

# código.py

## GALÁXIA 12

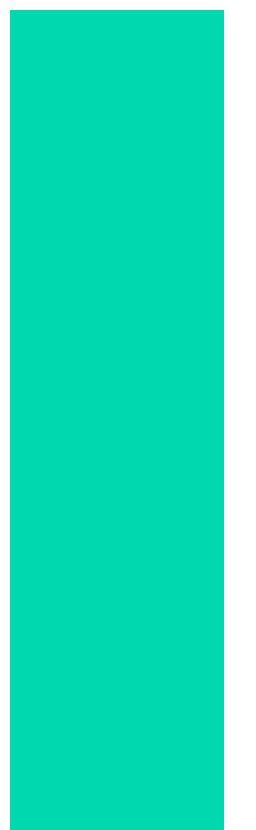
Modelos de  
Análise Técnica



## Introdução

Olá, seja bem-vindo à Galáxia 12 sobre modelos de análise técnica.

Nesta galáxia vamos explorar nossos modelos de investimentos voltados para análise gráfica, com backtest e o resultado de diversas estratégias como MACD, Bollinger Bands, RSI entre outras.



## Mundo 1

### 1.1. – Análise Técnica e Factor Investing

A análise técnica é baseada no estudo e análise do gráfico de preço e volume, focando nessas tendências, buscando identificar padrões e tendências – seja contra ou a favor – para prever movimentos futuros, através de indicadores e modelos, e tomar alguma decisão, seja de compra ou venda. Os principais elementos da análise técnica são:

- Utilização de gráficos, principalmente candlesticks e buscando identificar esses padrões e tendências.
- Utilização de indicadores técnicos como: médias móveis, Bollinger Bands, Moving Average Convergence Divergence (MACD), Hi-Lo, Relative Strength Index (RSI) e etc.
- Identificação de níveis de preços e volumes em que um ativo encontra o suporte e resistência, baseado no histórico.

A grosso modo, a análise técnica é um modelo de momentum. Diferentemente do Factor Investing, que permite a utilização de uma variedade maior de dados na tomada de decisões, sendo mais aberto a interpretações e análises, ou seja, mais amplo e flexível, indo muito além somente de preço e volume, permitindo a criação de fatores a partir de quaisquer dados, desde a idade dos CEO's até análise de redes sociais.

Sempre que estiver criando um modelo de análise técnica, leve em consideração o custo de oportunidade de manter apenas a ação. Compare o desempenho do modelo com o simples investimento "buy and hold" na empresa. Se o "buy and hold" apresentar um retorno maior, reconsidere a utilidade do modelo. Avalie que, se a ação teve um desempenho superior em momentos positivos sem que o modelo tenha capturado esses ganhos, é importante repensar a eficácia do modelo desenvolvido.

## Mundo 2

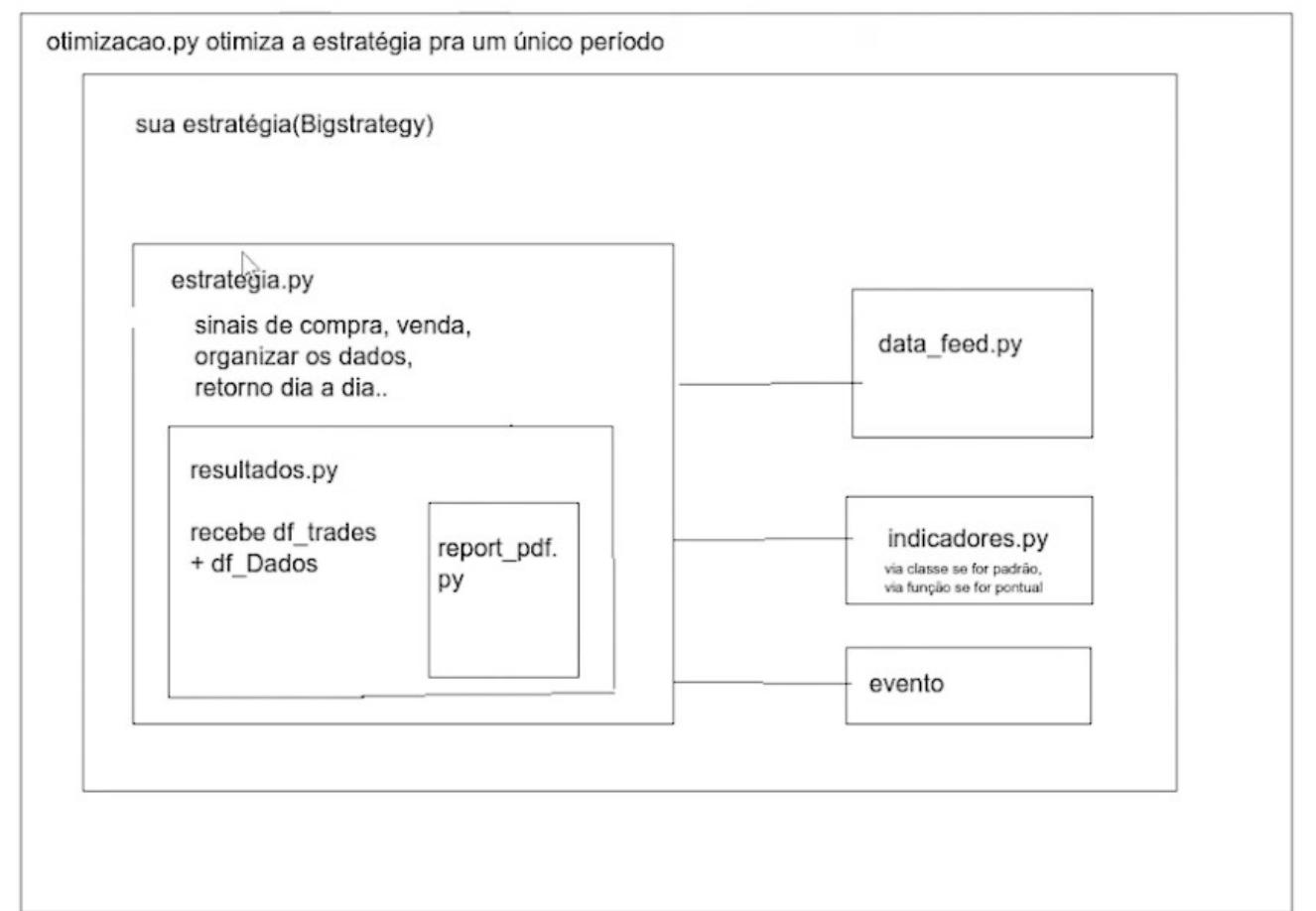
### 2.1. – Funcionamento da biblioteca de análise técnica

Esta seção será voltada para explicação do funcionamento da biblioteca de trading que foi estruturada para que seja possível realizar o back-testing de análise técnica.

É crucial que neste momento do código.py, o conceito de classe esteja em dia, pois a biblioteca utiliza bastante desse conceito. Como bem sabe, classes têm heranças, ou seja, uma classe pode estar contida dentro de outra classe, e assim sucessivamente. Portanto, foi criado um sistema de otimização para que seja possível trabalhar com códigos genéricos.

Dito anteriormente no mundo 1, a análise técnica se resume à preço e volume, logo, conseguimos criar arbitragens que servem para qualquer modelo de investimento, dado que a base dos modelos será sempre a mesma. Através da imagem abaixo, torna-se mais fácil de compreender o funcionamento.

otimizacao\_movel.py, que faz o WalkForwardAnalysis



O funcionamento da estratégia.py é condicionado à entrada de três elementos essenciais: dados, indicadores e eventos. É aqui na estratégia onde são emitidos os sinais de compra e venda, a organização dos dados, o retorno dia a dia e etc. Como os modelos de análise técnica são resumidos a preço e volume, são sempre esses os primeiros passos, assim como os resultados também serão avaliados da mesma forma. Portanto, dentro da classe estratégia.py, temos outra classe chamada de resultados.py, que receberá os dados organizados e os trades diretamente do estratégia.py, e será calculado o resultado do seu modelo e reportado em um pdf.

Data\_feed.py: Inserção de dados e suas características. Esse tratamento dos dados na entrada permite que seja utilizada qualquer base de dados nesta biblioteca. Afinal, está sendo indicado o caminho do parquet e as colunas que serão usadas, além da data inicial e final, e todo o tratamento.

```

46 if __name__ == "__main__":
47
48     acao = "PETR4"
49
50     dados = ReadData(
51
52         caminho_parquet = r'c:\Users\lsiqu\dev\base_dados_br\cotacoes.parquet',
53         tem_multiplas_empresas=True,
54         empresa_escolhida=acao,
55         nome_coluna_empresas = 'ticker',
56
56         data_inicial = "2010-01-01",
57         data_final = "2023-04-30",
58
59         formato_data = ('%Y-%m-%d'),
60
61         coluna_data = 0,
62         abertura = 12,
63         minima = 15,
64         maxima = 13,
65         fechamento = 11,
66         volume = 9
67     )
68

```

# código.py

Indicadores.py: Como o próprio nome diz, trata-se dos indicadores. Será utilizado como classe se for indicador padrão, e como método caso seja um indicador pontual. No exemplo do código abaixo, está sendo utilizado o método 'fazendo\_indicadores' de uma média móvel ao longo do tempo.

Também existe uma classe de indicadores prontos, chamada de '*Make-Indicator*', para apenas devolver um vetor com os indicadores. Em '*Make-Indicator*' é possível criar e acrescentar seu próprio indicador.

```
def fazendo_indicadores(self):
    self.sma_rapida = MakeIndicator().media_movel_simples(self.dados.fechamento, 7)
    self.sma_lenta = MakeIndicator().media_movel_simples(self.dados.fechamento, 40)

    self.lista_indicadores = [self.sma_lenta, self.sma_rapida]
```

```
class MakeIndicator():

    def __init__(self):
        pass

    def media_movel_simples(self, coluna_preco, periodo):
        sma = coluna_preco.rolling(periodo).mean().dropna()
        return sma

    def media_movel_exponencial(self, coluna_preco, periodo):
        ewa = coluna_preco.ewm(span=periodo, min_periods=periodo).mean().dropna()
        return ewa

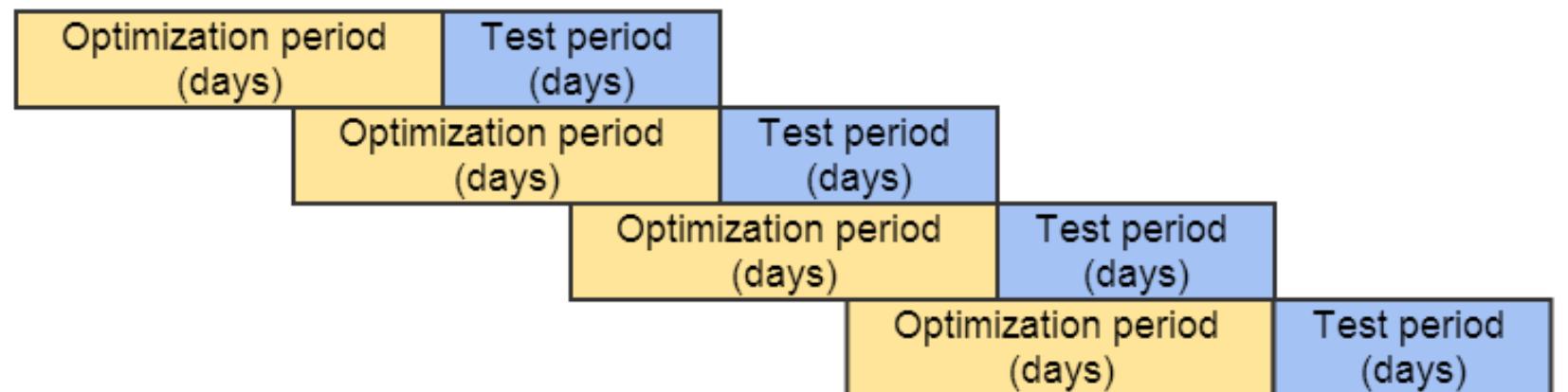
    def RSI(self, coluna_preco, periodo):
        ### Fórmula RSI:
        #100 - 100/(1 + mediaRetornosPositivos / mediaRetornosNegativos)
        retornos = coluna_preco.pct_change().dropna()
        retornos_positivos = retornos.apply(lambda x: x if x > 0 else 0)
        retornos_negativos = retornos.apply(lambda x: abs(x) if x < 0 else 0)
        media_retornos_positivos = retornos_positivos.rolling(window = periodo).mean().dropna()
        media_retornos_negativos = retornos_negativos.rolling(window = periodo).mean().dropna()
        rsi = 100 - 100/(1 + media_retornos_positivos/media_retornos_negativos)
        return rsi
```

# código.py

Evento: Será definido o que acontecerá com o seu modelo de investimento e como será sua atuação. Por exemplo: no código abaixo, temos um modelo de regressão à média, no qual se a média móvel rápida estiver acima da média móvel lenta, será executada a ordem de compra, caso contrário, será executada a ordem de venda.

```
def evento(self, data, i):
    if self.sma_rapida[data] > self.sma_lenta[data]:
        if self.comprado:
            pass
        else:
            self.compra(inverter=True)
    elif self.sma_rapida[data] < self.sma_lenta[data]:
        if self.vendido:
            pass
        else:
            self.venda(inverter = True)
            self.comprar_cdi()
```

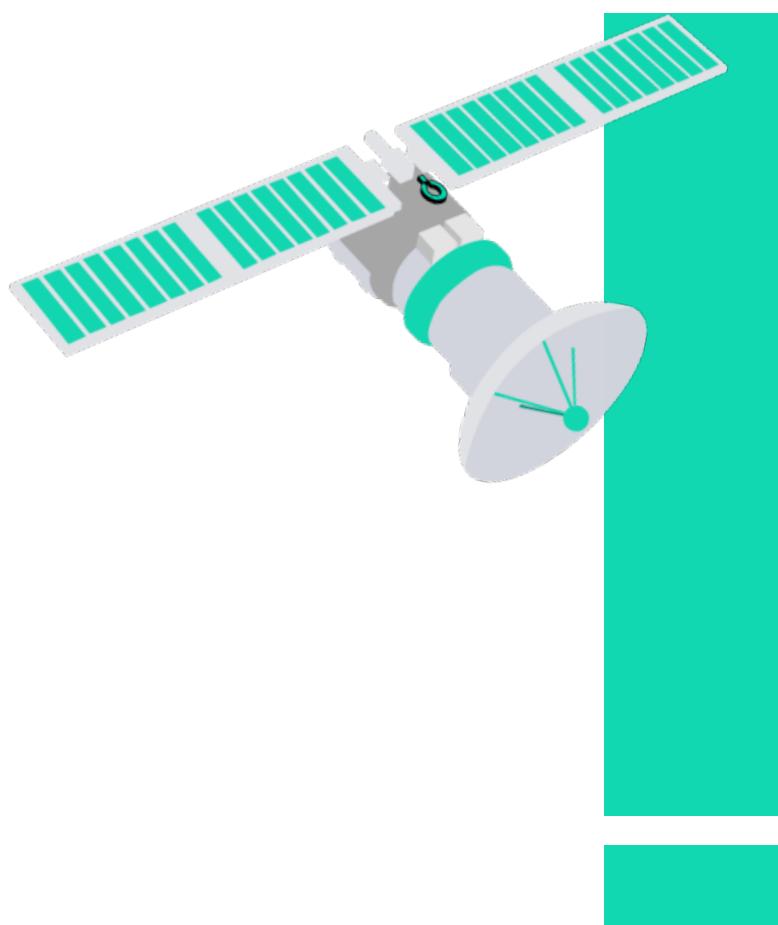
Quanto à otimização.py, o código é basicamente a otimização da estratégia para um único período. Portanto, os dados serão coletados e a estratégia é otimizada em um período para ser testada no período posterior, essa é a intenção do código.



Geralmente são separadas em 80/20, que é separar 80% da amostra para otimização e somente 20% para teste. Mas há um problema de "por qual motivo vamos otimizar a estratégia uma única vez, se é possível otimizar em várias janelas diferentes?". Ou seja, é melhor realizar uma janela móvel de otimização do que uma única vez.

Além disso, se a otimização for ruim, o modelo será jogado fora pois não funcionou. Portanto, é melhor otimizar em janelas móveis. Sendo assim, será testado o melhor período de média móvel em 3 anos, e colocado em prática no período de testes em 1 ano. Em outras palavras, 3 anos de otimização para 1 ano de teste.

1 year		0.5 year	
In-sample	Out-sample	In-sample	Out-sample
2015.06.30	2015.12.31	2016.06.30	2016.12.31
2017.06.30	2017.12.31	2018.06.30	



Resumindo, terá um código com as ações genéricas e inerentes a qualquer modelo, onde adiciona-se o específico: dados, indicadores e eventos. Após isso, a estratégia será otimizada tanto para apenas um período quanto para todos, e colocada em exercício no período de teste. Assim está estruturada a biblioteca de trading para análise técnica.

## Mundo 3

### 3.1. – Funcionamento da classe de dados (data\_feed.py)

Como ensinado no mundo anterior, nesta classe serão inseridos os dados e suas características. Esta classe funciona para qualquer tipo de dados, basta indicar ao programa quais as colunas que precisam ser retornadas, e será feito o tratamento das funções, colunas e filtragem de dados. É por conta disso que neste programa existem vários argumentos que são obrigatórios, a fim de retornar sempre o mesmo padrão.

Será enviado o endereço dos seus arquivos onde esteja o parquet das cotações para o ReadData, mas lembre-se, a forma como será feita a coleta das cotações é através do mesmo código utilizado na galáxia 11, onde é passado o dataframe do parquet dessas cotações.

```
def pegar_cotacoes(self):  
  
    response = requests.get(f'https://api.fintz.com.br/bolsa/b3/avista/cotacoes/historico/arquivos?classe=ACOES&preencher=true',  
                           headers=self.headers)  
  
    link_download = (response.json()['link'])  
  
    urllib.request.urlretrieve(link_download, f"cotacoes.parquet")  
  
    df = pd.read_parquet('cotacoes.parquet')  
  
    colunas_pra_ajustar = ['preco_abertura', 'preco_maximo', 'preco_medio', 'preco_minimo']  
  
    for coluna in colunas_pra_ajustar:  
  
        df[f'{coluna}_ajustado'] = df[coluna] * df['fator_ajuste']  
  
    df['preco_fechamento_ajustado'] = df.groupby('ticker')['preco_fechamento_ajustado'].transform('ffill')  
  
    df = df.sort_values('data', ascending=True)  
  
    df.to_parquet('cotacoes.parquet', index = False)
```

Depois, informe ao programa se existem múltiplas empresas, ou não (linha 53).

- Caso exista, é necessário uma filtragem (linha 25-27) da empresa escolhida (linha 64), e informar qual o nome da coluna que existe essa informação (linha 55).

```
if tem_multiplas_empresas:  
    self.dados = self.dados[self.dados[nome_coluna_empresas] == empresa_escolhida]
```

- Caso exista somente uma empresa, basta informar em "tem\_multiplas\_empresas = False", e apagar a empresa\_escolhida + nome\_coluna\_empresa. Quanto ao apagar, é opcional, somente informar como False, já é o suficiente.
- Continuando no programa, deve ser informado o período de data para filtragem, bem como o formato que essa data está sendo enviada.
- E por fim, diga as colunas que serão utilizadas no data frame, mas informe com o número. Por exemplo, como a coluna\_data é 0, pois ela é o próprio index. Isso será útil na organização do programa.

```
if __name__ == "__main__":  
  
    dados_petr = ReadData(  
        caminho_parquet = r'c:\Users\marce\OneDrive\Documentos\Varos\base_dados_br\cotacoes.parquet',  
        tem_multiplas_empresas=False,  
        empresa_escolhida='PETRA',  
        nome_coluna_empresas = 'ticker',  
        data_inicial = "2023-01-01",  
        data_final = "2023-04-30",  
        formato_data = ('%Y-%m-%d'),  
        coluna_data = 0,  
        abertura = 12,  
        minima = 15,  
        maxima = 13,  
        fechamento = 11,  
        volume = 9  
)
```

Após ter enumerado as colunas, vamos partir para a parte principal desta classe.

A partir daí, as colunas serão organizadas, pois será passada todas as colunas em 'organizando\_colunas', envie uma lista com as colunas que deseja que esteja presente no data frame, dê um "iloc" para buscar essas colunas com base na posição numérica. E por fim, será organizado e renomeado numa ordem padrão em 'self.dados.columns'.

```
def organizando_colunas(self):  
    colunas_escolhidas = [self.coluna_data, self.abertura, self.minima, self.maxima, self.fechamento, self.volume]  
    self.dados = self.dados.iloc[:, colunas_escolhidas]  
    # mudando pra ordem padrão  
    self.dados.columns = ['data', 'abertura', 'minima', 'maxima', 'fechamento', 'volume']
```

Existem duas opções ao realizar backtesting de análise técnica: utilizar os preços originais ou os preços ajustados.

Prós da utilização do preço original: Será calculado o indicador verdadeiro, através da informação real daquele período, ou qual era a média móvel de fato naquele dia, e calcular os preços reais.

Contra da utilização do preço original: Há problemas com desdobramento (split) e agrupamento (inplit) de ações, pois isso gera falsos sinais de compra e venda, e não é possível calcular a rentabilidade do modelo, afinal, muitas empresas desdobram durante sua trajetória.

Exemplificando, pense no cálculo de uma média móvel de uma empresa, que possui as seguintes cotações na semana de: 10, 11, 9, 12, 11. Após essa semana, a empresa realiza um desdobramento de ações e as cotações saltam para: 100, 101, 99, 102, 101. Sua média móvel também haverá um salto e irá gerar sinais falsos de compra e venda. Nesse caso, o ideal é utilizar os preços ajustados.

Sempre será gerado um arquivo parquet com um formato consistente, onde a data é o índice e as colunas selecionadas são organizadas na mesma ordem. Esses dados são totalmente tratados e organizados de maneira padronizada. A partir desse ponto, o restante do processo se torna mais fácil de ser padronizado.



## Mundo 4

### 4.1. – Rodando a estratégia (Bigstrategy)

É a hora de finalmente entender os códigos por trás da nossa estratégia. Nesse exemplo, será usada a estratégia de média móvel.

Para a construção de um modelo de média móvel, temos que calcular uma média rápida de um período de tempo, e uma média lenta num período maior.

Será testado a média móvel 7 com 40 (períodos), porém somente a estratégia, como no exemplo abaixo.

Não será feita a otimização dos dados neste mundo, há um capítulo destinado para isso.

sua estratégia(Bigstrategy)

estrategia.py

sinais de compra, venda,  
organizar os dados,  
retorno dia a dia..

resultados.py

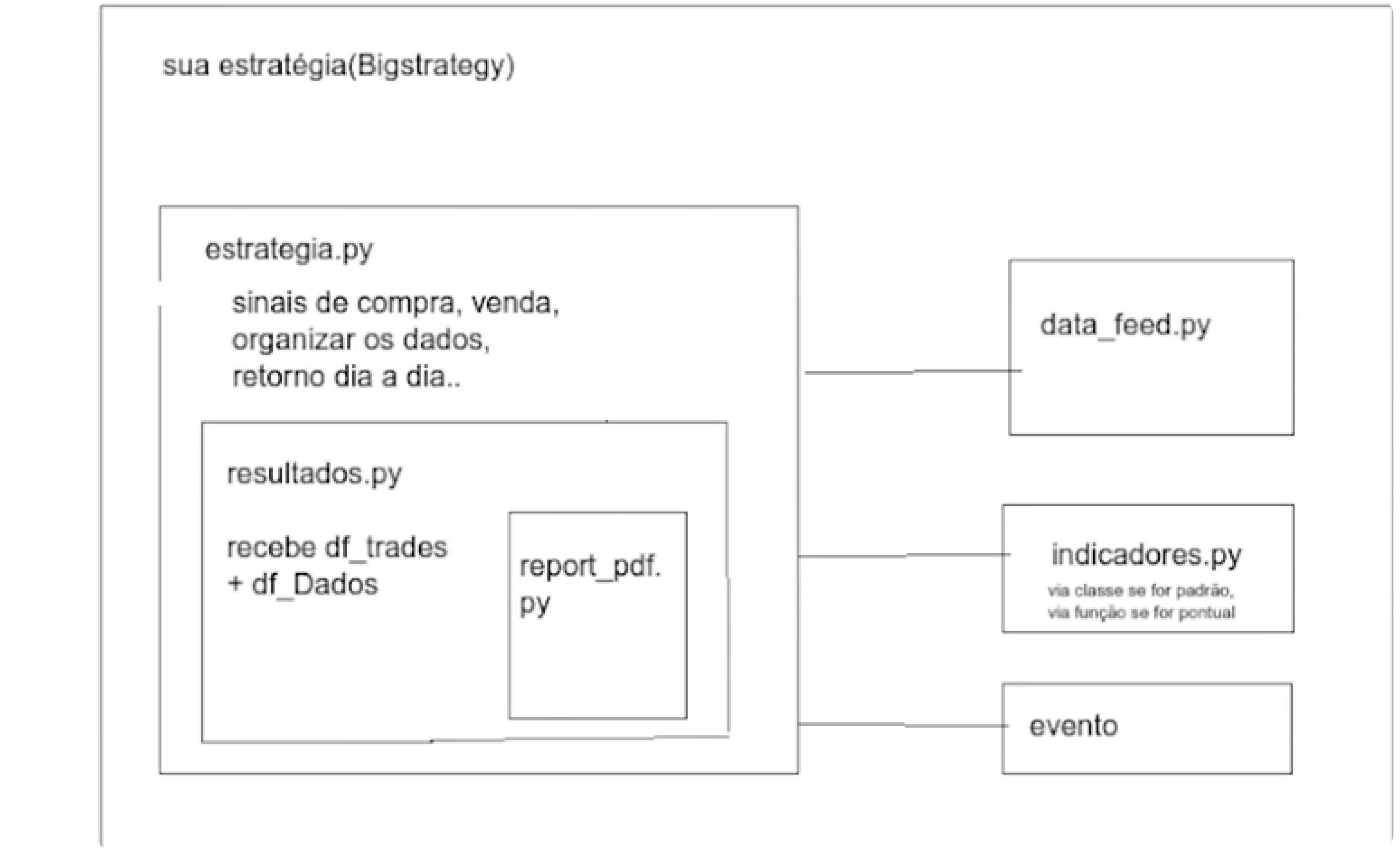
recebe df\_trades  
+ df\_Dados

report\_pdf.  
py

data\_feed.py

indicadores.py  
via classe se for padrão,  
via função se for pontual

evento



## 4.2. – Usando o Quick Start

Inicia-se a explicação do programa através do arquivo 'quickstart\_mm.py' e, em seguida, o 'estrategia.py'.

Temos a classe principal, "MM\_estrategia", que está herdando a classe BigStrategy. Como todo código de programação orientado à objeto, é necessário realizar o método init para inicialização.

```
class MM_estrategia(BigStrategy):
    def __init__(self):
        super().__init__()
```

### 4.2.1. – Indicadores e Evento

Nesses métodos é onde o programa começa de verdade. Essa é a única parte do programa que você irá de fato codar, visto que todos os comportamentos padrões já foram capturados pela classe mãe "BigStrategy".

Para a estratégia de média móvel, é necessário a criação de uma média rápida e uma lenta. Portanto, vamos criar esses indicadores e incorporar ao método 'fazendo\_indicadores'. Depois, criamos uma lista de indicadores, que deve existir para organizar as datas no estrategia.py.

Em evento, uma vez que a média móvel rápida esteja maior que a média móvel lenta, isso indica que o ativo está em tendência de alta, sendo recomendada a sua compra. Se a posição já estiver comprada, irá pular esta parte do código. Se não, emitimos um sinal de compra.

Do contrário, quando a média móvel rápida estiver menor que a média móvel lenta, indica uma tendência de baixa, sendo recomendada a venda do ativo. Se a posição estiver comprada, zeramos a compra. Essa é a lógica sem operar vendido, somente comprado.

Operar vendido funciona da mesma forma, modificando somente a lógica final. Ou seja, se a média móvel rápida for menor que a média móvel lenta, e estiver vendido, o código será pulado. Se não estiver vendido, vamos zerar o sinal de compra e vender.

E, se o CDI estiver ativado em seu programa quando estiver operando vendido na ação, será emitido um sinal de venda da ação e a compra do CDI.

```
def fazendo_indicadores(self):

    self.sma_rapida = MakeIndicator().media_movel_simples(self.dados.fechamento, 7)
    self.sma_lenta = MakeIndicator().media_movel_simples(self.dados.fechamento, 40)

    self.lista_indicadores = [self.sma_lenta, self.sma_rapida]

def evento(self, data, i):
    if self.sma_rapida[data] > self.sma_lenta[data]:
        if self.comprado:
            pass
        else:
            self.compra(inverter=True)

    elif self.sma_rapida[data] < self.sma_lenta[data]:
        if self.vendido:
            pass
        else:
            self.venda(inverter = True)
```

Ao gerar os sinais de compra ou venda de uma ação, saiba que o retorno do sistema é o retorno da ação ou carteira.

Então, se um sinal de compra for gerado e a posição estiver comprada, o retorno do seu modelo será o retorno da ação em que você está comprado. Por exemplo, se você comprou ações da Petrobras, o retorno do seu modelo será o retorno da Petrobras.

No caso de operar vendido, o processo é invertido: se a ação sobe 2%, você tem uma perda de 2%. Ou então, se você zerar a posição, o retorno é zerado.

Caso prefira um modelo que não opere vendido, faça mudanças no método `evento`. Em `self.compra`, remova a opção `inverter = True`, e em `self.venda`, modifique para `zerar = True`. Isso permitirá a operação somente de compra no modelo, desabilitando a possibilidade de venda a descoberto.

```
def evento(self, data, i):  
  
    if self.sma_rapida[data] > self.sma_lenta[data]:  
        if self.comprado:  
            pass  
        else:  
            self.compra()  
  
    elif self.sma_rapida[data] < self.sma_lenta[data]:  
        if self.vendido:  
            pass  
  
        else:  
            self.venda(zerar = True)
```

#### 4.2.2. – Gerando o PDF

Primeiramente, adicione a estratégia `MM\_estrategia()` , que consiste na média móvel. Esta estratégia será implementada como uma classe.

Posteriormente, inclua a taxa de corretagem, pois isso afeta a sua rentabilidade no longo prazo.

Além disso, é importante definir o caminho dos dados e das imagens a serem utilizados para gerar o PDF, em `add\_caminhos()` .

Também é necessário fornecer detalhes sobre os dados e suas características em `add\_data(dados)` . E finalizando, o CDI é adicionado através do `add\_cdi()` .

Por fim, a estratégia será executada pelo `run\_strategy()` , o que resultará na geração de um PDF com os resultados através do `make\_report()` .

```
start_estrategia = MM_estrategia()
start_estrategia.corretagem = 0.0005
start_estrategia.add_caminhos(caminho_dados=r'c:\Users\lsiqu\dev\base_dados_br',
| | | | | | | | | | caminho_imagens=r'c:\Users\lsiqu\dev\PDFS_BACKTEST\imagens')
start_estrategia.add_data(dados)
start_estrategia.add_cdi()
start_estrategia.run_strategy()
start_estrategia.make_report(nome_arquivo = r'c:\Users\lsiqu\dev\PDFS_BACKTEST\
| analise_tecnica\mm_teste.pdf')
print(start_estrategia.df_trades)
```

Lembre-se, todo o código em Quick Start está sendo alimentado pelo Estratégia.py. Como dito anteriormente, a classe MM\_Estrategia está herdando a classe BigStrategy, o qual será explicado.

## 4.3. – Funcionamento da estratégia

Vamos finalmente adentrar no código principal deste programa, que é o código da estratégia. Primeiro, deve ser feita a importação das bibliotecas

```
import pandas as pd
from resultados import MakeReportResult
import warnings
from pandas.errors import SettingWithCopyWarning
warnings.simplefilter(action="ignore", category=SettingWithCopyWarning)
import numpy as np
import os
```

### 4.3.1. – Init e variáveis

Como todo código de programação orientado à objeto, é necessário realizar o método init para inicialização de todas as variáveis.

A variável `self.numeros\_do\_trade` é utilizada para monitorar a quantidade de trades realizados durante o backtesting. Ao contrário do factor investing, que pode envolver estratégias executadas em intervalos regulares, como a cada 21 dias.

O programa funciona através de estados, como comprado ou vendido. Logo, `self.comprado/self.vendido/self.zerou\_venda/self.zerou\_compra`, são os estados em que o programa pode operar.

Cada sinal é acionado quando um indicador cruza uma média ou atinge um determinado valor, resultando na abertura de uma posição de trade. Essa variável é dinâmica e registra a quantidade de operações realizadas com base nos sinais gerados pelo sistema de trading.

Todos os parâmetros que possuem `None`, significa que serão adicionados dados posteriormente, como por exemplo em `self.dados` e `dados\_cdi`.

```
class BigStrategy():

    def __init__(self):
        self.novo_trade = False
        self.df_trades = pd.DataFrame(columns = ['data', 'retorno', 'numero_trade', 'sinal'])
        self.numero_do_trade = 0
        self.comprado = False
        self.vendido = False
        self.lista_indicadores = []
        self.dados = None
        self.zerou_venda = False
        self.zerou_compra = False
        self.barra_executada = 99999999999999999999999999999999
        self.cdi = False
        self.fechamento = None
        self.dados_cdi = None
        self.escolher_data = False
        self.data_inicial = None
        self.data_final = None
        self.parametro1 = None
        self.parametro2 = None
        self.corretagem = 0
        self.caminho_imagens = None
        self.caminho_benchmarks = None
```

Esse é um print de um recorte do DataFrame, onde é selecionado apenas as quatro informações necessárias para calcular todos os resultados.

0	2010-03-03	0.000000	NaN	venda	0.000000	1.000000		NaN	1.000000
1	2010-03-04	-0.002281	1.0	compra	-0.002281	0.997719	-0.002281	0.997719	
2	2010-03-05	0.016389	2.0	NaN	0.014071	1.016389	0.016389	1.014071	
3	2010-03-08	-0.002622	2.0	NaN	0.011411	0.997378	0.013724	1.011411	
4	2010-03-09	0.021911	2.0	NaN	0.033572	1.021911	0.035935	1.033572	
...	...	...	...	...	...	...	...	...	...
3255	2023-04-24	0.018784	110.0	NaN	0.867443	1.018784	0.044437	1.867443	
3256	2023-04-25	-0.003951	110.0	NaN	0.860065	0.996049	0.040311	1.860065	

### 4.3.2. – Adicionando data, CDI e os caminhos

O método `add\_data` serve para importar os dados, como bem ensinado no mundo anterior.

O método de adicionar o CDI com o `add\_cdi`, é o que torna possível operar comprado e vendido. E funciona da seguinte maneira: Quando se vende um ativo, como ações da Petrobras, é possível utilizar os recursos obtidos dessa venda para realizar investimentos em outras áreas, como por exemplo, investir no CDI ou em outras modalidades de renda fixa. Obtendo lucro com a queda do preço do ativo.

Mas, este método é opcional, podendo ser adicionado o CDI ou não, ou até mesmo inserir outra estratégia de seu interesse.

E por último, o método de adicionar caminho das imagens utilizadas no pdf e do benchmark através do `add\_caminhos`, funcionando na mesma mecânica da galáxia de Factor Investing.

```
def add_data(self, class_data):
    self.dados = class_data.dados

def add_cdi(self):
    if self.caminho_benchmarks == None:
        self.dados_cdi = pd.read_parquet('./cdi.parquet')
    else:
        self.dados_cdi = pd.read_parquet(f'{self.caminho_benchmarks}/cdi.parquet')

    self.dados_cdi['data'] = pd.to_datetime(self.dados_cdi['data'])
    self.dados_cdi = self.dados_cdi.set_index('data')

def add_caminhos(self, caminho_imagens, caminho_dados):
    self.caminho_imagens = caminho_imagens
    self.caminho_benchmarks = caminho_dados
    os.chdir(caminho_imagens)
```

### **4.3.3. – Rodando a estratégia**

Hora de rodar a estratégia. Inicialmente, será feita a técnica de reset, que é uma condição estabelecida para reiniciar o estado do sistema durante a otimização de uma estratégia. Essa condição é acionada ao final da execução da estratégia, independente se o sistema terminar comprado, vendido ou zerar as vendas, por exemplo.

Após o reset do data frame, a estratégia é executada novamente. A partir daí, rodamos o restante dos métodos que são: fazer o indicador, organizar as datas, calcular retorno e fazer a iteração.

```
def run_strategy(self, reset = True):  
  
    if reset:  
  
        self.numero_do_trade = 0  
        self.comprado = False  
        self.vendido = False  
        self.cdi = False  
        self.zerou_venda = False  
        self.zerou_compra = False  
        self.barra_executada = 9999999999999999999999999999999  
  
    else:  
  
        pass  
  
    self.df_trades = pd.DataFrame(columns = ['data', 'retorno', 'numero_trade', 'sinal'])  
  
    self.fazendo_indicadores()  
    self._organizando_datas()  
    self._calculando_retorno()  
    self._iteracao()
```

#### 4.3.4. – Indicadores e evento

Conforme visto no quick start, o indicador e o evento é definido posteriormente, porque ambos são elementos altamente flexíveis e personalizáveis dentro de um sistema de trading. Esses componentes têm a capacidade de serem adaptados e ajustados de acordo com as necessidades específicas do operador ou da estratégia de investimento.

```
def fazendo_indicadores(self):  
    pass
```

```
def evento(self, data, i = None):  
    pass
```

#### 4.3.5 – Organizando as datas

No método de organizar as datas, funciona dessa maneira. A filtragem ocorrerá para datas que sejam maiores ou iguais à data inicial escolhida e menores ou iguais à data final escolhida, garantindo que os dados selecionados estejam dentro do período definido.

Em seguida, aplicamos um filtro para os indicadores. Para cada indicador configurado no quick start, faremos a filtragem dos dados de acordo com a data escolhida.

Se a data não for especificada, será selecionada a data máxima entre as datas mínimas dos indicadores configurados. Exemplo: Se, indicador n° 1 inicia em 27/11, indicador n°2 inicia em 10/12, e indicador n°3 inicia em 23/12. É utilizado a data 23/12, caso contrário, ainda não existirá o indicador n°3, pois ele surgiu em 23/12.

Exemplificando diretamente no código: A média móvel rápida (7 dias) estará pronta antes da média móvel lenta (40 dias). Portanto, é necessário escolher a data de início do programa com base na média móvel lenta (40 dias). Caso inicie o programa pela média móvel rápida (7 dias), a média móvel lenta ainda não estará disponível.

```
def _organizando_datas(self):

    if self.escolher_data:
        self.dados_filtrados_data = self.dados[(self.dados.index >= self.data_inicial) &
                                                (self.dados.index <= self.data_final)]

        indicadores_filtrados = []
        for indicador in self.lista_indicadores:
            indicador = indicador[(indicador.index >= self.data_inicial) &
                                   (indicador.index <= self.data_final)]
            indicadores_filtrados.append(indicador)

        self.lista_indicadores = indicadores_filtrados

    else:
        datas_maximas = [create_indicador.index[0] for create_indicador in self.lista_indicadores]
        data_mais_recente = max(datas_maximas) #o maximo porque precisamos do mais recente
        self.dados_filtrados_data = self.dados[self.dados.index >= data_mais_recente]
        indicadores_filtrados = []
        for indicador in self.lista_indicadores:
            indicador = indicador[indicador.index >= data_mais_recente]
            indicadores_filtrados.append(indicador)

        self.lista_indicadores = indicadores_filtrados
```

### 4.3.6 – Calculando o retorno

No método para calcular o retorno, utilize os dados filtrados, obtidos através do `organizando\_datas`, conforme o período selecionado. Em seguida, calcule o retorno da coluna "fechamento" com base nesses dados filtrados.

Agora, com os dados e os retornos disponíveis, prossiga com a iteração e conduza o backtest.

```
def _calculando_retorno(self):  
  
    self.dados_filtrados_data.loc[:, 'retorno'] = self.dados_filtrados_data.fechamento.pct_change().copy()  
  
    if self.fechamento != None:  
        retorno_primeiro_dia = self.dados_filtrados_data.fechamento[0]/self.fechamento - 1  
        self.dados_filtrados_data.retorno[0] = retorno_primeiro_dia
```

#### 4.3.7 – Iteração

O método de iteração acompanha um fluxo de lógica complexo, que gerencia diferentes condições e estados do sistema para tomada de decisões. Essas decisões podem ser de compra, venda ou manutenção de posição com base nas datas e nos sinais gerados pelo sistema de trading.

O processo envolve a construção de listas com as 4 informações essenciais extraídas do DataFrame. O retorno inicial é zero ( $i = 0$ ).

Para cada linha em uma possível data, realizaremos uma iteração no DataFrame (`self.novo\_trade`), verificando todos os estados do sistema. Se estiver iniciando um novo trade, o número do trade será incrementado em 1.

```
def _iteracao(self):  
    datas = []  
    retornos = []  
    numero_trades = []  
    sinais = []  
    i = 0  
  
    for data, linhas in self.dados_filtrados_data.iterrows():  
        if self.novo_trade:  
            self.numero_do_trade = self.numero_do_trade + 1
```

Se um sinal de compra de um ativo for gerado no dia 20, a compra do ativo ocorrerá no dia 21, e a rentabilidade só será contabilizada no dia seguinte à compra. Portanto, o evento de compra é registrado apenas no fechamento do mercado (`self.evento(data=data, i=i)`), indicando que a compra foi feita no dia anterior e no dia atual haverá um valor NaN.

Em seguida, será ajustado o retorno, subtraindo a corretagem.

```
if self.comprado: #contabilizar a rent do dia

    datas.append(data.date())
    numero_trades.append(self.numero_do_trade)

    if self.novo_trade:

        if sinais != []:
            sinais[-1] = 'compra'
            sinais.append(np.nan)

        else:
            sinais.append('compra')

    retornos.append(linhas['retorno'] - self.corretagem)

else:
    retornos.append(linhas['retorno'])
    sinais.append(np.nan)
```

Depois, será verificado se está em posição vendida (`self.vendido`), seguindo a lógica similar à posição comprada, de forma invertida.

```
elif self.vendido:

    datas.append(data.date())
    numero_trades.append(self.numero_do_trade)

    if self.novo_trade:

        if sinais != []:
            sinais[-1] = 'venda'
            sinais.append(np.nan)

        else:
            sinais.append('venda')

    retornos.append((linhas['retorno'] * -1) - self.corretagem)

else:
    sinais.append(np.nan)
    retornos.append(linhas['retorno'] * -1)
```

Após, será verificado se foi zerada a venda (`self.zerou\_venda`). Nesse caso, o retorno será zero no dia seguinte, ou seja, um retorno zerado.

```
elif self.zerou_venda:  
  
    datas.append(data.date())  
  
    try:  
  
        retorno_ontem = retornos[-1]  
  
        retornos[-1] = (retorno_ontem - self.corretagem)  
  
        retornos.append(0)  
  
        sinais[-1] = 'compra'  
  
        sinais.append(np.nan)  
  
    except IndexError:  
  
        retornos.append(- self.corretagem)  
        sinais.append('compra')  
  
    numero_trades.append(np.nan)  
  
    self.zerou_venda = False
```

A lógica para zerar a compra (`self.zerou\_compra`) é idêntica à posição de zerar a venda, mas de forma invertida.

```
elif self.zerou_compra:  
  
    datas.append(data.date())  
  
    try:  
  
        retorno_ontem = retornos[-1]  
  
        retornos[-1] = (retorno_ontem - self.corretagem)  
  
        retornos.append(0)  
  
        sinais[-1] = 'venda'  
  
        sinais.append(np.nan)  
  
    except IndexError:  
  
        retornos.append(- self.corretagem)  
        sinais.append('venda')  
  
        numero_trades.append(np.nan)  
        self.zerou_compra = False
```

E por fim, se estiver comprado no CDI (`self.cdi`), será adicionado o retorno do CDI na data correspondente.

```
else:  
    datas.append(data.date())  
    retornos.append(0)  
    numero_trades.append(np.nan)  
    sinais.append(np.nan)  
  
if self.cdi:  
    try:  
        retornos[-1] = retornos[-1] + self.dados_cdi.retorno[data]  
    except KeyError:  
        pass  
    self.novo_trade = False  
  
    self.evento(data = data, i = i)  
  
    i = i + 1
```

#### 4.3.8 – Sinais de compra e venda

O método `comprar` opera da seguinte maneira:

1. Inverter: Se estava em posição vendida e entrará numa posição comprada, então `vendido = False`, indicando que a posição de venda foi zerada. Nesse caso, um novo trade é gerado ao realizar a compra.
2. Zerar: Quando não está mais em posição vendida e começa a comprar apenas para zerar a venda que havia sido feita anteriormente.
3. Vou Comprar: Caso não esteja em nenhum estado, será feito apenas o comando de compra, marcando assim `self.comprado = True` e `self.novo\_trade = True`, o que significa entrar na posição de compra e iniciar um novo trade.

```
def compra(self, inverter = False, zerar = False):

    if inverter:
        self.vendido = False
        self.novo_trade = True
        self.comprado = True

    elif zerar:
        self.vendido = False
        self.zerou_venda = True

    else: #vou comprar
        self.comprado = True
        self.novo_trade = True
```

O método `venda` funciona com uma lógica similar ao método compra, mas invertido.

```
def venda(self, inverter = False, zerar = False):

    if inverter:
        self.comprado = False
        self.novo_trade = True
        self.vendido = True

    elif zerar:
        self.comprado = False
        self.zerou_compra = True

    else:
        self.vendido = True
        self.novo_trade = True
```

#### 4.3.9. – Compra e venda do CDI

Caso seu programa rode com a possibilidade de operar com o CDI, é possível aplicar a sua venda e compra. A inclusão do CDI pode ser realizada no seu backtesting. Quando estiver em posição vendida, basta ajustar e adicionar no quickstart. Em evento, para a condição "else", inclua `self.comprar\_cdi()` para realizar a compra do CDI.

```
def comprar_cdi(self):
    self.cdi = True

def vender_cdi(self):
    self.cdi = False
```

```
def evento(self, data, i):
    if self.sma_rapida[data] > self.sma_lenta[data]:
        if self.comprado:
            pass
        else:
            self.compra(inverter=True)

    elif self.sma_rapida[data] < self.sma_lenta[data]:
        if self.vendido:
            pass
        else:
            self.venda(inverter = True)
            self.compra_cdi()
```

#### 4.3.10. – Gerando o pdf

Crie o `df\_trades` , que contém informações como data, retorno, número do trade e o sinal.

As funções `make\_report` e `resultados` são idênticas às do código usado na Galáxia 11 de factor investing, sem alterações. O código foi replicado e retornará o `df\_trades` .

```
    self.df_trades['data'] = datas
    self.df_trades['retorno'] = retornos
    self.df_trades['numero_trade'] = numeros_trades
    self.df_trades['sinal'] = sinais
```

```
def make_report(self, nome_arquivo = 'backtest.pdf'):

    MakeReportResult(df_trades=self.df_trades, df_dados =
        self.dados_filtrados_data, caminho_imagens=self.caminho_imagens,
        caminho_benchmarks=self.caminho_benchmarks, nome_arquivo=nome_arquivo)
```

## Mundo 5

### 5.1. – Otimização do programa

Neste cenário, faremos otimizações utilizando o código para aprimorar os modelos e realizar backtesting da maneira mais eficaz possível.

A otimização é fundamental para a adaptação ao longo do tempo, pois o mercado é dinâmico e está em constante mudança. As volatilidades, condições macroeconômicas e diversos fatores evoluem ao longo do tempo, exigindo também a evolução e adaptação dos modelos de negociação. Por isso, estaremos constantemente adaptando nosso modelo para se adequar às novas condições do mercado.

Ao otimizar parâmetros para seu próprio benefício, limite-se a no máximo dois parâmetros. A tentativa de otimizar uma grande quantidade, como 10 ou 15 parâmetros, pode levar ao overfitting, prejudicando a qualidade dos resultados.

Evite intervalos muito pequenos entre os parâmetros durante a otimização. Ao definir os intervalos, por exemplo, use um step de 5 para evitar problemas de sobreajuste.

Um erro comum é otimizar o período inteiro, o que pode levar à incorporação de contextos diferentes do mercado, sem uma regra clara de otimização. Cada estágio da otimização deve representar um novo estado do mercado. Assim, é recomendado seguir uma janela móvel que fará a escolha da janela ideal, que varia o suficiente para refletir mudanças no mercado. Não divida em muitos períodos, pois o padrão é um ratio de 3:1 (três para um).

Entretanto, quanto maior o período que você possui para análise, maior é a flexibilidade para permitir um período de otimização de 4 para 1 ou até 5 para 1, ajustando-se à dinâmica do mercado ao longo do tempo.

## 5.2. – Otimização da estratégia para um período

Na montagem da otimização, é necessário escolher o `parâmetro 1` e `parâmetro 2`, assim como `anos\_otimizacao` e `anos\_teste`.

```
walk = WalkForwardAnalysis(estrategia = MM_estrategia(), class_dados = dados,
    parametro1= range(7, 29, 7), parametro2= range(30, 46, 5), anos_otimizacao=2, anos_teste=1,
    nome_arquivo = rf"c:\Users\lsiqu\dev\PDFS_BACKTEST\analise_tecnica\backtest_2pra1_{acao}_MM.pdf",
    caminho_dados_benchmarks =r'c:\Users\lsiqu\dev\base_dados_br',
    caminho_imagens= r'c:\Users\lsiqu\dev\PDFS_BACKTEST\imagens')
```

O limite máximo de tempo estabelecido para a otimização é de 4 para 1(em anos). Se ultrapassar esse limite, a otimização será anulada. Isso se deve ao fato de que um período de tempo muito extenso abrange diferentes condições de mercado, tornando a adaptação do modelo menos eficaz. Portanto, o objetivo é evitar a otimização em intervalos de tempo muito longos para garantir uma melhor adaptação do modelo às condições atuais do mercado.

Na construção do método `fazendo\_indicadores`, otimize definindo os parâmetros de otimização - `parametro 1` e `parametro 2`. Estes parâmetros correspondem à média móvel rápida e à média móvel lenta, respectivamente.

Ao otimizar, evitaremos utilizar um incremento de 1 para 1 nos parâmetros, pois a diferença entre modelos com período 7 e 40 (range(7, 40)) em comparação com 8 e 40 (range(8, 40)) é marginal e muito pequena, por exemplo

Por conta disso, será feita a otimização em um intervalo que salte de maneira mais ampla, para reduzir a quantidade de iterações. Nesse código, rodará 16 possibilidades para `parametro1` e `parametro2`, totalizando 16 modelos, considerando 4 valores possíveis para cada parâmetro. Isso será executado ao longo de intervalos de 2 em 2 anos.

Ao evitar o `step=1` nos parâmetros de otimização, a quantidade de modelos a serem testados é reduzida. E assim, otimizamos o tempo necessário para executar o código, sem comprometer significativamente a busca por configurações ideais.

```
def fazendo_indicadores(self):
    self.sma_rapida = MakeIndicator().media_movel_simples(self.dados.fechamento, self.parametro1)
    self.sma_lenta = MakeIndicator().media_movel_simples(self.dados.fechamento, self.parametro2)

    self.lista_indicadores = [self.sma_lenta, self.sma_rapida]
```

### 5.3. – Init

Dê o init para iniciar os parâmetros

```
class Optimize():

    def __init__(self, estrategia, class_dados, parametro1 = None, parametro2 = ()):
        self.parametro1 = parametro1
        self.parametro2 = parametro2

        self.estrategia = estrategia

        estrategia.dados = class_dados.dados
```

### 5.4. – Rodando a otimização

É a hora de rodar o método `run\_optimize`.

Neste método, calcule o retorno acumulado para cada backtesting realizado, considerando cada combinação de parâmetros.

Na estrutura condicional `if self.parametro2 != ()` é quando for utilizado 2 parâmetros, e será feita todas as combinações.

Já em `else:`, havendo apenas um parâmetro, a lógica será semelhante, porém com apenas um loop. Em qualquer caso, o objetivo é calcular o retorno acumulado para cada configuração de parâmetro testada durante o processo de backtesting.

```
def run_optimize(self):
    self.df_retorno_acum = pd.DataFrame(columns=['parametro1', 'parametro2', 'retorno'])

    if self.parametro2 != []:
        z = 0
        for i in self.parametro1:
            for j in self.parametro2:
                self.estategia.parametro1 = i
                self.estategia.parametro2 = j

                self.estategia.run_strategy()

                retorno = ((self.estategia.df_trades['retorno'] + 1).cumprod() - 1).iat[-1]

                self.df_retorno_acum.loc[z, 'parametro1'] = i
                self.df_retorno_acum.loc[z, 'parametro2'] = j
                self.df_retorno_acum.loc[z, 'retorno'] = retorno

                z = z + 1
    else:
        z = 0
        for i in self.parametro1:
            self.estategia.parametro1 = i

            self.estategia.run_strategy()

            retorno = ((self.estategia.df_trades['retorno'] + 1).cumprod() - 1).iat[-1]

            self.df_retorno_acum.loc[z, 'parametro1'] = i
            self.df_retorno_acum.loc[z, 'parametro2'] = None
            self.df_retorno_acum.loc[z, 'retorno'] = retorno

            z = z + 1
```

O `otimizacao\_mm.py` e o `otimizacao.py` rodam juntos.

## 5.5. – Walk Forward Analysis (WFA)

### 5.5.1. – Init

Na função `init` será elaborada a otimização móvel. Embora o código possa parecer extenso inicialmente, grande parte dele será repetido, pois esta etapa é a última do processo. Assim, o código engloba muitos dos elementos presentes no quick start, já que é a execução completa do processo.

```
class WalkForwardAnalysis():
    def __init__(self, estategia, class_dados, anos_otimizacao, anos_teste, nome_arquivo, parametro1, parametro2 = (), corretagem = 0,
                 caminho_imagens = None, caminho_dados_benchmarks = None):
        self.otimizacao = Optimiza(estategia, parametro1=parametro1, parametro2=parametro2, class_dados=class_dados)
        self.parametro1 = parametro1
        self.parametro2 = parametro2
        self.dados = class_dados.dados
        self.anos_otimizacao = anos_otimizacao
        self.anos_teste = anos_teste
        self.nome_arquivo = nome_arquivo
        self.otimizacao.estategia.corretagem = corretagem

        self.otimizacao.estategia.caminho_imagens = caminho_imagens
        self.otimizacao.estategia.caminho_dados_benchmarks = caminho_dados_benchmarks

        os.chdir(caminho_imagens)

        self.otimizacao.estategia.add_cdi()

        self.buscando_data_inicial()
```

### 5.5.2 – Escolhendo a data de início

No método `buscando\_data\_inicial`, a data inicial da primeira otimização corresponde ao filtro de data do indicador, explicado em Factor Investing. Esse filtro de data é aplicado para filtrar os dados conforme o parâmetro e alcançar a data mais recente.

Este código é mais extenso porque está sendo realizado a otimização de parâmetros e precisamos filtrar os dados com base nessa otimização. O backtesting deve começar em uma data específica.

Dentro do range de parâmetros, será buscado a data mínima associada ao pior cenário possível. Neste caso, a data com o pior range é 46.

Portanto, os indicadores estão sendo configurados para começar exatamente no ponto em que o indicador de média móvel de 46 está disponível. Se este indicador estiver disponível, todos os outros indicadores de todas as faixas também estarão disponíveis.

Assim, esse trecho de código garante que o backtesting inicie no dia correspondente ao pior cenário possível.

```
def buscando_data_inicial(self):
    parametro1_maximo = max(self.parametro1)
    parametro1_minimo = min(self.parametro1)

    datas_maximas = []

    if self.parametro2 != ():
        parametro2_maximo = max(self.parametro2)
        parametro2_minimo = min(self.parametro2)

        lista_parametros = [parametro1_maximo, parametro1_minimo]
        lista_parametro2 = [parametro2_minimo, parametro2_maximo]

    else:
        lista_parametros = [parametro1_maximo, parametro1_minimo]

    for parametro in lista_parametros:
        self.otimizacao.estrategia.parametro1 = parametro
        if self.parametro2 != ():
            self.otimizacao.estrategia.parametro2 = parametro2_maximo
            self.otimizacao.estrategia.fazendo_indicadores()
            lista = [create_indicador.index[0] for create_indicador in self.otimizacao.estrategia.lista_indicadores]
            datas_maximas = datas_maximas + lista
        if self.parametro2 != ():
            for parametro in lista_parametro2:
                self.otimizacao.estrategia.parametro1 = parametro1_maximo
                self.otimizacao.estrategia.parametro2 = parametro
                self.otimizacao.estrategia.fazendo_indicadores()
                datas_parametro2 = [create_indicador.index[0] for create_indicador in self.otimizacao.estrategia.lista_indicadores]
                datas_maximas = datas_maximas + datas_parametro2
                data_mais_recente = max(datas_maximas)
                self.data_inicial = data_mais_recente
```

### 5.5.3. – Rodando o WFA

Assim, após ter as datas disponíveis, inicie a execução do WFA (Walk-Forward Analysis).

A premissa é semelhante à do factor investing: se você teve um bom desempenho no factor investing, é provável que se saia bem aqui, pois a dinâmica é bastante similar.

Para conduzir o WFA, primeiro, identifique as datas disponíveis. Em seguida, defina quantas otimizações serão possíveis considerando os anos de testes, as datas disponíveis e os anos para a otimização. Ou seja, calculamos quantas otimizações podem ser realizadas, dado um conjunto específico de datas disponíveis.

```
def run_walk(self):
    self.otimizacao.estrategia.escolher_data = True
    datas_disponiveis = (self.otimizacao.estrategia.dados[self.otimizacao.estrategia.dados.index >= self.data_inicial]).index
    otimizacoes_posseiveis = (len(datas_disponiveis) - 252 * self.anos_otimizacao)/int(252 * self.anos_teste)

    lista_df_otimizacao_por_periodo = []
    lista_df_trades_oos = []

    df_analise_is_oos = pd.DataFrame(columns=['data_inicial_IS', 'data_final_IS', 'retorno_IS_am',
                                                'data_inicial_OOS', 'data_final_OOS', 'retorno_OOS_am', 'parametro1', 'parametro2'])

    soma_datas = 0
```

Em seguida, é criada a tabela o DataFrame em `df\_analise\_is\_oos`, que será apresentado no PDF, contendo os resultados e informações relevantes do processo de WFA.

**Tabela resultados otimização**

Data inicial IS	Data final IS	Retorno IS a.m.	Data inicial OOS	Data final OOS	Retorno OOS a.m.	Parâmetro 1	Parâmetro 2
2010-03-10	2012-03-19	1.37%	2012-03-20	2013-03-28	1.25%	28	40
2011-03-16	2013-03-28	1.57%	2013-04-01	2014-04-02	-1.06%	14	40
2012-03-19	2014-04-02	2.18%	2014-04-03	2015-04-09	2.09%	7	35
2013-03-28	2015-04-09	1.93%	2015-04-10	2016-04-19	2.84%	7	40
2014-04-02	2016-04-19	7.15%	2016-04-20	2017-04-24	1.7%	21	30
2015-04-09	2017-04-24	5.74%	2017-04-25	2018-05-02	1.61%	21	30
2016-04-19	2018-05-02	2.86%	2018-05-03	2019-05-13	-2.11%	28	30
2017-04-24	2019-05-13	1.64%	2019-05-14	2020-05-18	1.68%	21	35
2018-05-02	2020-05-18	0.63%	2020-05-19	2021-04-16	3.19%	14	35

O final da primeira parte do código é um `for in range`. Nele, erá identificado o loop, que irá de 0 até o número total de otimizações possíveis + 1. Por exemplo, se o modelo tem 10 anos e estamos otimizando numa proporção de 2:1, teremos um ano restante que será o modelo que estará atualmente em execução. Portanto, adicionamos o +1.

No backtesting, será executado somente no período da otimização, não considerando o período completo, a fim de evitar viés de informação. Isso será feito filtrando apenas a data inicial e final da otimização.

Em seguida, utilizaremos o método `run.optimize` para cada uma das otimizações possíveis dentro do loop. Em cada otimização, será executado o `run.optimize` para o período de datas escolhido.

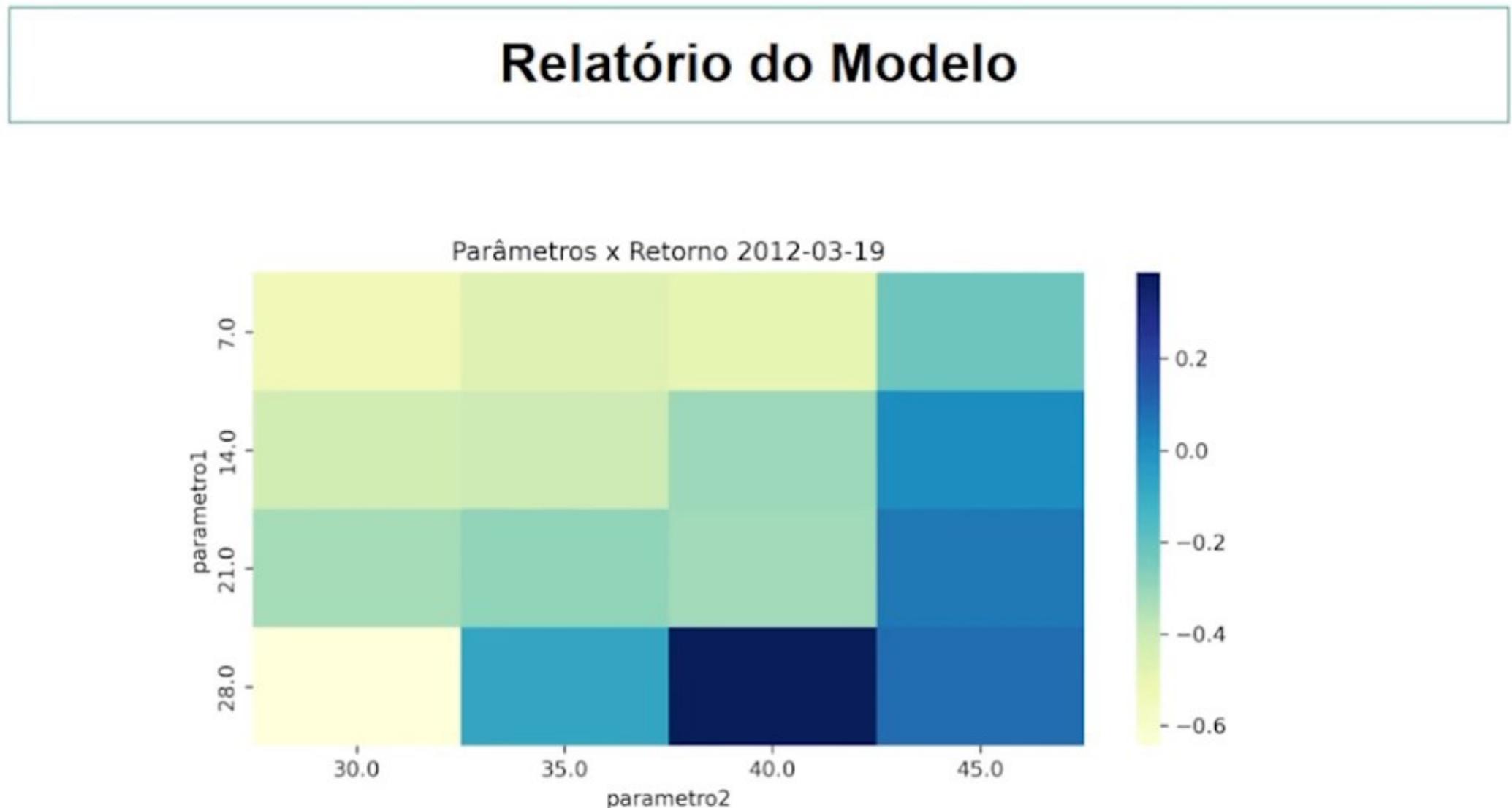
Depois, comece a rodar.

```
def run_walk(self):  
  
    self.otimizacao.estrategia.escolher_data = True  
  
    datas_disponiveis = (self.otimizacao.estrategia.dados[self.otimizacao.estrategia.dados.index >= self.data_inicial]).index  
  
    otimizacoes_possiveis = (len(datas_disponiveis) - 252 * self.anos_otimizacao)//int(252 * self.anos_teste)  
  
    lista_df_otimizacao_por_periodo = []  
    lista_df_trades_oos = []  
  
    df_analise_is_oos = pd.DataFrame(columns=['data_inicial_IS', 'data_final_IS', 'retorno_IS_am', 'data_inicial_OOS',  
                                              'data_final_OOS', 'retorno_OOS_am', 'parametro1', 'parametro2'])  
  
    soma_datas = 0  
  
    for i in range(0, otimizacoes_possiveis + 1):  
        self.otimizacao.estrategia.data_inicial = datas_disponiveis[int(0 + soma_datas)]  
        self.otimizacao.estrategia.data_final = datas_disponiveis[int((252 * self.anos_otimizacao) + soma_datas)]  
  
        self.otimizacao.run_optimize()
```

### 5.5.3.1. – In Sample

Na fase In Sample, ao chegarmos ao final do programa, iremos criar um heatmap utilizando o DataFrame que contém os retornos.

Dentro desse heatmap, vamos identificar o melhor modelo referente ao período de otimização para executar como teste. Isso será feito filtrando os parâmetros que geraram o melhor retorno, ajudando a identificar o conjunto de parâmetros que produz o melhor desempenho. No exemplo abaixo, o melhor modelo foi a combinação de parâmetro 28 com 40.



Esse modelo identificado como o melhor, será utilizado para o teste final, permitindo avaliar sua performance em um contexto diferente, a fim de validar sua eficácia.

**Tabela resultados otimização**

Data inicial IS	Data final IS	Retorno IS a.m.	Data inicial OOS	Data final OOS	Retorno OOS a.m.	Parâmetro 1	Parâmetro 2
2010-03-10	2012-03-19	1.37%	2012-03-20	2013-03-28	1.25%	28	40
2011-03-16	2013-03-28	1.57%	2013-04-01	2014-04-02	-1.06%	14	40
2012-03-19	2014-04-02	2.18%	2014-04-03	2015-04-09	2.09%	7	35

### 5.5.3.2. – Out of Sample

No Out of Sample (OOS), executaremos nossa estratégia utilizando os parâmetros ótimos determinados com base no heatmap. A configuração dos parâmetros será ajustada para rodar na próxima data, que corresponderá ao ano de teste.

Toda vez que o modelo finaliza seu período OOS e inicia um novo período, é crucial herdar algumas características do período anterior. Afinal, serão utilizados outros parâmetros. Por exemplo, se o OOS anterior finalizou com posição comprada em CDI, o novo OOS deve iniciar já nesse estado e, a partir daí, rodar o modelo de teste em relação ao período de otimização.

Para isso, foi feito um bloco de condicional `if-else` para verificar as condições do final do OOS anterior. Se o último período terminou com uma determinada condição (por exemplo, comprado em CDI), o novo OOS é configurado para iniciar nesse mesmo estado.

```
#isso aqui tem que ser uma lista pra fazer os heatmaps
lista_df_otimizacao_por_periodo.append(self.otimizacao.df_retorno_acum)

#rodar o melhor modelo pra extrair os parametros e o df trade in sample
melhor_modelo = self.otimizacao.df_retorno_acum[self.otimizacao.df_retorno_acum['retorno'] == self.otimizacao.df_retorno_acum['retorno'].max()]

parametro_otimo1 = melhor_modelo['parametro1'].iat[0]
parametro_otimo2 = melhor_modelo['parametro2'].iat[0]

df_analise_is_oos.loc[i, 'data_inicial_IS'] = datas_disponiveis[int(0 + soma_datas)].date()
df_analise_is_oos.loc[i, 'data_final_IS'] = datas_disponiveis[int((252 * self.anos_otimizacao) + soma_datas)].date()
df_analise_is_oos.loc[i, 'parametro1'] = parametro_otimo1
df_analise_is_oos.loc[i, 'parametro2'] = parametro_otimo2

self.otimizacao.estrategia.parametro1 = parametro_otimo1
self.otimizacao.estrategia.parametro2 = parametro_otimo2

self.otimizacao.estrategia.run_strategy()

trades_in_sample = self.otimizacao.estrategia.df_trades

dias_backtest = len(trades_in_sample)

retorno_acum = ((trades_in_sample['retorno'] + 1).cumprod() - 1).iat[-1]
retorno_am = (1 + retorno_acum) ** (21/dias_backtest) - 1

df_analise_is_oos.loc[i, 'retorno_IS_am'] = retorno_am
```

Em seguida, será executado o método `run\_strategy` com `reset = False`, justamente para iniciar em um estado específico, que corresponderá ao encerramento do último período de OOS. Essa abordagem permitirá iniciar o novo período de OOS de onde o período anterior foi encerrado, garantindo a continuidade das condições de estado do modelo.

```
#OUT OF SAMPLE

self.otimizacao.estrategia.data_inicial = datas_disponiveis[int((252 * self.anos_otimizacao + 1) + soma_datas)]

try:

    self.otimizacao.estrategia.data_final = datas_disponiveis[int((252 * (self.anos_otimizacao + self.anos_teste)) + soma_datas)]

except:

    self.otimizacao.estrategia.data_final = datas_disponiveis[-1]

if i != 0:

    self.otimizacao.estrategia.numero_do_trade = numero_do_trade
    self.otimizacao.estrategia.comprado = comprado
    self.otimizacao.estrategia.vendido = vendido
    self.otimizacao.estrategia.cdi = comprado_cdi
    self.otimizacao.estrategia.fechamento = fechamento
    self.otimizacao.estrategia.zerou_compra = zerou_compra
    self.otimizacao.estrategia.zerou_venda = zerou_venda

    self.otimizacao.estrategia.run_strategy(reset = False)

else:

    self.otimizacao.estrategia.run_strategy()

numero_do_trade = self.otimizacao.estrategia.numero_do_trade
comprado = self.otimizacao.estrategia.comprado
vendido = self.otimizacao.estrategia.vendido
comprado_cdi = self.otimizacao.estrategia.cdi
fechamento = self.otimizacao.estrategia.dados_filtrados_data.fechamento[-1]
zerou_compra = self.otimizacao.estrategia.zerou_compra
zerou_venda = self.otimizacao.estrategia.zerou_venda

trades_outof_sample = self.otimizacao.estrategia.df_trades
```

```
for z, indicador in enumerate(self.otimizacao.estrategia.lista_indicadores):

    trades_outof_sample[f'indicador{z}'] = indicador.values

    trades_outof_sample['data'] = pd.to_datetime(trades_outof_sample['data'])

    trades_outof_sample = pd.merge(trades_outof_sample, self.otimizacao.estrategia.dados, left_on = 'data', right_index = True)

    lista_df_trades_oos.append(trades_outof_sample)

dias_backtest = len(trades_outof_sample)

retorno_acum_oos = ((trades_outof_sample['retorno'] + 1).cumprod() - 1).iat[-1]
retorno_am_oos = (1 + retorno_acum_oos) ** (21/dias_backtest) - 1

df_analise_is_oos.loc[i, 'retorno_OOS_am'] = retorno_am_oos
df_analise_is_oos.loc[i, 'data_inicial_OOS'] = datas_disponiveis[int((252 * self.anos_otimizacao + 1) + soma_datas)].date()

try:

    df_analise_is_oos.loc[i, 'data_final_OOS'] = datas_disponiveis[int((252 * (self.anos_otimizacao + self.anos_teste)) + soma_datas)].date()

except:

    df_analise_is_oos.loc[i, 'data_final_OOS'] = datas_disponiveis[-1].date()

soma_datas = soma_datas + (252 * self.anos_teste)
```

#### 5.5.4. – Tratamento dos dados

Essa seção final do código será dedicada ao tratamento dos dados para o `MakeReportResult`. Essencialmente, consistirá na configuração dos DataFrames, filtrando os dados de forma a garantir a coerência das datas entre os diferentes conjuntos de dados usados. Isso é crucial para evitar desencontros ou inconsistências entre as datas nos DataFrames, evitando que o modelo tenha datas diferentes do índice de referência, do CDI, das ações, e assim por diante.

Resumidamente, será apenas um tratamento de datas para alinhar os dados corretamente, garantindo a consistência e coerência necessárias para o `MakeReportResult`. O restante do código não precisará ser modificado, a menos que seja necessário ajustar algo nas seções de evento, indicadores, dados e estratégia, intervalo de otimização e teste, nome e localização dos arquivos de dados.

```
#filtrando dados pra report (onde começa o df trades e termina)

data_filtro = pd.to_datetime(df_trades_out_of_sample['data'].iloc[0])

dados_filtrados_out_of_sample = self.dados[self.dados.index >= data_filtro]

dados_filtrados_out_of_sample['retorno'] = dados_filtrados_out_of_sample['fechamento'].pct_change()

df_trades_out_of_sample['cota'] = (df_trades_out_of_sample['retorno'] + 1).cumprod() - 1

if self.parametro2 !=():

    MakeReportResult(df_trades=df_trades_out_of_sample, df_dados = dados_filtrados_out_of_sample, otimizacao=True,
                      df_otimizacao = df_analise_is_oos, lista_df_heatmap_parametros = lista_df_otimizacao_por_periodo,
                      nome_arquivo = self.nome_arquivo, caminho_benchmarks=self.otimizacao.estrategia.caminho_benchmarks,
                      caminho_imagens=self.otimizacao.estrategia.caminho_imagens)

else:

    MakeReportResult(df_trades=df_trades_out_of_sample, df_dados = dados_filtrados_out_of_sample, otimizacao=True,
                      df_otimizacao = df_analise_is_oos, lista_df_heatmap_parametros = lista_df_otimizacao_por_periodo,
                      parametro1_only = True, nome_arquivo = self.nome_arquivo,
                      caminho_benchmarks=self.otimizacao.estrategia.caminho_benchmarks,
                      caminho_imagens=self.otimizacao.estrategia.caminho_imagens)
```

## Mundo 6

### 6.1. – Backtest de Média Móvel

Neste mundo, serão utilizados dados de outras fontes para a realização do backtest, afinal, muitas pessoas utilizam outra base de dados como: Bloomberg, TradingView, Investing.com, Yahoo Finance (o que será utilizado). Portanto, nesta galáxia de Análise Técnica é disponibilizada a integração com outras bases de dados para teste.

No programa, houve pouca mudança. Na função '`__name__ = "__main__"`', é necessário realizar o download do data frame da ação e depois ajustar as colunas que serão utilizadas – abertura, fechamento, máxima e mínima. Por fim, modifique os caminhos e passe "to.parquet", salvando as cotações da nossa ação.

Como está sendo feito somente com uma única cotação, o código é modificado para '`tem_multiplas_empresa = False`', ou simplesmente apague, como ensinado no Mundo 3.

```
if __name__ == "__main__":
    acao = "CSNA3"
    import yfinance as yf
    dados_yf = yf.download(acao + ".SA")
    dados_yf['fator_ajuste'] = dados_yf['Close']/dados_yf['Adj Close']
    ajustar = ['Open', 'High', 'Low']
    for ajuste in ajustar:
        dados_yf[ajuste] = dados_yf[ajuste]/dados_yf['fator_ajuste']
    dados_yf = dados_yf.drop(['Close', 'fator_ajuste'], axis = 1)
    dados_yf = dados_yf.reset_index()
    dados_yf.to_parquet(fr'c:\Users\lsiqu\dev\base_dados_br\cotacoes_{acao}.parquet')
    dados = ReadData(
        caminho_parquet = fr'c:\Users\lsiqu\dev\base_dados_br\cotacoes_{acao}.parquet',
        data_inicial = "2000-01-01",
        data_final = "2023-04-28",
        formato_data = ('%Y-%m-%d'),
        coluna_data = 0,
        abertura = 1,
        minima = 3,
        maxima = 2,
        fechamento = 4,
        volume = 5
    )
```

É necessário resetar o index para inserir a cotação dentro do ReadData, pois o Yahoo Finance automaticamente coloca as datas como índice, e em no código, tem que ser na coluna zero.

Ao utilizar dados antigos no backtest, é importante verificar se esses dados estão disponíveis. Por exemplo, no Yahoo Finance, os dados estão disponíveis apenas desde 2010, o que pode gerar bugs ao tentar acessar informações desde 2000. Para contornar essa situação, foi utilizado os dados do Ibovespa da nossa base de dados da Fintz. Esse cuidado é necessário para qualquer tipo de dado, como o Ibov, CDI, entre outros.

```
response = requests.get('https://api.fintz.com.br/indices/historico?indice=IBOV&dataInicio=2000-01-01',
                       headers=self.headers)

df = pd.DataFrame(response.json())

df = df.sort_values('data', ascending=True)

df.columns = ['indice', 'data', 'fechamento']

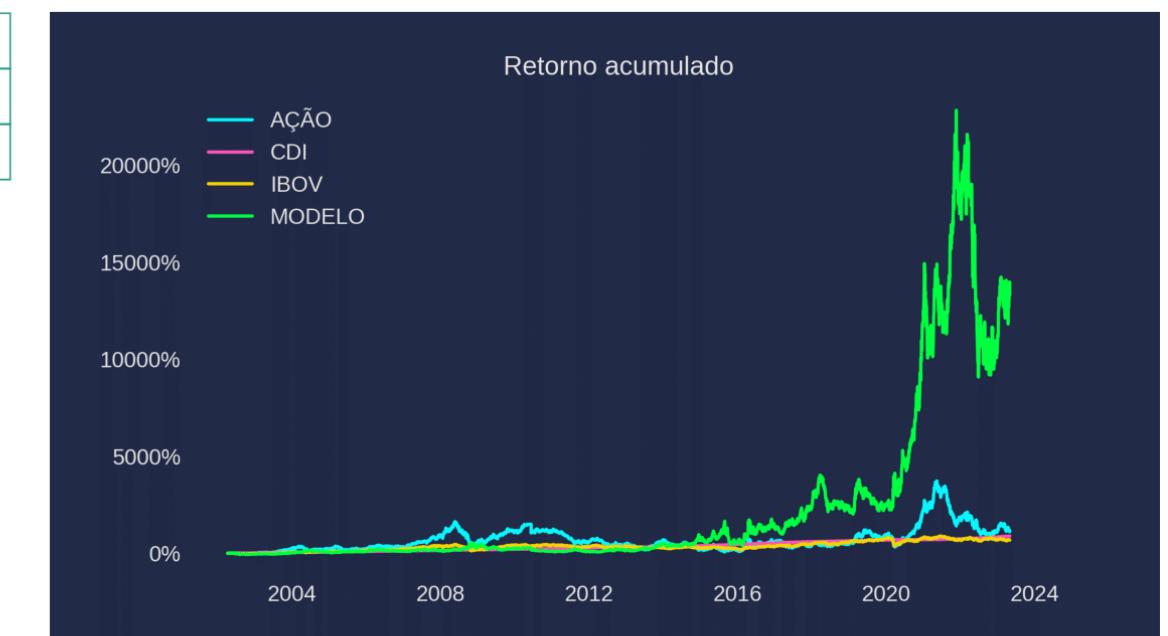
df = df.drop('indice', axis = 1)

df.to_parquet('ibov.parquet', index = False)
```

O resultado do backtest utilizando média móvel, com dados desde 2000, para a CSNA3 foi este.

### Relatório do Modelo

Dia inicial	2002-04-08
Dia final	2023-04-28
Dias totais	5228



### Estatísticas de Retorno e Risco

Retorno acum. modelo	13324.7%
Retorno acum. ativo	1166.81%
Retorno acum. CDI	885.47%
Retorno acum. IBOV	686.6%
Retorno a.a. modelo	26.64%
Vol 252d	53.94%
Índice Sharpe	0.28
VAR diário 95%	-5.08%
Drawdown máximo	-73.49%

### Estatísticas de Trade

Número de trades	176
% Operações vencedoras	42.61%
% Operações perdedoras	57.39%
Média de ganhos	23.0%
Média de perdas	-8.16%
Expec. matemática por trade	5.12%
Tempo médio de operação	29
Maior sequência de vitória	5
Maior sequência de derrotas	10

## Mundo 7

### 7.1. – Backtest Hi-Lo

O indicador HI-LO é utilizado para identificar novas tendências no mercado. Para isso, ele faz uso da média das máximas e mínimas dos preços da ação.

No método 'fazendo\_indicadores', será calculado a média móvel simples (SMA) usando o preço máximo e mínimo para o mesmo período. Neste caso, utilizaremos o período dos últimos 30 dias.

```
def fazendo_indicadores(self):  
  
    self.media_maxima = MakeIndicator().media_movel_simples(self.dados.maxima, self.parametro1)  
    self.media_minima = MakeIndicator().media_movel_simples(self.dados.minima, self.parametro1)  
  
    self.lista_indicadores = [self.media_maxima, self.media_minima]
```



A partir dessas médias máximas e mínimas, no método 'evento', gere os sinais de compra e venda. O código funciona na seguinte lógica: se o preço de fechamento estiver acima da média máxima, faça uma compra; se o preço de fechamento estiver abaixo da média mínima, faça uma venda. Simples assim.

No entanto, existe uma situação intermediária a ser considerada. O preço pode estar entre a média mínima e a média máxima, ou seja, em um limbo. E quando o preço está nesse limbo, o que faremos? Vamos manter a operação ativa?

Se o estado do programa estiver em uma posição de compra, e o preço cair abaixo da média máxima, não haverá problema, continue mantendo essa posição na empresa. A alteração na operação só ocorrerá se o preço cair abaixo da média mínima.

```
def evento(self, data, i):
    if self.dados.fechamento[data] > self.media_maxima[data]:
        if self.comprado:
            pass
        else:
            self.vender_cdi()
            self.compra(inverter=True)

    elif self.dados.fechamento[data] < self.media_minima[data]:
        if self.vendido:
            pass
        else:
            self.venda(inverter = True)
            self.comprar_cdi()
```

O TradingView é um ótimo lugar para a visualização das estratégias de análise técnica. No gráfico, as regiões em vermelho representam a média máxima e a posição de vendido, enquanto as regiões em verde representam a média mínima e a posição de comprador.



A média móvel do Hi-Lo tem algumas pequenas mudanças em relação à média móvel simples que utilizamos até aqui. Foi utilizado outra máxima de preço e o sinal foi alterado, em 'evento'.

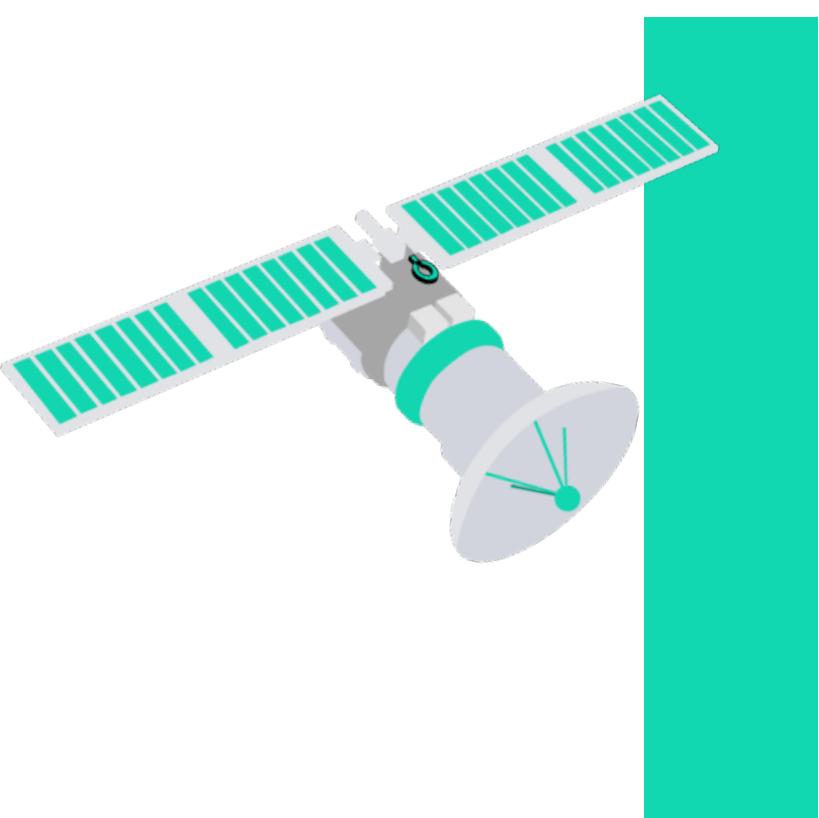
Ou seja, se o preço estiver acima do fechamento e estiver comprado, será enviado o comando 'pass'. Se não, venda o CDI e configure a estratégia com 'self.compra (inverter=True)', pois a operação será de long-short.

Se o preço estiver abaixo do fechamento e estiver vendido, será enviado o comando 'pass'. Caso contrário, venda a ação e compre o CDI, já que ao vender a ação, recebemos esse dinheiro para investir.

A otimização dos parâmetros é entre 15 e 50 períodos de 3 em 3, com 2 anos de otimização e 1 ano de teste.

```
walk = WalkForwardAnalysis(estrategia = hilo_estrategia(), class_dados = dados,
                           parametro1= range(15, 50, 3), anos_otimizacao=2, anos_teste=1,
                           nome_arquivo = rf"c:\Users\lsiqu\dev\PDFS_BACKTEST\analise_tecnica\backtest_2pra1_{acao}_HIL0.pdf",
                           caminho_dados_benchmarks =r'c:\Users\lsiqu\dev\base_dados_br',
                           caminho_imagens= r'c:\Users\lsiqu\dev\PDFS_BACKTEST\imagens')
```

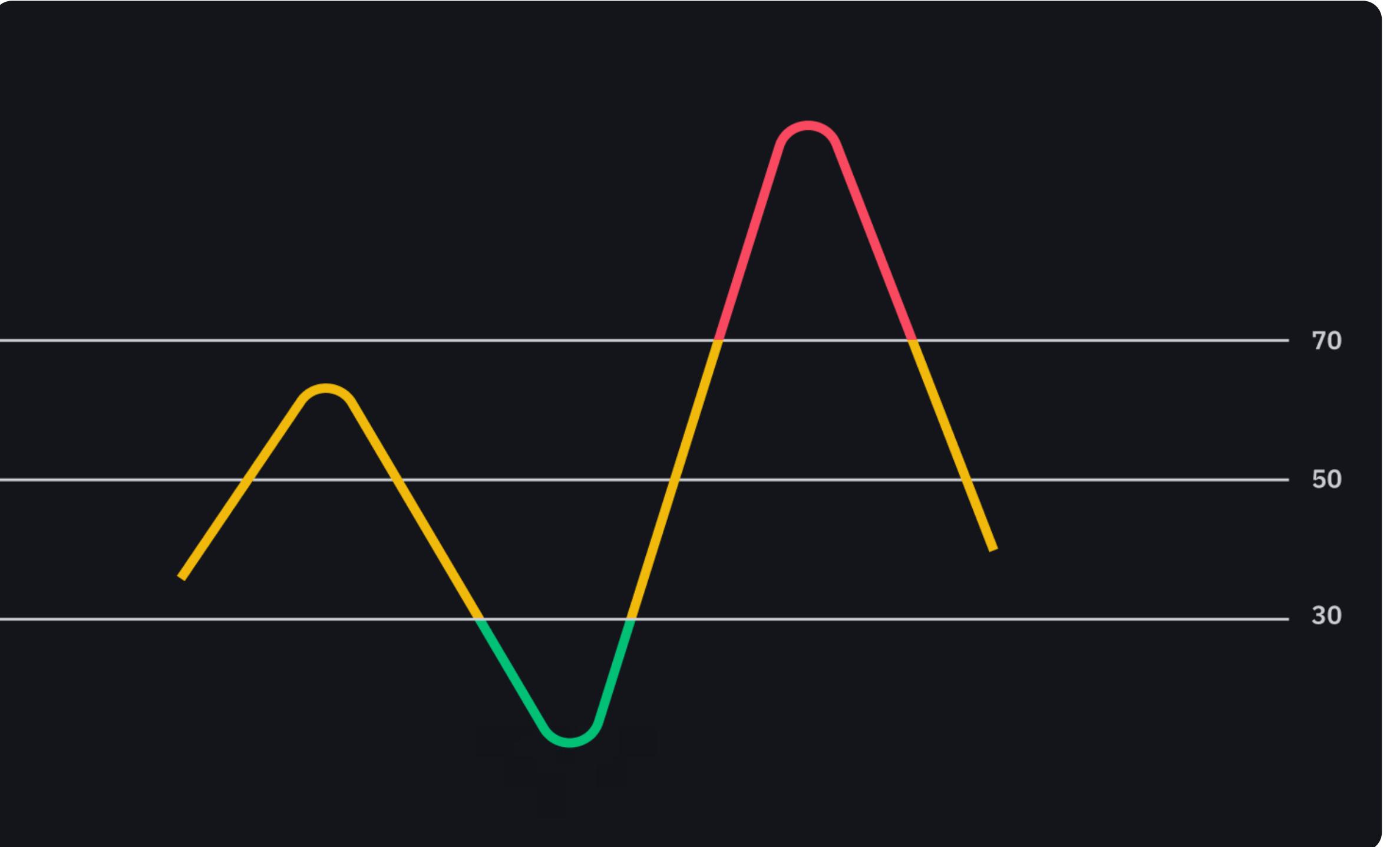
Ess é o resultado do backtest de Hi-Lo.



## Mundo 8

### 8.1. – Backtest RSI e RSI Invertido.

O indicador RSI (Índice de Força Relativa) é mais uma maneira de medir a força e a direção de uma tendência no preço de um ativo. Essa estratégia é conhecida como reversão à média, mas também pode ser utilizada para seguir tendências.



Para calcular o RSI, é feita uma média dos retornos positivos em relação aos retornos negativos, e esse valor é normalizado para oscilar entre 0 e 100. Se o indicador estiver próximo de 100, isso sugere que a empresa ou ativo está em uma tendência de alta bastante forte. Por outro lado, se o indicador estiver próximo de zero, isso indica uma tendência de baixa consideravelmente forte.

Será feito um backtest com PETR4 entre 2010 e 2023.

No arquivo de `indicadores`, existe a função para calcular o RSI. Nesta função, use o `pct\_change()` para obter os retornos percentuais dos dados de fechamento da Petrobras. Em seguida, utilize uma função lambda para separar os retornos positivos e negativos.

Na função lambda: Se  $X > 0$ , considere esse retorno positivo, caso contrário, será atribuído o valor zero. Se  $X < 0$ , considere esse retorno negativo; do contrário, será também atribuído o valor zero.

Depois, será calculado a média desses retornos. E por fim, faça a normalização dos resultados para obter o RSI.

```
def RSI(self, coluna_preco, periodo):  
    """ Fórmula RSI:  
  
    #100 - 100/(1 + mediaRetornosPositivos / mediaRetornosNegativos)  
  
    retornos = coluna_preco.pct_change().dropna()  
  
    retornos_positivos = retornos.apply(lambda x: x if x > 0 else 0)  
    retornos_negativos = retornos.apply(lambda x: abs(x) if x < 0 else 0)  
  
    media_retornos_positivos = retornos_positivos.rolling(window = periodo).mean().dropna()  
    media_retornos_negativos = retornos_negativos.rolling(window = periodo).mean().dropna()  
  
    rsi = 100 - 100/(1 + media_retornos_positivos/media_retornos_negativos)  
  
    return rsi
```

Definido o indicador RSI, é hora de definir seu evento.

Em `evento`, se o RSI for menor que 30 e não estiver comprado, será emitido um sinal de compra.

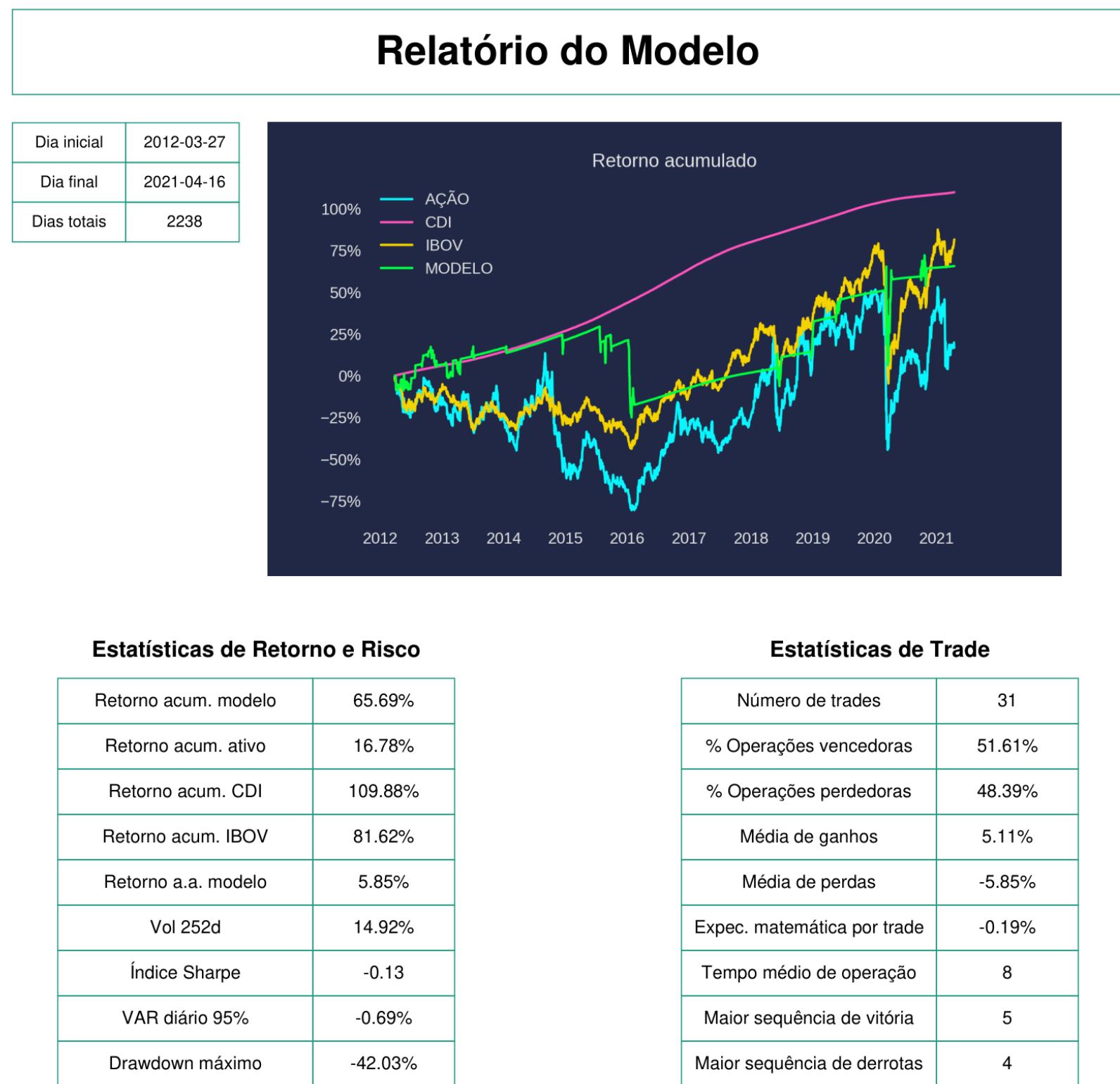
A ideia por trás dessa estratégia é a suposição de que, se o RSI estiver abaixo de 30, a empresa está propensa a uma reversão da média, indicando uma possível valorização futura.

Por outro lado, se o RSI for superior a 60 e caso esteja comprado, a posição será zerada. Após encerrar essa posição, entrará comprado em CDI. Este modelo de estratégia opera exclusivamente na condição de estar comprado.

```
def fazendo_indicadores(self):  
  
    self.rsi = MakeIndicator().RSI(self.dados.fechamento, self.parametro1)  
  
    self.lista_indicadores = [self.rsi]  
  
def evento(self, data, i):  
  
    if self.rsi[data] < 30 and (self.comprado == False): #reversão a média!  
  
        self.vender_cdi()  
  
        self.compra()  
        self.barra_executada = i  
  
    elif self.rsi[data] > 60 and self.comprado:  
  
        self.venda(zerar = True)  
  
        self.comprar_cdi()
```

E esse foi o resultado do modelo:

código.py



## Mundo 9

### 9.1. – Backtest MACD e MACD Invertido

O MACD (Moving Average Convergence Divergence) é um indicador que também se baseia em médias, mas de uma forma diferente.

Para isso, no método 'fazendo\_indicadores', calcule duas médias diferentes: uma média de longo prazo e outra de curto prazo. Ou seja, em vez de usar uma média comum que trata todos os dados da mesma forma, vamos usar a média móvel exponencial.

A média móvel exponencial dá uma maior importância aos dados mais recentes, ou seja, ela se concentra nos valores mais novos do que nos antigos. Portanto, a cotação de ontem tem um peso maior do que a cotação dos últimos 20 dias. Isso ajuda a perceber mudanças mais rapidamente, dando uma ideia melhor das tendências mais atuais do mercado.

Para criar o MACD, primeiro calcule a média de longo prazo e curto prazo. Em seguida, subtraia a média de curto prazo da média de longo prazo.

Depois, aplique a média exponencial para chegar na média do MACD.

Quanto aos sinais de compra ou venda, são gerados ao relacionar o valor do MACD sobre a média do MACD. Então, se o valor atual estiver acima da média naquela data, pode sinalizar um momento de compra; se estiver abaixo, pode indicar um momento de venda.

```
def fazendo_indicadores(self):

    self.media_longa = MakeIndicator().media_movel_exponencial(self.dados.fechamento, 26)
    self.media_curta = MakeIndicator().media_movel_exponencial(self.dados.fechamento, 12)

    self.MACD = self.media_longa - self.media_curta

    self.media_MACD = MakeIndicator().media_movel_exponencial(self.MACD, self.parametro1)

    self.lista_indicadores = [self.media_longa, self.media_longa, self.MACD, self.media_MACD]

def evento(self, data, i):

    if self.MACD[data] >= self.media_MACD[data]:
        #if self.comprado:
        if self.vendido:

            pass

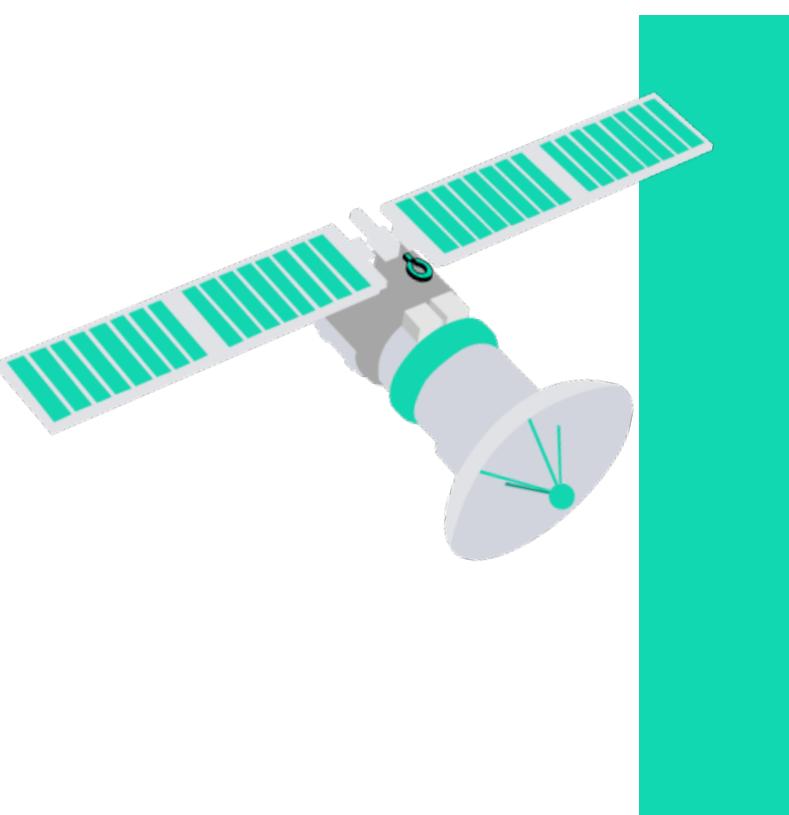
        else:
            # self.vender_cdi()
            # self.compra(inverter=True)
            self.venda(inverter=True)
            self.comprar_cdi()

    elif self.MACD[data] < self.media_MACD[data]:
        #if self.vendido:
        if self.comprado:

            pass

        else:
            # self.venda(inverter = True)
            # self.comprar_cdi()
            self.vender_cdi()
            self.compra(inverter=True)
```

E esse é o resultado do modelo.



Este modelo resultou em um com rentabilidade negativa. Em casos como este, experimente uma abordagem inversa. Em vez de comprar, considere a venda. Ao invés de seguir a tendência predominante, busque operar contra ela.

Essa nova estratégia pretende explorar a inversão de ações, ou seja, ao invés de seguir a direção do mercado, tente lucrar com movimentos contrários. Essa mudança de perspectiva pode proporcionar novas oportunidades de análise e estratégias, permitindo explorar possíveis cenários que não foram considerados anteriormente.

E foi esse o resultado do modelo operando na direção oposta ao mercado.

## Relatório do Modelo

Dia inicial	2012-03-08
Dia final	2021-04-16
Dias totais	2251



### Estatísticas de Retorno e Risco

Retorno acum. modelo	221.61%
Retorno acum. ativo	15.86%
Retorno acum. CDI	110.86%
Retorno acum. IBOV	83.46%
Retorno a.a. modelo	13.97%
Vol 252d	48.59%
Índice Sharpe	0.1
VAR diário 95%	-4.69%
Drawdown máximo	-69.71%

### Estatísticas de Trade

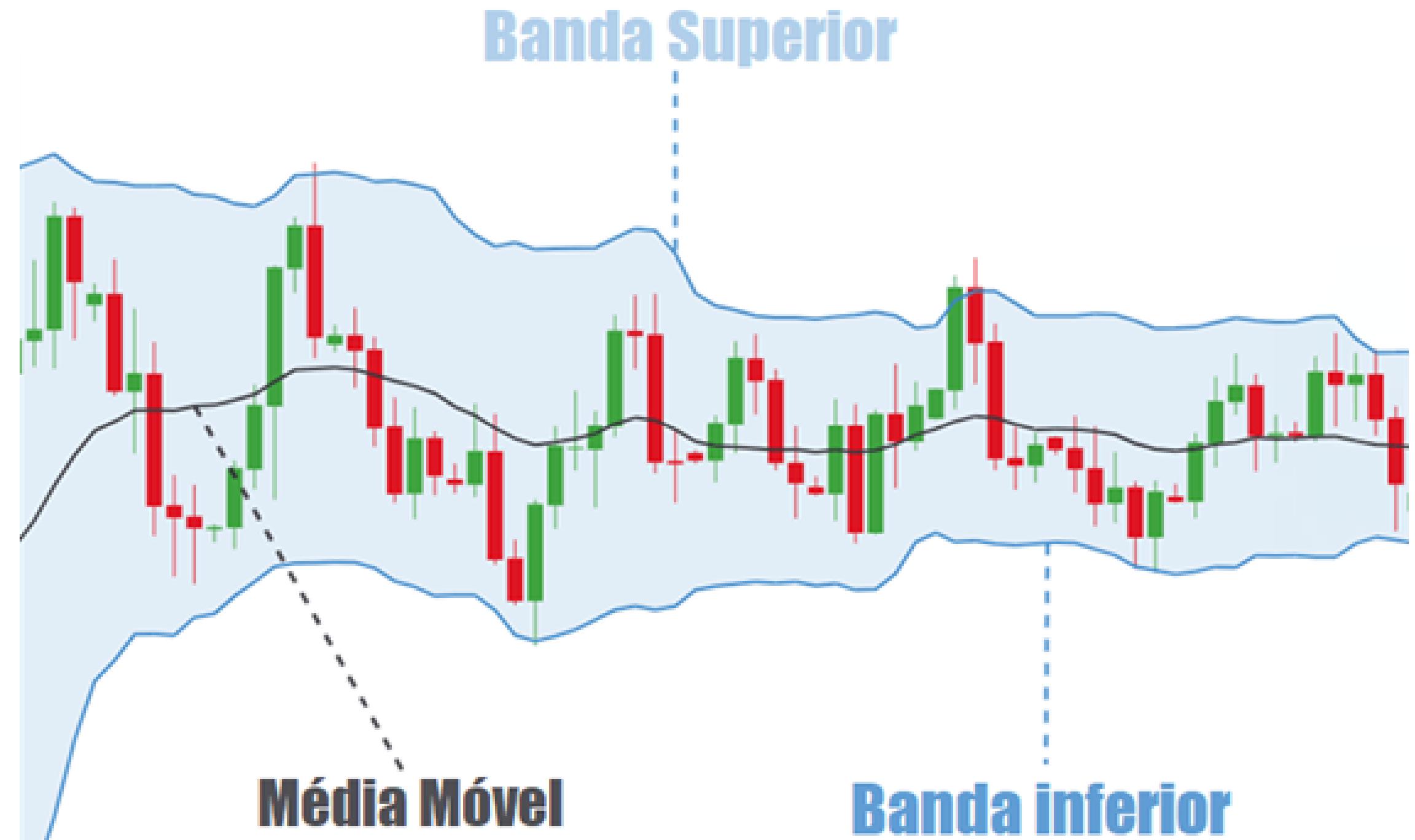
Número de trades	184
% Operações vencedoras	34.24%
% Operações perdedoras	65.76%
Média de ganhos	12.88%
Média de perdas	-4.83%
Expec. matemática por trade	1.23%
Tempo médio de operação	12
Maior sequência de vitória	5
Maior sequência de derrotas	10

## Mundo 10

### 10.1. – Backtest com Bollinger Bands

As Bollinger Bands são um indicador versátil que pode ser empregado de diversas formas, mas geralmente é visto como um indicador de reversão à média. Isso significa que quando o preço de uma ação se afasta em 2 desvios padrões, existe uma tendência de retorno em direção à média.

Por exemplo, se a média estiver em R\$24 e o desvio padrão da ação for R\$4, teremos uma linha a R\$24 (média), uma linha a R\$16 (2 desvios padrões abaixo) e outra linha a R\$32 (2 desvios padrões acima). Essas linhas podem indicar áreas de suporte, resistência ou oportunidades de reversão de tendência.



Crie uma estrutura que considere a coluna de preços, o período da média móvel e o número de desvios padrões. Primeiramente, gere a média móvel e, em seguida, calcule o desvio padrão dessa média móvel com base no período determinado.

Em seguida, defina os desvios padrões que serão usados para criar a borda superior e inferior das Bollinger Bands.

```
def bollinger_bands(self, coluna_preco, periodo_media, numero_desvios):

    sma = coluna_preco.rolling(periodo_media).mean().dropna()

    desvio_padrao = coluna_preco.rolling(periodo_media).std().dropna()

    borda_superior = sma + (desvio_padrao * numero_desvios)
    borda_inferior = sma - (desvio_padrao * numero_desvios)

    return sma, borda_superior, borda_inferior
```

Duas variáveis que devem ser otimizadas são: o período da média móvel a ser utilizado e o número de desvios padrões que serão considerados para as operações.

```
def fazendo_indicadores(self):

    self.media, self.borda_superior, self.borda_inferior = MakeIndicator().bollinger_bands(self.dados.fechamento,
periodo_media = self.parametro1,
numero_desvios=self.parametro2)

    self.lista_indicadores = [self.media, self.borda_superior, self.borda_inferior]
```

Em evento, se o preço de fechamento estiver acima da borda superior, será vendido. Se estiver comprado, vende invertendo, se não, vende e compra cdi.

Durante períodos de alta volatilidade, os movimentos podem ser bastante bruscos. Há situações em que a transição direta ocorre do estado de vendido para comprado, sem zerar a posição. Normalmente, ao comprar, o procedimento padrão é zerar a operação no meio, esperar a valorização da ação e, posteriormente, realizar a venda para novamente comprar. No entanto, em alguns cenários de alta volatilidade, podemos ter casos em que não realizamos operações sequenciais de inversão. Pode ser feita apenas zerar a posição sem abrir imediatamente uma nova posição, embora em outras situações seja possível comprar e, no dia seguinte, vender diretamente. Portanto, considere essas duas possibilidades no código: se já estiver comprado, inverta e venda. Se não tiver uma posição de compra, simplesmente venda e invista no CDI.

A mesma lógica é aplicada quando o fechamento estiver abaixo da borda inferior, mas de forma inversa.

Se o fechamento estiver acima da média, zere a posição de venda; se estiver abaixo da média, zere a posição de compra.

```
def evento(self, data, i):
    if self.dados.fechamento[data] > self.borda_superior[data]:
        if self.vendido:
            pass
        elif self.comprado:
            self.venda(inverter=True)
            self.comprar_cdi()
        else:
            self.venda()
            self.comprar_cdi()
    elif self.dados.fechamento[data] < self.borda_inferior[data]:
        if self.comprado:
            pass
        elif self.vendido:
            self.vender_cdi()
            self.compra(inverter=True)
        else:
            self.vender_cdi()
            self.compra()
    elif self.comprado:
        if self.dados.fechamento[data] > self.media[data]:
            self.venda(zerar=True)
            self.comprar_cdi()
    elif self.vendido:
        if self.dados.fechamento[data] < self.media[data]:
            self.compra(zerar=True)
            self.comprar_cdi()
```

E esse foi o resultado do modelo:

código.py



## Mundo 11

### 11.1. – Backtest com Bollinger Bands Invertido

Nesse mundo será explorado o uso do Bollinger Bands invertido para apenas duas operações. Há modificação somente no método 'evento'.

Essa estratégia funciona da seguinte maneira: ao identificar o preço ultrapassando a borda superior, execute uma compra. Faça a venda sómente quando a ação sair da borda inferior.

A mesma lógica se aplica para a venda: entre vendido quando estiver abaixo da borda inferior e compre ou inverta a operação apenas quando estiver acima da borda superior.

Siga uma lógica de comprar abaixo da média e vender acima dela, alternando entre essas ações conforme a movimentação dos preços.

```

def evento(self, data, i):

    if self.dados.fechamento[data] > self.borda_superior[data]:

        if self.vendido:

            pass

        elif self.comprado:

            self.venda(inverter=True)
            self.comprar_cdi()

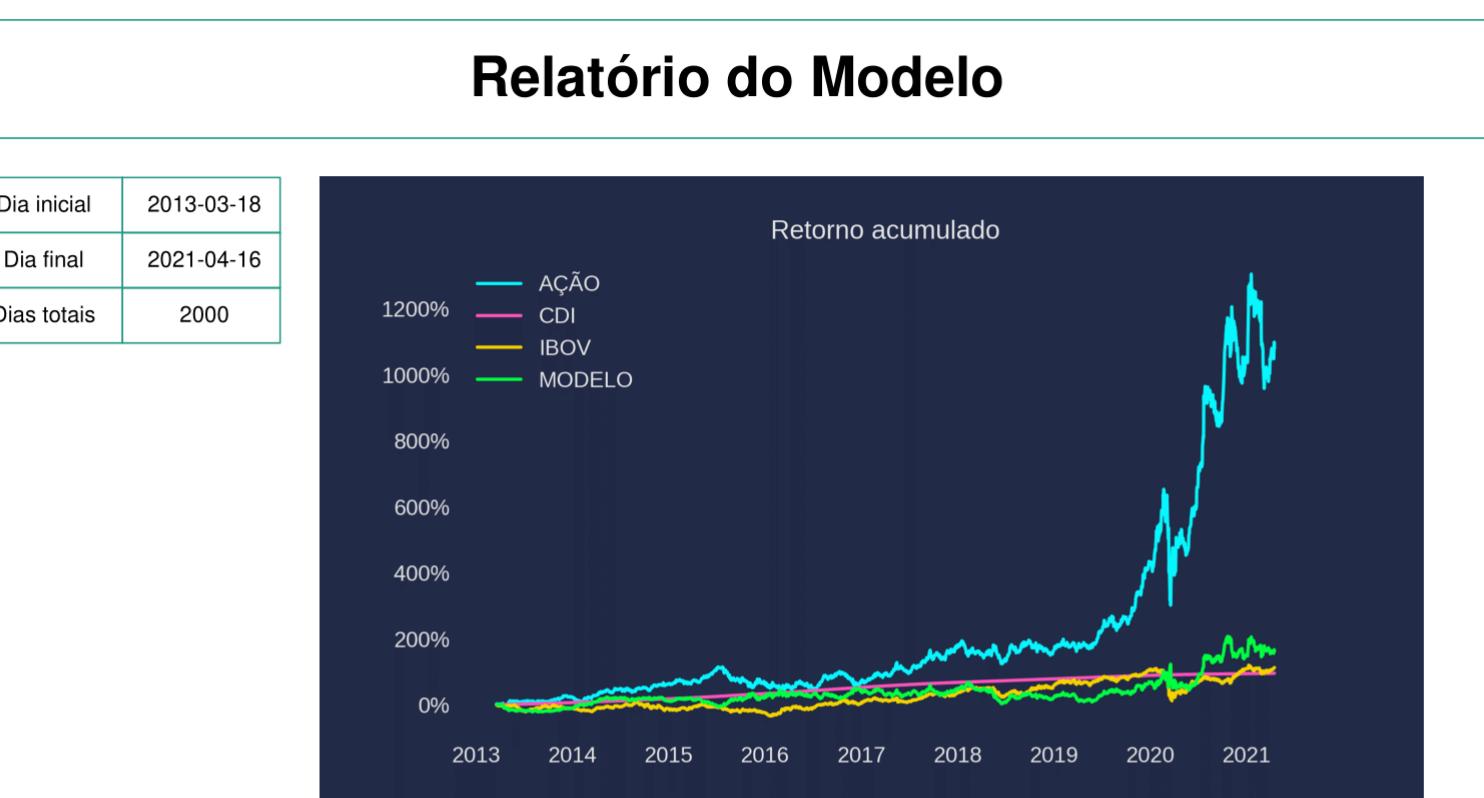
        else:

            self.venda()
            self.comprar_cdi()

```

E esse foi o resultado do modelo:

código.py



Estatísticas de Retorno e Risco

Retorno acum. modelo	162.75%
Retorno acum. ativo	1079.37%
Retorno acum. CDI	95.41%
Retorno acum. IBOV	112.97%
Retorno a.a. modelo	12.94%
Vol 252d	41.48%
Índice Sharpe	0.13
VAR diário 95%	-3.08%
Drawdown máximo	-38.75%

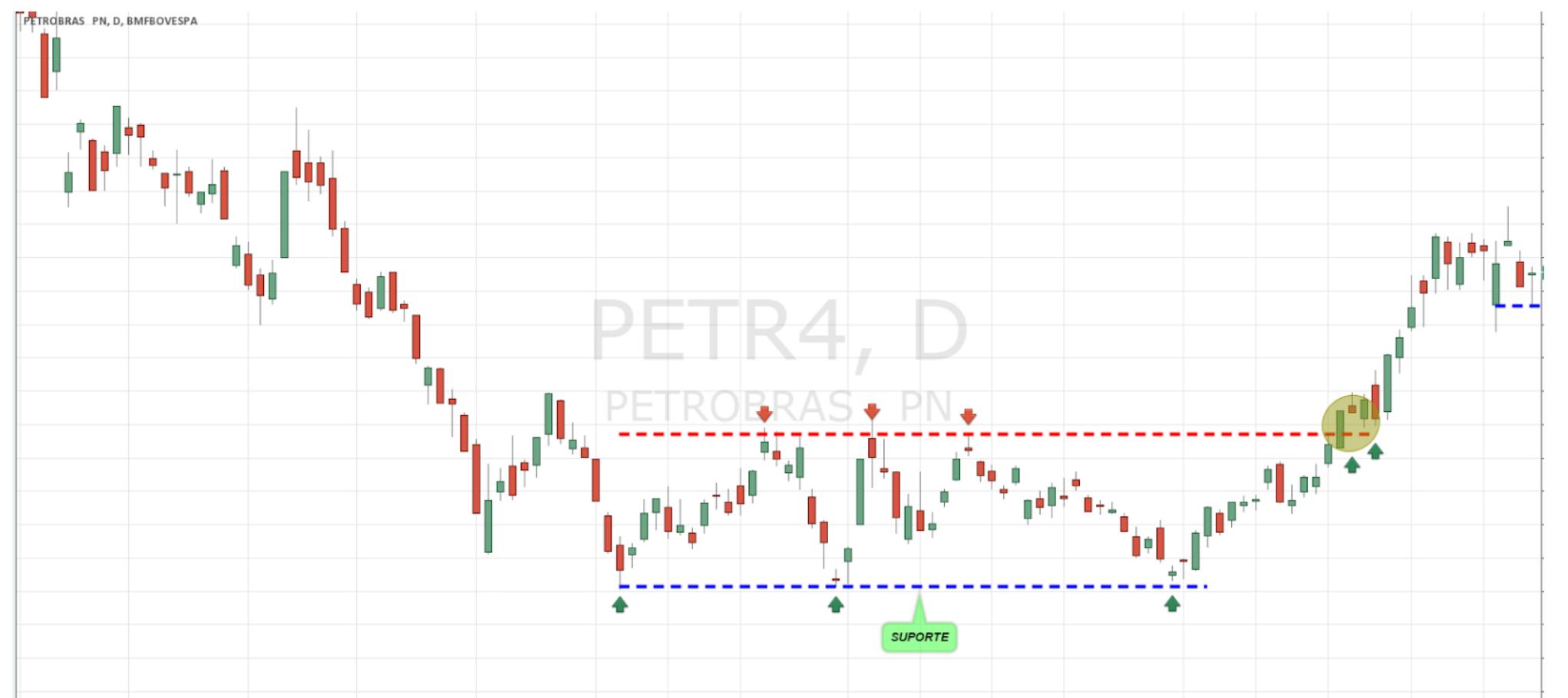
Estatísticas de Trade

Número de trades	70
% Operações vencedoras	45.71%
% Operações perdedoras	54.29%
Média de ganhos	9.25%
Média de perdas	-4.38%
Expec. matemática por trade	1.85%
Tempo médio de operação	28
Maior sequência de vitória	3
Maior sequência de derrotas	6

## Mundo 12

### 12.1. – Backtest de quebra de resistência envolvendo

Nesse backtest, será testado o modelo de quebra de resistência, porém será utilizado o volume, totalmente diferente do senso comum, onde utilizam preços.



A quebra de resistência máxima e mínima se refere aos valores extremos, alcançados durante um determinado período. Nesse contexto, o período escolhido para análise não será determinado previamente, mas sim otimizado para identificar o melhor intervalo de quebra de resistência para a nossa estratégia.

```
def quebra_resistencia_maximo(self, coluna_preco, periodo):  
    valor_maximo = coluna_preco.rolling(periodo).max().dropna()  
  
    return valor_maximo  
  
def quebra_resistencia_minimo(self, coluna_preco, periodo):  
    valor_minimo = coluna_preco.rolling(periodo).min().dropna()  
  
    return valor_minimo
```

Estamos lidando com uma quebra não apenas no preço, mas também no volume.

O parâmetro que será otimizado é o 'um', que representa o período de quebra de resistência a ser calculado. É através desse parâmetro que será definido o período para o preço e volume.

É importante considerar o intervalo de preço máximo e mínimo, pois é neste intervalo que as oportunidades de venda e compra são sinalizadas. Afinal, neste modelo, podemos entrar em uma posição vendida quando o preço cai e, em uma posição comprada quando o preço sobe.

```
def fazendo_indicadores(self):  
  
    self.quebra_resistencia_preco_maximo = MakeIndicator().quebra_resistencia_maximo(self.dados.fechamento, periodo = self.parametro1)  
    self.quebra_resistencia_preco_minimo = MakeIndicator().quebra_resistencia_minimo(self.dados.fechamento, periodo = self.parametro1)  
    self.quebra_resistencia_volume = MakeIndicator().quebra_resistencia_maximo(self.dados.volume, periodo = self.parametro1)  
  
    self.lista_indicadores = [self.quebra_resistencia_preco_maximo, self.quebra_resistencia_preco_minimo, self.quebra_resistencia_volume]
```

No método evento, existem três argumentos: o 'self', que se refere ao objeto atual; a 'data' que estamos utilizando para filtrar os dados; e o 'i', que representa a barra de execução, ou seja, indica se estamos na primeira, segunda, terceira barra e assim por diante.

O parâmetro 'i' é importante, porque em modelos de quebra de resistência, não há um sinal claro de saída. Por exemplo, quando você vende após uma quebra de resistência, não fica claro quando comprar novamente. Logo, é necessário definir um período de tempo específico para essa nova entrada, talvez baseado em uma inversão, o que envolveria estabelecer uma quantidade de dias ou barras.

Se o fechamento rompeu a resistência e não estamos no dia programado para zerar o modelo, temos a opção de selecionar esse dia específico para encerrar. Nesse caso, executamos 'self.barra\_executada = i'. Por exemplo, se executarmos na barra de número 50 e decidirmos zerar o modelo em 5 dias (parâmetro 2 = 5), então, mesmo que o rompimento da resistência ocorra na barra de número 55, não realizaremos nenhuma ação, ou seja, não faremos a compra nesse momento.

É crucial que ambas as quebras ocorram no mesmo dia: a quebra do preço e a quebra do volume. Não é suficiente apenas sair do intervalo estabelecido; é essencial que a saída ocorra com um volume consideravelmente alto para confirmar o sinal.

```
def evento(self, data, i):

    if self.dados.fechamento[data] == self.quebra_resistencia_preco_maximo[data] and (i != (self.barra_executada + self.parametro2)):
        #if (self.dados.fechamento[data] == self.quebra_resistencia_preco_maximo[data] and (i != (self.barra_executada + self.parametro2)) and
        #    self.dados.volume[data] == self.quebra_resistencia_volume[data]):

            if self.comprado:
                pass
            elif self.vendido:
                self.vender_cdi()
                self.compra(inverter=True)
                self.barra_executada = i
            else:
                self.vender_cdi()
                self.compra()
                self.barra_executada = i
```

Aplique a mesma lógica para o preço mínimo.

```
elif self.dados.fechamento[data] == self.quebra_resistencia_preco_minimo[data] and (i != (self.barra_executada + self.parametro2)):
    #elif (self.dados.fechamento[data] == self.quebra_resistencia_preco_minimo[data] and (i != (self.barra_executada + self.parametro2)) and
    #    self.dados.volume[data] == self.quebra_resistencia_volume[data]):

        if self.vendido:
            pass
        elif self.comprado:
            self.venda(inverter=True)
            self.comprar_cdi()
            self.barra_executada = i
        else:
            self.venda()
            self.comprar_cdi()
            self.barra_executada = i
```

Na função 'self.comprado' e 'self.vendido' trata da zeragem do modelo. Se o programa estiver comprado ou vendido, e o parâmetro 'i' atingir a barra executada mais o número de dias, o modelo é zerado.

```
elif self.comprado and (i == (self.barra_executada + self.parametro2)):  
    self.venda(zerar=True)  
    self.comprar_cdi()  
  
elif self.vendido and (i == (self.barra_executada + self.parametro2)):  
    self.compra(zerar=True)  
    self.comprar_cdi()
```

Informar os parâmetros, anos de otimização e anos de teste, caminho, corretagem e etc.

```
walk = WalkForwardAnalysis(estrategia = quebra_resistencia_estrategia(), class_dados = dados,  
parametro1= range(6, 20, 3), parametro2 = range(4, 33, 4),  
anos_otimizacao=3, anos_teste=1,  
nome_arquivo = rf"c:\Users\lsiqu\dev\PDFS_BACKTEST\analise_tecnica\backtest_2pra1_{acao}_RESISTENCIA_S_VOLUME.pdf",  
caminho_dados_benchmarks =r'c:\Users\lsiqu\dev\base_dados_br',  
caminho_imagens= r'c:\Users\lsiqu\dev\PDFS_BACKTEST\imagens',  
corretagem=0.00005)  
  
walk.run_walk()
```

código.py

## Relatório do Modelo

Dia inicial	2013-02-20
Dia final	2021-04-16
Dias totais	2018



### Estatísticas de Retorno e Risco

Retorno acum. modelo	192.55%
Retorno acum. ativo	56.32%
Retorno acum. CDI	96.36%
Retorno acum. IBOV	111.32%
Retorno a.a. modelo	14.34%
Vol 252d	47.62%
Índice Sharpe	0.11
VAR diário 95%	-4.79%
Drawdown máximo	-60.39%

### Estatísticas de Trade

Número de trades	147
% Operações vencedoras	40.14%
% Operações perdedoras	59.86%
Média de ganhos	11.98%
Média de perdas	-5.7%
Expec. matemática por trade	1.4%
Tempo médio de operação	12
Maior sequência de vitória	8
Maior sequência de derrotas	10

código.py

## Relatório do Modelo

Dia inicial	2013-02-20
Dia final	2021-04-16
Dias totais	2018



### Estatísticas de Retorno e Risco

Retorno acum. modelo	322.74%
Retorno acum. ativo	56.32%
Retorno acum. CDI	96.36%
Retorno acum. IBOV	111.32%
Retorno a.a. modelo	19.72%
Vol 252d	45.25%
Índice Sharpe	0.23
VAR diário 95%	-4.37%
Drawdown máximo	-73.34%

### Estatísticas de Trade

Número de trades	65
% Operações vencedoras	46.15%
% Operações perdedoras	53.85%
Média de ganhos	16.5%
Média de perdas	-7.2%
Expec. matemática por trade	3.74%
Tempo médio de operação	19
Maior sequência de vitória	3
Maior sequência de derrotas	5