Trabalho 1 de Econofísica

February 22, 2021

Edelson Luis Pinheiro Sezerotto Júnior, 288739

0.1 Introdução

O propósito deste trabalho é fazer uma série de análises estatísticas sobre um certo ativo, que foi escolhido como sendo o Bitcoin. Um dos motivos para essa escolha é que essa criptmoeda possui dados atualizados com uma grande frequência, o que permite análises com maior riqueza de detalhes; outro motivo é que o mercado de criptmoedas vem recebendo uma atenção cada vez maior, fazendo com que o Bitcoin seja uma moeda relevante para ser analisada. Para obter as séries temporais, foi usado um programa escrito em Python compartilhado pelo colega Luiz Guarinello, e que por sua vez foi adaptado da referência [1].

Em linhas gerais, os objetivos aqui são obter os retornos (linear, logarítmico e normalizado), e então calcular suas distribuições. Vamos então determinar qual distribuição estatística melhor se adequa aos dados, e também procurar pela existência de fatos estilizados. Na próxima seção são discutidos em detalhes os métodos utilizados.

0.2 Metodologia

Inicialmente, é necessário baixar as séries temporais. Para isso, foi utilizado um programa que as baixa através do API da Binance (referência [2]). Para sua utilização, são fornecidos dois argumentos: uma data de início d_0 e um período p. Com isso, ele baixa uma lista de dados coletados desde d_0 até a data d_f em que o programa é rodado, onde os dados estão separados entre si por intervalos de p minutos. As análises aqui serão feitas para três tuplas (d_0, p), listadas abaixo (Tabela 1). A coluna "periodo" mostra como p é escolhido no programa. Para todos os casos, d_f é a data 15/02/21.

d_o	p	periodo
01/11/20	60	1h
01/06/20	30	30m
01/02/21	1	1m

Para começar a escrever o programa, temos primeiro que importar algumas bibliotecas:

[145]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy
```

```
from math import *
```

Vamos definir alguns argumentos de entrada:

[187]:
```
periodo = '1m' #período da série temporal
w = 10**-5 #largura dos bins para plotar os histogramas
inc = 1000 #increase pra calcular resíduos
```

Três arquivos de dados foram previamente baixados, utilizando os valores da tabela 1. Agora vamos gerar um data frame:

[188]:
```
dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #gera um data frame a partir␣
 ↪dos dados
dfst = dfst.iloc[:, [0,4]] #pega as colunas de interesse (momento do registro␣
 ↪do dado e preço de fechamento)
dfst['timestamp'] = pd.to_datetime(dfst['timestamp']) #convertendo o formato␣
 ↪das datas

valores = dfst['close'] #gera uma lista para os preços de fechamento
N = len(valores)
datas = dfst['timestamp'].to_frame().drop(dfst.index[0]) #lista das datas
```

Deve-se notar que a lista das datas precisa ignorar o primeiro registro, afinal não é possível calcular seu retorno. Para calculá-los, usa-se o código abaixo:

[189]:
```
rlin = [(valores[i+1]-valores[i])/valores[i] for i in range(N-1)] #linear
rlog = [np.log10(valores[i+1])-np.log10(valores[i]) for i in range(N-1)]␣
 ↪#logarítmico

rmean = np.mean(rlog) #retorno médio
rstd = np.std(rlog) #desvio padrão
rnor = [(rlog[i]-rmean)/rstd for i in range(N-1)] #normalizado
```

Como pode ser notado, o retorno logarítmico foi calculado aqui utilizando base 10. Após isso podemos tomar a transformada de Fourier $F(r)$ de rlog, calcular a densidade espectral $S(r)$ através dos módulos quadrados dos valores da transformada, e obter as covariâncias $C(r)$ através da transformada inversa de $S(r)$, conforme o teorema de Wiener-Khinchin. Normalizando os valores $C(r)$ ao dividí-los por $C(0)$, obtemos as correlações entre os retornos. Fazemos o mesmo para as volatilidades, que nada mais são que os quadrados de rlog. As correlações de rlog e das volatilidades são obtidas com o código abaixo:

[190]:
```
fr = scipy.fft.fft(rlog) #transformada de fourier
sr = [fr[i].real**2+fr[i].imag**2 for i in range(len(fr))] #densidade espectral
Cr = scipy.fft.ifft(sr).real #covariâncias dos retornos
Crr = Crr = [Cr[i]/Cr[0] for i in range(len(Cr))]
Cr = Crr.copy() #correlações dos retornos

vol = [rlog[i]**2 for i in range(len(rlog))] #volatilidade
```

```
fv = scipy.fft.fft(vol) #transformada de fourier
sv = [fv[i].real**2+fv[i].imag**2 for i in range(len(fv))] #densidade espectral
Cv = scipy.fft.ifft(sv) #correlacao das volatilidades
Cvv = [Cv[i]/Cv[0] for i in range(len(Cv))]
Cv = Cvv.copy()

datasc = datas.copy() #datas para as correlacoes
datasc['Cr'] = Cr
datasc['Cv'] = Cv
datasc = datasc.head(int(len(datasc)/2))
```

0.3 Resultados e conclusões

0.3.1 Distribuições dos retornos

Antes de visualizarmos os retornos, é conveniente mostrar a série temporal, afinal é ela quem dita como eles se comportarão. Ela pode ser plotada com as linhas abaixo:

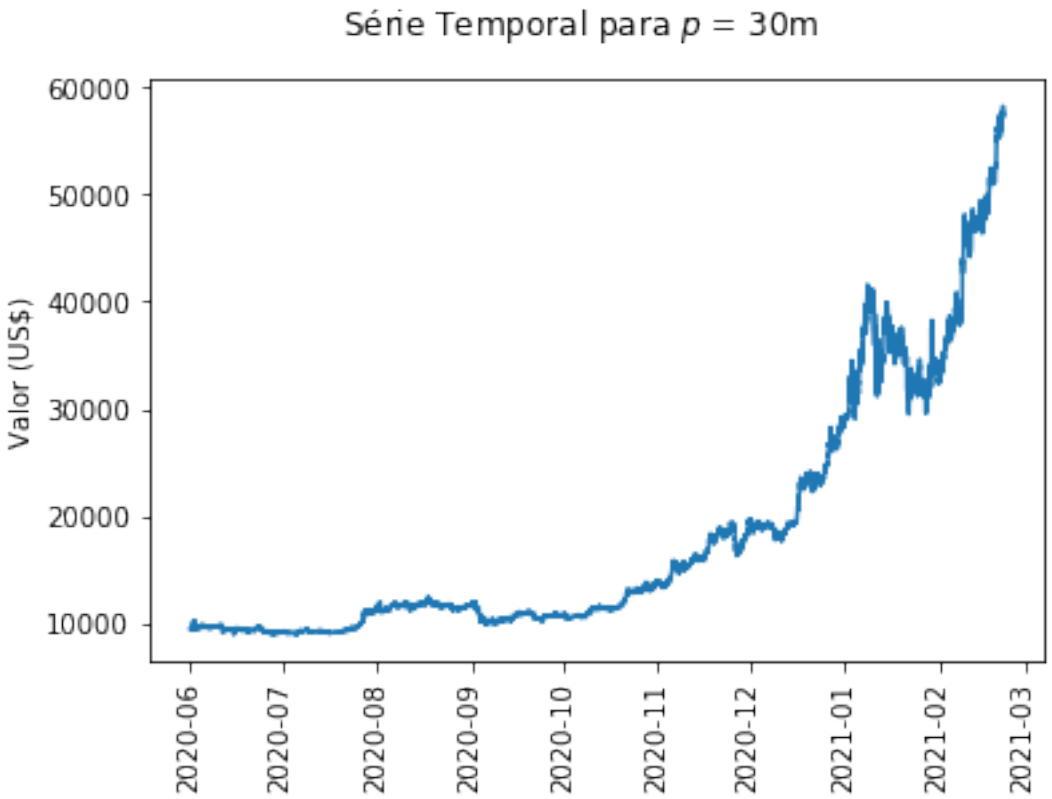
[185]:
```
plt.suptitle(f'Série Temporal para $p$ = {periodo}')
plt.ylabel('Valor (US$)')
plt.xticks(rotation=90)
plt.plot(dfst['timestamp'],valores)
```

[185]:
```
[<matplotlib.lines.Line2D at 0x7f90db6be250>]
```

Alterando a variável "periodo" para '30m' e '1m', respectivamente, obtemos os gráficos para $p = 30$ e $p = 1$ tendo os dados já devidamente baixados:

[144]:
```
plt.suptitle(f'Série Temporal para $p$ = {periodo}')
plt.ylabel('Valor (US$)')
plt.xticks(rotation=90)
plt.plot(dfst['timestamp'],valores)
```

[144]:
```
[<matplotlib.lines.Line2D at 0x7f90abe924c0>]
```



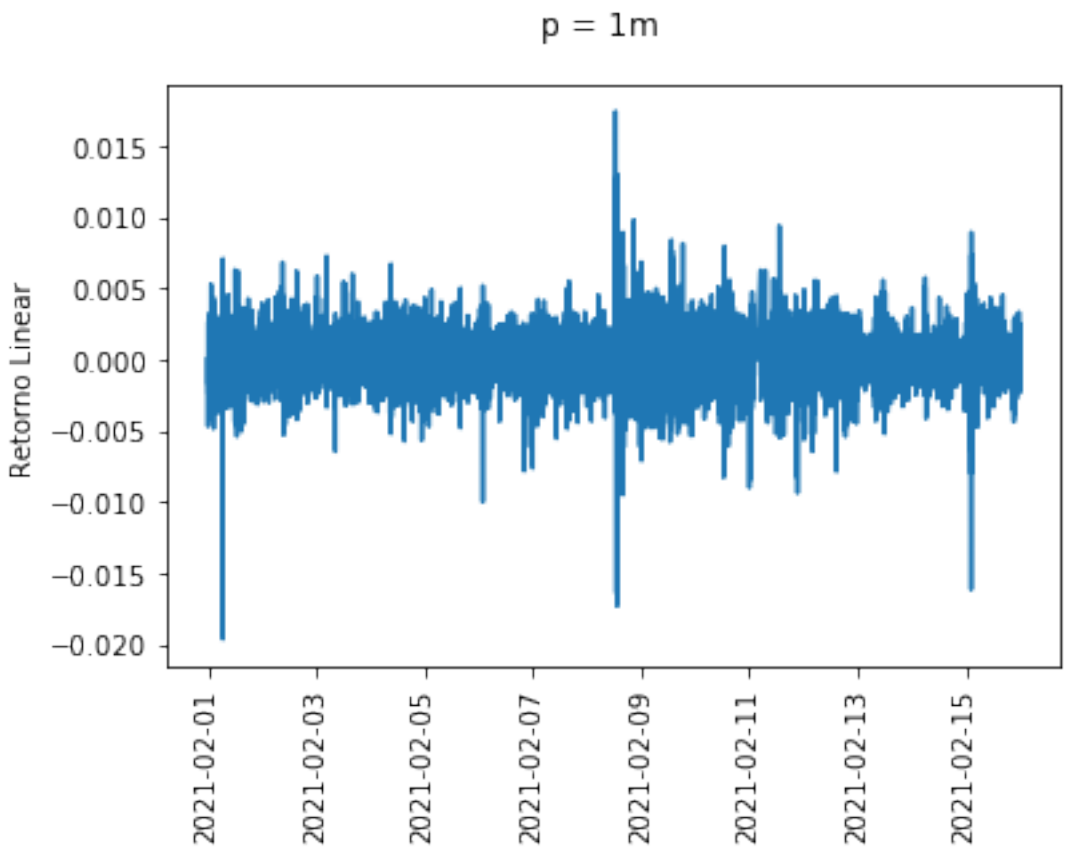
[83]:
```
plt.suptitle(f'Série Temporal para $p$ = {periodo}')
plt.ylabel('Valor (US$)')
plt.xticks(rotation=90)
plt.plot(dfst['timestamp'],valores)
```

[83]: [<matplotlib.lines.Line2D at 0x7f90db1e4d00>]

Nessa seção serão mostrados apenas os retornos para $p = 1$. Os outros valores de p serão usados na análise dos fatos estilizados. Abaixo temos os plots dos retornos:

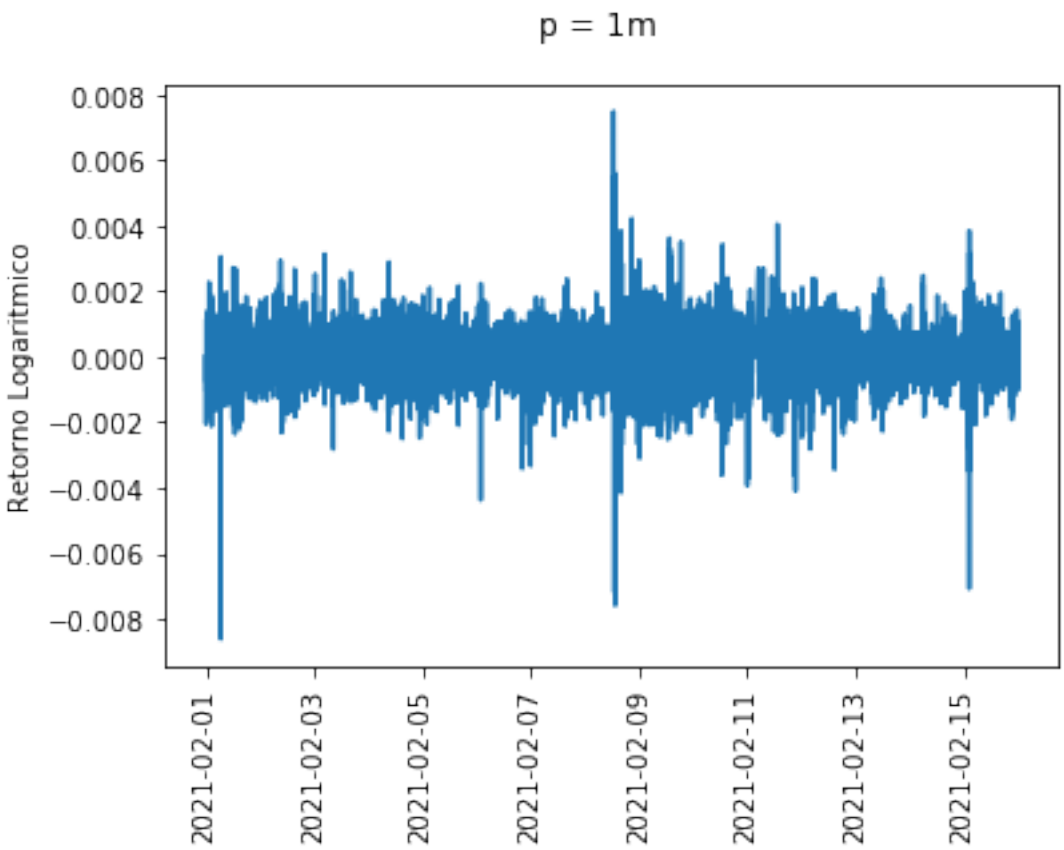
[161]:
```
plt.suptitle(f'p = {periodo}')
plt.ylabel('Retorno Linear')
plt.xticks(rotation=90)
plt.plot(datas,rlin)
```

[161]:
```
[<matplotlib.lines.Line2D at 0x7f90abf73970>]
```



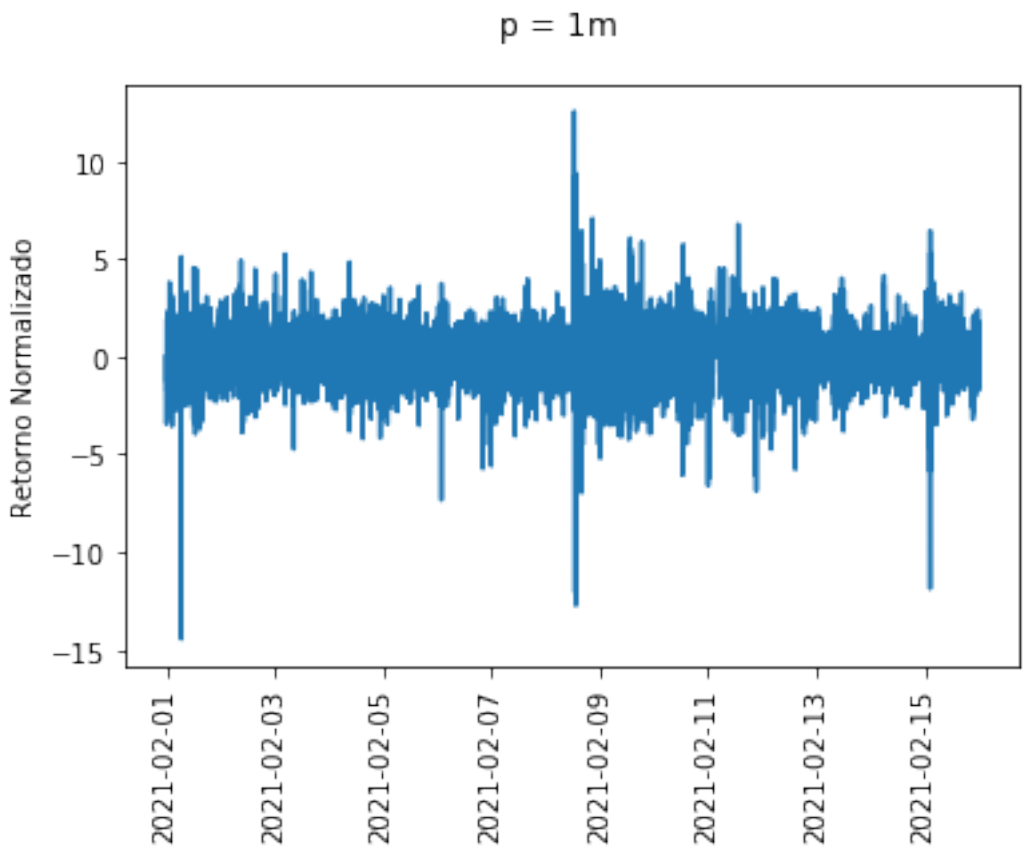
[163]:
```
plt.suptitle(f'p = {periodo}')
plt.ylabel('Retorno Logaritmico')
plt.xticks(rotation=90)
plt.plot(datas,rlog)
```

[163]: [<matplotlib.lines.Line2D at 0x7f90ab7c1b50>]



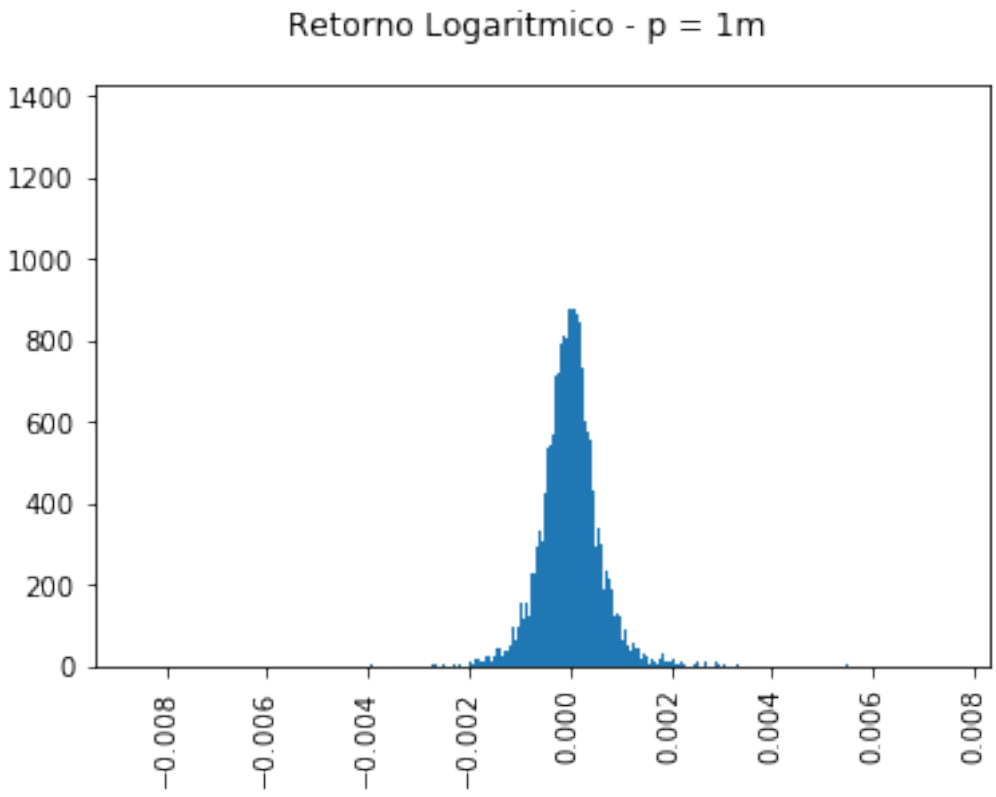
[164]:
```
plt.suptitle(f'p = {periodo}')
plt.ylabel('Retorno Normalizado')
plt.xticks(rotation=90)
plt.plot(datas,rnor)
```

[164]: [<matplotlib.lines.Line2D at 0x7f90ab763d90>]



Como era de se esperar devido às suas definições, todos têm a mesma aparência - o que muda são as escalas do eixo y. Apesar de os gráficos acima nos informarem os retornos, eles tornam difícil visualizar o quão distribuídos eles são. Isso é importante porque um investidor estará interessado em saber quão frequentemente acontece de um retorno ser positivo, ou de ele assumir um certo valor, por exemplo. Isso pode ser melhor visualizado utilizando um histograma. Vamos lembrar que um dos argumentos de entrada é w, a largura dos bins - os histogramas serão mostrados definindo w ao invés da quantidade de bins para ficar mais fácil de comparar os resultados (nessa análise o retorno utilizado será o logarítmico, mas poderia ser um dos outros).

[191]:
```
intervalos = []
c = min(rlog)
while c <= max(rlog):
    intervalos.append(c)
    c += w
plt.suptitle(f'Retorno Logaritmico - p = {periodo}')
plt.xticks(rotation=90)
dados = plt.hist(rlog,bins=intervalos,density=True)
max_gauss = 1/(rstd*(2*np.pi)**0.5)
```



0.3.2 Fit de distribuições estatísticas

Vamos agora fazer alguns fits nessa curva usando três distribuições estatísticas: gaussiana, Pareto e exponencial. Estamos interessados em modelar a cauda da distribuição dos retornos e analisaremos qual dessas três faz isso melhor descobrindo qual delas possui o menor resíduo r em relação às caudas, utlizando a equação abaixo:

$$r = \sqrt{\frac{\sum_{n=1}^{N}[f(x_n) - g(x_n)]^2}{N}}$$

onde N é a quantidade de retornos considerados; $\{x_n\}$ são os valores que definem os começos dos intervalos dos bins do histograma de densidades de probabilidade dos retornos logarítmicos; $f(x_n)$ é o valor do histograma para um dado x_n; $g(x_n)$ é o valor da função gaussiana avaliada para um dado x_n. Colocando em palavras, r nada mais é do que a média quadrática das diferenças dos valores da gaussiana em relação aos valores obtidos com os dados.

Nota-se aqui que estamos interessados em analisar os resíduos apenas para as caudas, e portanto devemos definir onde elas começam. Na literatura não há uma convenção para isso, e nesse trabalho consideraremos que elas começam a partir da metade do intervalo positivo de bins (os bins referentes a retornos negativos não serão considerados para o cálculo devido à simetria das distribuições).

3.2.1 Distribuição gaussiana A distribuição gaussiana é dada pela fórmula:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

onde μ é a média e σ o desvio padrão, os quais acima foram calculados através de, respectivamente, rmean e rstd. Para saber quais os limites entre os quais a gaussiana deve ser calculada, rodamos o trecho abaixo:

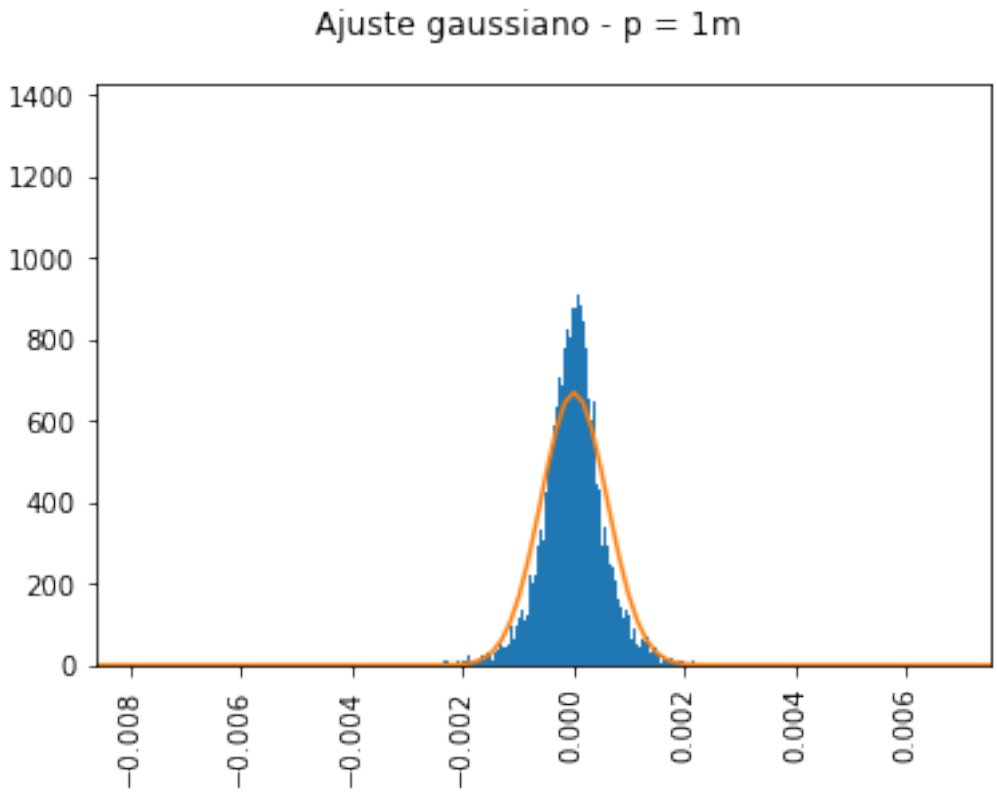
[90]:
```
intervalos = []
c = min(rlog)
while c <= max(rlog):
    intervalos.append(c)
    c += w
dados = plt.hist(rlog,bins=intervalos,density=True)
plt.clf()
```

```
<Figure size 432x288 with 0 Axes>
```

O valor mínimo do intervalo fica armazenado em dados[1][0] e o máximo em dados[1][-1]. Agora podemos gerar a gaussiana, que é mostrada abaixo junto com o histograma (o trecho abaixo já calcula também o resíduo). Deve-se reparar que para fazer a comparação, o histograma deve mostrar as densidades de probabilidade, e não números de contagens, e isso é feito usando o argumento density=True. Esse argumento faz com que cada valor de contagem seja dividido pela soma de todas as contagens (transformando as contagens em probabilidades), e esse valor é dividido por w (transformando probabilidades em densidades de probabilidade).

[192]:
```
max_gauss = 1/(rstd*(2*np.pi)**0.5)
x = np.linspace(dados[1][0], dados[1][-1], 100) #range da gaussiana
y = [max_gauss*np.e**(-(x[i]-rmean)**2/(2*rstd**2)) for i in range(len(x))]
plt.suptitle(f'Ajuste gaussiano - p = {periodo}')
plt.xticks(rotation=90)
plt.xlim([dados[1][0], dados[1][-1]])
plt.hist(rlog,bins=intervalos,density=True)
plt.plot(x,y)

lx = list(dados[1])
lx.pop()
ly = list(dados[0])
lx_copy = []
ly_copy = []
for i in range(len(lx)): #vou selecionar so valores positivos de lx
    if lx[i] > 0:
        lx_copy.append(lx[i])
        ly_copy.append(ly[i])
lx = lx_copy.copy()
ly = ly_copy.copy()
```

```
lx = lx[int(len(lx)/2):] #seleciona a ultima metade de lx
ly = ly[int(len(ly)/2):] #seleciona a ultima metade de ly

le = [max_gauss*np.e**(-(lx[i]-rmean)**2/(2*rstd**2)) for i in range(len(lx))]␣
↪#valores estimados pela distribuicao
residuo = 0
for i in range(len(le)): residuo += (ly[i]-le[i])**2
residuo /= len(le)
residuo = residuo**0.5
residuo_gaussiana = residuo
```



3.2.2 Distribuição de Pareto Essa é definida pela equação:

$$P(x) = \frac{\alpha {x_0}^2}{x^{\alpha+1}}$$

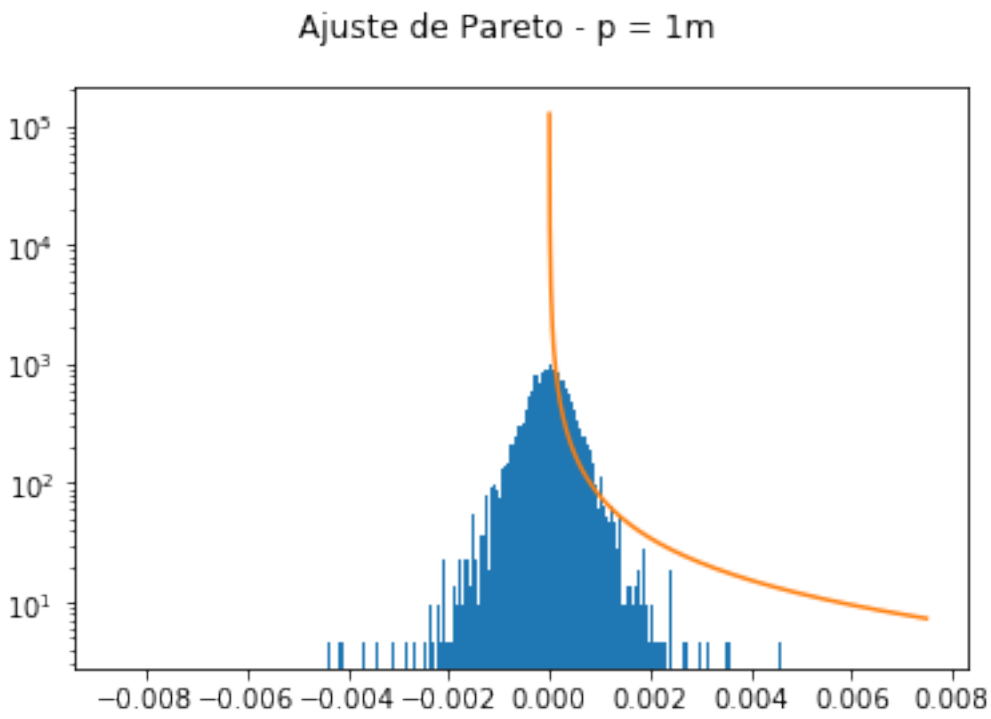
onde x_0 foi escolhido empiricamente e α é determinado pelo Estimador de Hill:

$$\alpha = \frac{N}{\sum_{n=1}^{N} ln(\frac{x_n}{x_0})}$$

onde N é a quantidade de retornos positivos.

Implementamos $P(x)$ com o código abaixo. Para comparar com o histograma, o eixo y foi plotado em escala logarítmica, porque os valores iniciais de $P(x)$ são muito grandes. Isso no entanto não é um problema porque o que estamos realmente interessados é na cauda da distribuição.

[193]:
```
xo = 10**-5
x = np.linspace(dados[1][0], dados[1][-1], 100)
vxo = list(dados[1]) #lista contendo todos os valores de x
vxo.pop()
vx = [] #lista contendo apenas os x positivos
for i in vxo:
    if i > 0: vx.append(i)
n = len(vx) #qtd de bins
alpha = [log(vx[i]/xo) for i in range(n)]
alpha = n/sum(alpha)
x = vx
y = [alpha*xo**alpha/(vx[i]**(alpha+1)) for i in range(n)]
plt.suptitle(f'Ajuste de Pareto - p = {periodo}')
plt.yscale('log')
plt.hist(rlog,bins=intervalos,density=True)
plt.plot(x,y)

lx = list(dados[1])
lx.pop()
ly = list(dados[0])
lx_copy = []
ly_copy = []
for i in range(len(lx)): #vou selecionar so valores positivos de lx
    if lx[i] > 0:
        lx_copy.append(lx[i])
        ly_copy.append(ly[i])
lx = lx_copy.copy()
ly = ly_copy.copy()
lx = lx[int(len(lx)/2):] #seleciona a ultima metade de lx
ly = ly[int(len(ly)/2):] #seleciona a ultima metade de ly

le = [alpha*xo**alpha/(lx[i]**(alpha+1)) for i in range(len(lx))] #valores␣
 ↪estimados pela distribuicao
residuo = 0
for i in range(len(le)): residuo += (ly[i]-le[i])**2
residuo /= len(le)
residuo = residuo**0.5
residuo_pareto = residuo
```



3.2.3 Distribuição exponencial A distribuição exponencial é dada pela seguinte equação:

$$E(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ \lambda e^{\lambda x} & x < 0 \end{cases}$$

onde λ pode ser determinado pelo método de máxima verossimilhança. Para isso, primeiro definimos uma função $L(x)$ como sendo o produtório dos valores de $E(x)$ para um conjunto de valores $\{x_n\}$:

$$L(\lambda) = \prod_{n=1}^{N} E(x_n) = \lambda^N \prod_{n=1}^{N} e^{-\lambda x_n}$$

Tomando o logaritmo natural de ambos os lados:

$$ln[L(\lambda)] = Nln(\lambda) - \lambda \sum_{n=1}^{N} x_n$$

Derivando em relação a λ e igualando a 0 para obter o máximo da função:

$$\frac{d[ln(L)]}{d\lambda} = \frac{N}{\lambda} - \sum_{n=1}^{N} x_n = 0$$

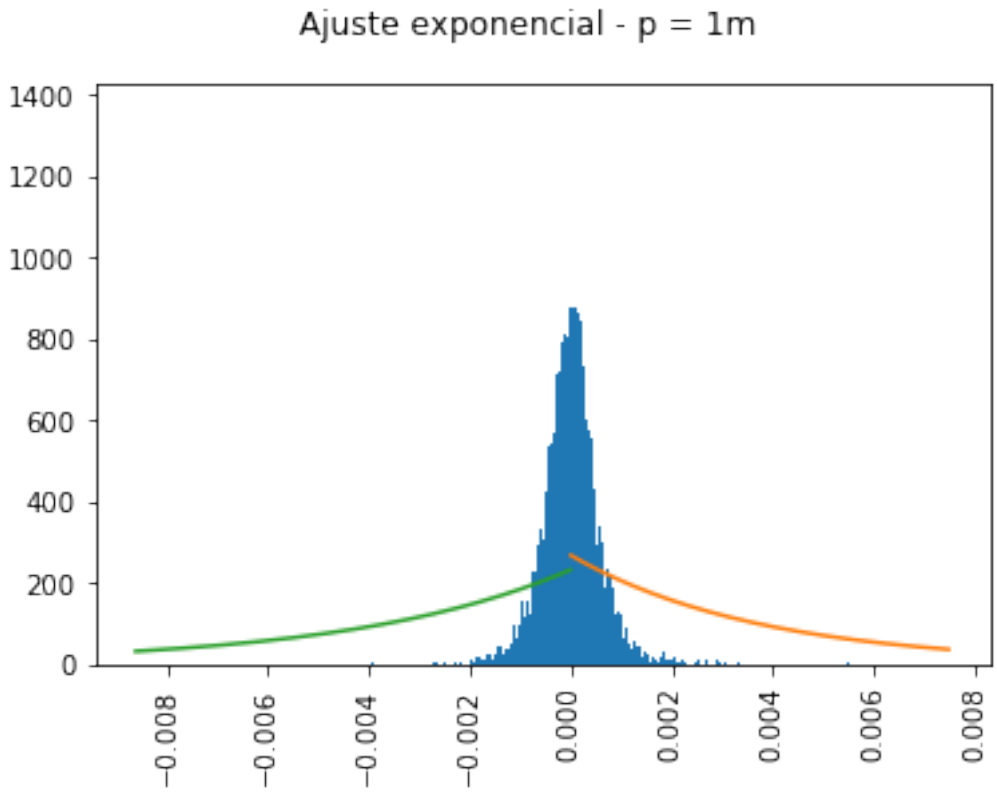
Concluimos que

$$\lambda = \frac{N}{\sum_{n=1}^{N} x_n}$$

Deve-se notar que para o cálculo acima foi suposto o caso $x \geq 0$. Caso contrário, o resultado ainda é válido se considerarmos os módulos de x e usarmos a forma de $E(x)$ para $x < 0$. Assim, a distribuição exponencial deve ser definida por partes (uma para os valores positivos e outra para os negativos). Implementamos isso com o código abaixo:

[194]:
```
x = np.linspace(dados[1][0], dados[1][-1], 100)
vxo = list(dados[1]) #lista contendo todos os valores de x
vxo.pop()
vx = [] #lista contendo apenas os x positivos
for i in vxo:
    if i >= 0: vx.append(i)
n = len(vx) #qtd de bins
l = n/sum(vx)
xp = vx
yp = [l*e**(-l*xp[i]) for i in range(n)]

vxo = list(dados[1]) #lista contendo todos os valores de x
vxo.pop()
vx = [] #lista contendo apenas os x negativos
for i in vxo:
    if i < 0: vx.append(i)
n = len(vx) #qtd de bins
l = abs(n/sum(vx))
xn = vx
yn = [l*e**(l*xn[i]) for i in range(n)]
plt.suptitle(f'Ajuste exponencial - p = {periodo}')
plt.hist(rlog,bins=intervalos,density=True)
plt.xticks(rotation=90)
plt.plot(xp,yp)
plt.plot(xn,yn)

lx = list(dados[1])
lx.pop()
ly = list(dados[0])
lx_copy = []
ly_copy = []
for i in range(len(lx)): #vou selecionar so valores positivos de lx
    if lx[i] > 0:
        lx_copy.append(lx[i])
        ly_copy.append(ly[i])
lx = lx_copy.copy()
ly = ly_copy.copy()
```

```
lx = lx[int(len(lx)/2):] #seleciona a ultima metade de lx
ly = ly[int(len(ly)/2):] #seleciona a ultima metade de ly

le = [l*e**(-l*lx[i]) for i in range(len(lx))] #lista de valores exponenciais
residuo = 0
for i in range(len(le)): residuo += (ly[i]-le[i])**2
residuo /= len(le)
residuo = residuo**0.5
residuo_exponencial = residuo
```



Por inspeção visual, podemos ver que a distribuição gaussiana é a que melhor modela as caudas dos dados analisados. Isso pode ser confirmado analisando os valores dos resíduos:

[195]:
```
print(f'Gaussiana: {residuo_gaussiana}')
print(f'Pareto: {residuo_pareto}')
print(f'Exponencial: {residuo_exponencial}')
```

```
Gaussiana: 0.831174722943922
Pareto: 10.983468369787992
Exponencial: 66.9636226831943
```

Com os dados acima verificamos que a gaussiana fornece de longe o melhor ajuste, seguido da distribuição de Pareto e por fim da distribuição exponencial (que nesse caso possui um resíduo tão

elevado que acaba não sendo útil para modelar os retornos).

0.3.3 Fatos estilizados

Fatos estilizados são fatos que costumam ocorrer de forma recorrente. Nesse trabalho vamos analisar dois deles: gaussianidade acumulada e clusters de volatilidade.

3.3.1 Gaussianidade acumulada Gaussianidade acumulada se refere ao fenômeno pelo qual, conforme vamos pegando cada vez mais dados, os histogramas das densidades de probabilidade dos retornos vão ficando cada vez mais ajustados a uma curva gaussiana. Para verificar se isso de fato ocorre, precisamos de alguma métrica que nos diga o quão bem ajustado um conjunto de dados está em relação a alguma distribuição. Conforme já definido anteriormente, fazemos isso calculando o resíduo desses dados.

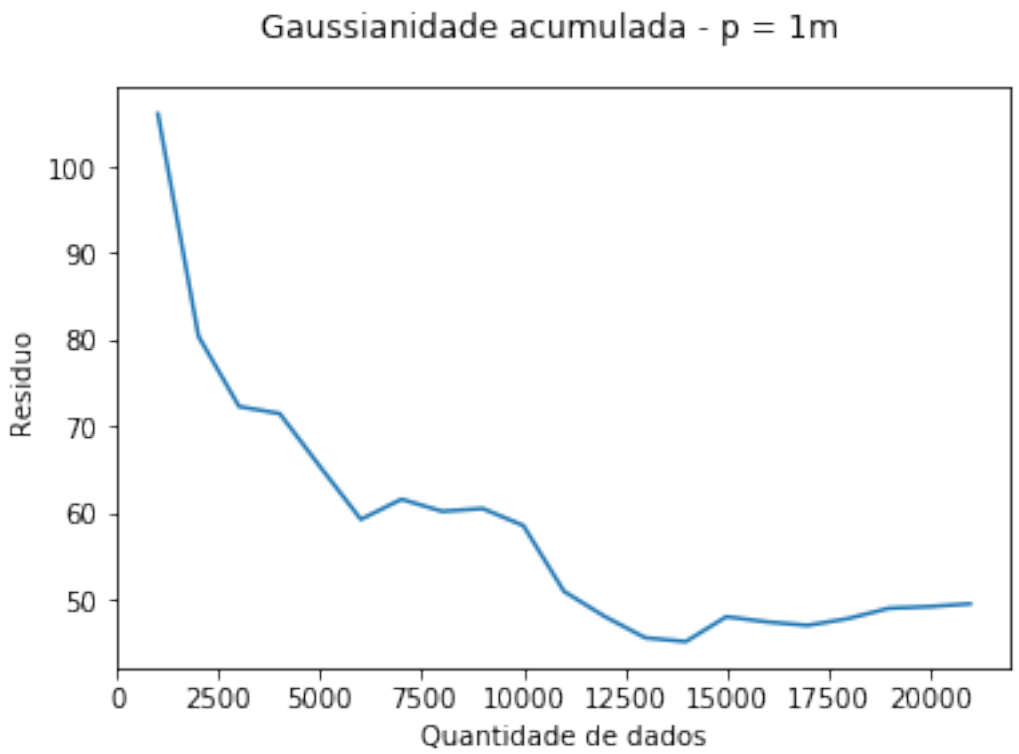
Devemos calcular uma série de valores r para ver como os resíduos evoluem ao longo do tempo. O que se espera é que eles vão diminuindo (idealmente, tendendo a zero), pois quanto menor o resíduo, melhor é o ajuste. Para isso, cria-se uma função que se utiliza de boa parte do código que já foi apresentado até aqui. Para ela se passa um argumento h (head) que faz com que ela pegue apenas os h primeiros dados da base e retorne o resíduo. Essa função é chamada por um loop e a cada nova iteração h é acrescentado em i unidades, onde i é definido pela variável "inc". Vamos então obter a curva dos resíduos para $p = 1$:

[170]:
```
def calcular_residuos(head):
    ######################## PRE-PROCESSAMENTO DOS DADOS␣
↪##########################
    dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #data frame da serie␣
↪temporal
    dfst = dfst.iloc[:, [0,4]] #pegando apenas as colunas de interesse
    dfst['timestamp'] = pd.to_datetime(dfst['timestamp']) #convertendo o␣
↪formato das datas
    dfst = dfst.head(head)

    valores = dfst['close']
    N = len(valores)
    datas = dfst['timestamp'].to_frame().drop(dfst.index[0])
  ␣
↪#################################################################################

    ##################### CALCULANDO OS RETORNOS␣
↪#################################
    rlin = [(valores[i+1]-valores[i])/valores[i] for i in range(N-1)] #linear
    rlog = [np.log10(valores[i+1])-np.log10(valores[i]) for i in range(N-1)]␣
↪#logaritmico

    rmean = np.mean(rlog)
    rstd = np.std(rlog)
    rnor = [(rlog[i]-rmean)/rstd for i in range(N-1)] #normalizado
```

```
  ␣
↪#############################################################################

    ######################## VERIFICANDO CORRELACOES #################
    fr = scipy.fft(rlog) #transformada de fourier
    sr = [fr[i].real**2+fr[i].imag**2 for i in range(len(fr))] #densidade␣
↪espectral
    Cr = scipy.fft.ifft(sr).real #correlacao


    vol = [rlog[i]**2 for i in range(len(rlog))] #volatilidade
    fv = scipy.fft(vol) #transformada de fourier
    sv = [fv[i].real**2+fv[i].imag**2 for i in range(len(fv))] #densidade␣
↪espectral
    Cv = scipy.fft.ifft(sv).real #correlacao
  ␣
↪#############################################################################

    ### Histograma do retorno logaritmico com bin width definido
    intervalos = []
    c = min(rlog)
    while c <= max(rlog):
        intervalos.append(c)
        c += w
    plt.suptitle('Retorno Logaritmico')
    plt.xticks(rotation=90)
    dados = plt.hist(rlog,bins=intervalos,density=True)
    plt.clf()

    #calculandoo o residuo
    vx = list(dados[1])
    vx.pop()
    vg = [max_gauss*np.e**(-(vx[i]-rmean)**2/(2*rstd**2)) for i in␣
↪range(len(vx))]
    residuo = 0
    for i in range(len(vg)): residuo += (dados[0][i]-vg[i])**2
    residuo /= len(vg)
    residuo = residuo**0.5

    return residuo

periodo = '1m'
inc = 1000

dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #data frame da serie temporal
qtd_dados = []
residuos = []
```

```
c = inc
while c <= len(dfst):
    residuo = calcular_residuos(c)
    residuos.append(residuo)
    qtd_dados.append(c)
    c += inc

plt.suptitle(f'Gaussianidade acumulada - p = {periodo}')
plt.xlabel('Quantidade de dados')
plt.ylabel('Residuo')
plt.plot(qtd_dados,residuos)
```

[170]: [<matplotlib.lines.Line2D at 0x7f90aa3b4310>]

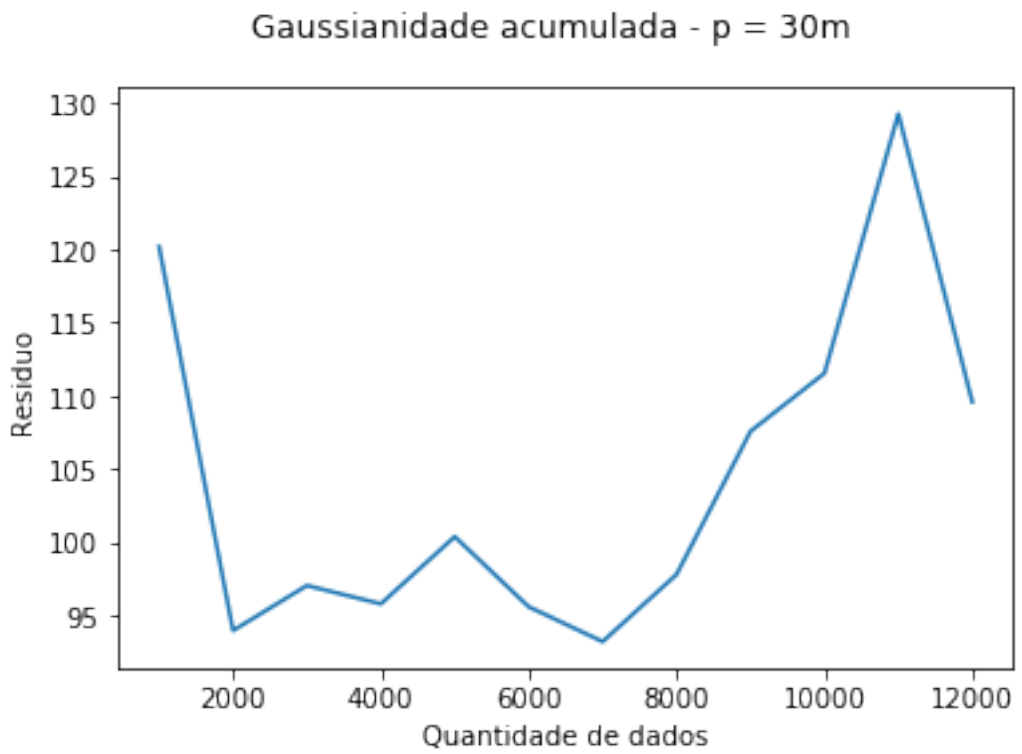


Como pode ser visto, a curva diminui conforme pegamos mais dados, confirmando o fenômeno da gaussianidade acumulada. Agora vejamos o comportamento da curva para $p = 30$:

[173]:
```
periodo = '30m'
inc = 1000

dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #data frame da serie temporal
qtd_dados = []
residuos = []
```

```
c = inc
while c <= len(dfst):
    residuo = calcular_residuos(c)
    residuos.append(residuo)
    qtd_dados.append(c)
    c += inc

plt.suptitle(f'Gaussianidade acumulada - p = {periodo}')
plt.xlabel('Quantidade de dados')
plt.ylabel('Residuo')
plt.plot(qtd_dados,residuos)
```

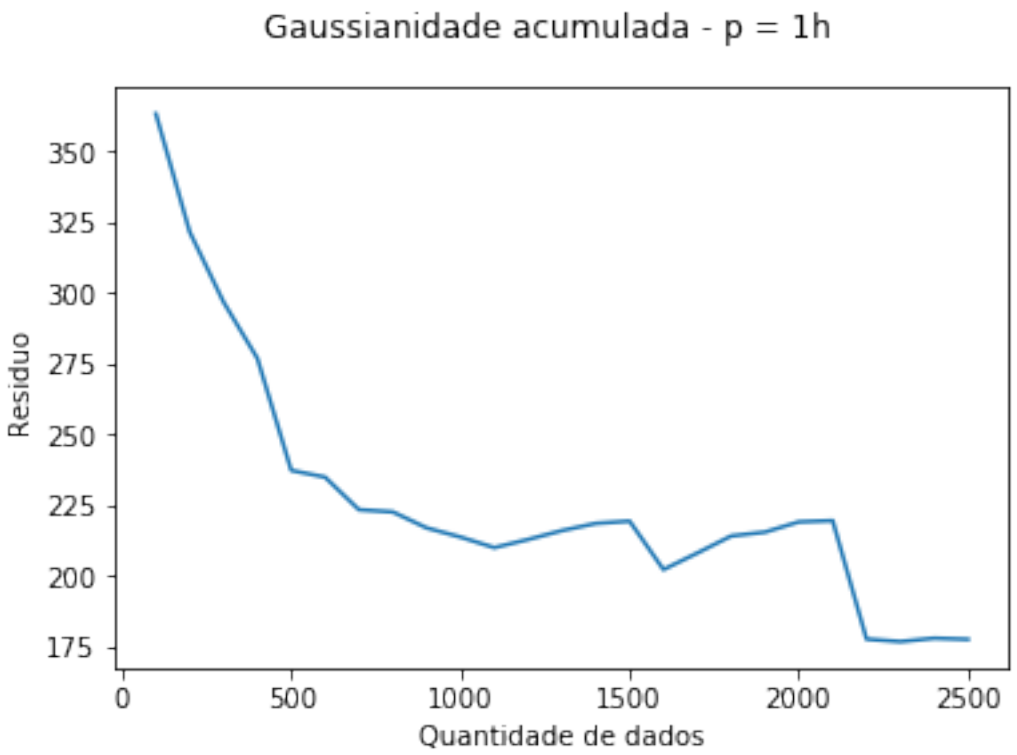
[173]: [<matplotlib.lines.Line2D at 0x7f909bc9a220>]



Nesse caso o fenômeno ocorre até certo ponto, mas a partir de cerca de 7000 dados o resíduo volta a crescer. Isso talvez aconteça porque a densidade de dados disponíveis por intervalo de tempo em relação à figura anterior é bem menor (30 vezes menor, nesse caso), e devemos lembrar que nossas medidas estatísticas são sempre mais precisas conforme maior a quantidade de dados. Ainda assim, após cerca de 11000 dados o resíduo começa a decrescer rapidamente, mostrando uma tendência de se aproximar de uma gaussiana novamente. Por último temos o gráfico para $p = 60$ (nesse caso i será reduzido para 100 porque temos menos dados):

[186]:
```
periodo = '1h'
inc = 100

dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #data frame da serie temporal
qtd_dados = []
residuos = []
c = inc
while c <= len(dfst):
    residuo = calcular_residuos(c)
    residuos.append(residuo)
    qtd_dados.append(c)
    c += inc

plt.suptitle(f'Gaussianidade acumulada - p = {periodo}')
plt.xlabel('Quantidade de dados')
plt.ylabel('Residuo')
plt.plot(qtd_dados,residuos)
```

[186]: [<matplotlib.lines.Line2D at 0x7f90a0a92be0>]

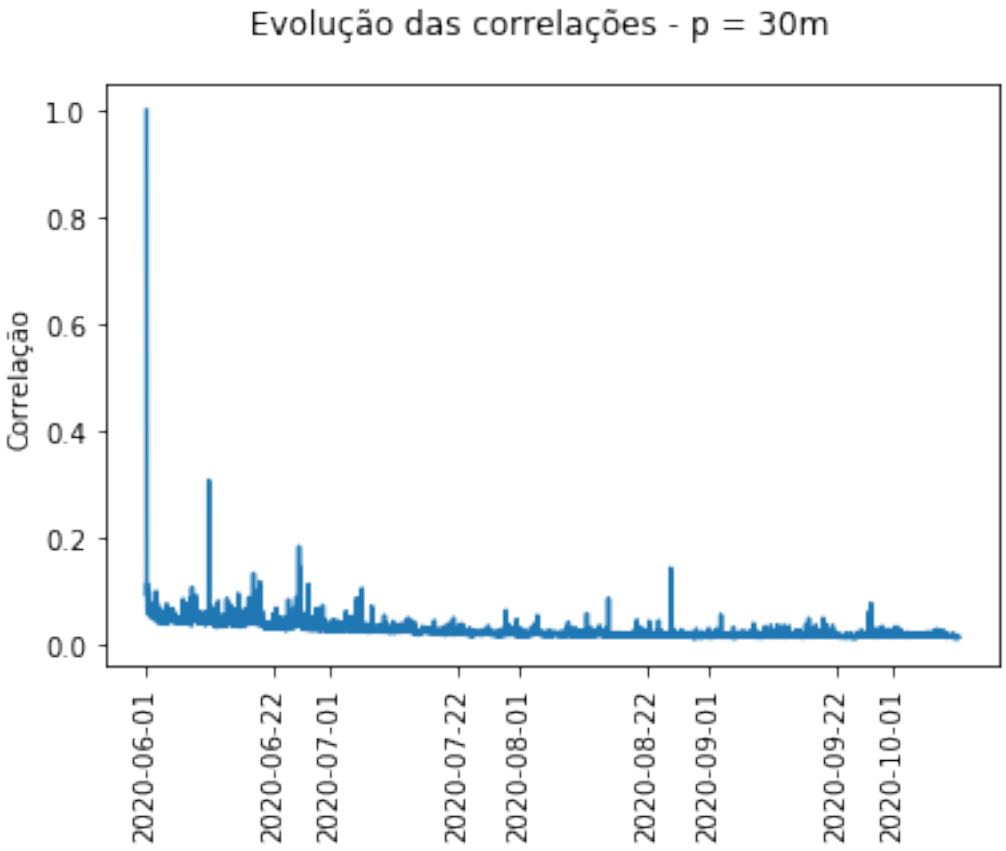


E de novo o fenômeno é observado. Com essas análises confirmamos que esse primeiro fato estilizado ocorre para o Bitcoin. Em seguida falaremos sobre clusters de volatilidade.

3.3.2 Clusters de volatilidade Volatilidade é definida como sendo o valor quadrado do retorno (que aqui novamente será considerado como sendo o logarítmico, mas poderia ser de outro tipo). A volatilidade é uma medida de inércia do mercado: quando o mercado está aquecido, com muitas pessoas comprando e vendendo, a situação tende a continuar assim, e o mesmo acontece para períodos de pouca movimentação. Dizer que há um cluster de volatilidade significa que, durante um certo período de tempo, a volatilidade mantém valores parecidos e portanto possuem uma alta correlação entre si. Isso é diferente do que se espera para os retornos, que não mantém relação uns com os outros (do contrário, seria possível prevê-los e todo mundo poderia lucrar ao mesmo tempo). Para perceber a formação de clusters, é necessário considerar intervalos de tempo longos, e por isso, para vê-los, vamos usar os dados para $p = 30$. As correlações das volatilidades são obtidas com um código contido na seção anterior e podem ser plotadas com os comandos abaixo (devido ao fato de que as correlações são simétricas em relação ao valor central, podemos plotar apenas metade do intervalo de tempo considerado):

[155]:
```
plt.xticks(rotation=90)
plt.suptitle(f'Evolução das correlações - p = {periodo}')
plt.ylabel('Correlação')
plt.plot(datasc['timestamp'],datasc['Cv'])
```

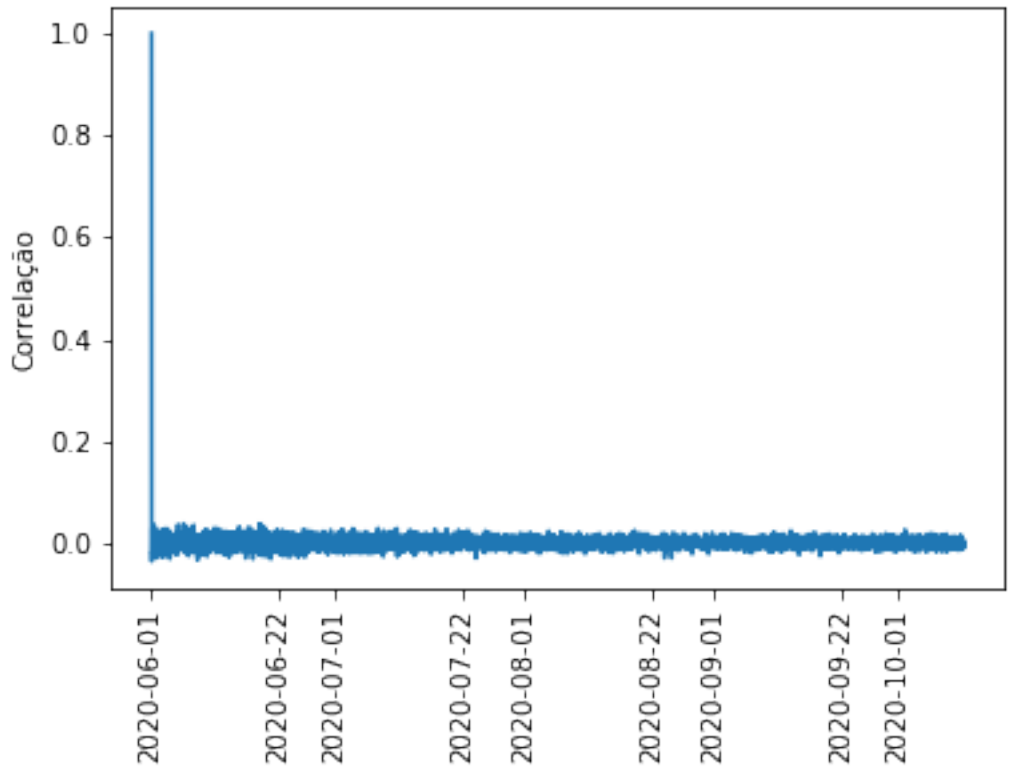
[155]: [<matplotlib.lines.Line2D at 0x7f90db3ef340>]



Como pode ser visto, há períodos em que a correlação das volatilidades aumenta, dando origem a regiões de maior estagnação nos retornos do mercado. Abaixo fazemos o plot para as correlações dos retornos:

[156]:
```
plt.xticks(rotation=90)
plt.ylabel('Correlação')
plt.plot(datasc['timestamp'],datasc['Cr'])
```

[156]: [<matplotlib.lines.Line2D at 0x7f90abf63730>]



Vemos pela imagem acima que a correlação dos retornos fica oscilando em torno de valores próximos de 0, indicando que os retornos não estão relacionados entre si. Com isso concluímos a verificação dos fatos estilizados.

0.4 Referências

[1] https://medium.com/swlh/retrieving-full-historical-data-for-every-cryptocurrency-on-binance-bitmex-using-the-python-apis-27b47fd8137f (acesso em 15/02/21)

[2] https://www.binance.com/en (acesso em 15/02/21)