: t_alpha_FF5MOM Target Variable Plots



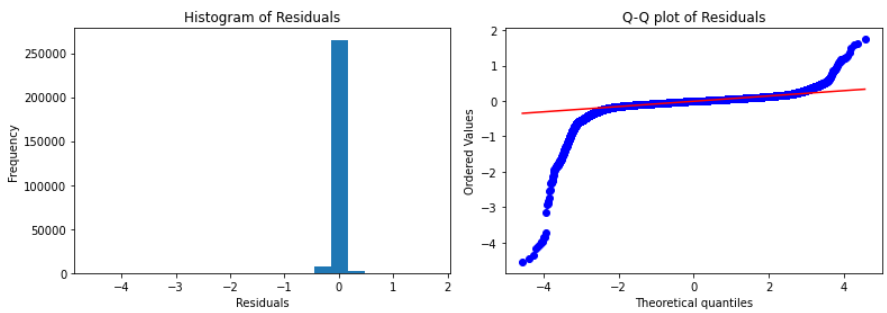
OlS Model Unadjusted



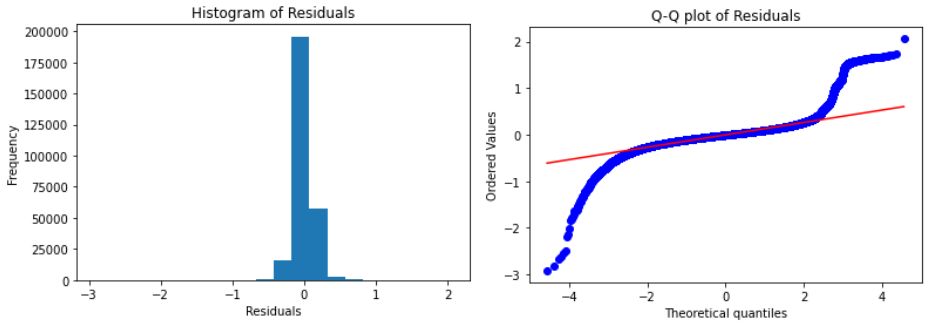
OlS Adjsuted 01 Plots



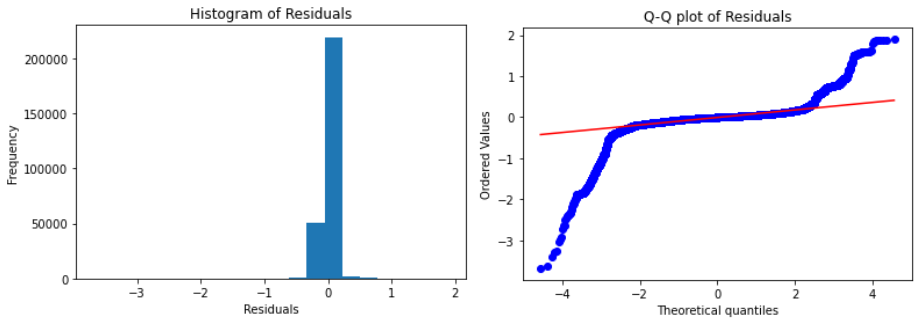
OLS Adjusted 02 Plots



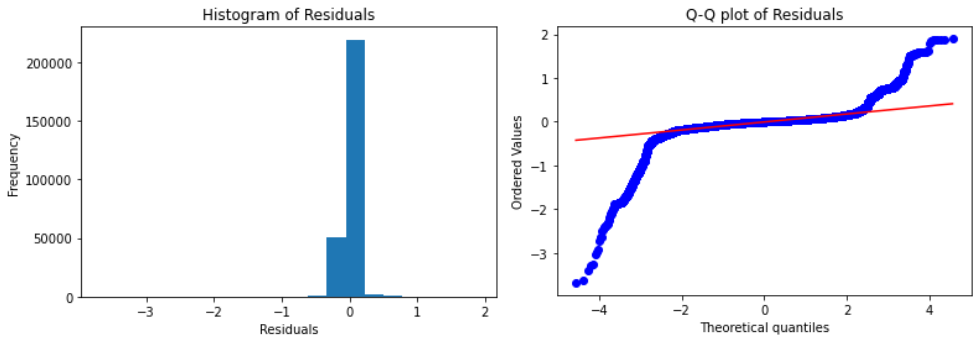
Gradient Boosting Plots



Random Forest Unadjusted Plots

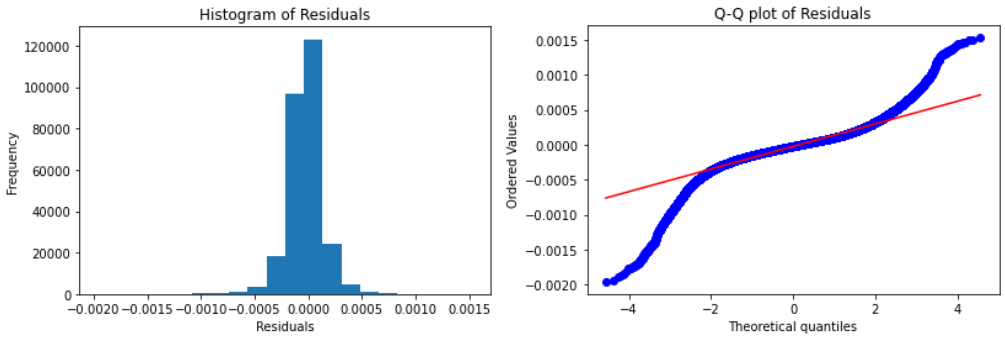


Random Forest Gridsearch Plots

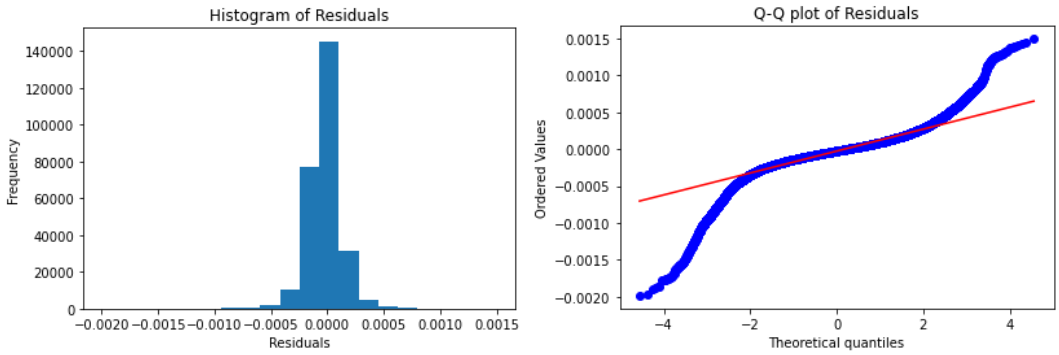


Random Forest Randsearch Plots

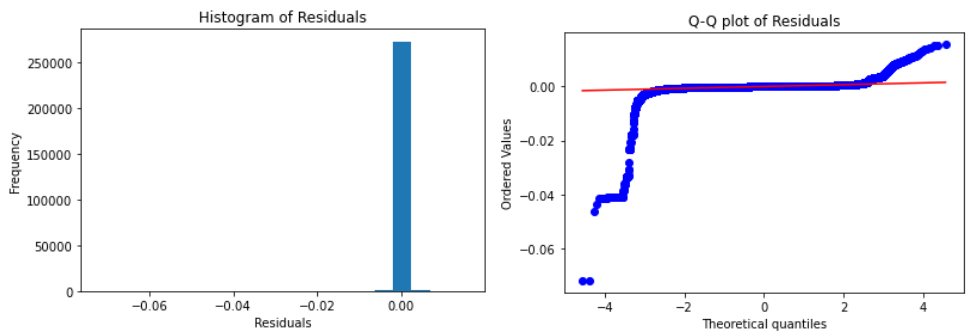
Appendix 2: Alpha FF5MOM Target Variable Plots



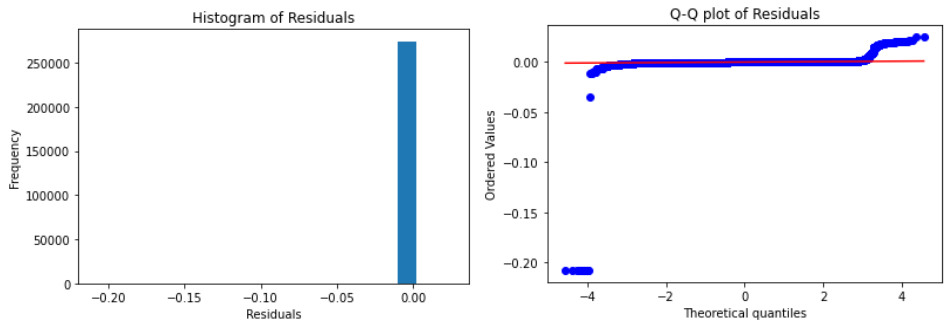
OLS Unadjusted Plots



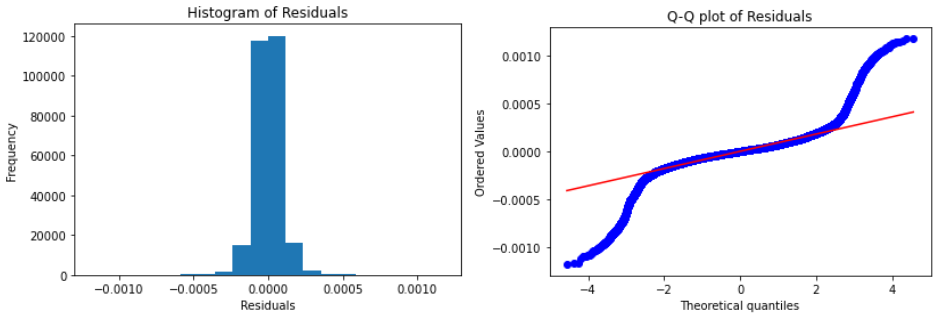
OLS Adjusted 01 Plots



Random Forest Base Model Plots



Random Forest Gridsearch



OLS Adjusted 02