Capítulo 1

Introdução às Séries Temporais: Uma Abordagem Prática em Python

Rogério de Oliveira, Orlando Y. Albarracin, Gustavo Rocha da Silva

Abstract

This short course is a practical course that introduces the main concepts of time series and how to manipulate and create time series models in Python. Includes the implementation and analysis of ARIMA-type statistical models and implementations that employ classical and deep machine learning. Practical exercises are included in the course and a final project is suggested. As support material for the course, the language of this text and its references were adapted for this purpose. In the end, the participant is expected to be able to understand, analyze and make predictions from series of data in different contexts, and apply these tools in their own research and professional practice.

Resumo

Este minicurso é um curso prático que apresenta os principais conceitos de séries temporais e como manipular e criar modelos de séries temporais em Python. Inclui a implementação e análise de modelos estatísticos do tipo ARIMA e implementações que empregam aprendizado de máquina clássico e profundo. Exercícios práticos estão incluídos no curso e é sugerido um projeto final. Sendo material de apoio ao curso, a linguagem deste texto e suas referências foram adaptadas para esse propósito. Ao final, espera-se que o participante seja capaz de compreender, analisar e fazer previsões a partir de séries de dados em diferentes contextos, e aplicar essas ferramentas na sua própria pesquisa e prática profissional.

1. Introdução

Séries temporais fazem parte de nosso dia a dia e podem ser encontradas em praticamente qualquer área, das ciências físicas e engenharias, às áreas de saúde e de negócios. Análises do comportamento da série, busca de padrões e sazonalidades, simulações e previsões são alguns dos resultados úteis que podemos obter de séries de dados, o que se tornou uma necessidade e um desafio em muitos campos. Este minicurso oferece uma introdução prática à análise e previsão de séries temporais utilizando a linguagem de programação `Python`. Embora existam várias formas de se abordar o problema de séries temporais, este minicurso se concentra no uso de modelos auto-regressivos integrados de médias móveis (ARIMA), um dos modelos estatísticos mais aplicados a séries temporais, e no uso de técnicas de aprendizado de máquina. O curso traz

exemplos, aplicações e exercícios com dados sintéticos e reais de diversas áreas. Ao final, espera-se que o participante seja capaz de entender, analisar e fazer previsões de séries de dados, podendo aplicar essas ferramentas em suas próprias pesquisas e prática profissional.

O curso está baseado no livro **Introdução às Séries Temporais: Uma Abordagem Prática em Python** (Oliveira, Albarracin e Silva, 2024), dos mesmos autores deste curso. Uma versão online está disponível no site do livro, https://github.com/Introducao-Series-Temporais-em-Python, onde podem ser encontrados, incluindo este texto, todos os códigos e dados empregados, os exercícios e soluções, e outros materiais complementares.

2. Introdução às séries temporais

```
#@markdown `imports`
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
# import matplotlib.ticker as ticker
import warnings

from datetime import datetime as dt

path = 'https://github.com/Introducao-Series-Temporais-em-Python/Book/raw/main/Data/'

course_path = 'https://github.com/Rogerio-mack/SBSI_2024_ts_short_course/raw/main'

# plt.style.use([ 'grayscale' , 'https://github.com/Introducao-Series-Temporais-em-Python/Bo
plt.style.use([ 'https://github.com/Introducao-Series-Temporais-em-Python/Book/raw/main/tsp.

plt.rcParams['axes.titlesize'] = 12.0
plt.rcParams['figure.titlesize'] = 12.0

#@markdown `tspplot()`
def tspplot(ts=None,label=None,title=None,ax=None,linestyle='solid',alpha=1,lw=1,nr_xticks=N

  import matplotlib.ticker as ticker

  if ax is None:
    fig, ax = plt.subplots()

  if label is not None:
    ax.plot(ts, label=label, linestyle=linestyle, lw=lw)
  else:
```

```
    ax.plot(ts, linestyle=linestyle,lw=lw,alpha=alpha)

  if title is not None:
    ax.set_title(title)

  if nr_xticks is not None:
    ax.xaxis.set_major_locator(ticker.MaxNLocator(nr_xticks))

  if nr_yticks is not None:
    ax.yaxis.set_major_locator(ticker.MaxNLocator(nr_yticks))

  if label is not None:
    plt.legend()

  plt.tight_layout()

  return


#@markdown `tspdisplay()`
class tspdisplay(object):
    # Adaptado de https://jakevdp.github.io/PythonDataScienceHandbook/index.html
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <hr>
    <h3 style='font-family:"Courier New", Courier, monospace'>{0}</h3><hr>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a + '.head()')._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a + '.head()'))
                           for a in self.args)

#@markdown `tspsimulate()`
import ipywidgets as widgets
from statsmodels.tsa.seasonal import seasonal_decompose

def tspsimulate(nr_months=60, beta_0=1, beta_1=2, beta_2=0, beta_3=0, alpha=0.2, seasons_mor
  np.seed = 123
  index_dates = pd.date_range(start='1/1/2000', periods=nr_months, freq='M')

  t = np.arange(0,len(index_dates),1)
```

```
    n = len(t)

    T = beta_0 + beta_1*t + + beta_2*t**2 + beta_3*t**3
    S = np.sin( np.arange(0,len(index_dates),1) * 2 * np.pi/seasons_month_len)
    S = S # only positives values
    S = S * T.mean() * alpha # scale
    R = np.random.sample(n)
    R = R # only positives values
    R = R * T.mean() * noise # scale

    if model == 'additive':
      Y = T + S + R
    else:
      Y = T * S * R

    Y = Y + np.abs(Y.min()) + 1 # just to create only positive values

    df = pd.DataFrame()
    df.index = index_dates
    df['t'], df['Y'], df['T'], df['S'], df['R'] = t, Y, T, S, R

    global ts1
    ts1 = df

    decomp = seasonal_decompose(df.Y, model=model, period=seasons_month_len)

    if show_plot:
      fig = decomp.plot()
      fig.set_size_inches((9, 7))

      if title is not None:
        plt.suptitle(title,y=1.05)

      plt.tight_layout()
      plt.show()

    # plt.plot(df['T'])
    # plt.show()

    # plt.plot(df['S'])
    # plt.show()

    return ts1

tspsimulate_ts = widgets.interactive(tspsimulate,  alpha=(0.2,10), beta_0=(-10,10), beta_1=(
```

```
# tspsimulate_ts

#@markdown `tspdecompose()`
def tspdecompose(df,model='additive',title=None):
  from statsmodels.tsa.seasonal import seasonal_decompose

  decomp = seasonal_decompose(df, model='multiplicative')

  fig = decomp.plot()
  fig.set_size_inches((9, 7))

  if title is not None:
    plt.suptitle(title,y=1.05)

  plt.tight_layout()
  plt.show()

  return
```

Uma *Série Temporal* é uma sequência de observações registradas em intervalos de tempo regulares.

Essas observações são *medidas* tomadas ou encontradas a tempos regulares, e você certamente já se deparou com dados como valores anuais do PIB, preços diários de ações e commodities, quantidade de *hits* diários em página Web ou site, ou aumento de temperatura global anual. São dados muito comuns e medidos a intervalos regulares (hora, dia, mês etc.). Séries com intervalos muito curtos (segundos ou menos) são ainda encontradas na física e biomedicina, e intervalos muito longos, de décadas ou mais, são encontradas na astronomia e geologia. Aqui a maior parte dos exemplos e exercícios empregam séries diárias, mensais ou anuais, mas os mesmos procedimentos são igualmente aplicáveis a qualquer série.

```
#@markdown
datasets = pd.read_csv(path + 'datasets.csv', index_col=0)

fig, ax = plt.subplots(3,2,figsize=(12,10))

show_files = ['gas_consumption.csv', 'pharma_sales.csv', 'google_trends_tesla.csv', 'covid_b

title_files = ['Consumo de gás para a produção de energia elétrica',
               'Volume mensal de vendas de anti-histamínicos',
               "Consultas 'Tesla' no Google (Google Trends)",
               'Número de novos casos de COVID no Brasil' ]

for i, axis in enumerate(fig.axes):
  if i < 4:
    df = pd.read_csv(path + show_files[i],index_col=0,parse_dates=True)
```

```
    tspplot(df[ df.columns[0] ],title=title_files[i],ax=axis,nr_xticks=5)
  else:
    ax = fig.axes[4::]

import matplotlib.ticker as ticker

# fig, ax = plt.subplots(1,2,figsize=(12,3.5))

global_violence = pd.read_csv(course_path + '/data/global_violence.csv',index_col=0,parse_da
data = global_violence[ global_violence['Category'] == 'Serious assault'][['Country', 'VALUE
data = data.sort_index()
for country in data.Country.unique():
  ax[0].plot(data[ data.Country == country ].VALUE, label=country, lw=1)
ax[0].set_title('Casos de assalto por 100.000 habitantes')
ax[0].xaxis.set_major_locator(ticker.MaxNLocator(5))
ax[0].legend()

GDP_BR_AR = pd.read_csv(course_path + '/data/GDP_BR_AR.csv',index_col=0,parse_dates=True)
sp500 = pd.read_csv(course_path + '/data/sp500.csv',index_col=0,parse_dates=True)

for country in GDP_BR_AR.country.unique():
  ax[1].plot(GDP_BR_AR[ GDP_BR_AR.country == country ]['GDP (Current USD)']/100000, label=co

ax2 = ax[1].twinx()
ax2.plot(sp500[sp500.index.year > 1983]['Adj Close'].resample('Y').mean(), 'k:', label='SP50

ax[1].set_title('GDP Brasil, Argentina e SP500')
ax[1].xaxis.set_major_locator(ticker.MaxNLocator(5))

ax2.legend()
ax[1].legend()

plt.tight_layout()
plt.show()
```

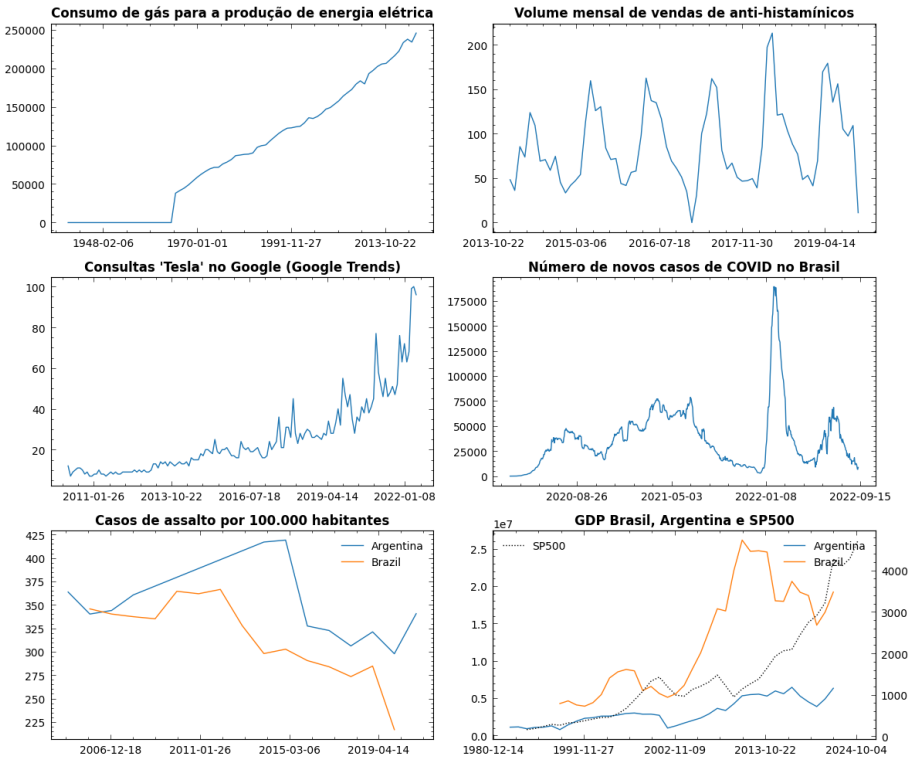


Figura 1. Diversos exemplos de séries temporais em diferentes áreas de domínio.

De acordo com o seu objetivo você pode estar interessado em diferentes tarefas aplicadas a uma série temporal:

1. Fazer previsões de valores futuros
2. Entender o mecanismo gerador da série
3. Descrever e comparar o comportamento da série
4. Procurar periodicidades e padrões relevantes
5. Identificar anomalias
6. Simular a série de dados

e, certamente, fazer previsões de valores futuros desempenha um papel bastante importante. Em todos os casos busca-se criar um modelo que se pretende útil para análise e aqui nos concentraremos unicamente em modelos auto-regressivos integrados de médias móveis (ARIMA) e de aprendizado de máquina supervisionado para previsões de valores futuros. Ambos são *modelos paramétricos* (possuem um número finito de parâmetros) e modelam as séries no *domínio do tempo* (diferentemente de modelos que empregam a frequência ou o espaço de estados), e estão entre os modelos mais amplamente aplicados na economia, finanças, engenharias e outras áreas.

2.1. Decomposição de séries temporais

A ideia de construir um modelo é a de criarmos uma simplificação útil dos dados e no caso de séries temporais o modelo de decomposição é o procedimento clássico.

Em geral decompomos uma série temporal em componentes onde cada um busca modelar um tipo de padrão ou comportamento da série:

- **Tendência**
- **Sazonalidade**
- **Resíduos**

A **tendência** apresenta o comportamento da série no longo prazo, o aumento ou diminuição dos valores da série no longo prazo, como o crescimento da temperatura global ano a ano. A componente **sazonalidade** apresenta o padrão sazonal da série, como mudanças que ocorrem com alguma periodicidade ao longo do tempo. É o caso dos acréscimos e decréscimos de temperatura que ocorrem ao longo das estações do ano, independentemente da elevação das temperaturas no longo prazo. Por último, os **resíduos**, apresentam o comportamento estocástica, ou *aleatório* da série, e que não sabemos explicar. Correspondem, por exemplo, às diferenças de temperatura entre dois dias consecutivos da mesma estação causadas por inúmeros fatores como a maior presença de nuvens ou de raios solares naqueles dias. Essa componente também pode ser denominada de *erro aleatório* ou *ruído branco*.

Vários desses comportamentos podem ser observados nas séries de Figura 1, como a tendência linear crescente do consumo de gás para geração de energia, as sazonalidades anais da venda de anti-histamínicos etc.

Em geral, a tendência inclui uma outra componente denominada **ciclo**, mas que por simplicidade denominamos unicamente de tendência. Ciclos e mudanças sazonais são comportamentos bem diferentes nas séries temporais. A sazonalidade é um comportamento recorrente que se repete a intervalos fixos, regulares (a maior temperatura em certas estações do ano, o maior número de visitas em um site de entretenimento aos finais de semana). Já os ciclos são comportamentos recorrentes mas que ocorrem a intervalos não regulares. Vulcões e terremotos, por exemplo, têm um comportamento recorrente, mas não sabemos quando irão ocorrer, e uma série que represente as temperaturas ou o tremor em torno da cratera do Vulcão Eyjafjallajökull apresentará comportamentos repetitivos, mas que não são sazonais. O mesmo ocorre com os ciclos econômicos que alternam recessão e crescimento, e não sabemos quando irá ocorrer a próxima crise ou o estouro de uma *bolha* do mercado. Nas séries de Figura 1, por exemplo, podemos observar alguns 'ciclos' de séries financeiras (GDP e SP500) e das ondas de variantes da pandemia de COVID.

2.2. Séries temporais em Python

Aquisição dos dados de uma série temporal pode ser feita na forma de dados tabulares. Em `Python`, o `Pandas` fornece suporte a vários tipos de formato de arquivos como `.csv`,`.xlsx`, `.json`, `.html`, `.hdf5` e `.sql` para a criação de um `DataFrame`.

```
# basic imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

```
df = pd.read_csv(course_path + '/data/capea-consulta-cafe.csv')
df.head()
```

```
      Data  vista R$  vista US$
0  01/2014    288.98     119.88
1  02/2014    366.32     153.96
2  03/2014    437.24     187.79
3  04/2014    449.45     201.45
4  05/2014    429.28     193.22
```

O `Pandas` é uma biblioteca para manipulação de dados tabulares e oferece várias funcionalidades para a seleção e transformação dos dados.

```
df[ df['vista R$'] > 1400 ][ ['Data', 'vista R$'] ]
```

```
       Data  vista R$
95  12/2021   1452.15
96  01/2022   1482.59
97  02/2022   1485.35
```

Um caso particularmente importante quando se trata de séries temporais trata-se da manipulação de datas.

```
df.dtypes
```

```
Data          object
vista R$     float64
vista US$    float64
dtype: object
```

```
# seleção errônea dos dados com o atributo Data no formato de caracteres (object)
df[ df['Data'] > '10/2023' ].head()
```

```
       Data  vista R$  vista US$
10  11/2014    460.96     180.61
11  12/2014    455.20     172.39
22  11/2015    469.39     124.29
```

```
23  12/2015    479.32     123.94
34  11/2016    556.74     166.83
```

```
# seleção correta dos dados com o atributo Data no formato de datetime
df.Data = pd.to_datetime(df.Data)
df[ df['Data'] > '10/2023' ].head()
```

```
          Data  vista R$  vista US$
118 2023-11-01    888.00     181.31
119 2023-12-01    974.46     198.90
120 2024-01-01   1003.74     204.34
```

```
df.Data = pd.to_datetime(df.Data)
df[ df['Data'].dt.year > 2023 ].head()
```

```
          Data  vista R$  vista US$
120 2024-01-01   1003.74     204.34
```

2.1.1. Time index

Em Python, muitas funções úteis para a manipulação de séries temporais como resample, gráficos e uso de outros pacotes, requerem que o atributo de tempo da série temporal esteja representado no índice dos dados.

```
df.index = pd.to_datetime(df.Data)
df = df.drop(columns='Data')
df.head()
```

```
            vista R$  vista US$
Data
2014-01-01    288.98     119.88
2014-02-01    366.32     153.96
2014-03-01    437.24     187.79
2014-04-01    449.45     201.45
2014-05-01    429.28     193.22
```

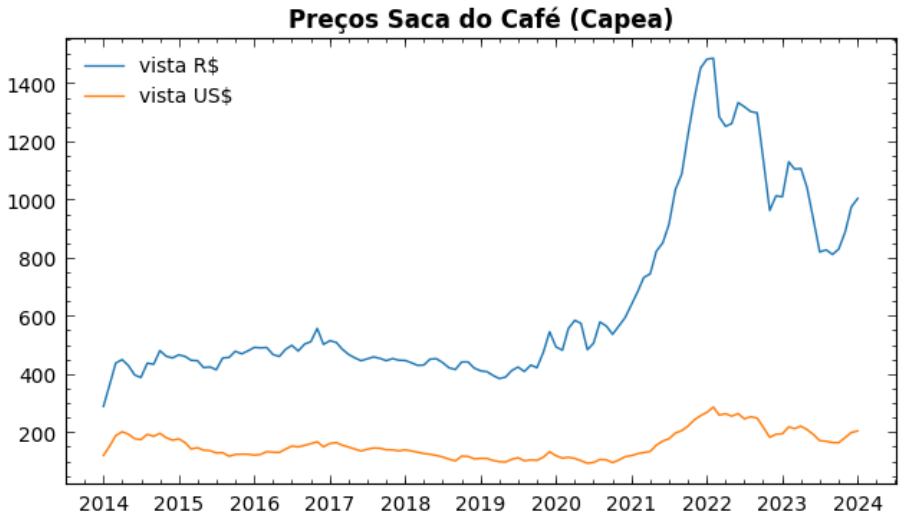
```
# resample dos dados por ano
df_year = df.resample('Y').mean()
df_year.head()
```

```
              vista R$   vista US$
Data
2014-12-31  418.572500  177.984167
2015-12-31  451.494167  137.497500
2016-12-31  494.522500  142.853333
2017-12-31  465.690833  146.050833
2018-12-31  435.646667  120.087500
```

2.1.2. Gráficos

Gráficos de séries de dados são fundamentais para explorar e entender o comportamento dos dados e podem ser obtidos com bibliotecas como `matplotlib` ou `seaborn`. Um índice do tipo `datetime` permite que identificar os dados como uma série temporal e formatar a escala de tempo dos gráficos.

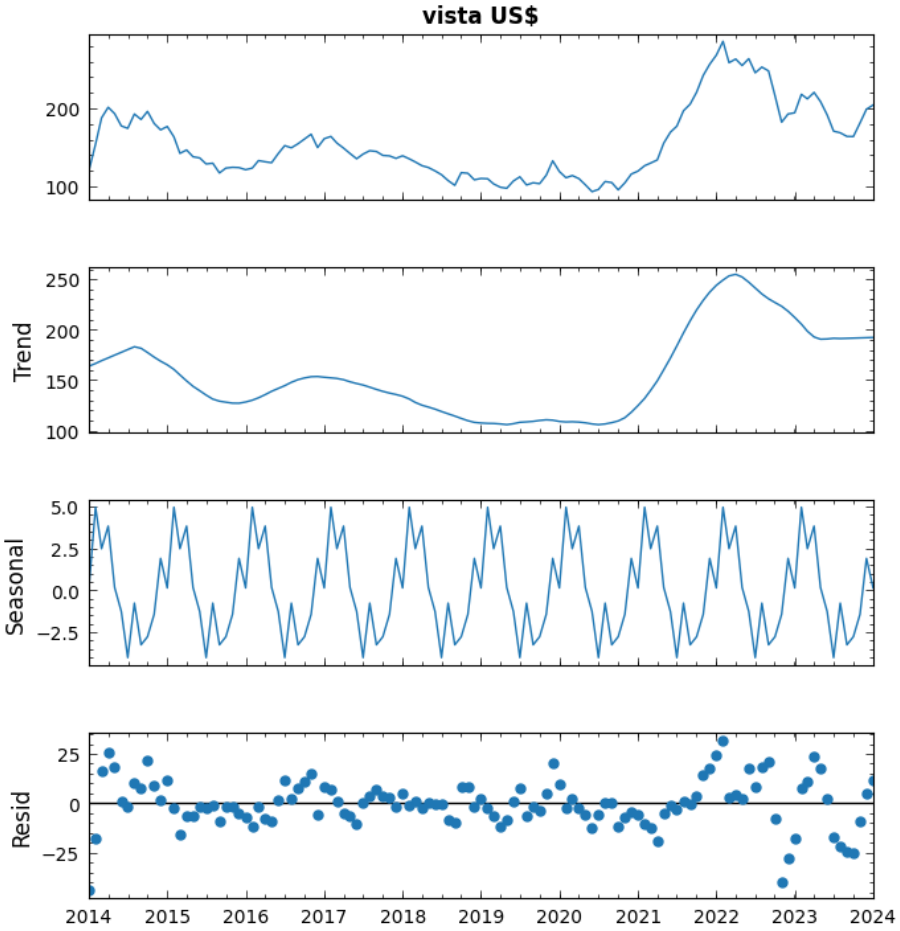
```
plt.plot(df,label=['vista R$','vista US$'])

plt.title('Preços Saca do Café (Capea)')
plt.legend()
plt.show()
```



2.1.3. Decompondo uma série temporal

O principal pacote para modelos estatísticos de séries de dados em `Python` é o `statsmodel`, e a função `seasonal_decompose()` permite decompor uma série em suas componentes de tendência, sazonalidade e resíduos.

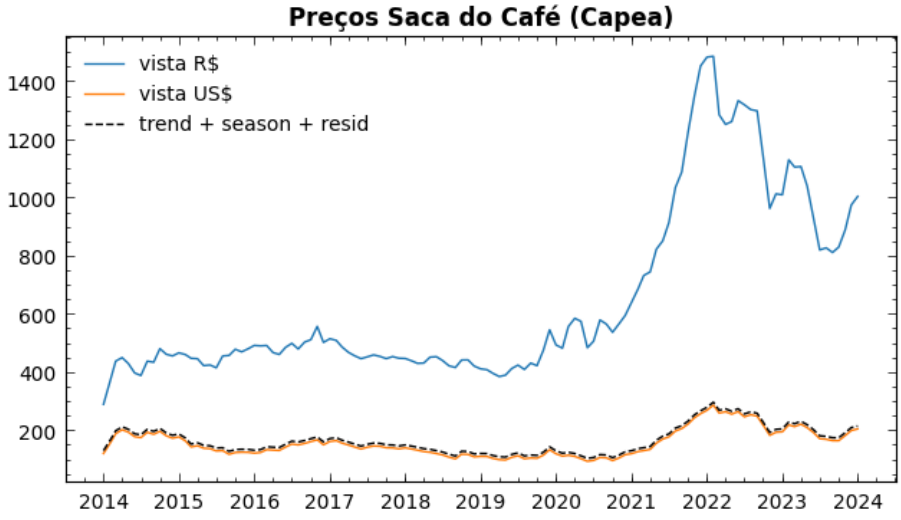
```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(df['vista US$'], model='additive', extrapolate_trend=1)

fig = result.plot()
fig.set_size_inches((7, 8))
plt.show()
```




```
df_compose = pd.DataFrame()

df_compose['trend'] = result.trend
df_compose['seasonal'] = result.seasonal
df_compose['resid'] = result.resid

df_compose['compose'] = df_compose.sum(axis=1)
df_compose.head()
```

Data	trend	seasonal	resid	compose
2014-01-01	163.774583	0.142035	-44.036618	119.88
2014-02-01	166.537917	4.960345	-17.538262	153.96
2014-03-01	169.301250	2.486304	16.002446	187.79
2014-04-01	172.064583	3.833470	25.551946	201.45
2014-05-01	174.827917	0.168887	18.223196	193.22

```
plt.plot(df,label=['vista R$','vista US$'])
plt.plot(df_compose['compose']+10,color='k',linestyle='dashed',label='trend + season + resid

plt.title('Preços Saca do Café (Capea)')
plt.legend()
plt.show()
```



2.3. Séries aditivas e multiplicativas

A depender da natureza da série as componentes de tendência, sazonalidade e resíduo podem ser combinadas de forma aditiva ou multiplicativa, isto é:

$$Y_t = T_t + S_t + R_t$$

para modelos aditivos, ou:

$$Y_t = T_t \times S_t \times R_t$$

para modelos multiplicativos, sendo Y_t a série observada e T_t, S_t, R_t respectivamente as componentes de tendência, sazonalidade e resíduo.

Idealmente você pode ter em mente que séries aditivas apresentam valores que, embora apresentem variações, têm uma variação limitada ao longo do tempo. Já séries multiplicativas, em geral, apresentam um comportamento explosivo, ou exponencial. De qualquer modo, lembre-se que os modelos são simplificações que buscam ser úteis, e uma série, por exemplo a atividade solar a cada mês, não tem qualquer obrigação de se comportar de forma aditiva ou multiplicativa, e nem sempre é muito simples identificar se uma série é aditiva ou multiplicativa.

2.4. Estacionariedade

Decompor uma série já traz informações bastante úteis, como a tendência de seus valores futuros, padrões sazonais e seu grau de incerteza. Mas na maior parte dos casos estamos interessados em fazer previsões mais assertivas dos valores futuros.

Para isso, a maior parte dos modelos assume que a série que se deseja prever é uma série *estacionária*. Uma série estacionária, é uma série em que as estatísticas dos dados, isto é, média, variância e covariância *não mudam ao longo do tempo*. A *estacionariedade no sentido amplo* significaria a ausência completa de tendência e sazonalidade da série. Mas, em geral, é suficiente e mais comum a *estacionariedade no sentido restrito*, podendo haver mudanças das estatísticas em algum momento (como as sazonalidades) mas não *ao longo do tempo*.

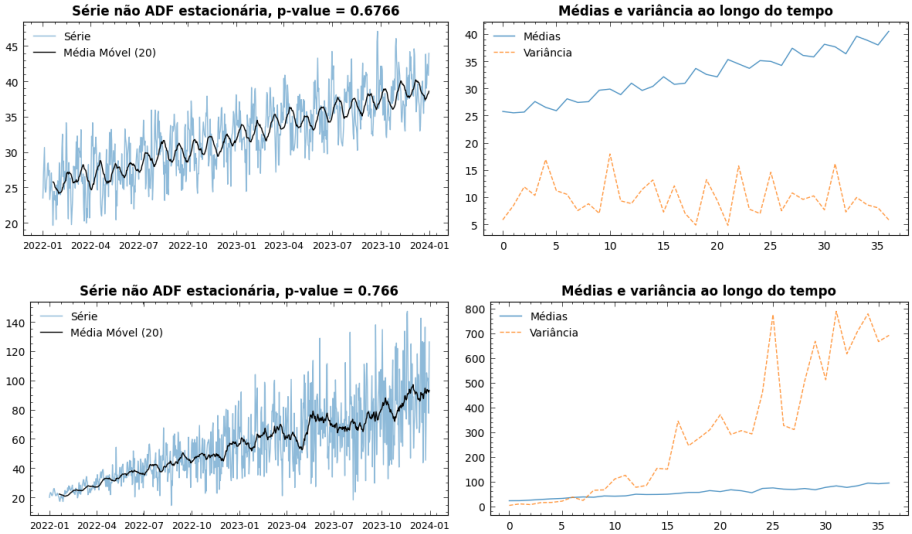
```
no_stacionary_avg = pd.read_csv(course_path + '/data/no_stacionary_avg.csv',index_col=0,pars
no_stacionary_var = pd.read_csv(course_path + '/data/no_stacionary_var.csv',index_col=0,pars
stacionary = pd.read_csv(course_path + '/data/stacionary.csv',index_col=0,parse_dates=True)
```

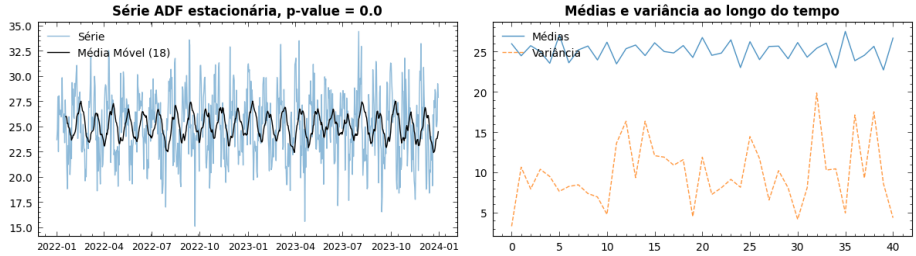
```
#@markdown `PLOT`
def PLOT(df,ADF_result):

  fig, ax = plt.subplots(1,2,figsize=(12,3.5))
  ax[0].plot(df,label='Série',alpha=0.5,lw=1)
  # plt.setp(ax[0].get_xticklabels(), fontsize=9)
  ax[0].tick_params(axis='x', labelsize=9)
  ax[0].plot(df.rolling(ADF_result[2]).mean(),label='Média Móvel (' + str(ADF_result[2]) + '

  if ADF_result[1] < 0.05:
    ax[0].set_title('Série ADF estacionária, p-value = ' + str(np.round(ADF_result[1],4)))
  else:
    ax[0].set_title('Série não ADF estacionária, p-value = ' + str(np.round(ADF_result[1],4)

  ax[0].legend(loc='upper left')

  grouped_means = [df['values'][i:i+ADF_result[2]].mean() for i in range(0, len(df), ADF_res
  grouped_var = [df['values'][i:i+ADF_result[2]].var() for i in range(0, len(df), ADF_result

  ax[1].plot(range(len(grouped_means)), grouped_means, label='Médias',alpha=0.8)
  ax[1].plot(range(len(grouped_var)), grouped_var, label='Variância',alpha=0.8, linestyle='d

  ax[1].set_title('Médias e variância ao longo do tempo')
  ax[1].legend(loc='upper left')

  plt.tight_layout()
  plt.show()
```

```
  return
```

```
from statsmodels.tsa.stattools import adfuller, kpss

def ADF(df,verbose=False):
  result = adfuller(df)
  if verbose:
    print('ADF Statistic: %f' % result[0])
    print('p-value: %f' % result[1])

    if result[1] < 0.05:
      print('Série é ADF estacionária')
    else:
      print('Série é não ADF estacionária')

  return result
```

```
PLOT(no_stacionary_avg, ADF(no_stacionary_avg))
PLOT(no_stacionary_var, ADF(no_stacionary_var))
PLOT(stacionary, ADF(stacionary))
```





Há vários testes estatísticos para verificar a estacionariedade de uma série temporal, mas os testes mais comuns são o teste **Augmented Dickey Fuller ("ADF")** e o teste **Kwiatkowski-Phillips-Schmidt-Shin ("KPSS")** e que, basicamente, verificam a presença de tendência na série e entendimento do domínio e a análise exploratória dos dados são essenciais para uma interpretação adequada desses testes. Os gráficos anteriores ilustram diferentes casos de estacionariedade e não estacionariedade com os respectivos p-values do teste ADF.

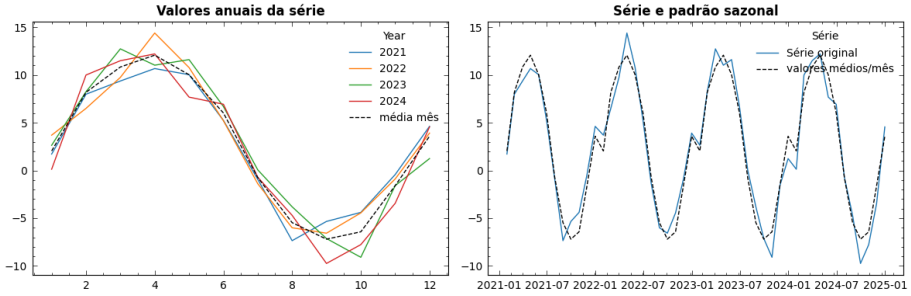
Tanto no ADF como no KPSS teste, a hipótese nula (estacionariedade) é rejeitada para p-value > 0.05. Assim, **para p-value < 0.05, diremos que a série é estacionária**.

2.5. Sazonalidade

A inspeção visual das séries desempenha um papel fundamental na identificação de sazonalidades, como também da estacionariedade apesar dos testes disponíveis. Gráficos de agrupamentos com os valores médios por dia, mês etc. são bastantes empregados e constituem a base de muitos modelos de sazonalidade.

```
# Criando uma série de periodicidade anual
date_range = pd.date_range(start='2021-01-01', periods=4*12, freq='M')
seasonal_pattern = np.sin(np.linspace(0, 2*np.pi, 12)) # Repetindo o padrão mensal (de jane
values = np.tile(seasonal_pattern, 4) # Repetindo o padrão sazonal para os 4 anos
values = 10*values + 5*np.random.sample(len(values))

time_series = pd.DataFrame({'Value': values},index=date_range)

# agregando os valores
group_month = pd.DataFrame(time_series.groupby([time_series.index.year,time_series.index.mon
group_month.columns = ['year','month','values']

fig, ax = plt.subplots(1,2,figsize=(12,4))

for y in group_month.year.unique():
 ax[0].plot(group_month[ group_month.year == y ].month,group_month[ group_month.year == y ]

ax[0].plot(time_series.index.month.unique(),time_series.groupby([time_series.index.month]).m
ax[0].set_title('Valores anuais da série')
```

```
ax[0].legend(title='Year',loc='upper right')

ax[1].plot(time_series,label='Série original')
ax[1].plot(time_series.index,np.tile(time_series.groupby([time_series.index.month]).mean().v
ax[1].set_title('Série e padrão sazonal')
ax[1].legend(title='Série',loc='upper right')

plt.tight_layout()
plt.show()
```



2.5.1. Peridiograma

Apesar da importância e predominância da inspeção visual, o **peridiograma** pode ser uma ferramenta bastante útil e mais adiante introduziremos os gráficos autocorrelação que também nos ajudam a identificar sazonalidades.

O peridiograma apresenta a distribuição das frequências em um sinal ao longo do tempo e as frequências mais presentes podem ser empregadas para identificar as sazonalidades presentes na série.

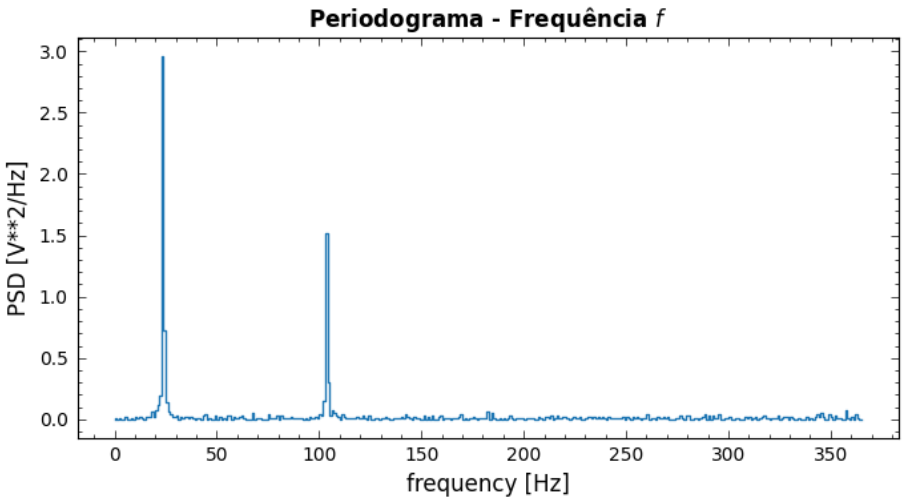
$$P_i = \frac{\text{Total de Períodos}}{freq_i}$$

onde, P_i é a periodicidade da frequência $freq_i$. Pode haver inúmeras frequências, mas podemos nos limitar as mais predominantes (1, 2 ou 3 mais presentes).

```
from scipy import signal
fig, ax = plt.subplots(figsize=(7,4))

frequencies, spectrum = signal.periodogram(stacionary['values'],fs=len(stacionary))

ax.step(frequencies, spectrum)

ax.set_title('Periodograma - Frequência $f$')
ax.set_xlabel('frequency [Hz]')
ax.set_ylabel('PSD [V**2/Hz]')
```

```
dfft = pd.DataFrame()
dfft['frequencies'], dfft['spectrum'] = frequencies, spectrum
dfft['periods'] = len(stacionary) / dfft['frequencies']
display(dfft.sort_values('spectrum',ascending=False).head())

plt.tight_layout()
plt.show()
```

	frequencies	spectrum	periods
24	24.0	2.960242	30.416667
104	104.0	1.515986	7.019231
25	25.0	0.723724	29.200000
105	105.0	0.305413	6.952381
23	23.0	0.188924	31.739130



2.6. Diferenciação e Log: eliminando a tendência e sazonalidade

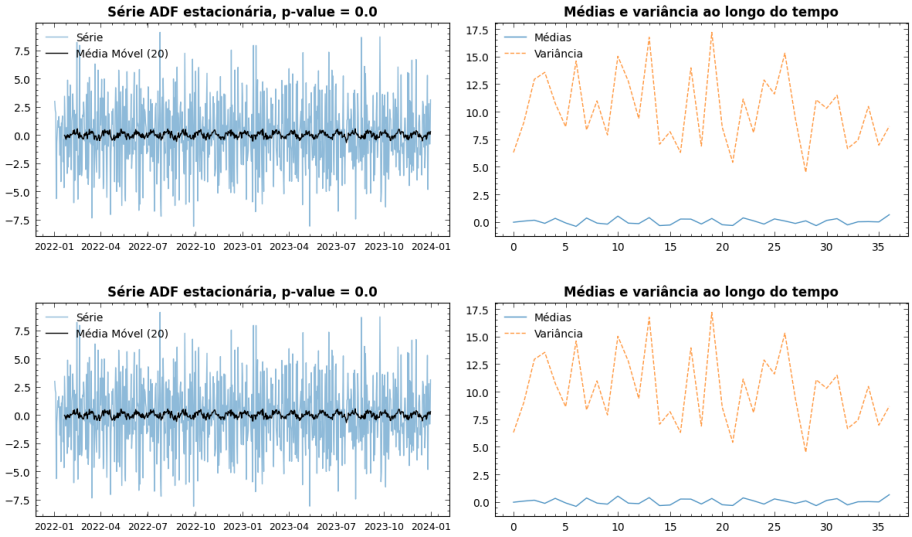
Um dos métodos mais simples para se reduzir a tendência de uma série temporal é construir uma nova série por **diferenciação**. Nesta nova série o valor no intervalo de tempo atual é calculado como a diferença entre o valor original e o valor no intervalo de tempo anterior.

$$\Delta^1 y_t = y_t - y_{t-1}$$

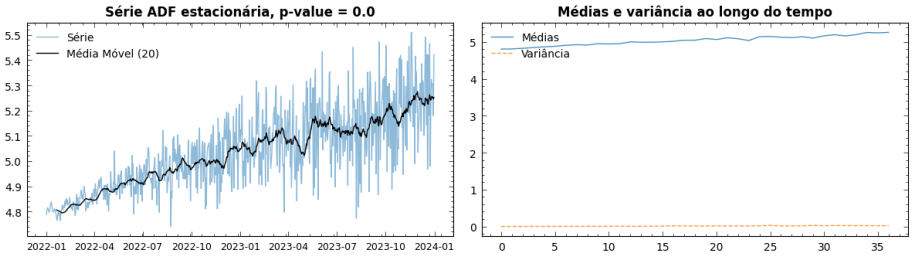
$$\Delta^d y_t = y_t - y_{t-d}$$

A diferenciação acima é de ordem 1 e diferenciações maiores podem ser aplicadas para, por exemplo, eliminar uma tendência polinominal. Para tendências exponenciais pode ser necessário aplicar, do mesmo modo, a transformação *logarítimica* da série (tornando-a linear) antes de se aplicar a diferenciação.

```
PLOT(no_stacionary_avg.diff().dropna(), ADF(no_stacionary_avg.diff().dropna()))
PLOT(no_stacionary_avg.diff().dropna(), ADF(no_stacionary_avg.diff().dropna()))
```




```
no_stacionary_varlog = no_stacionary_var.copy() + 100
no_stacionary_varlog['values'] = np.log(no_stacionary_varlog['values'])

PLOT(no_stacionary_varlog.dropna(), ADF(no_stacionary_varlog.diff().dropna()))
```




```
#@markdown
# nota: para reconstrução da série original a partir da série diferenciada com cumsum()
no_stacionary_avg['y'] = no_stacionary_avg['values']
no_stacionary_avg['y.diff()'] = no_stacionary_avg['y'].diff()

no_stacionary_avg['y_cumsum()'] = no_stacionary_avg['y'].diff()
no_stacionary_avg.at['2022-01-01','y_cumsum()'] = no_stacionary_avg.iloc[0]['y']
no_stacionary_avg['y_cumsum()'] = no_stacionary_avg['y_cumsum()'].cumsum()
```

```
# no_stacionary_avg.head()
```

2.7. Resíduos

Os pressupostos do modelo ARIMA incluem a estacionariedade da série temporal (ou a estacionariedade por diferenciação), mas também a independência dos resíduos e a normalidade dos resíduos.

O termo resíduo é ser empregado para designar a diferença entre os valores reais e o ajuste obtido da série por algum modelo e consiste em uma componente *não explicada* série.

Essa diferença pode ser medida de diversas formas, mas ao final todas refletem as diferenças da série real Y_t e os valores estimados $\hat{Y}_t$ que queremos minimizar. Por serem muito empregadas algumas dessas medidas recebem nomes especiais:

- Erro Médio Absoluto (*Mean Absolute Error*), $MAE = \frac{1}{n}\sum|Y_t - \hat{Y}_t|$
- Erro Médio Quadrático (*Mean Square Error*), $MSE = \frac{1}{n}\sum(Y_t - \hat{Y}_t)^2$
- Raiz do Erro Médio Quadrático (*Root Mean Square Error*), $RMSE = \sqrt{ \frac{1}{n} \sum (Y_t - \hat Y_t)^2 } $
- Erro Percentual Absoluto Médio (*Mean Absolute Percentage Error*), $MAPE = \frac{1}{n}\sum|\frac{Y_t - \hat{Y}_t}{Y_t}|$

Todas medidas que podem ser fácil e diretamente calculadas ou pode-se empregar algum pacote.

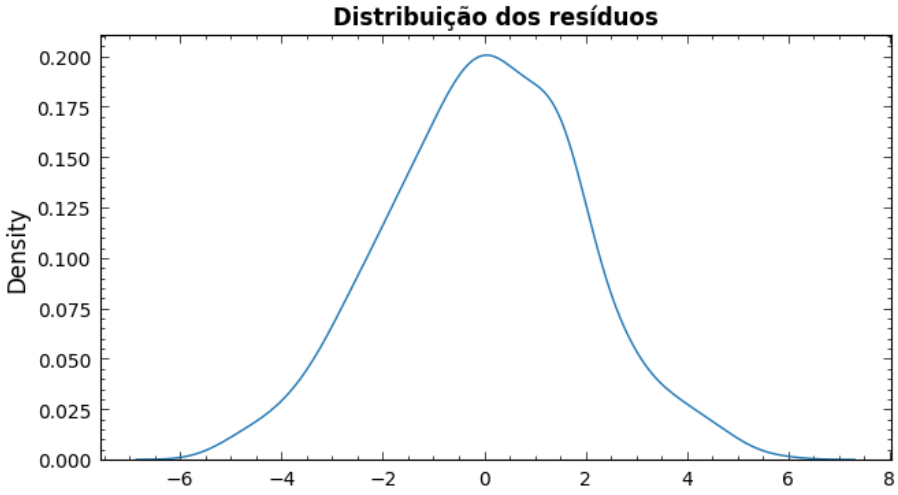
```
import statsmodels.tools.eval_measures as eval_measures

def error_measures(y, y_pred):

    mae = np.mean(np.abs(y_pred - y))                    # MAE
    mse = np.mean((y_pred - y)**2)                       # MSE
    rmse = eval_measures.rmse(y, y_pred, axis=0)         # RMSE from statsmodels
    # rmse = np.mean((y_pred - y)**2)**.5                # RMSE
    mape = np.mean(np.abs(y_pred - y)/np.abs(y))         # MAPE

    metrics = {'MSE':mse, 'MAE': mae, 'RMSE':rmse, 'MAPE':mape}

    for key, value in metrics.items():
      print(f'{key}: \t {value:.4f}')

    return metrics

result = seasonal_decompose(stacionary, model='additive')

non_NA = pd.merge(stacionary, result.trend, how='inner', left_index=True, right_index=True)
```

```
y = stacionary.loc[non_NA]['values']
y_pred = result.trend.loc[non_NA] + result.seasonal.loc[non_NA]

_ = error_measures(y,y_pred)
```

```
MSE:   3.6922
MAE:   1.5335
RMSE:   1.9215
MAPE:   0.0628
```

Mas além de buscar minimizar o erro, ou resíduos, queremos que seus valores sejam independentes, no sentido de não estarem correlacionados, e que apresentem uma distribuição normal.

```
sns.kdeplot(y - y_pred)

plt.title('Distribuição dos resíduos')
plt.show()
```



A normalidade dos resíduos pode ser verificada através de um gráfico de distribuição ou do tipo *qqplot*, o que é mais comum que o uso de testes de hipótese de normalidade. Para independência dos resíduos pode-se empregar um gráfico de autocorrelação dos resíduos. Veremos esse gráfico mais adiante como também uma função `plot_diagnostics()` do `statsmodels` que agrega todos esses gráficos para uma melhor análise dos resíduos.

2.8. Exercícios e referências

Além do livro base deste curso há um grande número de livros texto que tratam desde os conceitos de séries temporais. Montgomery et al. (2015) e Chatfield

(1996) são livros texto clássicos e um ótimo texto nacional é Morettin e Toloi (2006). Eles tratam dos conceitos de séries temporais e de modelos estatísticos, mas não trazem código ou implementações. Hyndman e Athanasopoulos (2018) é um livro texto bastante tradicional de séries temporais e que também traz exemplos de implementação, embora empregue a linguagem R, bastante empregada no tratamento de séries de dados. O texto também tem uma versão disponível online. Box et. al. (2015) é um clássico e uma referência obrigatória para quem quer se aprofundar formalmente em modelos estatísticos para séries temporais.

Para conceitos introdutórios de Python, `Pandas` e aprendizado de máquina clássico, Vanderplas (2016) é um texto muito útil, possui um seção dedicada a séries temporais e, assim como o material deste curso e nosso livro texto, está disponível online em formato de notebooks Python. Também disponíveis online, Oliveira (2022) é uma introdução útil para visualização de dados em Python, incluindo séries de dados, e Kong et al. (2020) apresenta a implementação de vários métodos numéricos em Python.

Os sites de documentação das bibliotecas Pandas (2024) e statsmodels (2024) são úteis para a melhor compreensão de muitas das funções empregadas nesta e nas seções seguintes.

Por último, para complementar esta seção, De Gooijer e Hyndman (2006) é um artigo curto que revisa, sem fórmulas ou implementações, as ideias do principais modelos de séries temporais clássicos e como eles evoluíram antes da introdução dos modelos de aprendizado de máquina.

Os exercícios dessa seção e suas soluções podem ser acessadas no material complementar do curso exerc_parte1_introd.

3. Modelos autoregressivos

A ideia principal dos modelos autoregressivos, incluindo o ARIMA, consiste em modelar a dependência serial dos dados, uma vez que, na maioria das séries, observa-se que os valores recentes estão correlacionados com seus valores passados e que a força dessa dependência diminui quando considerados valores mais distantes no tempo. Assim, é razoável pensar que o valor de amanhã das vendas de uma safra ou do volume de chuvas está correlacionado com os mesmos valores observados ontem, ou no dia anterior, e que essa correlação diminui conforme nos afastamos no tempo.

3.1. Modelos de regressão e autoregressão

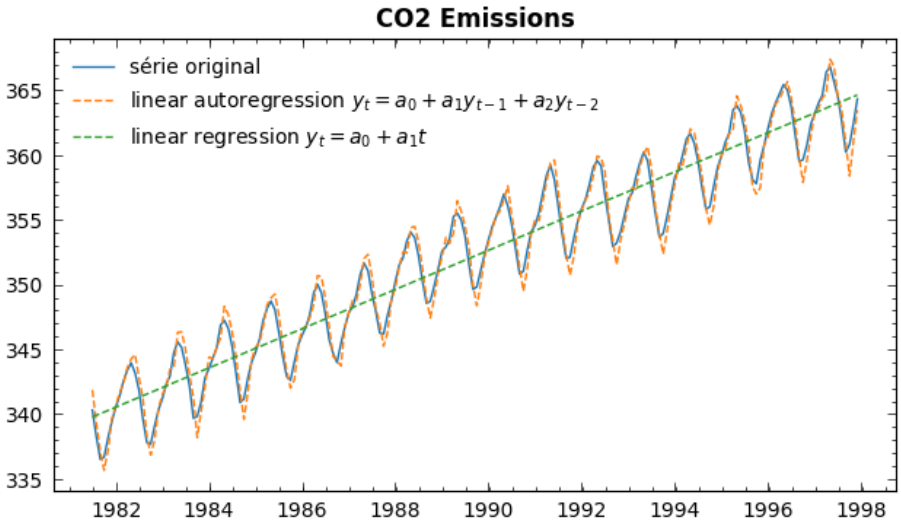
Modelos de regressão se baseiam em variáveis independentes para prever a variável dependente, os modelos de autoregressão usam os próprios valores passados da variável dependente para fazer previsões futuras. Para séries temporais, em geral, faz mais sentido empregarmos a forma autoregressiva que empregar o tempo

(que seria única variável independente em uma única série de dados). Entretanto, em ambos os casos o cálculo dos coeficientes pode ser feito do mesmo modo (em geral um método de mínimos quadrados).

```
co2 = pd.read_csv(path + 'co2.csv',index_col=0,parse_dates=True)

co2['CO2_t-1'] = co2.CO2.shift()
co2['CO2_t-2'] = co2.CO2.shift().shift()
co2 = co2.dropna()
co2.head()
```

Date	time	CO2	CO2_t-1	CO2_t-2
1981-07-01	2	340.32	342.08	342.74
1981-08-01	3	338.26	340.32	342.08
1981-09-01	4	336.52	338.26	340.32
1981-10-01	5	336.68	336.52	338.26
1981-11-01	6	338.19	336.68	336.52

```
import statsmodels.api as sm

# regressão
X = co2[['time']]
y = co2[['CO2']]

X = sm.add_constant(X)
y_regression = sm.OLS(y, X).fit().predict()

# auto regressão
X = co2[['CO2_t-1','CO2_t-2']]

X = sm.add_constant(X)
y_autoregression = sm.OLS(y, X).fit().predict()

# show results
plt.plot(co2.CO2,label='série original')
plt.plot(co2.index,y_autoregression, '--', label='linear autoregression $y_t = a_0 + a_1 y_
plt.plot(co2.index,y_regression, '--', label='linear regression $y_t = a_0 + a_1 t$')

plt.legend(loc='upper left')
plt.title('CO2 Emissions')
plt.show()
```



3.2. Modelo ARIMA

Existem vários tipos de modelos, empregando diferentes princípios, para análise e previsões de séries temporais. O modelo ARIMA (Autorregressivos Integrados de Médias Móveis), é um modelo de análise estatística amplamente utilizado para modelar séries temporais estacionárias e não estacionárias, e constituir a base de modelos mais complexos (ARIMAX, VARIMAX, SARIMAX, ARCH, GARCH etc. envolvendo variáveis exógenas, sazonalidades e volatilidade de séries temporais). O ARIMA é, portanto, um modelo fundamental.

O modelo ARIMA consiste nos seguintes componentes:

- $AR(p)$: Termo autoregressivo que incorpora a dependência entre uma observação e uma série de observações defasadas até a ordem p, o que pode ser escrito do seguinte modo:

$$Y_t = \phi_0 + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + ... + \phi_p Y_{t-p} + \epsilon_t,$$

onde ϵ_t é o erro do modelo.

- $I(d)$: Termo integrado que envolve diferenciar na ordem d os dados da série temporal para torná-los estacionários.

$$W_t = \Delta^d Y_t = Y_t - Y_{t-d}$$

e portanto,

$$Y_t = W_t + Y_{t-d}$$

- $MA(q)$: Termo de média móvel que leva em conta a dependência entre uma observação e um erro residual de um modelo de média móvel de ordem q.

$$Y_t = \mu + e_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + ... + \theta_q\epsilon_{t-q},$$

onde μ é a média da série.

Tanto o modelo AR como o modelo MA são conceitualmente e podem ser calculados como uma regressão linear do valor atual da série, respectivamente sobre seus valores passados (AR) e os termos de erro com relação à média móvel (MA). Ao final o modelo ARIMA completo pode ser escrito como:

$$W_t = \underbrace{\Delta^d Y_t}_{I(d)}$$

$$W_t = \underbrace{\phi_0 + \phi_1 W_{t-1} + \phi_2 W_{t-2} + ... + \phi_p W_{t-p}}_{AR(p)} + \underbrace{\epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + ... + \theta_q\epsilon_{t-q}}_{MA(q)},$$

$$Y_t = W_t + Y_{t-d}$$

Assim, o modelo ARIMA é a junção de modelos que podem também ser empregados de forma independente como segue.

\begin{array}{c c c} \hdashline \text{Model}&\text{ARIMA(p,q,d)}&\text{Tipo de Série}\ \hdashline AR(p) & ARIMA(p, 0, 0) & \text{estacionária} \ MA(q) & ARIMA(0, 0, q) & \text{estacionária} \ ARMA(p,q) & ARIMA(p, 0, q) & \text{estacionária} \ ARIMA(p, d, q) & ARIMA(p, d, q) & \text{não estacionária} \ \hdashline \end{array}

E em todos os casos como vimos assume-se a independência e a normalidade dos resíduos da série.

3.2.1. Exemplo 1

Considerando a série Y_t estacionária podemos considerar os seguintes modelos de ordem 1:

\begin{array}{c c } \hdashline \text{Modelo}&\text{ARIMA(p,q,d)}\ \hdashline AR(1) \space \text{ou} \space ARIMA(1,0,0) & Y_t=\phi_0 + \phi_1Y_{t-1} + \epsilon_t \ I(1) \space \text{ou} \space ARIMA(0,1,0) & Y_t= Y_{t-1} + \epsilon_t \ MA(1) \space \text{ou} \space ARIMA(0,0,1) & Y_t= \epsilon_t + \theta_1 \epsilon_{t-1} \ ARMA(1,1) \space \text{ou} \space ARIMA(1,0,1) & Y_t=\phi_0 + \phi_1Y_{t-1} + \epsilon_t + \theta_1 \epsilon_{t-1} \ \hdashline \end{array}

3.2.2. Exemplo 2

Considerando a série Y_t estacionária para a diferenciação de ordem 1, podemos construir a série estacionária:

$$W_t = Y_t - Y_{t-1}$$

e, então, o modelo:

\begin{array}{c c } \hdashline \text{Modelo}&\text{ARIMA(p,q,d)}\ \hdashline ARIMA(1,1,1) & W_t=\phi_0 + \phi_1W_{t-1} + \epsilon_t + \theta_1 \epsilon_{t-1} \ \hdashline \end{array}

Em que é ajustado um modelo ARMA(1,1) à série diferenciada W_t.

3.2.3. Exemplo 3

Modelos de *suavização*, uma outra técnica bastante empregada em séries temporais, também podem ser derivados de modelos ARIMA como o modelo básico de suavização (ARIMA(0,1,1)), o modelo de *Holt Amortecido* (ARIMA(0,1,2)) e o método linear de Holt (ARIMA(0,2,2)). E o modelo ARIMA Sazonal, SARIMA(p,d,q)(P,D,Q), consiste em um modelo ARIMA em que a componente sazonal é também modelada do mesmo modo com os parâmetros P, D, Q.

3.3. Autocorrelação e autocorrelação parcial

A correlação de duas variáveis x, y refere-se à sua dependência linear e é dada por:

$$\rho(x, y) = \frac{cov(x, y)}{\sqrt{var(x)var(x)}}$$

Em séries temporais, nos referimos à *autocorrelação* à correlação entre valores da mesma série para intervalos de tempo diferentes e, por exemplo a correlação do valor atual com o valor do instante anterior da série é dado por:

$$\rho(x_t, x_{t-1}) = \frac{cov(x_t, x_{t-1})}{\sqrt{var(x_t)var(x_{t-1})}}$$

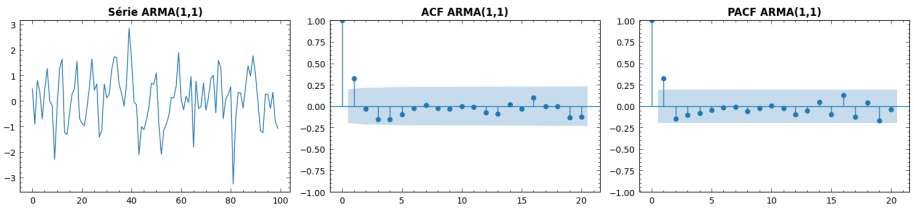
A função ACF (*Autocorrelation Function*) fornece os valores de autocorrelação para diferentes defasagens de valores $(x_{t-1}, x_{t-2}, ...)$ e fornece uma boa estimativa para os valores q do modelo AR. A função ACF considera todos os valores intermediários, isto é, a correlação de x_t e x_{t-k} inclui cadeia de dependência de todos os têrmos entre os dois valores. A função PACF (*Partial Autocorrelation Function*) estima os valores de autocorrelação para diferentes defasagens de valores $(x_{t-1}, x_{t-2}, ...)$ somente para o último elemento, excluindo as dependências anteriores. Ela fornece uma boa estimativa para os valores p do modelo AR.

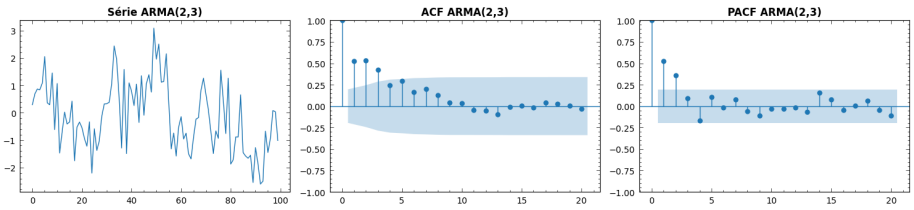
\begin{array}{c c c c} \hdashline \text{Função} & MA(q) & AR(p) & ARMA(p,q) \ \hdashline ACF & \text{Desprezível após } q & \text{Decaimento}^1 & \text{Decaimento}^1 \text{após } q \ PACF & \text{Decaimento}^1 & \text{Desprezível após } p & \text{Decaimento}^1 \text{após } p \ \hdashline \end{array}

[1] *Decaimento tipo exponencial ou sinusoide*

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

def plot_ts_acf_pacf(ts,model_name):
  fig, (ax1, ax2, ax3) = plt.subplots(1, 3,figsize=(15,3.5))

  ax1.plot(ts,linestyle='solid',alpha=1,lw=1)
  ax1.title.set_text('Série ' + model_name)
  plot_acf(ts, ax=ax2, title='ACF ' + model_name)
  y=plot_pacf(ts, ax=ax3, title='PACF ' + model_name)

  plt.tight_layout()
  plt.show()
  return
```

```
import statsmodels.api as sm
from statsmodels.tsa.arima_process import arma_generate_sample
np.random.seed(1234)

model_name = 'ARMA(1,1)'
ar = np.array([1,-0.3,0,0,0,0]) # list ar_coefs has the form [1, -a_1, -a_2, ..., -a_p]
ma = np.array([1,0.3,0,0,0,0])  # list ma_coefs has the form [1, m_1, m_2, ..., m_q]
ts = arma_generate_sample(ar, ma, nsample=100)
plot_ts_acf_pacf(ts,model_name)

model_name = 'ARMA(2,3)'
ar = np.array([1,-0.4,-0.3,0,0,0])
ma = np.array([1])
ts = arma_generate_sample(ar, ma, nsample=100)
plot_ts_acf_pacf(ts,model_name)
```





3.4. Aplicando um modelo ARIMA

Após uma exploração e entendimento dos dados a aplicação do modelo ARIMA envolve, desse modo, os seguintes passos:

1. **Análise da estacionariedade da série**, e escolha do parâmetro d de diferenciação que torna a série estacionária nos casos em que não é estacionária.
2. **Verificação da autocorrelação e autocorrelação parcial da série**, identificando os potenciais valores p (através da PACF) e q (através da ACF) do modelo.
3. **Análise dos modelos e seleção dos parâmetros**, seguindo alguma métrica de desempenho do modelo como o critério de informação de Akaike (AIC) ou o critério Bayesiano de Schwarz (BIC) [1].
4. **Análise dos resíduos**, observando-se a normalidade e independência dos resíduos.
5. **Previsões**, com o modelo selecionado.

[1] Note que a escolha de valores p (através do PACF) e q (através ACF) influenciam-se mutuamente e, por exemplo, não pode haver uma autocorrelação parcial de ordem 1 com uma correlação de ordem 0.

3.4.1. Análise da estacionariedade da série

Vamos produzir uma série sintética simulando uma série ARIMA(2,1,3). A série original é não estacionária mas a série diferenciada de ordem 1 torna-se estacionária. Obtemos então nosso parâmetro $d = 1$.

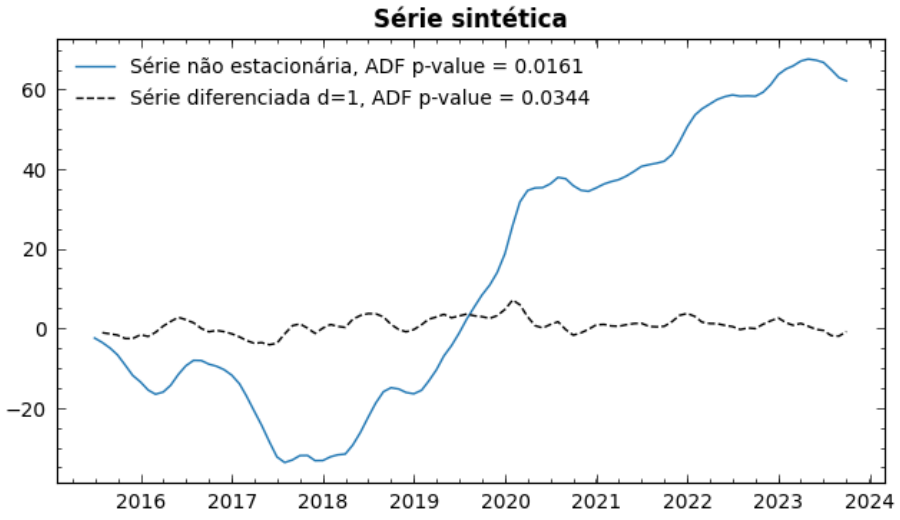
```
#@markdown
def simulate_ARIMA(phi = np.array([0]), theta = np.array([0]), d = 0, t = 0, mu = 0, sigma =
# adaptado de https://github.com/TOMILO87
    """ Simulate data from ARMA model (eq. 1.2.4):

    z_t = phi_1*z_{t-1} + ... + phi_p*z_{t-p} + a_t + theta_1*a_{t-1} + ... + theta_q*a_{t-

    with d unit roots for ARIMA model.

    Arguments:
    phi -- array of shape (p,) or (p, 1) containing phi_1, phi2, ... for AR model
    theta -- array of shape (q) or (q, 1) containing theta_1, theta_2, ... for MA model
```

```
d -- number of unit roots for non-stationary time series
t -- value deterministic linear trend
mu -- mean value for normal distribution error term
sigma -- standard deviation for normal distribution error term
n -- length time series
burn -- number of discarded values because series beginns without lagged terms
seed -- numpy random state

Return:
x -- simulated ARMA process of shape (n, 1)

Reference:
Time Series Analysis by Box et al.
"""
np.random.seed(seed)

# add "theta_0" = 1 to theta
theta = np.append(1, theta)

# set max lag length AR model
p = phi.shape[0]

# set max lag length MA model
q = theta.shape[0]

# simulate n + q error terms
a = np.random.normal(mu, sigma, (n + max(p, q) + burn, 1))

# create array for returned values
x = np.zeros((n + max(p, q) + burn, 1))

# initialize first time series value
x[0] = a[0]

for i in range(1, x.shape[0]):
    AR = np.dot(phi[0 : min(i, p)], np.flip(x[i - min(i, p) : i], 0))
    MA = np.dot(theta[0 : min(i + 1, q)], np.flip(a[i - min(i, q - 1) : i + 1], 0))
    x[i] = AR + MA + t

# add unit roots
if d != 0:
    ARMA = x[-n: ]
    m = ARMA.shape[0]
    z = np.zeros((m + 1, 1)) # create temp array

    for i in range(d):
```

```
            for j in range(m):
                z[j + 1] = ARMA[j] + z[j]
            ARMA = z[1: ]
        x[-n: ] = z[1: ]

    return x[-n: ]


df = simulate_ARIMA(phi = np.array([0.8]), theta = np.array([0.6, 0.2]), d=1, n = 100)
df = pd.DataFrame(df.flatten(),columns=['values'])
df.index = pd.date_range(start='6/1/2015', periods = 100, freq='M')
df.head()
```

```
              values
2015-06-30 -2.453310
2015-07-31 -3.577104
2015-08-31 -4.972580
2015-09-30 -6.694570
2015-10-31 -9.273456
```

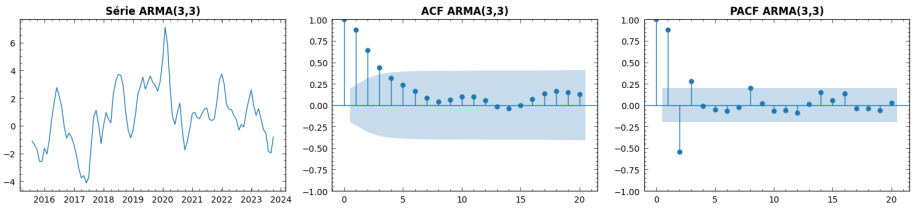
```
plt.plot(df, label = 'Série não estacionária, ADF p-value = ' + str(np.round(adfuller(ts)[1]
plt.plot(df.diff(), 'k--', label = 'Série diferenciada d=1, ADF p-value = ' + str(np.round(a

plt.legend()
plt.title('Série sintética')
plt.show()
```



3.4.2. Verificação da autocorrelação e autocorrelação parcial da série

Os parâmetros p e q são obtidos respectivamente da análise dos gráficos de autocorrelação parcial e autocorrelação, e obtemos assim os valores $p = 3$ e $q = 3$.

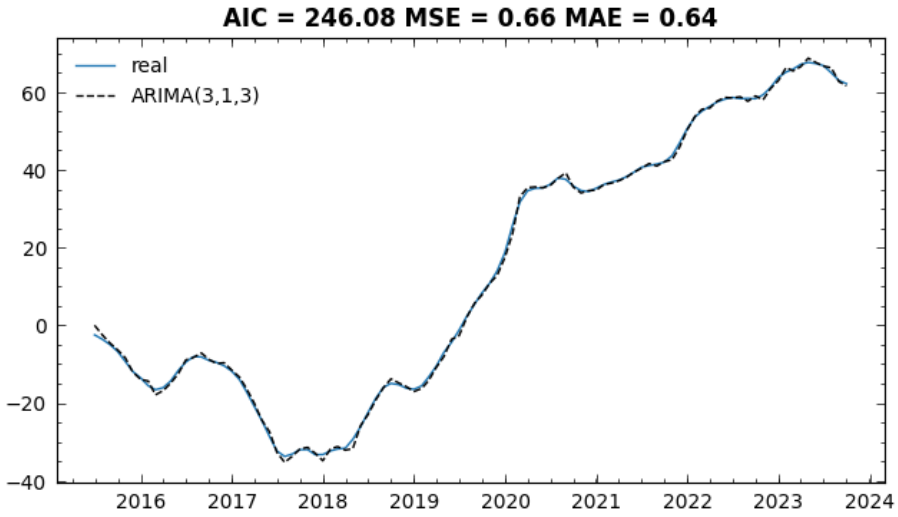
```
plot_ts_acf_pacf(df.diff().dropna(),'ARMA(3,3)')
```



3.4.3. Análise dos modelos e seleção dos parâmetros

Podemos então aplicar esses parâmetros e analisar o ajuste da série e métricas do modelo obtido.

```
from statsmodels.tsa.arima.model import ARIMA

p = 3; d = 1; q = 3
model = ARIMA(df, order=(p, d, q))
results = model.fit(method_kwargs={'maxiter':700})

plt.plot(df,label='real')
plt.plot(results.fittedvalues, 'k--', label='ARIMA(' + str(p) + ',' + str(d) + ',' + str(q)
plt.title(f'AIC = {results.aic:.2f} MSE = {results.mse:.2f} MAE = {results.mae:.2f}')

plt.legend()
plt.show()
```



O sumário do `statsmodels` fornece uma série de informações para análise do modelo.

```
print(results.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  values   No. Observations:                  100
Model:                 ARIMA(3, 1, 3)   Log Likelihood                -116.041
Date:                Sat, 13 Jan 2024   AIC                            246.083
Time:                        04:16:33   BIC                            264.248
Sample:                    06-30-2015   HQIC                           253.432
                         - 09-30-2023
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.4131      0.357      3.955      0.000       0.713       2.113
ar.L2         -1.2381      0.399     -3.106      0.002      -2.019      -0.457
ar.L3          0.6095      0.198      3.072      0.002       0.221       0.998
ma.L1          0.1014      0.378      0.269      0.788      -0.639       0.841
ma.L2          0.4784      0.297      1.612      0.107      -0.103       1.060
ma.L3          0.3000      0.208      1.442      0.149      -0.108       0.708
sigma2         0.5939      0.085      6.959      0.000       0.427       0.761
===================================================================================
Ljung-Box (L1) (Q):                   0.03   Jarque-Bera (JB):                 1.76
Prob(Q):                              0.87   Prob(JB):                         0.41
Heteroskedasticity (H):               0.70   Skew:                             0.31
Prob(H) (two-sided):                  0.32   Kurtosis:                         3.21
```

```
==============================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Ele consiste de 3 partes. Na primeira parte há informações gerais do modelo, como ordem do modelo, número de amostras e métricas como o AIC e BIC. Na segunda, há os coeficientes estimados, sua significância e intervalo de confiança. A terceira parte traz indicadores para análise dos resíduos.

Os gráficos das funções PACF e ACF sugerem valores de p e q a serem empregados. Mas podemos buscar alguma métrica de desempenho do modelo, como uma métrica de erro ou o AIC, para busca dos melhores parâmetros.

```
warnings.filterwarnings('ignore')
model_list = []; AIC_list = []

d = 1
for p in range(2,5):
  for q in range(2,5):
    model = ARIMA(df, order=(p, d, q))
    results = model.fit(method_kwargs={'maxiter':700})
    model_list.append('ARIMA(' + str(p) + ',' + str(d) + ',' + str(q) + ')')
    AIC_list.append(np.round(results.aic,4))

results_df = pd.DataFrame({'model': model_list, 'AIC': np.array(AIC_list) }).sort_values('AI
display(results_df)

warnings.filterwarnings('ignore')
```

```
          model       AIC
2  ARIMA(2,1,4)  248.3544
6  ARIMA(4,1,2)  247.5014
1  ARIMA(2,1,3)  247.3347
4  ARIMA(3,1,3)  246.0826
3  ARIMA(3,1,2)  246.0429
8  ARIMA(4,1,4)  245.6695
0  ARIMA(2,1,2)  245.4423
5  ARIMA(3,1,4)  245.4241
7  ARIMA(4,1,3)  244.4067
```

Assim, para o melhor desempenho pela métrica AIC podemos escolher os valores $p = 2, d = 1, q = 4$ e, então, analisar se os erros obtidos satisfazem as premissas do modelo.

```
p = 2; d = 1; q = 4
model = ARIMA(df, order=(p, d, q))
results = model.fit(method_kwargs={'maxiter':700})
print(results.summary())
```

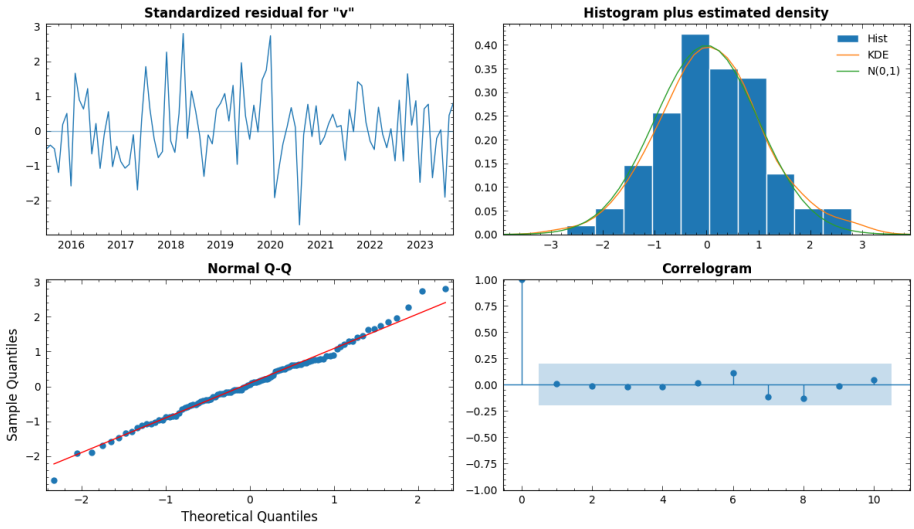
```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  values   No. Observations:                  100
Model:                 ARIMA(2, 1, 4)   Log Likelihood                -117.177
Date:                Sat, 13 Jan 2024   AIC                            248.354
Time:                        04:16:37   BIC                            266.520
Sample:                    06-30-2015   HQIC                           255.704
                         - 09-30-2023
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.4308      0.556      2.574      0.010       0.341       2.520
ar.L2         -0.4610      0.464     -0.994      0.320      -1.370       0.448
ma.L1          0.0946      0.562      0.168      0.866      -1.007       1.197
ma.L2         -0.3271      0.386     -0.847      0.397      -1.084       0.430
ma.L3         -0.3064      0.165     -1.857      0.063      -0.630       0.017
ma.L4         -0.1494      0.187     -0.799      0.424      -0.516       0.217
sigma2         0.6086      0.095      6.402      0.000       0.422       0.795
===================================================================================
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):                 0.84
Prob(Q):                              0.92   Prob(JB):                         0.66
Heteroskedasticity (H):               0.72   Skew:                             0.15
Prob(H) (two-sided):                  0.35   Kurtosis:                         3.34
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

3.4.4. Análise dos resíduos

Na parte dos indicadores de resíduos os valores `Prob` correspondem a p-valores das métricas indicadas (no exemplo, p-value Ljung-Box $= 0.92 > 0.05$ indica que os resíduos são ruído branco e p-value Jarque-Bera $= 0.66 > 0.05$ indica que os resíduos tem uma distribuição normal).

A análise dos resíduos ainda é normalmente complementada com análise dos gráficos da função `plot_diagnostics()` para, além da normalidade, verificarmos a independência dos valores de erro. No exemplo, como podemos ver, os erros não estão corelacionados.

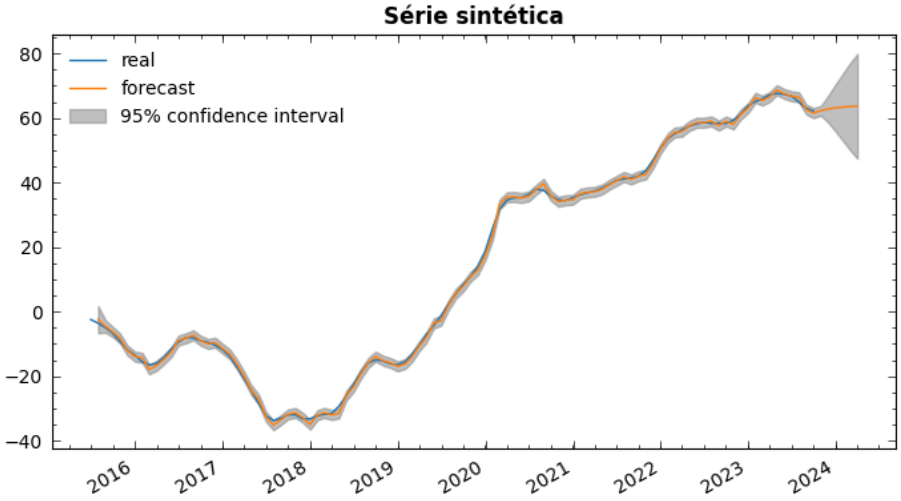
```
fig = plt.figure(figsize=(12,7))

results.plot_diagnostics(fig=fig)

plt.tight_layout()
plt.show()
```



3.4.5. Previsões

Uma vez selecionado o modelo podemos fazer previsões de valores futuros da série. Por exemplo, a predição para os próximos 6 meses.

```
from statsmodels.graphics.tsaplots import plot_predict

fig, ax = plt.subplots(figsize=(7,4))
ax.plot(df,label='real')
plot_predict(results, start=pd.to_datetime('2015-07-31'), end=pd.to_datetime('2024-03-31'),a
# plot_predict(results, start=1, end=len(ts) + 6,ax=ax) # aqui, funciona o mesmo modo

ax.set_title('Série sintética')
plt.tight_layout()
plt.show()
```




```
results.forecast(6)
```

```
2023-10-31    62.313898
2023-11-30    62.791316
2023-12-31    63.171009
2024-01-31    63.401228
2024-02-29    63.555601
2024-03-31    63.670354
Freq: M, Name: predicted_mean, dtype: float64
```

3.5. Exercícios e referências

As referências da seção anterior, Montgomery et al. (2015), Chatfield (1996), Morettin e Toloi (2006) são livros texto acessíveis e bastante úteis aqui para compreender os modelos ARIMA e seus submodelos, enquanto Box et. al. (2015) é um texto avançado.

Análise e implementações de modelos clássicos em R podem ser encontradas em Coghlan (2024), Shmueli et al. (2016) e Hyndman e Athanasopoulos (2018). Peixeiro (2022) traz implementações em Python com o pacote `statsmodels`, sendo mais próximo das implementações empregadas aqui. Nielsen (2019) é outro texto voltado para modelos práticos de séries temporais em Python, incluindo modelos de espaço de estados e de aprendizado de máquina.

Por último a documentação statsmodels (2024) é essencial para a compreensão das implementações e exercícios desta seção que podem ser acessados no material complementar do curso em exerc_parte2_arima.

4. Aprendizado de máquina

Os modelos autoregressivos, como vimos, baseiam-se em modelos de regressão linear. Modelos de aprendizado de máquina supervisionado também podem ser empregados para fazer previsões de valores futuros de séries temporais e têm tido um sucesso grande em fornecer previsões mais precisas para séries muito complexas, em particular modelos baseados em redes neurais profundas desenvolvidas com `PyTorch` ou `TensorFlow/Keras`, ou modelos híbridos como o Greykite (2024) (LinkedIn) e Prophet (2024) (Meta). As diferenças entre um modelo estatístico e um modelo de aprendizado de máquina residem na capacidade de precisão e explicabilidade de cada modelo e, de modo bastante simples, poderíamos dizer que trata-se de uma escolha entre fazer previsão mais precisa (modelos de aprendizado de máquina) ou compreender os processos subjacentes aos dados (modelos estatísticos), não excluindo o caso em que ambas as técnicas podem ser aplicadas.

Independente se empregamos um modelo de aprendizado de máquina clássico (como regressores de árvores de decisão, florestas de árvores aleatórias e k-vizinhos mais próximos) ou de redes neurais, incluindo aprendizado profundo (*deep learning*), o processo é bastante semelhante partindo por empregar os valores defasados da série como variáveis preditoras (como vimos no modelo autoregressivo anteriormente). Este processo está representado na figura 2.

Figura 2. Esquema geral do aprendizado de máquina (regressão) e da engenharia de atributos para séries temporais.

Em todos os casos, dado uma classe de modelos que desejamos empregar para aproximar uma série, o modelo é ajustado até que se alcance um erro mínimo por alguma métrica (como o MSE). Diferentemente de um modelo estatístico, como o ARIMA, a aplicação do modelo de não requer nenhum pressuposto como estacionariedade, não sazonalidade ou independência e normalidade dos resíduos, sendo seu único objetivo aproximar ao máximo a série que se deeja modelar. Esse aspecto de *caixa-preta* ou de *força-bruta*, principalmente nos modelos de aprendizado profundo, é uma das principais críticas na aplicação desses tipo de modelo em séries temporais. Entretanto, eles podem obter resultados de previsibilidade bastante bons em casos muito complexos.

São inúmeras as classes de modelos e bibliotecas disponíveis. Mas sendo o esquema geral de aplicação do método o mesmo, vamos apresentar uma aplicação com modelos clássicos de aprendizado empregando o pacote `scikit-learn` e outra de aprendizado profundo, com `TensorFlow/Keras`, o que para os nossos propósitos parece ser suficiente.

4.1. Scikit-learn

O `scikit-learn` é a principal biblioteca de aprendizado de máquina de modelos clássicos (não profundo), incluindo modelos de aprendizado supervisionado (classificação, regressão) e não supervisionado (clusterização, detecção de anomalias,

redução de dimensionalidade). Para predição de séries temporais empregamos modelos regressores e uma relação dos modelos disponíveis encontram-se a seguir.

```
# alguns dos estimadores de regressão disponíveis no scikit-learn
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from xgboost.sklearn import XGBRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import BayesianRidge
from sklearn.linear_model import ElasticNet
from sklearn.kernel_ridge import KernelRidge
from sklearn.linear_model import SGDRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
```

Cada um desses modelos tem um *princípio*, ou fundamento, mas que não entraremos em detalhe aqui. Assim, um estimador como `MLPRegressor` empregará uma rede neural para aproximar uma série, enquanto um `DecisionTreeRegressor` empregará uma árvore de decisão baseada na distribuição dos valores. Quaisquer desses estimadores podem ser igualmente aplicados.

4.1.1. Exemplo 1

Vamos empregar a série `co2` utilizada antes. Já criamos os dados defasados da série para serem as variáveis preditoras do modelos. Desse modo, faremos fazer o ajuste do modelo para,

$$(CO2_{t-1}, CO2_{t-2}) \longrightarrow CO2_t$$

```
df = co2
df.head()
```

Date	time	CO2	CO2_t-1	CO2_t-2
1981-07-01	2	340.32	342.08	342.74
1981-08-01	3	338.26	340.32	342.08
1981-09-01	4	336.52	338.26	340.32
1981-10-01	5	336.68	336.52	338.26
1981-11-01	6	338.19	336.68	336.52

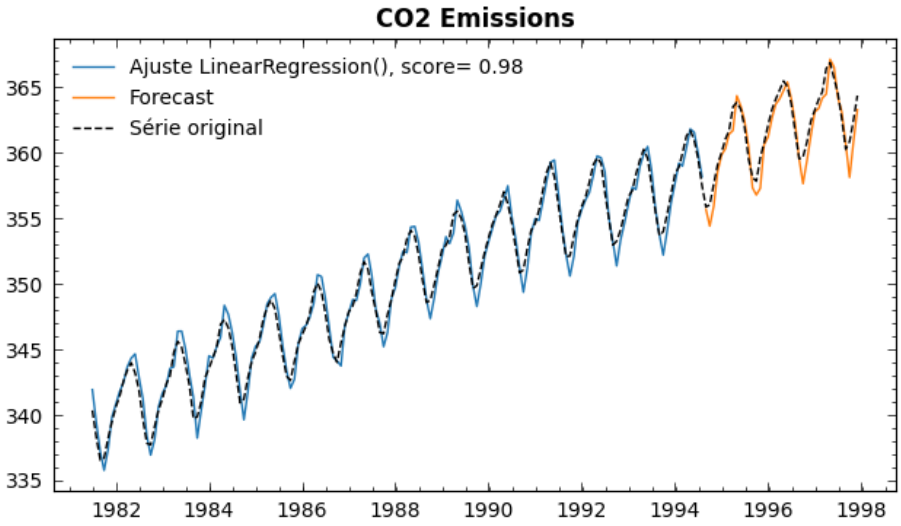
Um conceito importante na construção de modelos de aprendizado de máquina é o conceito de **conjuntos de treinamento e teste**. A ideia é separar dados empregados para o treinamento do modelo daqueles que são empregados para mensurar o seu desempenho. Há várias formas de fazer isso e, tipicamente, são empregados um percentual de dados aleatórios para treinamento (80% por exemplo) e teste (20%). Mas uma alternativa ao tratarmos de séries temporais

considerarmos para teste os dados mais recentes (por exemplo, os 20% que correspondem aos dados mais recentes).

```
# conjuntos de treinamento e teste
train_size = int(len(df) * 0.80)
train_data, test_data = df.iloc[0:train_size], df.iloc[train_size:len(df)]
```

O código a seguir é um padrão para o uso de estimadores do `scikit-learn` e o estimador (modelo) empregado a seguir pode ser substituído por quaisquer dos outros regressores disponíveis.

```
X_train, y_train, X_test, y_test = train_data[['CO2_t-1','CO2_t-2']], train_data[['CO2']], t

# modelo
model = LinearRegression()
# model = DecisionTreeRegressor() # try this!

# treinamento do modelo
model.fit(X_train, y_train)

# pedição com o modelo treinado
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# show results
plt.plot(df.iloc[0:train_size].index, train_predict,label='Ajuste ' + str(model) + ', score=
plt.plot(df.iloc[train_size:len(df)].index, test_predict,label='Forecast')
plt.plot(df.CO2,'k--',label='Série original',lw=1)

plt.legend(loc='upper left',fontsize=10)
plt.title('CO2 Emissions')
plt.show()

_ = error_measures(y_test.values.flatten(), test_predict.flatten())
```




```
MSE:   1.1527
MAE:   0.8495
RMSE:   1.0737
MAPE:   0.0023
```

4.1.2. Exemplo 2

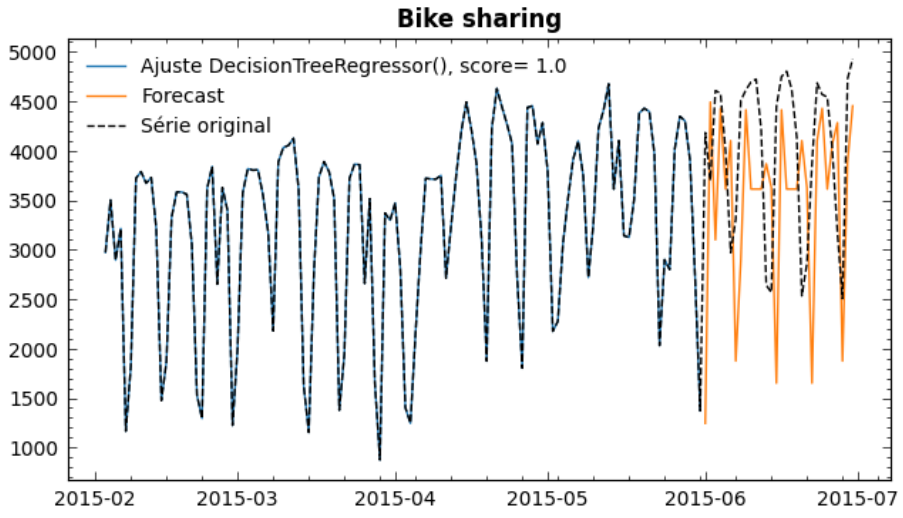
Um conjunto de dados mais complexo pode, entretanto, exigir um modelo mais robusto.

```
df = pd.read_csv(course_path + '/data/dados_bike_cnt.csv',index_col=0,parse_dates=True)

# valores defasados
df['cnt_t-1'] = df['cnt'].shift()
df['cnt_t-2'] = df['cnt'].shift(2)
df = df.dropna()
df.head()
```

```
             cnt  cnt_t-1  cnt_t-2
timestamp
2015-02-03  2972   3556.0   1117.0
2015-02-04  3502   2972.0   3556.0
2015-02-05  2894   3502.0   2972.0
2015-02-06  3214   2894.0   3502.0
2015-02-07  1165   3214.0   2894.0
```

```
# conjuntos de treinamento e teste
train_size = int(len(df) * 0.80)
train_data, test_data = df.iloc[0:train_size], df.iloc[train_size:len(df)]
```

```
X_train, y_train, X_test, y_test = train_data[['cnt_t-1','cnt_t-2']], train_data[['cnt']], t

# modelo
model = DecisionTreeRegressor()

# treinamento do modelo
model.fit(X_train, y_train)

# predição com o modelo treinado
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# show results
plt.plot(df.iloc[0:train_size].index, train_predict,label='Ajuste ' + str(model) + ', score=
plt.plot(df.iloc[train_size:len(df)].index, test_predict,label='Forecast')
plt.plot(df.cnt,'k--',label='Série original',lw=1)

plt.legend(loc='upper left',fontsize=10)
plt.title('Bike sharing')
plt.show()

_ = error_measures(y_test.values.flatten(), test_predict.flatten())
```




```
MSE:    1516080.7667
MAE:    1014.1000
RMSE:    1231.2923
MAPE:    0.2694
```

4.2. TensorFlow/Keras

O `TensorFlow` é uma biblioteca para a implementação de aprendizado profundo que permite criar modelos bastante complexos e uso de recursos de processamento em gpu, e o `Keras` fornece uma interface de mais alto nível para sua programação. Diferentes tipos de redes podem ser empregados e vamos empregar aqui dois modelos, um modelo de camadas sequenciais (ou rede multilayer perceptron) e um modelo LSTM (long short-term memory) que é uma arquitetura de rede neural recorrente bastante empregados em modelos de séries temporais. Modelos profundos vem tendo sucesso no tratamento de dados muitos complexos que e envolvem grandes volumes de dados, grande ruído e variabilidade, além de séries múltiplas de dados.

4.2.1. Exemplo 1

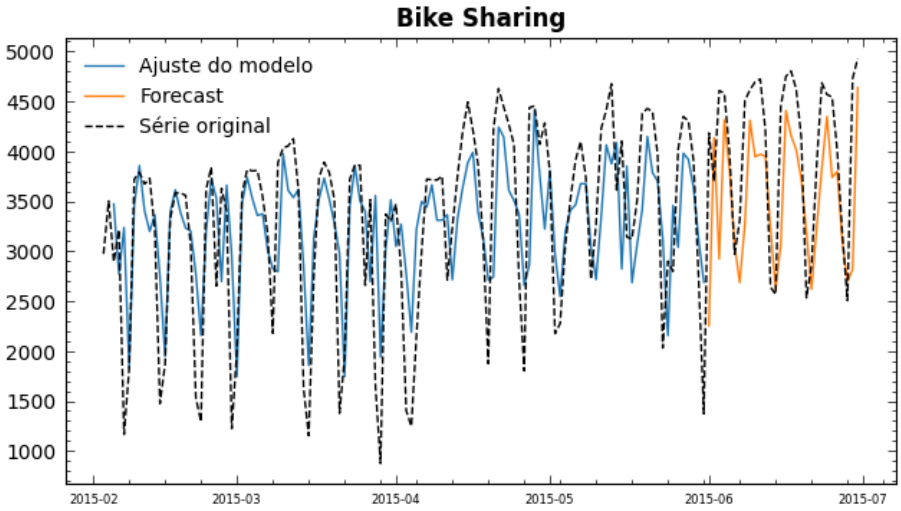
Empregamos os mesmos dados de Bike Sharing do exemplo anterior para buscarmos resultados melhores com um modelo neural. O uso de pacotes de aprendizado profundo é em geral mais complexo e o leitor pode buscar mais detalhes da programação envolvida aqui na documentação do Keras (2024).

Em um modelo neural é comum normalizarmos os dados (que são desnormalizados na previsão dos valores). O código a seguir implementa um modelo sequencial, mas que pode ser substituído pelo modelo recorrente LSTM comentado no código. Ambos os modelos apresentam um resultado melhor que o obtido pelo modelo clássico anterior.

```
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM

cnt_original = df[['cnt']] # salva antes da normalização
df = df[['cnt']]

# normalizando os dados
scaler = MinMaxScaler(feature_range=(0, 1))
df['cnt'] = scaler.fit_transform(df)

# valores defasados normalizados
df['cnt_t-1'] = df['cnt'].shift()
df['cnt_t-2'] = df['cnt'].shift(2)
df = df.dropna()

# conjuntos de treinamento e teste
train_size = int(len(df) * 0.80)
train_data, test_data = df.iloc[0:train_size], df.iloc[train_size:len(df)]

X_train, y_train, X_test, y_test = train_data[['cnt_t-1','cnt_t-2']], train_data[['cnt']], t
```

```
# modelo MLP ou LSTM
model = Sequential()

# MLP, MAPE ~ 0.14
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(64, activation='relu'))

# LSTM, MAPE ~ 0.16 # comente as linhas acima e empregue estas para o modelo LSTM
# model.add(LSTM(units=1024, return_sequences=True, input_shape=(X_train.shape[1], 1)))
# model.add(LSTM(units=32))

model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# treinamento do modelo
model.fit(X_train, y_train, epochs=50, batch_size=16, verbose=0)

# predição com o modelo treinado
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# invertendo a normalização
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)

y_test = scaler.inverse_transform(y_test)

# show results
plt.plot(df.iloc[0:train_size].index, train_predict,label='Ajuste do modelo ')
plt.plot(df.iloc[train_size:len(df)].index, test_predict,label='Forecast')
plt.plot(cnt_original,'k--',label='Série original',lw=1)

plt.xticks(fontsize=6)
plt.legend(loc='upper left',fontsize=10)
plt.title('Bike Sharing')
plt.show()

_ = error_measures(y_test.flatten(), test_predict.flatten())
```

```
4/4 [==============================] - 0s 7ms/step
1/1 [==============================] - 0s 26ms/step
```




```
MSE:    650567.3823
MAE:    621.2500
RMSE:    806.5776
MAPE:    0.1504
```

4.2.2. Exemplo 2

Até aqui empregamos somente os valores da própria série para a previsão de novos valores. Entretanto nossa série pode ser influenciada por variáveis exógenas. O volume de vendas dependendo da taxa de inflação, o consumo de energia dependendo no volume de chuvas no período e assim por diante. No exemplo a seguir incluímos o dados de humidade para melhorar a nossa previsão do volume de compartilhamento de bicicletas, uma vez que para uma maior humidade do clima (chuva) é esperado uma menor procura por bicicletas. O resultado melhora a previsão anterior, embora um modelo com mais elementos seja necessário.

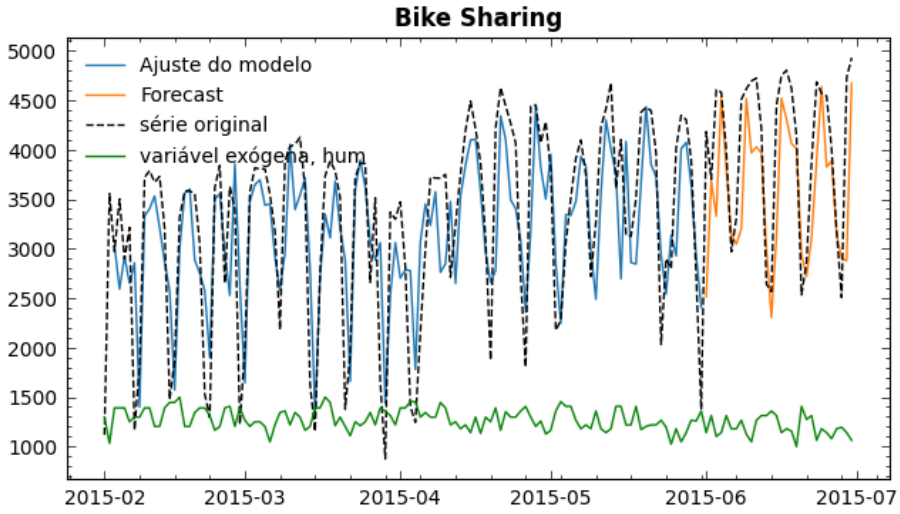
```
df_cnt_hum = pd.read_csv(course_path + '/data/dados_bike_cnt_hum.csv',index_col=0,parse_date
df_cnt_hum.head()
```

```
             cnt   hum
timestamp
2015-02-01  1117  87.0
2015-02-02  3556  70.0
2015-02-03  2972  93.0
2015-02-04  3502  93.0
2015-02-05  2894  93.0
```

```
df = df_cnt_hum[['cnt']]

# normalizando os dados
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
df['cnt'] = scaler.fit_transform(df)

# valores defasados normalizados
df['cnt_t-1'] = df['cnt'].shift()
df['cnt_t-2'] = df['cnt'].shift(2)

# incluindo uma variável exógena
scaler2 = MinMaxScaler(feature_range=(0, 1))
df[['hum']] = scaler2.fit_transform(df_cnt_hum[['hum']])

df = df.dropna()

# conjuntos de treinamento e teste
train_size = int(len(df) * 0.80)
train_data, test_data = df.iloc[0:train_size], df.iloc[train_size:len(df)]

X_train, y_train, X_test, y_test = train_data.drop(columns='cnt'), train_data[['cnt']], test

# modelo MLP, MAPE ~ 0.12
model = Sequential()

model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(64, activation='relu'))

model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# treinamento do modelo
model.fit(X_train, y_train, epochs=50, batch_size=16, verbose=0)

# predição com o modelo treinado
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)

# invertendo a normalização
train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)

y_test = scaler.inverse_transform(y_test)

# show results
plt.plot(df.iloc[0:train_size].index, train_predict,label='Ajuste do modelo ')
plt.plot(df.iloc[train_size:len(df)].index, test_predict,label='Forecast')
```

```
plt.plot(df_cnt_hum.cnt,'k--',label='série original',lw=1)

# deslocando a escala apenas para exibição dos dados de hum
hum = MinMaxScaler(feature_range=(1000, 1500)).fit_transform(df_cnt_hum[['hum']]).flatten()
plt.plot(df_cnt_hum.index,hum,'g-',label='variável exógena, hum',lw=1,alpha=0.95)

plt.legend(loc='upper left',fontsize=10)
plt.title('Bike Sharing')

plt.show()

_ = error_measures(y_test.flatten(), test_predict.flatten())
```




```
MSE:   517079.1224
MAE:   520.1264
RMSE:   719.0821
MAPE:   0.1259
```

4.3. Exercícios de referências

Kong et al. (2020) e Vanderplas (2016) trazem uma introdução prática ao aprendizado de máquina clássico (não profundo) em Python com `scikit-learn`. Entretanto, assim como a maioria dos livros textos de aprendizado de máquina, o foco está em modelos de classificação e regressão de valores, e não em séries temporais. Se você quiser saber mais sobre implementações de aprendizado profundo Zhang et al. (2022) é um texto avançado, completo e online que traz exemplos em `TensorFlow`, `PyTorch` e `MXNET`. Nielsen (2019) traz modelos práticos de aprendizado clássico e profundo de séries em Python, mas os modelos

de redes profundas empregam `MXNET`, bem menos empregado que as bibliotecas `TensorFlow` e `PyTorch`.

Embora tenham um *esquema geral* simples de implementação (engenharia dos atributos, ajuste do modelo e previsão), os possíveis modelos e suas implementações, sejam de aprendizado clássico ou profundo, são bastante variados e a consulta da documentação das bibliotecas scikit-learn (2024), TensoFlow (2024) e Keras (2024) é essencial para entendimento dos exemplos aqui e para que você possa realizar as suas implementações.

Para esta seção, os exercícios e soluções podem ser acessados em exerc_parte3_ml.

6. Conclusão

Neste minicurso você pode entender alguns dos conceitos fundamentais de séries temporais e como lidar com séries temporais em `Python` e `Pandas`. Pode aprender como construir e analisar modelos estatísticos do tipo ARIMA (incluídos os modelos AR, MA, ARMA) com a biblioteca `statsmodels`, e a partir deles você poderá facilmente evoluir modelos mais complexos como o SARIMA, SARIMAX. Você também aprendeu como aplicar modelos de aprendizado de máquina clássicos e de aprendizado profundo com as bibliotecas `scikit-learn` e `TensorFlow/Keras` a partir da engenharia dos atributos da série, incluindo o uso de variáveis exógenas.

Esse poderá ser um ferramental bastante útil para você, na vida profissional ou para seus projetos de pesquisa, e a ferramenta certa - aprendizado de máquina, ou estatística clássica - e o modelo específico a ser utilizado, em última análise, dependerá do contexto mais amplo do seu problema.

Há um grande número de temas que não foram abordados aqui como intervalos de confiância, diversos outros modelos estatísticos, critérios de seleção de modelos, conceitos de ajuste no aprendizado de máquina ou validação cruzada, etc. mas contamos que você encontrará aqui um ótimo ponto de partida para se aprofundar nesses pontos quando precisar.

Material complementar

Exercícios, conjuntos de dados e outros materiais do curso podem ser acessados em https://github.com/Introducao-Series-Temporais-em-Python/minicurso-SBC-SBSI-2024.

Referências

Box, G. E., Jenkins, G. M., Reinsel, G. C., Ljung, G. M. (2015). **Time series analysis: forecasting and control**. John Wiley & Sons.

Chatfield, C. (1996) **The analysis of time series: an introduction.** Chapman and hall/CRC.

Coghlan, A. (2024) **Welcome to a little book of R for time series**. Parasite Genomics Group, Cambridge, U.K. https://a-little-book-of-r-for-time-series.readthedocs.io/en/latest/. Acesso em: 12 jan. 2024.

De Gooijer, J. G., Hyndman, R. J. (2006) **25 years of time series forecasting.** International journal of forecasting, v. 22, n. 3, p. 443-473.

Greykite. Disponível em: https://linkedin.github.io/greykite/. Acesso em: 12 jan. 2024.

Hyndman, R. J., Athanasopoulos, G. (2018) **Forecasting: principles and practice**.

Keras. Disponível em: https://keras.io/. Acesso em: 12 jan. 2024.

Kong, Q., Siauw, T., Bayen, A. (2020) **Python Programming and Numerical Methods: A Guide for Engineers and Scientists**. Academic Press, 2020. Disponível em: https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html. Acesso em: 12 jan. 2024.

Montgomery, D. C., Jennings, C. L.; Kulahci, M. (2015) **Introduction to time series analysis and forecasting**. John Wiley & Sons.

Morettin, P. A., Toloi, C. (2006) **Análise de séries temporais**. In: Análise de séries temporais.

Nielsen, A. (2019) **Practical time series analysis: Prediction with statistics and machine learning**. O'Reilly Media.

Oliveira, R. (2022). **Visualizacao de Dados em Python**. ISBN: 978-65-5545-511-3. Coleção Conexão Inicial. Editora Mackenzie.

Oliveira, R., Albarracin, O. Y. E., Silva, G. R. (2024) **Introdução às Séries Temporais: Uma Abordagem Prática em Python** (in printing). Coleção Conexão Inicial. Editora Mackenzie. Disponível em: https://github.com/Introducao-Series-Temporais-em-Python Acesso em: 12 jan. 2024.

Pandas. Disponível em: https://pandas.pydata.org/. Acesso em: 12 jan. 2024.

Peixeiro, M. (2022) **Time Series Forecasting in Python**.

Prophet. Disponível em: https://facebook.github.io/prophet/. Acesso em: 12 jan. 2024.

scikit-learn. Disponível em: https://scikit-learn.org/stable/. Acesso em: 12 jan. 2024.

Shmueli, G., Lichtendahl Jr. K. C. (2016) **Practical time series forecasting with r: A hands-on guide**. Axelrod schnall publishers.

statsmodels. Disponível em: https://www.statsmodels.org/stable/index.html. Acesso em: 12 jan. 2024.

TensorFlow. Disponível em: https://www.tensorflow.org/. Acesso em: 12 jan. 2024.

Vanderplas, J. (2016) **Python data science handbook: Essential tools for working with data.** O'Reilly Media, Inc., 2016. Disponível em: https://jakevdp.github.io/PythonDataScienceHandbook/ Acesso em: 12 jan. 2024.

Zhang, A. et al. (2021) **Dive into deep learning.** arXiv preprint arXiv:2106.11342, 2021. Disponível em: https://d2l.ai/. Acesso em: 12 jan. 2024.