Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Automated Formulaic Alpha Generation for Quantitative Investing using Evolutionary Algorithms

Bachelor's Thesis

Olin Geimer

geimero@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Zhao Meng
Prof. Dr. Roger Wattenhofer

March 13, 2022

Acknowledgements

I would like to thank Prof. Dr. Roger Wattenhofer and the Distributed Copmuting Group for giving me this opportunity to dive deeper into the exciting intersection of Machine Learning and Financial Markets in the frame of this research project.

Furthermore, I would like to thank Zhao Meng for his comprehensive and understanding but frank guidance and supervision throughout the past couple of months.

Last but not least I would like to thank Dr. Yunpu Ma for our regular insightful and inspiring meetings and his sympathetic support throughout this thesis.

Abstract

A cosmic ray consists of mostly highly energetic protons that emanate from the sun, the Milky Way and distant galaxies. By colliding with particles in our atmosphere they trigger a chain reaction that leads to so called cosmic-ray showers of lower energetic particles like pions [1]. In modern biology these are held responsible for inducing the random genetic mutations that led to the development of life on our planet as we know it [2].

In the frame of this thesis we will explore how we can make use of these random mutations of the genetic representation of competing candidates to find functions that correlate well with the stock market. This will yield a set of formulaic alphas that are used in quantitative investing to recognise patterns in a stock's price development and trade on them accordingly. We will evaluate the performance of a set of eight formulaic alphas that are generated by a genetic program on the Nasdaq 100 and realize that they are highly correlated with the development of the federal reserve's balance sheet. The assessment of a simple trading algorithm's performance increase allows the assumption that the generated formulaic alphas help to recognise patterns in the stock market.

Contents

Chapter 1

Introduction

When looking at the stock market and observing the general price developments of individual stocks they can mostly be described by the companies predicted performance, alternative investment opportunities and macroeconomic developments, like changes in the inflation rate or the promised returns on treasury bonds. However, the more detailed price movements depend much more on the combination of the behaviours of all the stock market participants. This could for example be a learned rule where to set a stop loss or the definition of a buying signal when a stock price brakes its recently tested highs. The assumption for this thesis is, that the sum of the behaviours of all the stock market participants generates patterns. If we are able to find and recognize these we can try to monetize on our knowledge of probable future behaviour of the collection of participants by aligning our trading strategy with the pattern's price development. Since finding these patterns comes with the potential to make large amounts of money, many stock market participants are constantly trying to discover them. When they do, they change their trading behaviour to monetize on their knowledge. However, when too many participants do this, the pattern itself changes its meaning and thus the knowledge of it becomes less valuable. This leads to a race to find the most useful and meaningful patterns and a big secrecy around which ones a participant discovered and uses for his/her trading decisions.

1.1 Capturing Patterns in Price Developments

To find and make use of patterns in price developments, we first need a way to describe them. A lot of the time a combination of so called formulaic alphas and machine learning algorithms are used to capture the dependencies of different input parameters and to extract a trading decision from them.

1.1.1 Formulaic Alphas

A formulaic alpha (also called "alpha") is a function that takes multiple stock related parameters of a past time period. The value it returns can be used as

a signal or prediction for the price of the next time period. In the paper [3] some of the most well known and most used formulaic alphas are presented. To understand better what a formulaic alpha is, let us take a look at an example:

$$Alpha\#101 = \frac{Close - Open}{High - Low} \tag{1.1}$$

Let us assume our pattern takes into consideration the Low, High, Opening and Closing price of a trading day of a stock. The result of this computation can be used as one of many input signals for a trading algorithm. Thus we can compute this formula for a specific stock and then use the outcome to decide if we buy a long or short position.

1.1.2 Trading Bot

Some of the patterns might not be captured by a single formulaic alpha, but by a certain combination of alphas. A trading bot in this thesis will take a set of formulaic alphas as features and output a trading recommendation. When training it on labeled data (the label "1" means that buying at the closing price today and selling at the closing price of the following day would be profitable; "0" means unprofitable) the bot will learn that some specific combinations of alphas are predicting the stock market behaviour better than the individual alphas. These specific combinations are somewhat a "pattern of patterns" and thus some of the patterns we extract from stock market behaviour are captured in the trading bot that combines the alphas predictions to make a trading decision.

1.2 The Process of Finding Formulaic Alphas

The sourcing for novel formulaic alphas is a very hard and creativity seeking task. In this thesis we will explore how we can make use of genetic programming to automate this process.

1.2.1 Automation using Evolutionary Algorithms

The term evolutionary algorithm describes a set of machine learning approaches that are strongly leaning on the concept behind the actual natural evolution process. This means it's based on the idea of "survival of the fittest". A set of candidates is defined that are competing in solving a certain task - in our case: predicting stock market price developments. By measuring how well a candidate solved the given task we can choose the best ones and cross them in the hope that their off-spring will be even better. Genetic programming is one of the ways to implement this evolution motivated approach. Here we simulate some of the

coincidences of crossovers of different candidates that happen in real life, and also introduce some random mutations in the candidates genetic representation.

Since the search space for the optimal solution is very large and for reasons of computational resources and time constraints can't be fully explored, we cannot guarantee to find the best possible solution. It is actually quite improbable for us to find that best solution. However, evolutionary algorithms - and genetic programs in specific - are great tools to find decently well performing solutions in a reasonable amount of time.

Chapter 2

Alphas and Trading Decisions

To make it clear how data is combined into formulaic alphas and then used to generate a trading decision I will explain the flow of data through the algorithm in the following.

2.1 Data Sourcing and Feature Computation

In the first step we source the financial inter-day data of all the companies contained in an index I from YahooFinance!. We denote the data up until day t of stock s as $d_{t,s}$. Given a certain time period and I we get a set of data points $\mathbf{D_I} = \{d_{t1,s1}, d_{t2,s1}, d_{t1,s2}, d_{t2,s2}, \dots\}$ (where $s1, s2 \in I$ and $t1, t2 \in [startDate, endDate]$) that we can use to train our algorithms and test our results.

Given a vector of formulaic alphas $\mathbf{A} = \begin{bmatrix} a_1 & a_2 & a_3 & \dots \end{bmatrix}^T$, where $\mathrm{a_i}$ could for example be equation 1.1, we denote $a_i(d_{t,s})$ as the value that the formulaic alpha a_i returns when given the data $d_{t,s}$. It follows naturally that we describe the vector of results that a vector $\mathbf{A}$ of alphas generates given data $d_{t,s}$ as $\mathbf{A}(\mathbf{d_{t,s}}) = \begin{bmatrix} a_1(d_{t,s}) & a_2(d_{t,s}) & a_3(d_{t,s}) & \dots \end{bmatrix}$.

2.2 Generating Formulaic Alphas with Evolutionary Algorithms

As we already saw we can make use of the idea of "survival of the fittest" to solve problems. But which concrete steps do we need to take if we want to predict stock market price developments?

First of we need to find a **"genetic representation"** of our candidates that can easily be mutated or crossed with the genetic representation of other candidates. In our case this will be the tree representation of a formulaic alpha - but more about this later. The genetic representation also will implicitly define

the size of our search space, which - for us - will contain all possible formulas that take the data defined in $d_{t,s}$ as arguments. In an attempt to not overfit on the given training data and decrease the search space our algorithm will only consider formulas whos tree representation has a depth that is smaller than a certain value.

To be able to measure how well a candidate (here: formulaic alpha) performs on the given task, we need to define a measure of **"fitness"**. For this we will make use of the information coefficient (IC) as presented in [4] - so the correlation between the value that the formulaic alpha returns and the actual price development. Since for making reasonable trading decisions it is important to find a set of alphas that are not too strongly correlated, we will also take into consideration how similar a candidate's predictions are to the ones of previously found formulaic alphas.

The approach of the **genetic program** is the following. First we will generate a starting population of candidates (formulaic alphas) and their genetic representations. In the next step we compute the fitness (IC with actual price development) value of each candidate. These performance measurements will then be taken into consideration when composing a new generation of candidates. For creating the new generation we will take on the best ones of the last generation (this is called elitism), crossover candidates that are pulled from the previous generation, induce random mutations in well performing candidates and insert fresh ones. We will repeat the same process of creating a follow-up generation multiple times until a computational limit or a certain fitness value is reached. The candidates in the last generation are then a collection of possibly well-performing formulaic alphas.

This whole process is non-deterministic in the hope that mutations and crossovers of random well performing candidates will create even better off-springs. Thus, different runs of the algorithm will probably and hopefully end up in different local optima of the search space.

To train this genetic program we will need a lot of examples like $d_{t,s} \in D_{generate} \subseteq D$ and the corresponding possibly realized returns $r_{t,s}$. From this set $D_{generate}$ the algorithm is going to get randomly sampled data for which it is supposed to predict an expected return. The result of the formulaic alphas is then compared to the actual return to compute the fitness. The algorithm only uses **non-timeseries data**. This means it only takes into consideration a couple of intraday parameters of a single trading day to predict future behaviour.

2.3 The Trading Bot

2.3.1 Training a Trading Bot

Let us assume we are trying to train a trading bot that predicts if buying a stock at today's closing price and selling it at the closing price after a holding period of h days will be profitable. We can create labels for historical data of a stock s, by comparing the closing price on date t with the closing price on date $t + h$. If that trade was made, the actual relative return (for a fixed holding period h) is described by

$$\mathbf{r_{t,s}} = \frac{closing(d_{t+h,s}) - closing(d_{t,s})}{closing(d_{t,s})} \tag{2.1}$$

How we choose the label for our trading bot depends not only on the actual return but also on the cost c that a trade triggers (usually around 0.3%). If we want to teach our bot to only trade if the expected return is bigger than the cost that is caused, we choose the label as follows:

$$\mathbf{l_{t,s}} = \begin{cases} 1, \text{ if } r_{t,s} - c > 0 \\ 0, \text{ if } r_{t,s} - c \leq 0 \end{cases} \tag{2.2}$$

After defining a vector of alphas $\mathbf{A}$, sourcing enough data $\mathbf{d_{t,s}}$ and extracting optimal trading decision labels $\mathbf{l_{t,s}}$, we can train our trading bot in a supervised learning manner. For a layout we choose a combination of multiple decision trees that each are trained independently. A trading decision is then generated by a majority vote of the different trees. For the training this setup will take $\mathbf{A}(\mathbf{d_{t,s}})$ as features and $\mathbf{l_{t,s}}$ as labels.

2.3.2 Decision Trees for Trading

Similar to the paper [5] we will use decision trees to turn a set of formulaic alphas into a trading decision. However, unlike in [5], we will not use them to rank all possible investment opportunities in an index and then buy the one with the highest probability to yield good returns. Our approach will be much simpler. Our trading bot will just take a stock's information and decide if buying at the closing price of the day of the information and selling at the next day's closing price will be profitable or not. This of course includes the trading fee of 0.3% per trade.

We will train three different decision trees (CatBoost, LightGBM and XGBoost) individually to make trading decisions. The final trading decision will then be dependent on the "majority vote" of the three decision trees.

Chapter 3

Genetic Program

3.1 Genetic Representation of Formulaic Alphas

Genetic Representation

For us a formulaic alpha will basically just be function that takes a set of arguments - in our case the open, closing, high, low, return (of the previous day in percent) and volume of a trading day - and returns a value that correlates with the stock market closing price of the following day. Let us consider the example equation that we have already seen 1.1:

$$Alpha\#101 = \frac{Close - Open}{High - Low} \tag{3.1}$$

To find a "genetic representation" that we can easily modify by mutating it or crossing it with other candidates' genetic representations, we have to think of other ways to represent this function. One of the more intuitive ways is to think of a function as a tree, where all non-leaf nodes are sub-functions (like division, addition, negate, absolute value, sin) and all leafs are either a real number or one of our input variables. To derive our genetic representation of a formulaic alpha it is easier to think of the way a function would be implemented if each sub-function was a method in programming that is called by handing over arguments - like in 3.2.

$$Alpha\#101 = div(sub(Close, Open), sub(High, Low)) \tag{3.2}$$

Since the arity of every sub-function is clearly defined we can even leave out the brackets and commas (3.3).

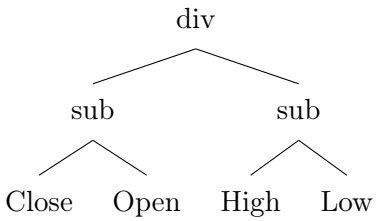
$$Alpha\#101 = div\ sub\ Close\ Open\ sub\ High\ Low \tag{3.3}$$

We finally save this into a list (3.4).

$$Alpha\#101 = [div, sub, Close, Open, sub, High, Low] \tag{3.4}$$

In the algorithms implementation this list format of a function 3.4 is its genetic representation.

However, intuitively it is easier to imagine that we are working with a tree structure. We will visualize crossovers and mutations with this more understandable representation.



3.1.1 Crossovers and Mutations

In the course of our evolutionary process we will mutate and crossover different functions to create new off-springs that might perform better than their predecessors.

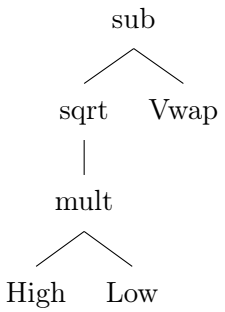
Crossover

Let us first take a look at how we crossover two given functions. The underlying idea is that if we have two well performing candidates - the parent and the donor - the mixture of these could yield an even better performance [6]. To come up with this "off-spring" we choose random sub-trees in both functions and just exchange the $parent_{subtree}$ with the $donor_{subtree}$. Let us have a look at an example.

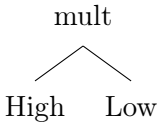
We take 1.1 as parent and the following example program from [3] as donor:

$$Alpha\#41 = \sqrt{High * Low} - Vwap \tag{3.5}$$

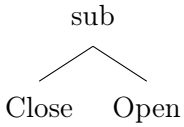
In the tree representation this is:



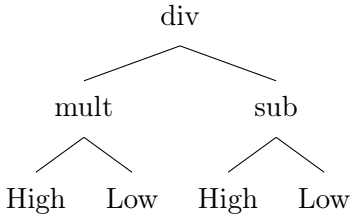
From our donor we now take a random subtree $donor_{subtree}$ - let's say:



In the parent we also choose a random subtree $parent_{subtree}$ - let's say:



We replace $parent_{subtree}$ in the parent by $donor_{subtree}$. This is how we end up with a new off-spring:



During this whole procedure, we are careful not to choose $donor_{subtree}$ as deeper than $parent_{subtree}$. Besides other things this would lead to bloating of the formulas and possible over-fitting on the given training data on which the performance of the candidates is evaluated.

Mutation

A mutation follows the idea of genetic mutations as they happen in the real world [7]. With a set probability we introduce some random changes in the genetic representation of a function. We have three different types of mutations that we can use to introduce changes into a parent function:

- sub-tree mutation:

  Once we understood the concept of a crossover between two functions, this is quite simple. Instead of taking another existing function as a donor, we randomly generate a function - of which we do not know the fitness - and take that as a donor for a crossover.

- hoist mutation:

  In this mutation we randomly choose a sub-tree of our parent function $parent_{subtree}$. We also draw a random sub-tree of that $parent_{subtree}$ which results in $parsub_{subtree}$. Now we simply replace $parent_{subtree}$ in the parent by $parsub_{subtree}$. This mostly leads to a shortening of the function and thus helps to counteract the bloating of formulas.

- point mutation:

  The point mutation is the most intuitive one. It simply chooses a node in the parent at random and replaces it with a random node of the same category. Thus, if that node is a sub-function (like add, div, abs, sin, ...) we replace that node with another randomly drawn sub-function that has the same arity. Respectively we do the same thing if the node is a real number or one of the input parameters.

These mutations are not always applied, since this would tamper with the performance of the generated functions too much. With a certain (quite low) probability we apply these mutations on off-springs that were created in a crossover of two functions. The hope is that this will lead to a better exploration of the large search space and thus enable us to find good solutions that we would otherwise not be on the path of finding.

3.2 Fitness Evaluation

To decide which ones of our created off-springs and their parents is allowed to move on to the next generation, we need a measure of their predicting performance on the stock market. In the process of genetic programs this is called the "fitness of a candidate".

3.2.1 Correlation with Stock Market Developments

As suggested by the paper [5] we can use the information coefficient (IC) presented in [4] to evaluate how well a candidate's predictions correlate with the actual stock market. Over the available historical training data it takes the average Pearson correlation between the development according to the candidate and the actual return if a long position was bought.

$$IC_{true} = \frac{1}{T}\sum_{t=1}^{T} corr(candidate_{predicted}(t), return_{actual}(t)) \tag{3.6}$$

For obvious reasons we want to find functions that have an IC that is as high as possible, since this allows us to be more confident in the prediction that an

alpha makes. Thus, to evaluate the performance of a candidate we compute the IC_{true} between its predicted development ($candidate_{predicted}$) and the actual price development ($return_{actual}$).

3.2.2 Correlation with Already-Discovered Alphas

If we just take IC_{true} as the measure of fitness of an alpha we find ourselves with the problem that the results of running the evolution process multiple times are very likely to converge to similar local optima. To keep exploring the search space more, we will also take into consideration how different a candidate is to a set of given "other functions" (*others*, OF). We do this by looking at the maximum correlation the candidate's predictions have with the predictions of *others*.

$$IC_{others} = \max_{i \in OF} \frac{1}{T} \sum_{t=1}^{T} corr(candidate_{self,\,predicted}(t), candidate_{i,\,predicted}(t)) \tag{3.7}$$

3.2.3 Overall Fitness

When evaluating the overall fitness of a candidate/potential alpha we have to take both the IC_{true} and the IC_{others} into consideration to make sure that we explore more of the search space and don't converge to solutions similar to the "other functions" *others*. For this purpose we define a max_ic. Every function that has a IC_{others} greater than this value will be inadmissible. From the available information, this is very similar to the approach that was chosen in [5]:

$$fitness = \begin{cases} 0, & \text{if } IC_{others} > max\_ic \\ IC_{true}, & \text{if } IC_{others} \leq max\_ic \end{cases} \tag{3.8}$$

However, in the context of our genetic program we will try to already incorporate the IC_{others} before it passes the max_ic. We do this in the following way:

$$fitness = \begin{cases} 0, & \text{if } IC_{others} > max\_ic \\ IC_{true} - p * IC_{others}, & \text{if } IC_{others} \leq max\_ic \end{cases} \tag{3.9}$$

The choice of p here symbolizes how strongly we want the population to diverge from "other functions" *others* even before the limit max_ic is reached. However, we have to be careful not to choose p too large, since otherwise IC_{others} dominates the fitness evaluation. This would mean that from the fitness value we learn too little about how the candidate actually performs on the true stock developments IC_{true}. We add this part to the fitness evaluation in the hope that it will steer the functions in the population away from already found alphas,

even before the maximal allowed correlation - max_ic - with already discovered alphas *others* is reached. This could intuitively be interpreted as already telling the functions that they are developing into the same direction as "other functions" *other*, before they become too similar. Otherwise we suddenly set their fitness to zero and disqualify a member of the population that could have been used to search another part of the search space.

While running the algorithm we choose p to be around values in the region $[0.01, 0.1]$. The reasoning behind this is that when ran multiple times evolutionary algorithms have a strong tendency of converging to very similar solutions and thus an IC_{others} of close to 1. We are hoping to find functions that have an IC_{true} in the range of $[0.05, 0.1]$. Thus, to make sure that

$$p * IC_{others} \not> IC_{true} \tag{3.10}$$

we have to choose

$$p \leq \frac{IC_{true}}{IC_{others}} \tag{3.11}$$

Which leaves us with approximately $p \in [0.05, 0.1]$. Since very short functions are unlikely to have a good performance IC_{true}, we include even lower values for p which leads to $p \in [0.01, 0.1]$.

Because of this we make the punishment for a high correlation with IC_{others} in a candidate's fitness dependent on the depth of the candidate. Through comparison of multiple test runs I found the population to explore the search space quite well while not getting pushed away from the already found alphas too much for the following developments of p and max_ic (dependent on the depth of the candidate d, where $d \in [0, 6]$):

$$p(d) = max((d^2 - 4 * d) * 0.01,\ 0.01) \tag{3.12}$$

$$max\_ic(d) = 1 - 0.06125 * d \tag{3.13}$$

3.3 Population Choice: Parents vs Off-springs

Given a list of candidates - also called a population - we need to come up with a next improved population that contains candidates that perform better than the initial ones. By doing this over and over again we hope to end up with a list of candidates that are way better at solving the given task than our initial population. In this context we will look at populations as "generations".

3.3.1 Generating the Next Population - Algorithm

In the following we will take a look at how we create a next generation when given a last generation and a list of "*other*" functions from which we want to differ. For this we draw random parents from our last generation, cross them and apply mutations. This creates off-springs that will compete with their parents for a spot in the next generation. How this process works exactly can be seen in Algorithm 1 and is described in further detail in the following explanations.

Code: Explanation A

We randomly draw two parents from the given *last_gen* (last generation). These are then crossed over to create two off-springs. To introduce "random genetic changes" we apply the presented mutations with a set probability p to the off-springs. After evaluating the fitness of the two parents and two off-springs we create a list - called *candidates* - that contains the four functions and their fitness values.

Code: Explanation B

From the *candidates* list we delete the functions that are already contained in the list of functions we are creating for the *next_generation*, since we do not want functions to appear multiple times in the same population. This would be a waste of resources since these "double functions" could be used to explore another part of the search space and push the population into similar local optima. However, since deleting functions in *candidates* might lead to zero or only one function being contained in *candidates*, we need to deal with the case that the functions left in *last_generation* are not diverse enough to generate equations different from the ones in *next_generation*. This is done as described in "Explanation C".

Code: Explanation C

If the *last_generation* only contains a maximum of three equations and "doesn't manage to create equations that are not contained in the next generation yet" for more than 20 rounds, we risk introducing equations being double in the population. We do this by applying a concept called elitism: we take the best performing functions from the *last_generation* and copy them into the *next_generation*.

Algorithm 1 Choosing the next generation

Given: *last_gen*, *others*, *train_data*

next_generation ← []
without_change ← 0
elitism ← best four functions from *last_gen* according to fitness
population_size ← *len*(*last_gen*)

while *len*(*next_generation*) < *population_size* **do**
▷ For the following: Explanation A
parent_a, *parent_b* ← *random.sample*(*last_gen*)
$offspring\_a \leftarrow crossover(parent = parent\_a, donor = parent\_b)$
$offspring\_b \leftarrow crossover(parent = parent\_b, donor = parent\_a)$

apply mutations to $offspring\_a$ with probability p
apply mutations to $offspring\_b$ with probability p

$fit\_par\_a \leftarrow fitness(parent\_a, others, train\_data)$
$fit\_par\_b \leftarrow fitness(parent\_b, others, train\_data)$
$fit\_of\_a \leftarrow fitness(offspring\_a, others, train\_data)$
$fit\_of\_b \leftarrow fitness(offspring\_b, others, train\_data)$

$candidates \leftarrow [(parent\_a, fit\_par\_a), (parent\_b, fit\_par\_b)]$
$candidates.extend([(offspring\_a, fit\_of\_a), (offspring\_b, fit\_of\_b)])$

▷ For the following: Explanation B
delete functions from *candidates* that are in *next_generation*

if *len*(*candidates*) ≥ 2 **then**
add the best two functions from *candidates* to *next_generation*
without_change ← 0
remove *parent_a* and *parent_b* from *last_gen*
else
without_change+ = 1
end if
▷ For the following: Explanation C
if *without_change* > 20 &
len(*next_generation*) < *population_size* &
len(*last_population*) ≤ 3 **then**
add best function f from *elitism* to *next_generation*
delete f from *elitism*
end if
end while

return *next_generation*

3.3.2 Elitism

As shown in [8], the usage of the concept of "elitism" can help reduce bloat in populations and thus speed up the process of finding high quality candidates. The idea is quite simple: we choose the best performing candidates of the previous generation and add them to the next generation that we are trying to generate. It is important that we do not change the candidate at all. By doing this we make sure that our evolutionary process does not "accidentally forget" about the best performing candidates.

In our "Parent vs Off-spring" approach of evolving a generation, we only make use of elitism in cases where we can't come up with candidates that are not contained in our next generation ourselves. We know that the candidates we introduce into the next generation with the "elitism" approach will not be worse (in terms of fitness) than all the candidates in the last generation.

Later on we will also make use of this idea in a broader sense, when introducing the "$warm\_gp$" method in Chapter 4.2.

3.3.3 Competition between Parent and Off-spring

While actually generating new functions that will be put into a next generation we can take a lot of design decisions. For this algorithm we use a competition between pairs of parents (that are drawn from the last generation) and the off-springs that their crossovers create. The usage of this concept is roughly mentioned in the [5] paper and can be researched in more depth in [9].

One of the fundamental ideas is that we can make sure that while stepping through multiple generations, the performance of our candidates does not get worse. By having parents compete with their off-springs for a spot in the next generation, we also encourage our population to stay diverse so that not all candidates converge to a similar solution.

3.4 Running the Evolutionary Process

In the following we will make a given starting set of functions evolve and compete with each other. We do this by over and over applying Algorithm 1. Stepping through multiple generations we are using elitism and the concept of parent-offspring competition. This results in Algorithm 2. In the code implementation of this part of the algorithm we also include an "early-stopping" mechanism, that makes sure that in case the populations stop getting better for a lot of iterations we stop our algorithm and return the population. This helps us shorten the real-life run-time of our algorithm.

Algorithm 2 Running the genetic program

Given: *start_pop*, *others*

generation_limit ← 200
cur_pop ← *start_pop*
for each $g \in range(generation\_limit)$ **do**
    new_gen ← *algo*1(*last_gen* = *cur_pop*, *others*)
    cur_pop ← *new_gen*
end for

▷ Return the full last population

return *cur_pop*

3.4.1 Search Space Growth with Function Depth

In the actual implementation of Algorithm 2 the parameter *generation_limit* (set to 200) - similar to the fitness calculation - is dependent on the depth d of the candidates in the given *start_pop*. Functions of a lower depth will take less "iterations" (amount of usages of Algorithm 1) to explore most of the possible functions of a certain depth, thus we can choose the *generation_limit* relatively small. However, the size of the functions' search space for a growing depth d increases due to two reasons.

(For simplicity let us only consider functions whose tree representation is "full", meaning we only use sub-functions with an arity of two and leafs only exist at the maximal depth the tree has. This will yield a simple **lower bound** for the actual growth of the search space.)

1. The length of the function doubles every time we add another level of depth to the function space. The higher the depth d, the more the length of the function will grow with a further increase in depth. Thus, the amount of positions that a candidate "has to fill" grows like in 3.14.

$$amount\_positions = 2^d \tag{3.14}$$

2. With the *amount_positions* also the number of possible choices of functions for these positions $functions\_possible$ increases. The development of $functions\_possible$ dependent on the amount of positions that are to be taken by a sub-function in a candidate's tree representation $amnt\_pos\_for\_functions$ grows like shown in 3.15, where n is the amount of available sub-functions with an arity of two (like add, sub, div, mult, max, ...).

$$functions\_possible = n^{amnt\_pos\_for\_functions} \tag{3.15}$$

Thus for a fixed n the search space grows proportionally to 3.16 when increasing the depth of the functions, where k is the amount of possible leaf elements in the tree representation of a function (namely quantized real numbers and given parameters). The *generation_limit* when exploring this search space for a depth d should grow proportionally with 3.16 in case we want to guarantee that we allow the algorithm to explore the search space of functions with higher depth just as well as the search space of lower depth functions.

$$\mathcal{O}\left(n^{2^{d-1}} * k^{2^d - 2^{d-1}}\right) \tag{3.16}$$

In our genetic program, however, a population is very unlikely to explore the whole search space and will probably get stuck in a local optimum way before this.

When a population gets stuck in a local optimum it often just keeps losing diversity when running more evolution steps - according to observations when running the code. This implies that it loses the ability to explore distant parts of the search space, since the population loses the diversity of functions necessary to assemble totally unexplored combinations of sub-functions.

Thus the in 3.16 shown lower bound of the growth of the search space is interesting to understand from how many possible functions our genetic program tries to find the small subset that correlates well with the stock market. However, it should only be in the back of our heads when choosing the *generation_limit* of the genetic program 2 dependent on the populations' functions' depth d.

Chapter 4

Alpha-Generating Algorithm

4.1 The Overall Algorithm

In the following we will introduce the overall algorithm used to generate a diverse set of alphas. It makes use of the just presented genetic program (that we will call "$algo2$" in the pseudo-code).

Algorithm 3 Overall Alpha-Generating Algorithm

Given: $train\_data$
$final\_gp[0] \leftarrow [[Open], [Close], [Low], [High], [Volume], [Return\_yesterday]]$
$max\_tree\_depth \leftarrow 5$
$gene\_pool\_size \leftarrow 25$
$startup = []$
for each $d \in range(1, max\_tree\_depth + 1)$ **do**
  for each $f \in range(gene\_pool\_size)$ **do**
    $pop \leftarrow warm\_gp(gp = final\_gp[d-1][:],\ startup = startup[d])$
    $best \leftarrow algo2(start\_pop = pop,\ others = final\_gp[d][:])$
    $best\_cand \leftarrow best[0]$
    $final\_gp[d][f] \leftarrow best\_cand$

    ▷ Recycle the rest of the final population
    $startup[d].extend(best[1:])$
  end for
end for

return $final\_gp[max\_tree\_depth][:]$

$final\_gp$ is a list that contains a list of the best found functions for every depth d. Of course we start with a set of functions that have the depth zero - so just the parameters that we are given. To now generate the $gene\_pool$ at the next depth d - so the functions contained in $final\_gp[d]$ - we take into consideration the list of functions from the previous depth $d-1$, so $final\_gp[d-1]$. We apply a

method $warm\_gp$ that will take functions of the lower depth $d-1$ and recombine them into functions of depth d that will be used as the starting population pop for the genetic program.

Now that we have found a population pop of functions that are re-combinations of functions of lower depth $d-1$, we will start a genetic program we call $algo2$ that implements Algorithm 2. During the whole evolutionary process that $algo2$ executes we evaluate the fitness with respect to how different a candidate is from already found functions of depth d, so $final\_gp[d][:]$ (besides looking at the correlation with stock price developments). Finally the best function found by $algo2$ is added to our list of discovered functions of depth d, namely $final\_gp[d]$. The rest of the found functions will be added to a list called *startup* that will be handed over to the $warm\_gp$ method to recycle the other learned functions from the final generation and initialize the starting population pop for the next genetic program. When adding to the list *startup* we are careful to not add candidates that are already contained in *startup*.

The rough structure of this procedure is leaned on an idea proposed in [5]: We first try to find the best possible functions of depth $d-1$. We assume that well performing functions of higher depth d combine the features expressed by functions of depth $d-1$. Thus we iterate through different depths, starting with our raw parameters at $d=0$.

What was not presented in this paper [5] is that we recycle the "non-best" candidates the genetic program generates. This is done by collecting them in the *startup* list and considering them when initializing the next starting population of the genetic program.

4.2 Warm Gene Pool: $warm\_gp$

As proposed in [5] we are going to try to shorten the train-time and increase the performance of our final population by using an approach called the "warm start method". When we choose the initial population from which the genetic program starts, we could randomly generate functions to make up that starting population. However, since random functions are really improbable to predict stock market prices well, at first we are going to randomly generate more functions than needed. From these we will choose the set of functions that performs best with respect to the prediction of stock market developments and the difference to a list of already given functions in *others*. Later on we will also make use of the candidates collected in *startup* that went through the genetic program, but did not make it into the $final\_gp$.

Let us describe our motivation throughout $warm\_gp$ shown in Algorithm 4:

We start by generating a lot of functions ($factor\_more * population\_size$ many) and putting them all into a formula pool. During this we are careful not

Algorithm 4 warm_gp

Given: gp (gene pool), $startup$, $train\_data$

$population\_size \leftarrow 50$
$factor\_more \leftarrow 10$
$amount\_formulas \leftarrow (factor\_more * population\_size)$
$formula\_pool \leftarrow startup$

▷ Generating missing amount of candidates for the $formula\_pool$
while $len(formula\_pool) < amount\_formulas$ **do**
    $f \leftarrow$ random draw from sub-functions (like add, sub, sin etc.)
    if $f.arity == 1$ **then**
        $gene\_a \leftarrow$ random draw from gp
        $fresh\_candidate \leftarrow f(gene\_a)$
    else ▷ $f.arity == 2$
        $gene\_a \leftarrow$ random draw from gp
        $gene\_b \leftarrow$ random draw from gp
        $fresh\_candidate \leftarrow f(gene\_a, gene\_b)$
    end if
    $formula\_pool.extend(fresh\_candidate)$
end while

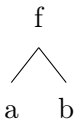
▷ Choosing the best candidates of $formula\_pool$
$pool\_fitness \leftarrow fitness(formula\_pool, train\_data)$
$pool = (formula\_pool, pool\_fitness)$
$sorted\_pool \leftarrow$ sort $pool$ descending according to $pool\_fitness$

▷ Try to decrease high correlation among returned candidates
$best\_cand\_twice \leftarrow sorted\_pool[: 2 * population\_size]$
$indeces \leftarrow [1, 3, 5, ..., (2 * population\_size - 1)]$
$best\_cand \leftarrow$ take candidates at positions $indeces$ from $best\_cand\_twice$

return $best\_cand$

to introduce formulas multiple times. To generate a formula of depth d, we draw a random "sub-function" f - like add, sub, sin etc. Depending on the arity of the drawn sub-function, we draw candidates a, b of depth $d-1$ from the gene pool given to the $warm\_gp$ algorithm in the argument gp. We then simply arrange the drawn candidates as arguments of f.

In a tree structure this would look like the following: (for arities two and one)



and

It can be easily seen that this simply adds another layer to the depth of the candidates a, b. Thus the depth of the resulting trees is: $(d-1)+1 = d$.

Given a list of candidates *startup* we have a better initial guess for the functions to put into the starting population of the following genetic program. This is why we included the candidates from *startup* in $formula\_pool$. Afterwards we will look for a subset (of this generated pool of formulas $formula\_pool$) that has the size of the needed population that will be returned. To start this process we evaluate the fitness of each candidate in the formula pool with respect to the correlation to stock price developments (fitness).

Since with a growing size of the *gene_pool* of the current depth d also the amount of candidates provided by *startup* will grow, we have to be careful not to return a starting population that is too highly correlated and lacks diversity. For this purpose we don't simply take the best $population\_size$ many candidates from *pool*, but only take every second candidate in *pool* until we have filled our starting population.

Chapter 5

Performance Assessment

Let us have a look at some of the formulaic alphas that the above described genetic program generates. We trained the algorithm four separate times with different gene pool and population sizes (mostly close to: Gene Pool Size = 25, Population Size = 40). Each training took roughly 44h on a GeForce RTX 2080 Ti.

For the following performance assessment we have to keep in mind that we just searched for alphas that only take the parameters Open, Close, Low, High, Volume and the yesterday's Return. There was no additional data provided that a formulaic alpha can process to return values that correlate with the stock market development. This includes that no time-series data was used.

We will refer to formulaic alphas that were generated using our genetic program as "genetic formulaic alphas".

5.1 Stock Market Patterns

5.1.1 Formulaic Alphas found by the Genetic Program

The Best Generalizing Alphas

Let us have a look at a subset of the formulaic alphas that were found by the genetic program. The diversity and performance of these formulas was evaluated on the data set on which the genetic program that generated them was trained, namely 2014 to 2019. We then chose a subset of formulaic alphas that are weakly correlated amongst each other but still perform well on the given time frame (2014 to 2019).

In the following these formulaic alphas and their performance on the stock market data (Nasdaq 100) from 2014 to 2021 are listed:

1. Correlation to stock market: 0.071

$$Genetic\#1 = \max\left(\frac{Low}{Open} * \frac{High}{Close},\ \cos(\min(Close,\ Open))\right) \tag{5.1}$$

2. Correlation to stock market: 0.065

$$Genetic\#2 = \left(\frac{Low}{Close} - Return\right) + \left(\frac{Low}{Close} * cos(Return)\right) \tag{5.2}$$

3. Correlation to stock market: 0.057

$$Genetic\#3 = min\left(0.938,\ \frac{Low}{Close}\right) \tag{5.3}$$

4. Correlation to stock market: 0.055

$$\begin{aligned} Genetic\#4 = {} & min(Low - Open,\ 0.133) \\ & + max(cos(Volume),\ High - Close) \end{aligned} \tag{5.4}$$

5. Correlation to stock market: 0.051

$$Genetic\#5 = max\left(min(Low - Open, 0.133),\ \frac{Low - Close}{\sqrt{(High)}}\right) \tag{5.5}$$

6. Correlation to stock market: 0.46

$$Genetic\#6 = \frac{(High - Close) - (Return)}{(Low + Return) + \frac{-0.177}{Close}} \tag{5.6}$$

7. Correlation to stock market: 0.45

$$Genetic\#7 = max\left(\frac{High}{Close} - |Return|,\ min\left(\frac{Low}{Open},\ \frac{High}{Close}\right)\right) \tag{5.7}$$

8. Correlation to stock market: 0.40

$$Genetic\#8 = \frac{0.386}{Low} + \frac{Low}{Close} \tag{5.8}$$

The average pairwise correlation of formulaic alphas in this set is 0.49.
The maximal pairwise correlation of formulaic alphas in this set is 0.59.
Their average correlation with the stock market is 0.054.

5.1.2 The Development of Stock Market Patterns

What is interesting about the performance of the formulaic alphas found with the genetic program is that there seems to be some sort of underlying force that changes the correlation of the formulaic alphas with stock market developments year by year.

The varying performance from year 2005 to 2021 of the eight "best generalizing alphas" can be seen in Figure 5.1.

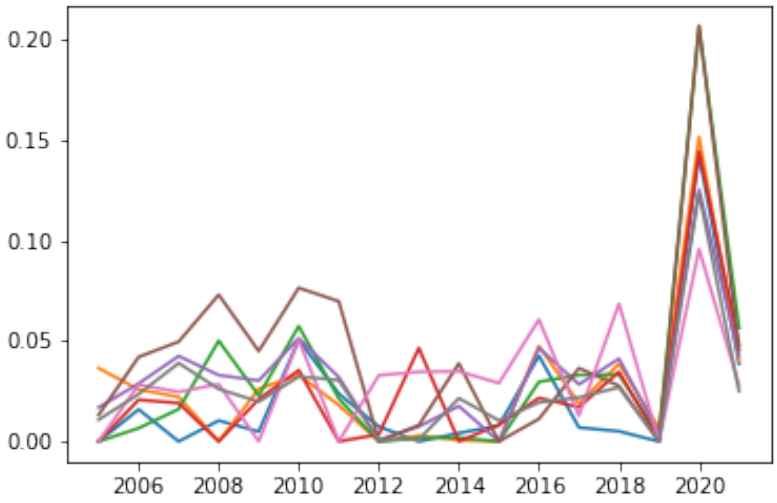


Figure 5.1: Best Generalizing Alphas
brown: *Genetic*#1 green: *Genetic*#2
blue: *Genetic*#3 grey: *Genetic*#4
purple: *Genetic*#5 red: *Genetic*#6
pink: *Genetic*#7 orange: *Genetic*#8

When we average the yearly performance over the generalizing alphas we can observe the following development visible in 5.2.

In the set of 101 formulaic alphas presented in [3], there only exist two alphas that can be computed using the same data as it is available for the candidates of our genetic program. These are the 101st (seen in 1.1) and the 54th (in 5.9) alpha.

$$Alpha\#54 = -1 * \frac{(Low - Close) * (Open^5)}{(Low - High) * (Close^5)} \tag{5.9}$$

Their correlation to the stock market price over the time frame 2005 to 2021 can be seen in Figure 5.3, where they are compared to the average performance

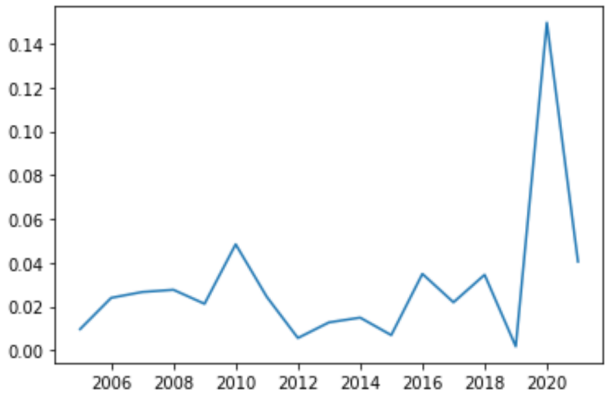


Figure 5.2: Average over Generalizing Alphas

of the "best generalizing alphas" (blue).

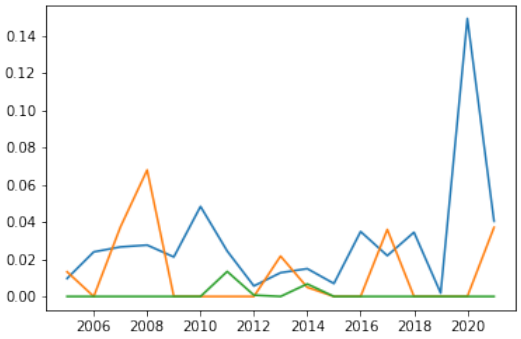


Figure 5.3: Genetic vs 101 Alphas
green: *Alpha*#101 orange: *Alpha*#54

When looking at this development, it is important to mention that the algorithms that generated the alphas whose performance we are measuring have been trained on data from 2014 to 2019. The generating process has not seen any data before 2014 or after 2019.

In an attempt to understand what drives the correlation development of the alphas generated with the genetic program (in Figure 5.2) we can look at multiple parameters that influence financial market behaviour and compute the correlation between the change in these parameters and the observed performance development of our eight genetic alphas (blue, left scale for the following figures).

Market Returns

Let us have a look at the development of the yearly closing price (in Figure 5.4) of the market that we evaluate the performance of the found alphas on, namely the Nasdaq 100 (red, right scale) [10].

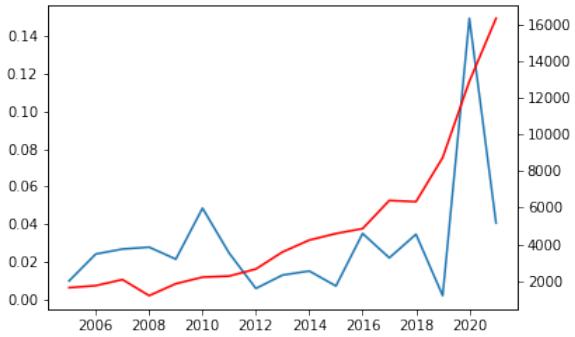


Figure 5.4: Nasdaq 100 Closing Prices

The Pearson correlation between the Nasdaq 100 and the performance development of the alphas is 0.51.

Looking at the resulting annualized returns (red, right scale) of the Nasdaq 100 in Figure 5.5 we find a correlation of 0.59.

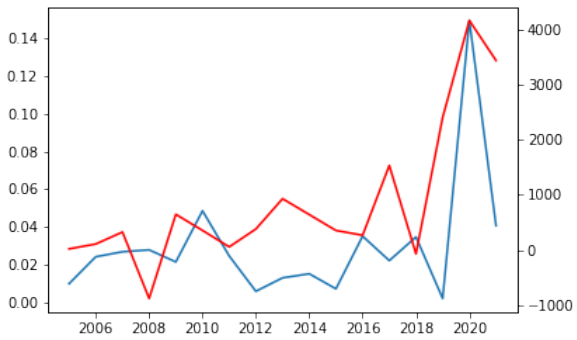


Figure 5.5: Nasdaq 100 Annualized Returns

Market Volatility

Alternatively we can also take a look at the annualized volatility that the Nasdaq 100 (red, right scale) was exposed to (Figure 5.6). For this we will make use of

the "CBOE NASDAQ 100 Volatility Index" (VXNCLS) and source our data from [11].

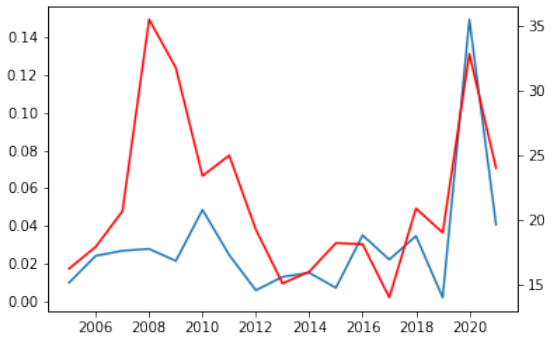


Figure 5.6: Nasdaq 100 Volatility

The Pearson correlation between the annualized VXNCLS and the performance development of the alphas is 0.54.

Balance Sheet of the Federal Reserve

It gets even more interesting when we take a look at the development of the balance sheet of the federal reserve [12] (red, right scale) in Figure 5.7.

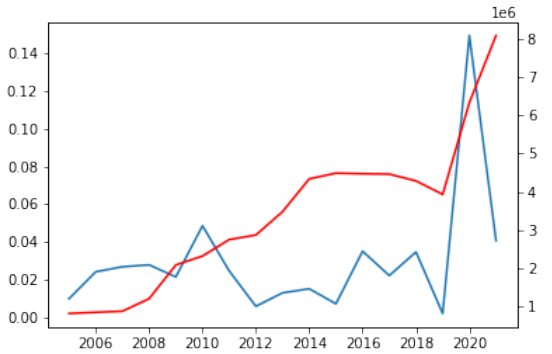


Figure 5.7: Fed Balance Sheet

The correlation between the federal reserve's balance sheet and the alpha performance from 2005 until 2021 is only 0.41. However, when we take a look at the changes in the balance sheet from year to year (Figure 5.8) and compare that to the development of the performance of our alphas we find a correlation of 0.73.

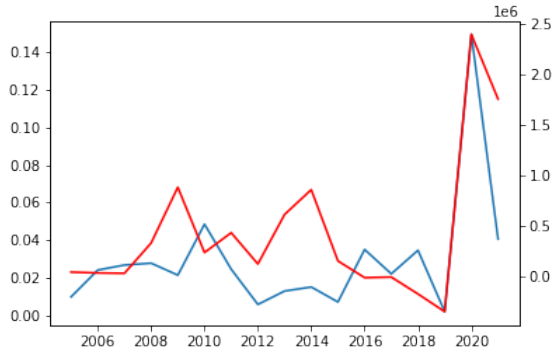


Figure 5.8: Changes in Fed Balance Sheet

To also get a feel for the development of the correlation of *Alpha*#101 and *Alpha*#54 with the just mentioned macroscopic developments we can also consider their correlation which can be found in Table 5.1.

Table 5.1: Annualized Correlations

	Alpha#54	Alpha#101
Stock Market Closing	0.031	0.0
Stock Market Returns	0.0	0.0
Stock Market Volatility	0.230	0.014
Fed Balance Sheet	0.0	0.0
Fed Balance Sheet Changes	0.053	0.066

What is striking about these results is that the formulaic alphas presented in [3] are almost not at all correlated with any of the considered macroscopic developments - unlike the genetic formulaic alphas. This allows the assumption that the alphas that can be found using genetic programs are in general more dependent on developments of the overall stock market than the classical formulaic alphas.

5.2 Trading Bot with Genetic Enhancements

To get a feel for the performance of the genetic formulaic alphas, we will train two simple trading bots - one with a subset of the 101 formulaic alphas in [3], and one which additionally uses the alphas generated with the genetic program. We then compare the returns these trading bots yield on unknown test data.

We will train the trading bots with data from 05.02.2014 until 04.02.2019,

and test with data from 05.02.2019 until 05.02.2021. The used data is from the 100 stocks that are contained in the Nasdaq 100 as of December 2021. To train the trading bots we explore the search space of hyper parameters for the used decision trees by applying a Bayesian optimization [13] with 500 samples and 20 random explorations. The best hyper parameters that this process returns are going to be used to train each final trading bot.

It is important when observing the returns of the different trading bots to consider that in the testing period of about two years with 100 stocks there exist way more possibilities for trading than trades that can actually be made. One day contains a trading option for each one of the 100 stocks in the Nasdaq 100. The trading bot decides for each one of them independently if returns could be made by buying at the days closing price and selling at the next days closing price. One strategy would be to evaluate the predicted returns of all stocks for a day, buy the one with the highest and sell it the next day. With our simple trading bot it might make sense to spread the money over all trades the bot sees to be profitable on a day. Thus we cannot make any statement about yearly returns and absolute figures like "total trade return". What is however interesting is the average performance of the trades the trading bot decides to make and the amount of trades it makes relative to the amount of trades it could have made. This could be interpreted as the amount of patterns the algorithm found.

The shown returns are already reduced by the trading cost of 0.3% per trade.

5.2.1 Trading Bot - using only the 101 Alphas

Since it is tricky to implement all 101 of the formulaic alphas presented in [3] we will only use a subset of 50 alphas. Most of them work on time-series data that goes back up until 280 days before the day for which the trading decision (buy long or don't buy) has to be made.

CatBoost Trading Bot

We train a CatBoost decision tree with the following hyper parameters that were found by the Bayesian optimization:

'bagging_temperature': 0.8638296708208457, 'border_count': 186, 'depth': 4, 'iterations': 378, 'l2_leaf_reg': 5, 'learning_rate': 0.03342078295759463, 'random_strength': 1.1005181643333055

On the unseen test data this will yield returns with the performance that can be seen in 5.9.

```
Trading options: 46718
Trades made: 1232
Winning trades: 729
Losing trades: 503
Average trade return: 0.007420899065847209
Total trade return: 9.142547649123761
```

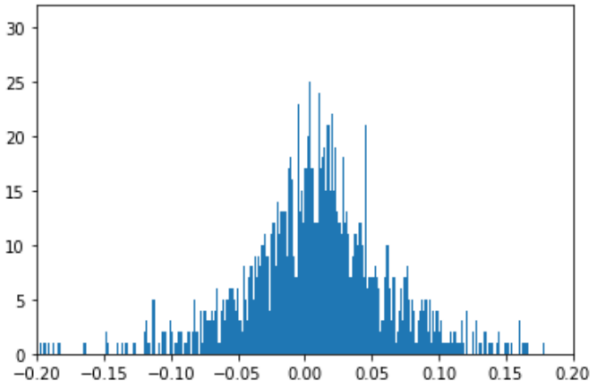


Figure 5.9: CatBoost Trading Bot Returns

XGBoost Trading Bot

We train a XGBoost decision tree with the following hyper parameters that were found by the Bayesian optimization:

'alpha': 0.03207593189500518, 'eta': 0.025628614081187853, 'gamma': 0.2902154 7757560384, 'max_depth': 4, 'eval_metric': 'aucpr', 'booster': 'gbtree'

On the unseen test data this will yield returns with the performance that can be seen in Figure 5.10.

LightGBM Trading Bot

We train a LightGBM decision tree with the following hyper parameters that were found by the Bayesian optimization:

'bagging_fraction': 0.7556865138865563, 'feature_fraction': 0.6322722245720651, 'lambda_l1': 4, 'lambda_l2': 25, 'learning_rate': 0.15259903879711864, 'max_depth': 9, 'min_data_in_leaf': 4574, 'min_gain_to_split': 1.6814097950 584883, 'num_leaves': 2180

On the unseen test data this will yield returns with the performance that can be seen in Figure 5.11.

```
Trading options: 46718
Trades made: 377
Winning trades: 210
Losing trades: 167
Average trade return: 0.006515181781562214
Total trade return: 2.4562235316489547
```

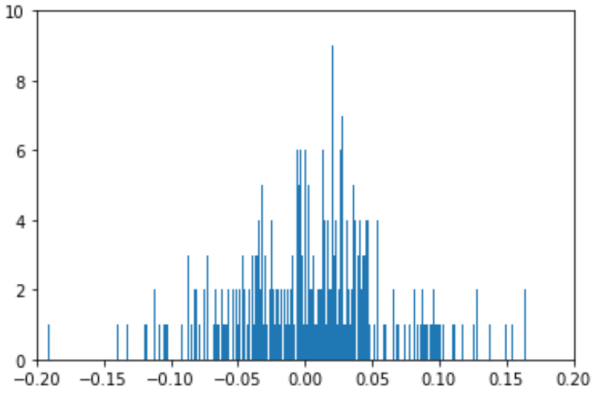


Figure 5.10: XGBoost Trading Bot Returns

Majority Vote for a Trading Decision

Now we take the three shown decision trees and make them act together to improve the trading performance. For each possible trade the CatBoost, XGBoost and LightGBM trading bot give their vote if the trade should be made or not. If the majority votes for "yes" (1), the trade is made. This leads to returns with the performance that can be seen in Figure 5.12:

5.2.2 Trading Bot - with Additional Genetic Features

We will train a second simple trading bot that will not only take the subset of the 101 formulaic alphas from [3] as input features, but also the eight "best generalizing alphas" presented in 5.1.1. We remember that the genetic alphas **only** take the six data points (Open, Close, Low, High, Volume and the previous day's Return) that are given for the trading day for which we want to decide if we should buy or sell.

CatBoost Trading Bot - Genetic

We train a CatBoost decision tree with the following hyper parameters that were found by the Bayesian optimization:

'bagging_temperature': 0.733659960722984, 'border_count': 28, 'depth': 6, 'iterations': 792, 'l2_leaf_reg': 20, 'learning_rate': 0.01, 'random_strength':

```
Trading options: 46718
Trades made: 4199
Winning trades: 2268
Losing trades: 1931
Average trade return: 0.002657683094081732
Total trade return: 11.159611312049194
```

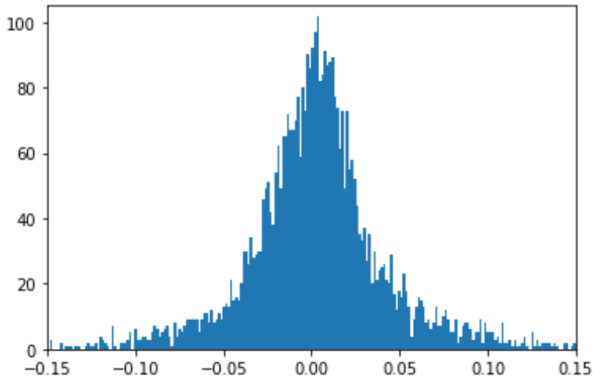


Figure 5.11: LightGBM Trading Bot Returns

```
Trading options: 46718
Trades made: 1093
Winning trades: 648
Losing trades: 445
Average trade return: 0.008759850026515676
Total trade return: 9.574516078981635
```

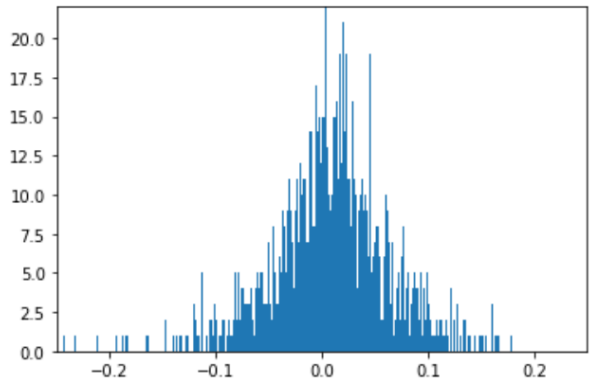


Figure 5.12: Majority-Vote Trading Bot Returns

1.7939742879332983

On the unseen test data this will yield returns with the performance that can be seen in 5.13.

We can see that the average return (from 0.74% to 1.04%) and the amount of winning trades found (from 729 to 890) both increased when we add the genetic formulaic alphas.

```
Trading options: 46718
Trades made: 1496
Winning trades: 890
Losing trades: 606
Average trade return: 0.01047135709716163
Total trade return: 15.665150217353798
```

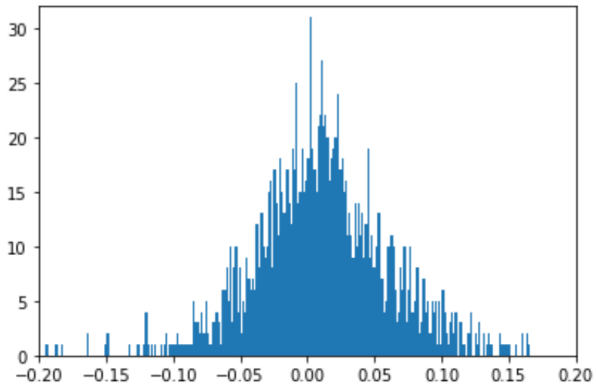


Figure 5.13: CatBoost Trading Bot Returns - Genetic

XGBoost Trading Bot - Genetic

We train a XGBoost decision tree with the following hyper parameters that were found by the Bayesian optimization:

'alpha': 0.051758279717249445, 'eta': 0.01864992642556822, 'gamma': 0.6851439 687822579, 'max_depth': 5, , 'eval_metric': 'aucpr', 'booster': 'gbtree'

On the unseen test data this will yield returns with the performance that can be seen in Figure 5.14.

As we can see the performance of the XGBoost trading bot has dramatically improved when we enhance the given input features with out generalizing alphas. Not only did the amount of found winning trades go up from 210 to 1010, but also the average return per trade was improved from 0.65% to 0.76%.

```
Trading options: 46718
Trades made: 1728
Winning trades: 1010
Losing trades: 718
Average trade return: 0.007627006713215816
Total trade return: 13.17946760043693
```

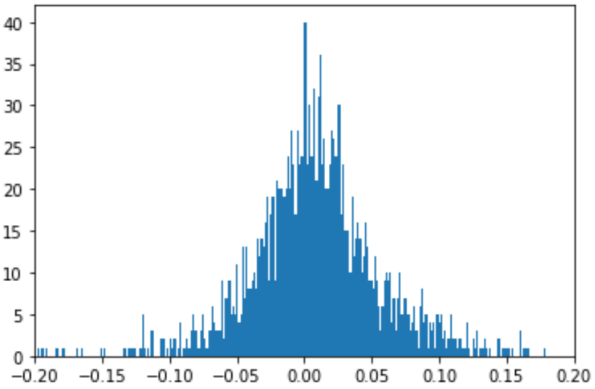


Figure 5.14: XGBoost Trading Bot Returns - Genetic

LightGBM Trading Bot - Genetic

We train a LightGBM decision tree with the following hyper parameters that were found by the Bayesian optimization:

'bagging_fraction': 0.7726724919682233, 'feature_fraction': 0.5834499497443935, 'lambda_l1': 21, 'lambda_l2': 6, 'learning_rate': 0.053237143963974255, 'max_depth': 5, 'min_data_in_leaf': 515, 'min_gain_to_split': 0.79172193339 59962, 'num_leaves': 299

On the unseen test data this will yield returns with the performance that can be seen in Figure 5.15.

Also the performance of the LightGBM based trading bot has improved in terms of average return per trade. However, it seems like the the genetic features made the trading bot more careful about making the decision to trade ("Trades made" went from 4199 to 2634).

Majority Vote for a Trading Decision - Genetic

We take the three genetically enhanced decision trees and make them act together to improve the trading performance. This yields the returns shown in 5.16.

As we can see, the performance of the overall trading bot did not increase in the "average trade returns", it actually did decrease a little bit from 0.00875 to 0.00849. However, the amount of successfully identified patterns (winning

```
Trading options: 46718
Trades made: 2623
Winning trades: 1490
Losing trades: 1133
Average trade return: 0.0048007762094175385
Total trade return: 12.592435997302204
```

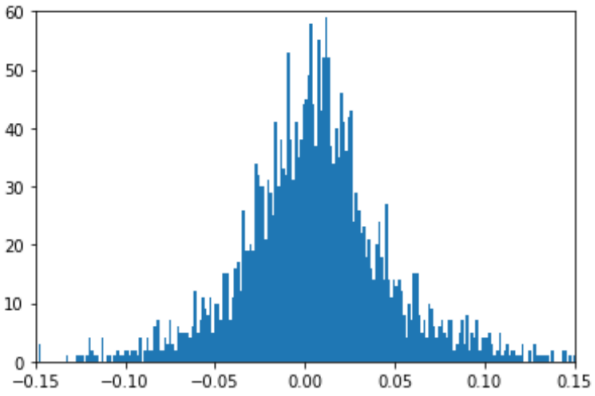


Figure 5.15: LightGBM Trading Bot Returns - Genetic

```
Trading options: 46718
Trades made: 1695
Winning trades: 997
Losing trades: 698
Average trade return: 0.008492490763720405
Total trade return: 14.394771844506087
```

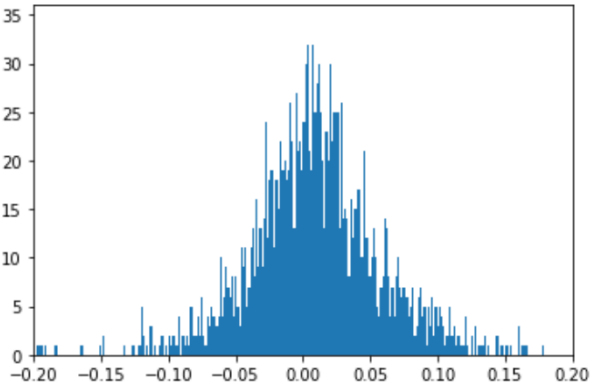


Figure 5.16: Majority-Vote Trading Bot Returns - Genetic

trades) went up from 648 to 997. Thus the average trade will yield a little lower returns for the genetically enhanced trading bot, but we will find more trading opportunities then with the standard bot. This makes sense since the found genetic alphas are supposed to find patterns in the stock market by correlating with it. Thus, the trading bot should find more patterns when they are included. In the whole "alpha generating" and "trading bot training" process there was never a ranking of higher or lower returns of a trade done. It was always just about profitable vs non-profitable trades (label "0" vs label "1").

This allows the assumption that the generated genetic formulaic alphas indeed help - even for a simple trading bot - to recognise patterns in the stock market.

Chapter 6

Conclusion

As we learned genetic programs - like the one presented - can be used to generate formulaic alphas that show a correlation to developments of stock prices. In the frame of this thesis we explored and back-tested the performance of such alphas that only consider six parameters of a single day to return a value that correlates to the true stock price development. The natural extension would now be to implement the same algorithm but also consider sub-functions that take time-series data as input parameters which will probably yield much better results. For this it would probably be a good start to quantize the length of the time-series data, to not increase the size of the search space 3.4.1 too much.

It was interesting to find a high correlation of the performance of genetic formulaic alphas with the changes in the federal reserves balance sheet. Additionally, we found that for a simple trading bot adding the generated features helps to recognise stock market patterns better. The overall approach and algorithm were roughly motivated by the method described in the paper [5]. Due to missing details for the implementation of the presented AutoAlpha algorithm, I had to come up with quite many own solutions to make the algorithm work. These included, but were not limited to the concept of *startup* in *warm_gp*, depth dependent fitness and generation limits and an adaption of the parent vs off-spring competition.

We can summarize that much speaks in favor of the statement that formulaic alphas generated with genetic programs are insightful and help with recognizing stock market patterns. However - unlike classical formulaic alphas like in [3] - their performance is correlated to some of the macroscopic changes of the stock market and thus have to be taken with caution when applied in an unknown economic environment.

Bibliography

[1] CERN. Cosmic rays: particles from outer space. [Online]. Available: https://home.cern/science/physics/cosmic-rays-particles-outer-space

[2] N. Globus and R. D. Blandford, "The chiral puzzle of life," May 2020.

[3] Z. Kakushadze, "101 formulaic alphas," Dec. 2015.

[4] R. C. Grinold and R. N. Kahn, "Active porfolio management," Jan. 2000.

[5] T. Zhang, Y. Li, Y. Jin, and J. Li, "Autoalpha: an efficient hierarchical evolutionary algorithm for mining alpha factors in quantitative investment," Feb. 2020.

[6] W. M. Spears and V. Anand, "A study of crossover operators in genetic programming," Jul. 1992.

[7] J. H. Holland, "Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence," Apr. 1992.

[8] R. Poli, N. F. McPhee, and L. Vanneschi, "Elitism reduces bloat in genetic programming," 2008.

[9] J. E. Smith and F. Vavak, "Replacement strategies in steady state genetic algorithms: Dynamic environments," in *Journal of Computing and Information Technology - CIT 7*, 1999.

[10] Annual development of the nasdaq 100 index from 1986 to 2021. [Online]. Available: https://www.statista.com/statistics/261720/annual-development-of-the-sunds-500-index/

[11] C. B. O. Exchange. Cboe nasdaq 100 volatility index. [Online]. Available: https://fred.stlouisfed.org/series/VXNCLS#0

[12] B. of Governors of the Federal Reserve System (US). Assets: Total assets: Total assets (less eliminations from consolidation): Wednesday level. [Online]. Available: https://fred.stlouisfed.org/series/WALCL#0

[13] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of bayesian methods for seeking the extremum," in *Towards Global Optimization, 2:117–129*, 1978.

Appendix A

Data - Correlation Computations

In the following I will list the concrete annualized numbers that were used when creating the figures used in the "Performance Assessment" chapter.

Table A.1: Annualized Nasdaq100 Data used for Computation of Correlations

	Nasdaq100 Closing	Nasdaq100 Returns	Nasdaq100 Volatility
2005	1645.2	24.08	16.29
2006	1756.9	111.7	17.97
2007	2084.93	328.03	20.72
2008	1211.65	-873.28	35.51
2009	1860.31	648.66	31.79
2010	2217.86	357.55	23.44
2011	2277.83	59.97	25.02
2012	2660.93	383.1	19.3
2013	3592	931.07	15.14
2014	4236.28	644.28	16.05
2015	4593.27	356.99	18.26
2016	4863.62	270.35	18.17
2017	6396.42	1532.8	14.06
2018	6329.96	-66.46	20.93
2019	8733.07	2403.11	19.06
2020	12888.3	4155.23	32.86
2021	16320.1	3431.8	24.05
Source	[10]	[10]	[11]
Used in	Figure 5.4, Table 5.1	Figure 5.5, Table 5.1	Figure 5.6, Table 5.1

Table A.2: Annualized Fed Data used for Computation of Correlations

	Fed Balance Sheet	Fed Balance Sheet Changes
2005	814064	39082
2006	845224	31160
2007	872610	27386
2008	1201361	328751
2009	2083665	882304
2010	2317319	233654
2011	2747857	430538
2012	2867655	119797
2013	3479263	611608
2014	4337664	858401
2015	4487953	150289
2016	4472130	-15823
2017	4462194	-9936
2018	4284306	-177888
2019	3930978	-353329
2020	6332639	2401662
2021	8091062	1758422
Source	[12]	[12]
Used in	Figure 5.7, Table 5.1	Figure 5.8, Table 5.1

Table A.3: Annualized Correlation of Genetic Alphas with Stock Market

	Genetic#1	Genetic#2	Genetic#3	Genetic#4
2005	0.0132	0	0	0.0108
2006	0.0421	0.0066	0.016	0.0236
2007	0.0497	0.0161	0	0.039
2008	0.073	0.0501	0.0104	0.0261
2009	0.045	0.0228	0.0049	0.0198
2010	0.0765	0.0574	0.0506	0.0323
2011	0.0697	0.0214	0.0238	0.0305
2012	0.0001	0	0.0074	0.0006
2013	0.0081	0.0017	0	0.001
2014	0.0389	0.0019	0.0041	0.0215
2015	0	0	0.0079	0.0104
2016	0.0112	0.0297	0.0426	0.0194
2017	0.0365	0.0331	0.0069	0.0219
2018	0.0277	0.0333	0.0051	0.0269
2019	0	0.0035	0	0
2020	0.2071	0.2062	0.1424	0.1235
2021	0.0454	0.0564	0.0387	0.025

Table A.4: Annualized Correlation of Genetic Alphas with Stock Market

	Genetic#5	Genetic#6	Genetic#7	Genetic#8
2005	0.017	0	0	0.0365
2006	0.0288	0.0207	0.0282	0.0258
2007	0.0425	0.0191	0.0246	0.0221
2008	0.033	0	0.0285	0
2009	0.0303	0.021	0	0.0262
2010	0.0514	0.0355	0.0506	0.0331
2011	0.032	0	0	0.0181
2012	0.0007	0.0035	0.0329	0
2013	0.0074	0.0465	0.0346	0.0029
2014	0.0176	0	0.0349	0.0006
2015	0	0.0083	0.029	0
2016	0.0466	0.0216	0.0607	0.0475
2017	0.0284	0.0171	0.0128	0.0189
2018	0.0412	0.0347	0.0686	0.0387
2019	0.004	0.0004	0	0.0067
2020	0.1254	0.1443	0.0958	0.1516
2021	0.0424	0.0481	0.0277	0.0403

Table A.5: Annualized Correlation of 101 Alphas with Stock Market

	Alpha#54	Alpha#101
2005	0.0133	0
2006	0	0
2007	0.0374	0
2008	0.068	0
2009	0	0
2010	0	0
2011	0	0.0134
2012	0	0.0007
2013	0.0217	0
2014	0.0049	0.0067
2015	0	0
2016	0	0
2017	0.036	0
2018	0	0
2019	0	0
2020	0	0
2021	0.0372	0