Trabalho 2 de Econofísica

March 24, 2021

Edelson Luis Pinheiro Sezerotto Júnior, 288739

0.1 Introdução

Neste trabalho serão utilizados alguns modelos estatísticos para simular e analisar o comportamento de uma série temporal de retornos associada a Bitcoins. Para obter a série temporal, foi usado um programa adaptado da referência [1].

Serão usados três modelos: ARMA, GARCH e cadeias de Markov, onde com os dois primeiros serão feitas simulações de séries temporais para ver o quão bem elas modelam os dados; e o modelo das cadeias de Markov será usado para extrair algumas informações a partir dos dados.

Inicialmente, é necessário baixar a série temporal. Para isso, foi utilizado um programa que as baixa através do API da Binance. 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 em que o programa é rodado, onde os dados estão separados entre si por intervalos de p minutos. Neste trabalho, a data inicial é 01/01/21, o período é de 5 minutos e a data final é 23/03/21.

Primeiramente, vamos importar algumas bibliotecas e definir alguns valores de entrada:

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import *
from datetime import datetime
import scipy
from arch import arch_model
import random as r

#valores de entrada
periodo = '5m' #periodo da serie temporal
m0, mf = 1, 3 #meses pelos quais os dados se estendem
ni = 100 #nro de iteracoes para a cadeia de markov
```

Agora geramos um DataFrame a partir dos dados (já baixados) e fazemos algum pré-processamento:

[3]:
```
dfst = pd.read_csv('BTCUSDT-'+periodo+'-data.csv') #data frame da serie temporal
dfst['timestamp'] = pd.to_datetime(dfst['timestamp']) #convertendo o formato␣
 ↪das datas
```

```
valores = dfst['close']
N = len(valores)
datas = dfst['timestamp'].to_frame().drop(dfst.index[0])
datas = datas.reset_index(drop=True) #reinicia os indices
```

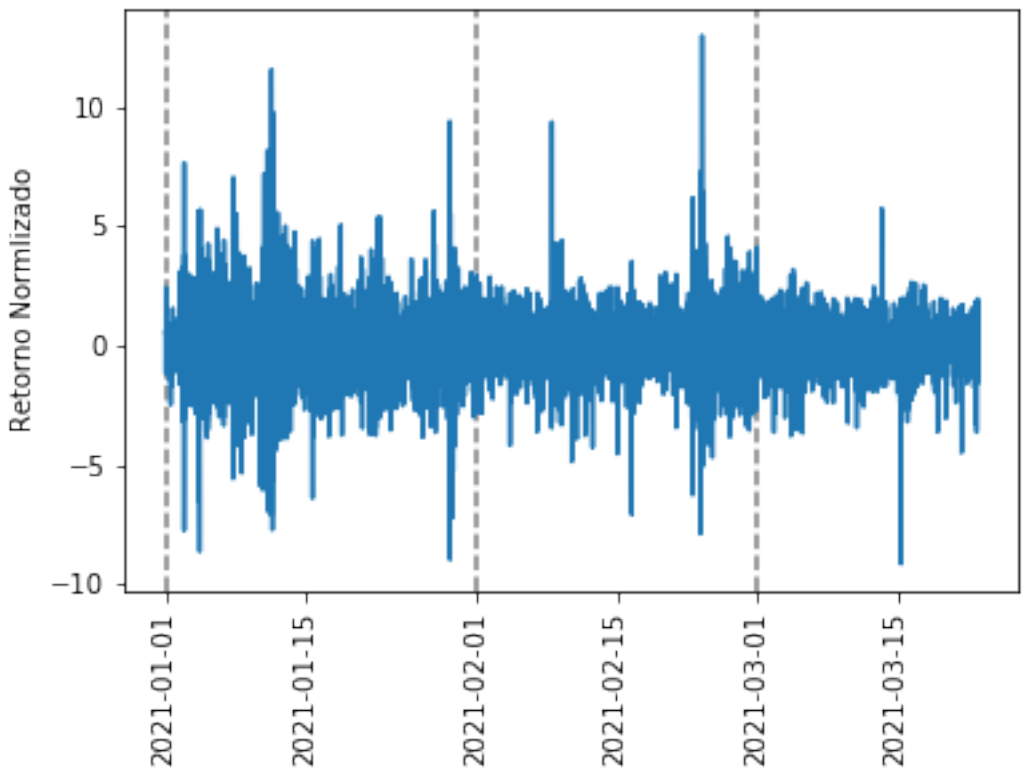
Tendo os dados já prontos, podemos obter os retornos com o trecho abaixo (aqui serão analisados os retornos normalizados):

[4]:
```
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
```

E agora podemos visualizar a série temporal dos retornos:

[5]:
```
for month in range(m0,mf+1): plt.axvline(datetime(2021,month,1),
 ↪linestyle='--', color='k', alpha=0.5) #datetime requer formato aaaa/mm/dd
plt.ylabel('Retorno Normlizado')
plt.xticks(rotation=90)
plt.plot(datas,rnor)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7fea0f882e50>]
```



0.2 Conceitos teóricos

Antes de começar a implementar os modelos, vamos definir uma função para calcular os resíduos das séries que serão simuladas. O resíduo r é definido como a média quadrática das diferenças dos valores simulados em relação aos valores reais obtidos dos dados e é uma maneira de quantificar o quão bem a simulação se adequa aos dados originais. O resíduo é calculado usando a seguinte equação:

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

onde N é a quantidade de retornos considerados; $\{x_n\}$ são as datas associadas aos retornos; $f(x_n)$ é o valor do retorno real para um dado x_n; $g(x_n)$ é o valor do retorno simulado para um dado x_n. Abaixo definimos a função para calculá-lo.

[6]:
```
def calcular_residuos(lista1,lista2):
    residuo = 0
    for i in range(len(lista1)): residuo += (lista1[i]-lista2[i])**2
    residuo /= len(lista1)
    residuo = residuo**0.5
    return residuo
```

0.2.1 Modelos estatísticos

ARMA(p,q) O ARMA(p,q) é um modelo que permite simular séries temporais para um conjunto de valores $\{y_t\}$ onde o elemento y_t é calculado pela expressão abaixo:

$$y_t = \mu + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}$$

onde μ é o valor médio da série temporal, que pode ser obtido dos dados reais; $\{\phi_i\}$ e $\{\theta_i\}$ são parâmetros que devem ser obtidos de maneira a fazer $\{y_t\}$ se adequar aos dados; e ϵ_t é um ruído obtido de uma distribuição gaussiana normalizada (isto é, com média 0 e desvio padrão 1). Neste trabalho serão usados $p = q = 1$, o que nos dá um modelo ARMA(1,1), descrito pela equação abaixo:

$$y_t = \mu + \phi y_{t-1} + \theta \epsilon_{t-1}$$

Conforme visto na aula do dia 22/02, ϕ e θ satisfazem as seguintes relações:

$$\gamma_0 = \phi\gamma_1 + \sigma_\epsilon^2 + \theta\sigma_\epsilon^2(\phi + \theta)$$

$$\gamma_1 = \phi\gamma_0 + \theta\sigma_\epsilon^2$$

onde γ_0 é a variância dos dados reais; γ_1 é a primeira autocovariância, que pode ser obtida pelo teorema de Wiener-Khinchin; e σ_ϵ^2 é a variância de $\{\epsilon_t\}$, que vale 1, dada a definição de ϵ_t. Considerando $\mu \approx 0$ (o que está de acordo com os dados reais, onde a média é da ordem de 10^{-18}) e fazendo $\sigma_\epsilon^2 \approx 1$ (o que também é amparado pelos dados), podemos obter ϕ e θ a partir das equações acima, o que nos dá as seguintes relações:

$$\theta = \sqrt{\gamma_0 - \frac{\gamma_1^2}{\gamma_0 - 1}}$$

$$\phi = \frac{\gamma_0 - (1 + \theta^2)}{\theta + \gamma_1}$$

GARCH(p,q) No modelo GARCH(p,q), y_t é calculado pela seguinte expressão:

$$y_t = \epsilon_t \sigma_t$$

onde ϵ_t é definido da mesma forma que anteriormente e σ_t é o desvio padrão de y_t, e é dado pela expressão:

$$\sigma_t = \sqrt{\omega + \sum_{i=1}^{p} \alpha_i y_{t-i}^2 + \sum_{i=1}^{q} \beta_i \sigma_{t-i}^2}$$

onde ω, $\{\alpha_i\}$ e $\{\beta_i\}$ são parâmetros que devem ser obtidos de maneira a fazer $\{y_t\}$ se adequar aos dados. Neste trabalho será usada uma biblioteca que determina esses coeficientes através de métodos numéricos. Verificou-se que para os valores $p = 1$ e $q = 3$ a simulação fica bem ajustada aos dados. Nesse caso, temos um modelo GARCH(1,3) no qual y_t é descrito pela expressão abaixo:

$$y_t = \epsilon_t \sqrt{\omega + \alpha_1 y_{t-1}^2 + \beta_1 \sigma_{t-1}^2 + \beta_2 \sigma_{t-2}^2}$$

Cadeia de Markov Uma cadeia de Markov é um modelo no qual temos um certo sistema que pode estar em diferentes estados $\{n\}$. Esse sistema pode passar de um estado n_i para um n_j com uma certa probabilidade p_{ij} que depende apenas de i e j, não sendo influenciada pelos estados acessados pelo sistema até chegar a n_i. Diferentemente dos modelos ARCH e GARCH, que serão usados para simular séries temporais, a cadeia de Markov será usada aqui para obter algumas informações sobre os retornos: estaremos interessados em obter os estados estacionários e os tempos de primeira passagem para cada estado.

Primeiramente, precisamos definir quais são os estados acessados pela série de retornos. Aqui definiremos três deles: "Crab" (C), "Bear" (Be), e "Bull" (Bu). Diremos que a série está no estado C no instante t se o retorno $|r_t|$ for menor ou igual ao desvio padrão σ dos dados, isto é, se $|r_t| \leq \sigma$; o estado Be ocorre quando $r_t < -\sigma$; e Bu é quando $r_t > \sigma$. Para facilitar a lógica dos cálculos mais à frente, vamos associar cada estado a um número, conforme a tabela abaixo:

Estado	Símbolo	Número
Crab	C	0
Bear	Be	1
Bull	Bu	2

ou seja, se for dito que o sistema está no estado 1, significa que o estado é Bear, e assim por diante.

Quando o sistema está em um estado i, ele tem uma certa probabilidade de ir para cada um dos três estados (o que inclui continuar no estado atual). A esse estado i podemos associar um vetor x_i que contém as probabilidades P_C, P_{Be}, e P_{Bu} de o sistema ir para um estado seguinte. Isto é, temos o seguinte vetor:

$$\vec{x_i} = \begin{pmatrix} P_{C,i} \\ P_{Be,i} \\ P_{Bu,i} \end{pmatrix}$$

A partir dos dados reais, podemos obter uma matriz P que nos dá um conjunto de probabilidades: $P_{C \to Be}$ é a probabilidade de que, caso o estado atual seja C, o estado seguinte seja Be, e assim por diante. Devemos notar que usando os símbolos numéricos, podemos escrever $P_{C \to Be}$ como P_{01}, e a mesma lógica se aplica às outras transições. Em sua forma completa, a matriz tem a seguinte definição:

$$P = \begin{pmatrix} P_{00} & P_{10} & P_{20} \\ P_{01} & P_{11} & P_{21} \\ P_{02} & P_{12} & P_{22} \end{pmatrix}$$

Então para obtermos, por exemplo, o termo $P_{21} = P_{Bu \to Be}$, contamos quantas vezes essa transição acontece nos dados reais e dividimos esse valor pela quantidade total de vezes em que o sistema se encontra no estado final, que nesse caso é Be. Utilizando essa mesma lógica podemos obter todos os valores de P. Tendo obtido a matriz, podemos calcular x_{i+1}, que nos dá as probabilidades de que o sistema vá para cada um dos estados no tempo $i + 1$:

$$P\vec{x_i} = \vec{x_{i+1}}$$

Conforme demonstrado na aula do dia 08/03, após muitas passagens temporais, x_i se tornará aproxamadamente constante, e nesse caso o sistema terá atingido estados estacionários, onde as probabilidades de estar em cada estado serão constantes. Mais a frente será mostrado que isso de fato ocorre para os dados analisados.

Além disso, estaremos interessados em calcular os tempos de primeira passagem (TPP) da série temporal. Em termos simples, o TPP associado à transição $C \to Bu$ é o tempo médio que leva para o sistema sofrer essa transição, e assim por diante. Nesse exemplo, o TPP pode ser denotado pelo símbolo μ_{02}. Para calcular um termo qualquer μ_{ij}, usamos a expressão abaixo:

$$\mu_{ij} = 1 + \sum_{k \neq j} P_{ik}\mu_{kj}$$

Com a expressão acima obtemos um sistema linear de 9 equações e 9 incógnitas, que pode ser representado pela expressão abaixo:

$$M\vec{\mu} = \vec{R}$$

onde M e $\vec{R}$ podem ser determinados usando a expressão para μ_{ij} e são exibidos abaixo:

$$M = \begin{pmatrix} 1 & 0 & 0 & -P_{01} & 0 & 0 & -P_{02} & 0 & 0 \\ 0 & 1-P_{00} & 0 & 0 & 0 & 0 & 0 & -P_{02} & 0 \\ 0 & 0 & 1-P_{00} & 0 & 0 & -P_{01} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-P_{11} & 0 & 0 & -P_{12} & 0 & 0 \\ 0 & -P_{10} & 0 & 0 & 1 & 0 & 0 & -P_{12} & 0 \\ 0 & 0 & -P_{10} & 0 & 0 & 1-P_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & -P_{21} & 0 & 0 & 1-P_{22} & 0 & 0 \\ 0 & -P_{20} & 0 & 0 & 0 & 0 & 0 & 1-P_{22} & 0 \\ 0 & 0 & -P_{20} & 0 & 0 & -P_{21} & 0 & 0 & 1 \end{pmatrix}$$

$$\vec{R} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

μ pode ser isolado na equação matricial acima aplicando a matriz inversa a M, M^{-1}, à direita de ambos os lados da expressão:

$$\vec{\mu} = M^{-1}\vec{R}$$

Explicitamente, μ é definido pela expressão abaixo:

$$\vec{\mu} = \begin{pmatrix} \mu_{00} \\ \mu_{01} \\ \mu_{02} \\ \mu_{10} \\ \mu_{11} \\ \mu_{12} \\ \mu_{20} \\ \mu_{21} \\ \mu_{22} \end{pmatrix}$$

onde μ_{00} é o tempo médio que leva para o sistema ir de C para C, e assim por diante. Com isso completamos a exposição teórica das análises que serão desenvolvidas na seção seguinte.

0.2.2 Implementação e resultados

Vamos agora escrever os cógidos para implementar as ideias desenvolvidas até aqui, começando pelo modelo ARMA(1,1).

ARMA(1,1) Vamos começar calculando alguns valores de interesse. O trecho abaixo nos dá a média dos retornos, os valores $\{\epsilon_t\}$ e a lista de autocovariâncias, da qual podemos obter γ_0 e γ_1 a partir do primeiro e segundo elementos, respectivamente:

[7]:
```
mu = np.mean(rnor)
erros = np.random.normal(0, 1, len(rnor))

fr = scipy.fft.fft(rnor) #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 #covariancia dos retornos
```

Com o trecho abaixo calculamos ϕ e θ:

[8]:
```
gama0, gama1 = Cr[0], Cr[1] #autocovariancias de interesse
theta = (gama0-gama1**2/(gama0-1))**0.5
phi = (gama1-theta)/gama0
```

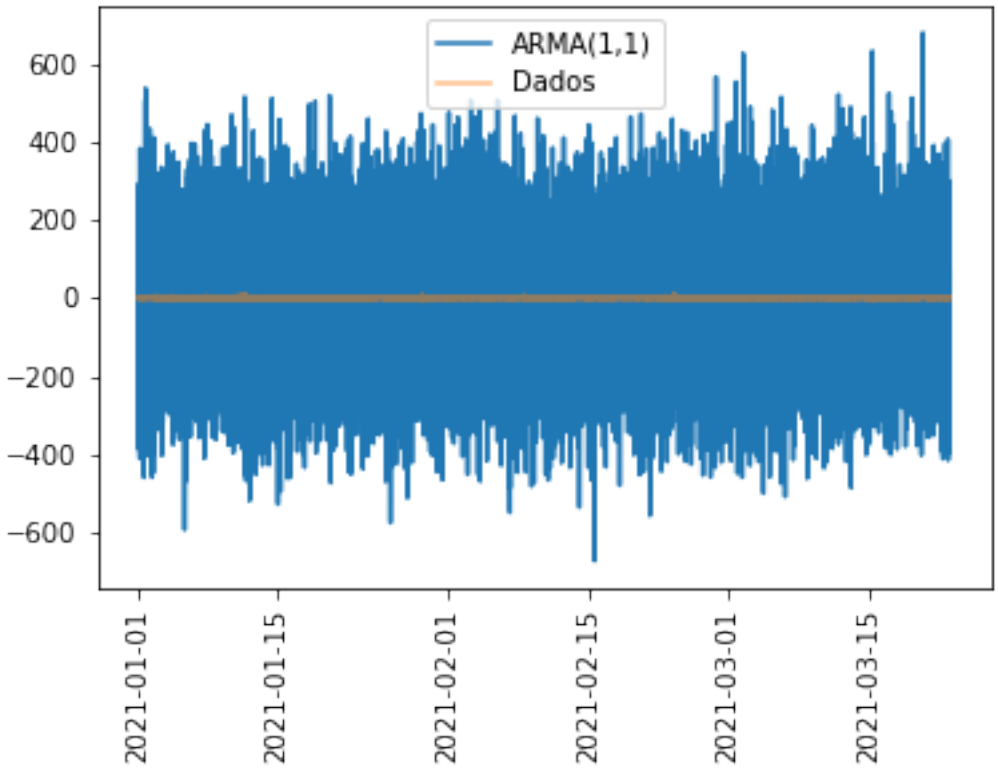
E com isso podemos calcular os valores $\{y_t\}$:

[9]:
```
r_arma = []
yt = rnor[0]
for i in range(len(rnor)):
    r_arma.append(yt)
    yt = phi*yt+theta*erros[i-1]+erros[i]
```

A série real é plotada com a simulada com os comandos abaixo:

[10]:
```
plt.xticks(rotation=90)
plt.plot(datas,r_arma,label='ARMA(1,1)')
plt.plot(datas,rnor,alpha=0.5,label='Dados')
plt.legend(loc='best')
```

[10]:
```
<matplotlib.legend.Legend at 0x7fea0f6a97c0>
```

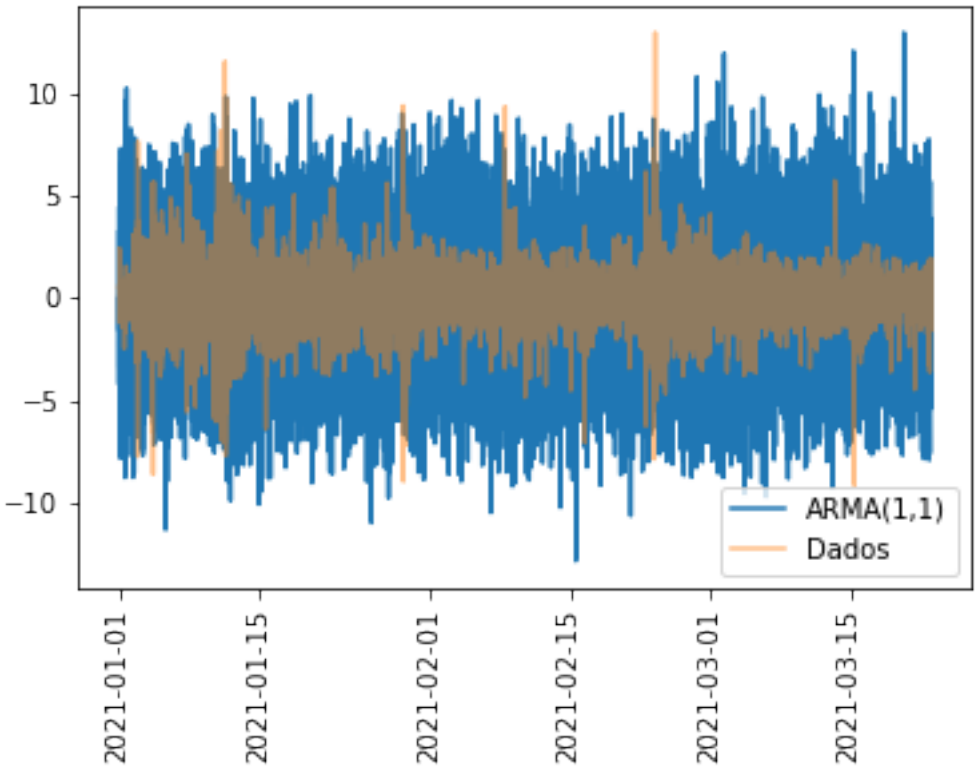


Percebemos que os valores simulados são em geral muito maiores do que os valores reais. Isso pode indicar que o modelo ARMA(1,1) não modela os dados adequadamente ou então pode ter havido algum erro na implementação no modelo. Abaixo é plotada uma figura onde os $\{y_n\}$ são reescalados de forma que seu máximo seja igual ao dos dados reais, a fim de comparar melhor o comportamento das duas séries:

[11]:
```
f = max(r_arma)/max(rnor)
if f <= 1: f = 1/f

plt.xticks(rotation=90)
plt.plot(datas,r_arma/f,label='ARMA(1,1)')
plt.plot(datas,rnor,alpha=0.5,label='Dados')
plt.legend(loc='best')
```

[11]:
```
<matplotlib.legend.Legend at 0x7fea0f4a7190>
```



Usando uma função já definida anteriormente, podemos obter o resíduo r_A para o modelo ARMA(1,1):

[14]:
```
res_arma = calcular_residuos(rnor,r_arma)
res_arma = round(res_arma, 2) #arredonda o valor para duas casas decimais
```

Temos aqui que $r_A = 152.42$. Esse valor longe de 0 confirma nossa conclusão baseada em inspeção visual de que o modelo não modelou os dados corretamente.

GARCH(1,3) A implementação do modelo GARCH nesse trabalho foi inspirada pela referência [2]. O código abaixo faz uma estimativa para as 5 constantes do modelo (ω, α_1, β_1, β_2, e β_3):

[15]:
```
test_size = int(len(rnor)*0.1)
train, test = rnor[:-test_size], rnor[-test_size:]
model = arch_model(train, p=1, q=3)
model_fit = model.fit()
model_fit.summary()
```

```
Iteration:      1,   Func. Count:      8,   Neg. LLF: 3153321457.139747
Iteration:      2,   Func. Count:     20,   Neg. LLF: 26863664.41126357
Iteration:      3,   Func. Count:     30,   Neg. LLF: 37514.677877405695
Iteration:      4,   Func. Count:     39,   Neg. LLF: 27220.39467635595
Iteration:      5,   Func. Count:     47,   Neg. LLF: 27190.95838495234
```

```
Iteration:      6,   Func. Count:     55,   Neg. LLF: 26932.116395374807
Iteration:      7,   Func. Count:     63,   Neg. LLF: 27138.48066071193
Iteration:      8,   Func. Count:     71,   Neg. LLF: 26926.702586524465
Iteration:      9,   Func. Count:     80,   Neg. LLF: 26893.908170454248
Iteration:     10,   Func. Count:     88,   Neg. LLF: 26881.709577727383
Iteration:     11,   Func. Count:     96,   Neg. LLF: 26880.363505298967
Iteration:     12,   Func. Count:    103,   Neg. LLF: 26880.349157357392
Iteration:     13,   Func. Count:    110,   Neg. LLF: 26880.34825430499
Iteration:     14,   Func. Count:    117,   Neg. LLF: 26880.3482392239
Iteration:     15,   Func. Count:    123,   Neg. LLF: 26880.34823922552
Optimization terminated successfully    (Exit mode 0)
            Current function value: 26880.3482392239
            Iterations: 15
            Function evaluations: 123
            Gradient evaluations: 15
```

[15]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:                          y   R-squared:                       0.000
Mean Model:             Constant Mean       Adj. R-squared:                  0.000
Vol Model:                      GARCH       Log-Likelihood:               -26880.3
Distribution:                  Normal       AIC:                           53772.7
Method:            Maximum Likelihood       BIC:                           53820.4
                                            No. Observations:                20979
Date:                Wed, Mar 24 2021       Df Residuals:                    20978
Time:                        22:40:19       Df Model:                            1
                                 Mean Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
mu         8.4059e-03  5.172e-03      1.625      0.104 [-1.731e-03,1.854e-02]
                           Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0170  3.882e-03      4.390  1.134e-05 [9.433e-03,2.465e-02]
alpha[1]       0.1139  1.463e-02      7.785  6.987e-15   [8.519e-02,  0.143]
beta[1]        0.4965      0.174      2.855  4.302e-03     [  0.156,  0.837]
beta[2]        0.0925      0.195      0.475      0.635     [ -0.289,  0.474]
beta[3]        0.2814  5.737e-02      4.906  9.306e-07     [  0.169,  0.394]
==============================================================================

Covariance estimator: robust
"""
```

Com isso podemos definir os valores das contantes:

[16]:
```
omega = 0.0170
alfa1 = 0.1139
beta1 = 0.4965
beta2 = 0.0925
beta3 = 0.2814
```

Com o trecho abaixo definimos valores iniciais para o modelo:

[17]:
```
r_garch = [rnor[0]]
yt = 1
sigmas = [1,1] #[sigma_t-2, sigma_t-1]
```

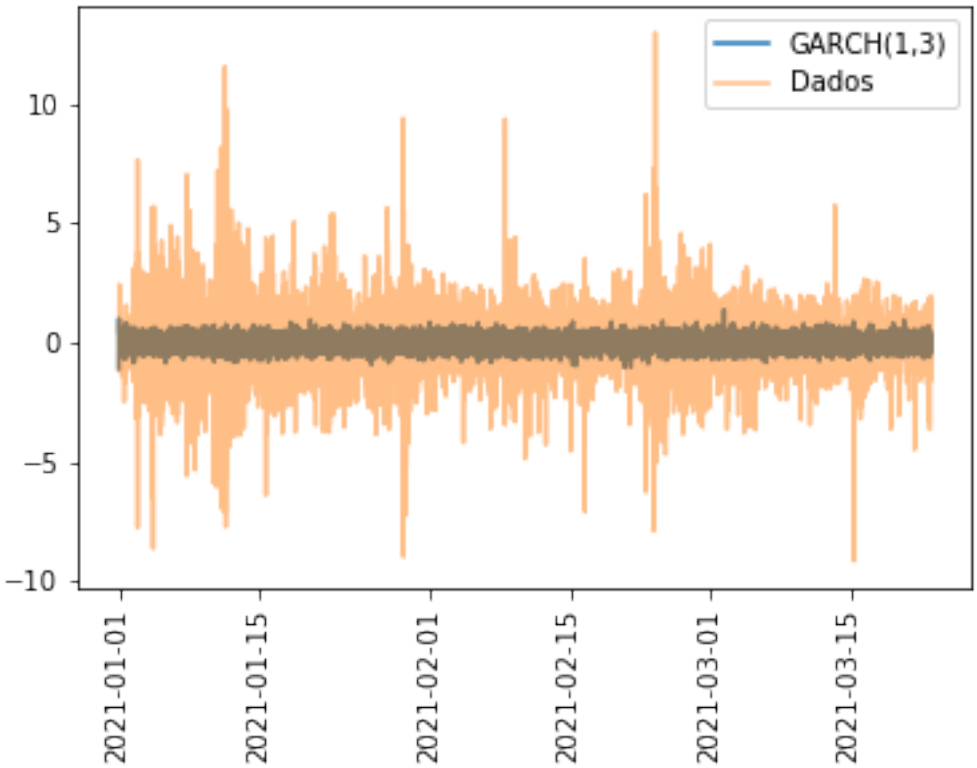
ou seja, definimos y_0 como sendo o primeiro retorno dos dados reais; $y_1 = 1$; e $\sigma_0 = \sigma_1 = 1$. Agora geramos os valores $\{y_t\}$ com os comandos a seguir:

[18]:
```
for i in range(len(rnor)-1):
    r_garch.append(yt)
    sigmat = (omega+alfa1*yt**2+beta1*sigmas[1]**2+beta2*sigmas[0]**2)**0.5
    yt = erros[i]*sigmat
    sigmas[0] = sigmas[1]
    sigmas[1] = sigmat
```

Vamos visualizar os resultados:

[19]:
```
plt.xticks(rotation=90)
plt.plot(datas,r_garch,label='GARCH(1,3)')
plt.plot(datas,rnor,alpha=0.5,label='Dados')
plt.legend(loc='best')
```

[19]:
```
<matplotlib.legend.Legend at 0x7fea0f083dc0>
```



Por inspeção visual, percebemos que o GARCH(1,3) fez uma modelagem mais precisa do que o ARCH(1,1). Isso pode ser confirmado através do resíduo r_G:

[20]:
```
res_garch = calcular_residuos(rnor,r_garch)
res_garch = round(res_garch, 2)
```

Aqui temos $r_G = 1.03$, que representa cerca de 0.67% de r_A.

Cadeia de Markov Primeiro geramos a matriz P_0 que contém as quantides de transições associadas a cada par de estados:

[21]:
```
P = [[0]*3,[0]*3,[0]*3]
std = np.std(rnor) #desvio padrao de rnor
e = ['C','C'] #estados inicial e final (Crab, Bear ou Bull)

lef = [0,0,0] #lista dos estados finais [C,Be,Bu]
#preenchendo P
for i in rnor:
    if abs(i) <= std: e[1] = 'C'
    elif i > std: e[1] = 'Bu'
    elif i < std: e[1] = 'Be'

    if e[1] == 'C':
```

```
        if e[0] == 'C': P[0][0] += 1
        elif e[0] == 'Be': P[0][1] += 1
        else: P[0][2] += 1
    elif e[1] == 'Be':
        if e[0] == 'C': P[1][0] += 1
        elif e[0] == 'Be': P[1][1] += 1
        else: P[1][2] += 1
    else:
        if e[0] == 'C': P[2][0] += 1
        elif e[0] == 'Be': P[2][1] += 1
        else: P[2][2] += 1

    if e[1] == 'C': lef[0] += 1
    elif e [1] == 'Be': lef[1] += 1
    elif e [1] == 'Bu': lef[2] += 1

    e[0] = e[1]
```

E então obtemos P normalizando P_0:

[22]:
```
for linha in range(3):
    for coluna in range(3):
        if coluna == 0: P[linha][0] /= lef[0]
        elif coluna == 1: P[linha][1] /= lef[1]
        else: P[linha][2] /= lef[2]
P = np.array([P[0],P[1],P[2]]) #cada lista representa uma linha
```

Os estados estacionários são obtidos com os comandos abaixo:

[23]:
```
r_mark = [] #retornos calculados por cadeias de markov
x = np.array([2/10, 7/10, 1/10])
yt = r.choices([-1,0,1], x)[0]

lp = [[],[],[]] #lista de probabilidades
for i in range(ni):
    lp[0].append(x[0])
    lp[1].append(x[1])
    lp[2].append(x[2])
    yt = r.choices([-1,0,1], x)[0]
    r_mark.append(yt)
    x = P.dot(x)
```

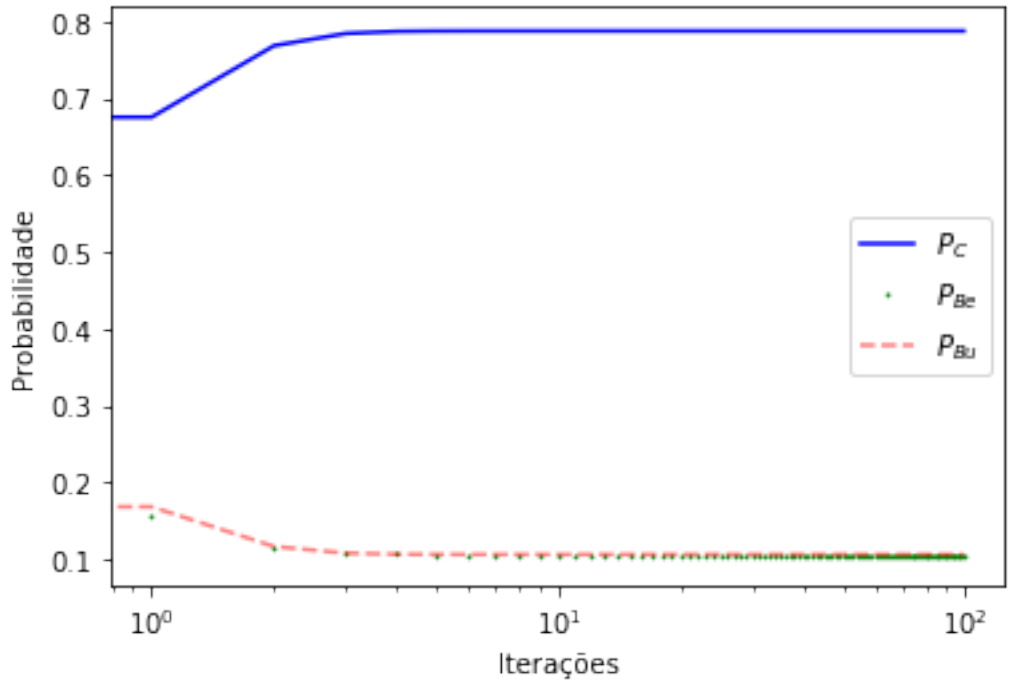
onde os elementos de $\vec{x_0}$ foram definidos arbitrariamente como $(0.2, 0.7, 0.1)$

Podemos visualizar a evolução dos elementos de $\vec{x}$ a cada iteração com o seguinte trecho:

[24]:
```
plt.plot(lp[0],label='$P_C$',color='blue')
plt.plot(lp[1],'v',markersize=1,label='$P_{Be}$',color='green')
plt.plot(lp[2],'--',color='red',label='$P_{Bu}$',alpha=0.5)
```

```
plt.xscale('log')
plt.legend(loc='best')
plt.ylabel('Probabilidade')
plt.xlabel('Iterações')
```

[24]: `Text(0.5, 0, 'Iterações')`



Como já era esperado, percebemos que após algumas iterações as probabilidades de encontrar o sistema em um certo estado se tornam constantes. Por exemplo, após atingir a estabilidade, o sistema se encontra aproximadamente 80% do tempo no estado C, e assim por diante.

Finalmente, o código abaixo calcula os tempos de primeira passagem:

[25]:
```
T = P.transpose() #P transposto

M = np.matrix([[1,0,0,-T[0][1],0,0,-T[0][2],0,0],
               [0,1-T[0][0],0,0,0,0,0,-T[0][2],0],
               [0,0,1-T[0][0],0,0,-T[0][1],0,0,0],
               [0,0,0,1-T[1][1],0,0,-T[1][2],0,0],
               [0,-T[1][0],0,0,1,0,0,-T[1][2],0],
               [0,0,-T[1][0],0,0,1-T[1][1],0,0,0],
               [0,0,0,-T[2][1],0,0,1-T[2][2],0,0],
               [0,-T[2][0],0,0,0,0,0,1-T[2][2],0],
               [0,0,-T[2][0],0,0,-T[2][1],0,0,1]])
```

```
Mi = np.linalg.inv(M)
R = np.array([1]*9)
del mu
mu = Mi.dot(R).transpose() #lista de primeiros passagens no formato de matriz
pp = []
for i in range(9): pp.append(mu[i].tolist()[0][0])
pp = np.array(pp) #lista de primeiras passagens
for i in pp: print(round(i,2))
```

```
1.27
10.45
10.34
1.56
9.48
9.15
1.51
9.62
9.41
```

Os valores de $\vec{\mu}$ são armazenados na lista pp e são os seguintes:

$$\mu_{C\to C} = 1.27$$
$$\mu_{C\to Be} = 10.45$$
$$\mu_{C\to Bu} = 10.34$$
$$\mu_{Be\to C} = 1.56$$
$$\mu_{Be\to Be} = 9.48$$
$$\mu_{Be\to Bu} = 9.15$$
$$\mu_{Bu\to C} = 1.51$$
$$\mu_{Bu\to Be} = 9.62$$
$$\mu_{Bu\to Bu} = 9.41$$

Ou seja, uma vez que o sistema está no estado Be, levará em média 1.56 unidades de tempo para ele ir para o estado C, e assim por diante.

0.2.3 Referências

[1] https://medium.com/swlh/retrieving-full-historical-data-for-every-cryptocurrency-on-binance-bitmex-using-the-python-apis-27b47fd8137f

[2] https://github.com/ritvikmath/Time-Series-Analysis