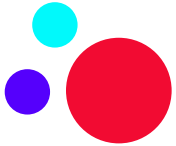
ML Time Series

infoShareAcademy.com



ML Time Series

Agenda

1. **Szeregi czasowe**
2. Modele liniowe i nieliniowe
3. Metryki

ML Time Series

Agenda

1. Szeregi czasowe
2. **Modele liniowe i nieliniowe**
3. Metryki

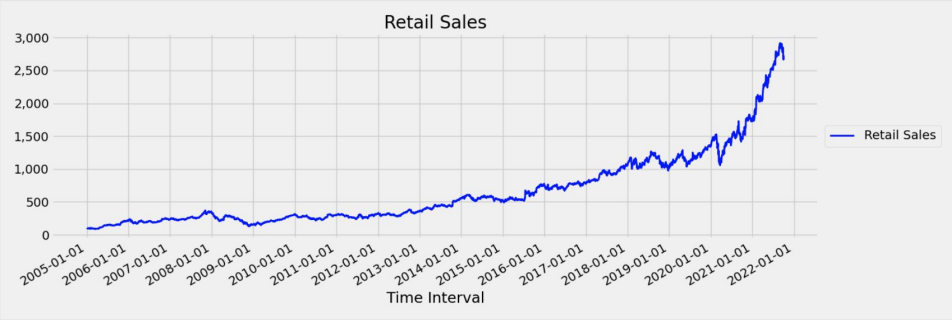
ML Time Series

Agenda

1. Szeregi czasowe
2. Modele liniowe i nieliniowe
3. **Metryki**

ML Time Series

Szeregi czasowe



infoShareAcademy.com

ML Time Series

Szeregi czasowe - zastosowanie

Serwisowanie:

Wykrywanie anomalii we wskaźnikach maszyn produkcyjnych pomaga uniknąć niespodziewanej awarii i reagować zawczasu.

Łańcuch dostaw:

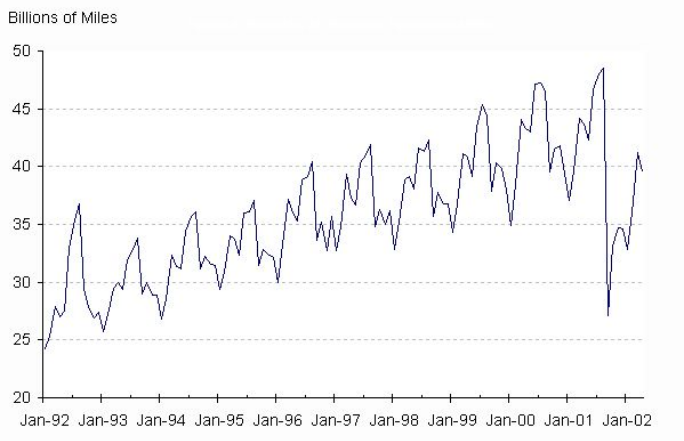
Przewidywanie przyszłego popytu na produkty firmy pozwala na zabezpieczenie surowców i efektywniejszą negocjację cen z dostawcami.

Marketing:

Śledzenie sprzedaży w czasie i czynników, które na nią wpływają, pomagają na precyzyjniejsze oszacowanie efektów kampanii reklamowych i trafniejsze decyzje dotyczące wykorzystania budżetu marketingowego.

ML Time Series

Komponenty szeregów czasowych



Sezonowość

Trend

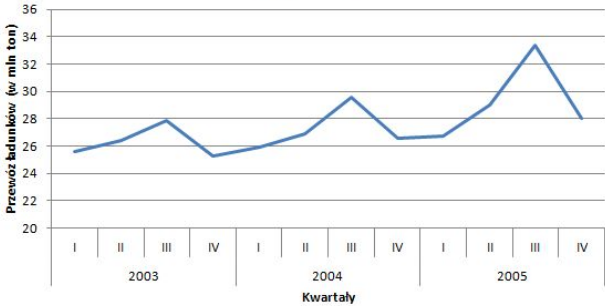
Cykliczność

Przypadkowość

https://www.bts.gov/archive/publications/transportation_indicators/october_2002/Special/A_Time_Series_Analysis_of_Domestic_Air_Seat_and_Passenger_Miles

ML Time Series

Sezonowość

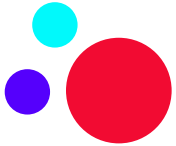


ML Time Series

Sezonowość

Główne cechy sezonowości:

1. Cykliczność w okresach krótkoterminowych.
2. Regularność i powtarzalność.
3. Wpływ na tendencje w danych.



ML Time Series

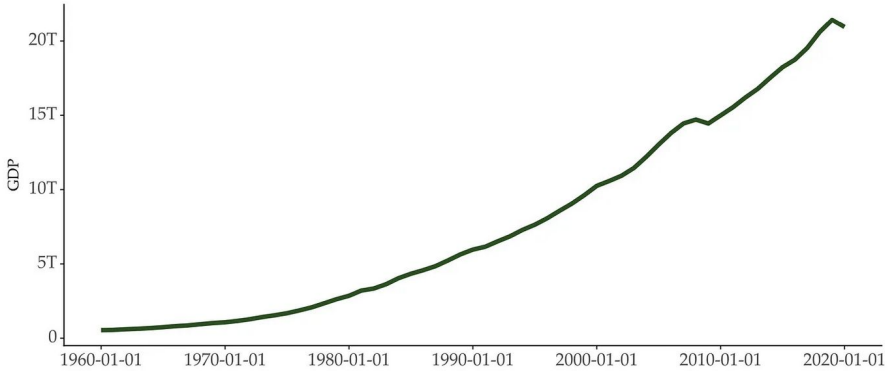
Sezonowość

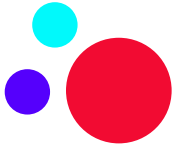
Jak radzić sobie z sezonowością:

1. Dekompozycja szeregów czasowych.
2. Profilowanie sezonowe.
3. Modelowanie sezonowości.

ML Time Series

Trend



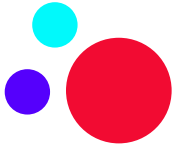


ML Time Series

Trend

Główne cechy trendu:

1. Długoterminowy charakter.
2. Kierunek zmiany.
3. Wpływ na analizę danych.



ML Time Series

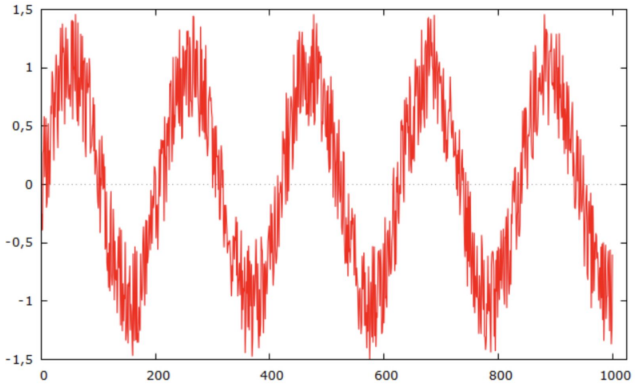
Trend

Jak radzić sobie z trendem:

1. Dekompozycja szeregów czasowych.
2. Modelowanie trendu.
3. Analiza różnic.

ML Time Series

Cykliczność



ML Time Series

Cykliczność

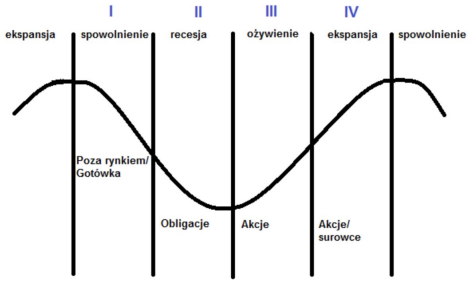
Główne cechy cykliczności:

1. Długość cyklu.
2. Niezależność od kalendarza.
3. Wzorce nieregularne.
4. Wpływ na analizę danych.

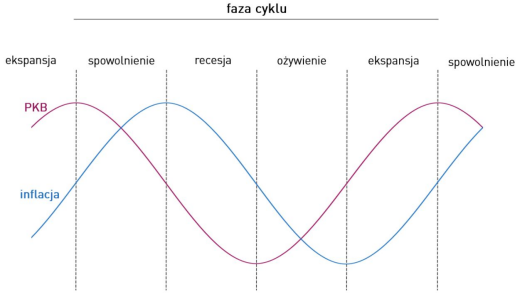
ML Time Series

Cykliczność

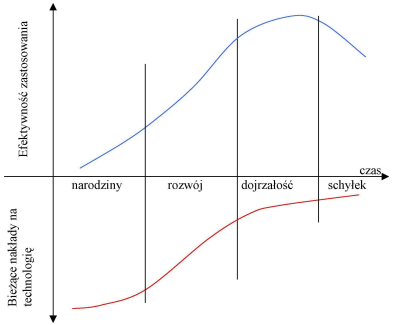
- Cykle koniunkturalne:

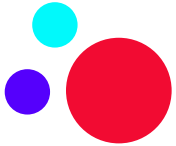


- Cykle inwestycyjne:



- Cykle technologiczne:





ML Time Series

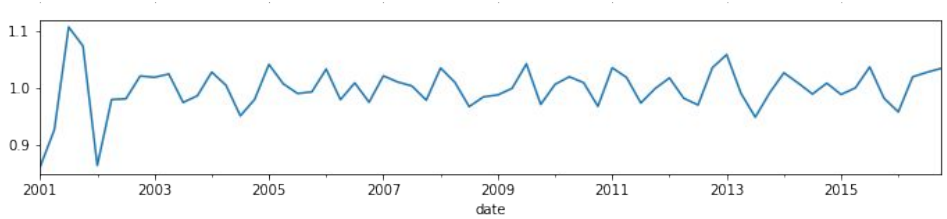
Cykliczność

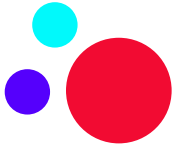
Jak radzić sobie z cyklicznością:

1. Analiza trendów długoterminowych.
2. Użycie zaawansowanych modeli.
3. Uwzględnianie czynników zewnętrznych.

ML Time Series

Przypadkowość



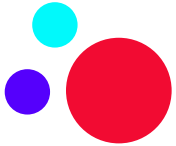


ML Time Series

Przypadkowość

Główne cechy przypadkowości:

1. Losowe fluktuacje.
2. Brak związku z innymi komponentami.
3. Wpływ na stabilność modeli.

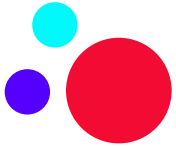


ML Time Series

Przypadkowość

Przykłady przypadkowości:

1. Szum w danych finansowych.
2. Zmienność w zachowaniu konsumentów.



ML Time Series

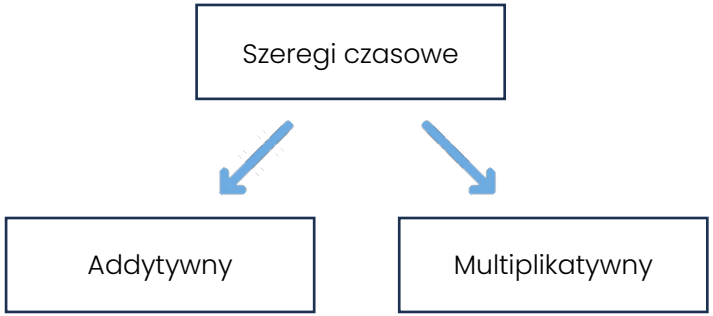
Przypadkowość

Jak radzić sobie z przypadkowością:

1. Modelowanie i eliminacja.
2. Rozpoznawanie anomaliach.
3. Uwzględnianie niestabilności.

ML Time Series

Typy szeregów czasowych

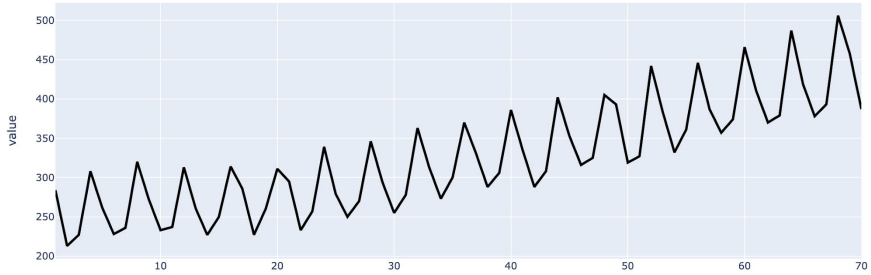


ML Time Series

Typ addytywny

$$Y_t = T_t + S_t + C_t + \mathcal{E}_t$$

Australian Beer Production



Time Series = Trend + Sezonowość + Cykliczność + Przypadkowość

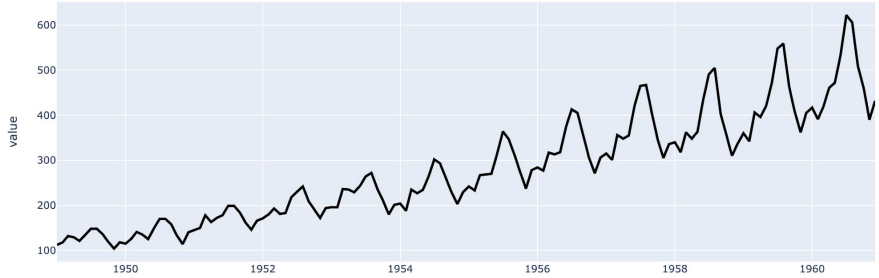
Przykład interpretacji: produkcja piwa w miesiąca letnich jest średnio o X butelek większa z tytułu działania zjawiska sezonowości.

ML Time Series

Typ multiplikatywny

$$Y_t = T_t \times S_t \times C_t \times \mathcal{E}_t$$

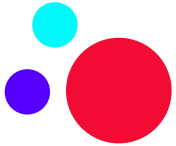
Airline Passenger Number



Time Series = Trend * Sezonowość * Cykliczność * Przypadkowość

Przykład interpretacji: liczba pasażerów linii lotniczych w miesiącach letnich jest średnio ok. X% większa z tytułu działania zjawiska sezonowości.

info Share ACADEMY

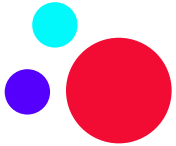


ML Time Series

Implementacja

```
AirPassengers = pd.read_csv("data/AirPassengers.csv")
AirPassengers["date"] = pd.to_datetime(AirPassengers["date"])
AirPassengers.rename(columns = {"value":"passengers"}, inplace = True)
```

	date	passengers
0	1949-01-01	112
1	1949-02-01	118
2	1949-03-01	132
3	1949-04-01	129
4	1949-05-01	121



ML Time Series

Implementacja

AirPassengers.sample(10)

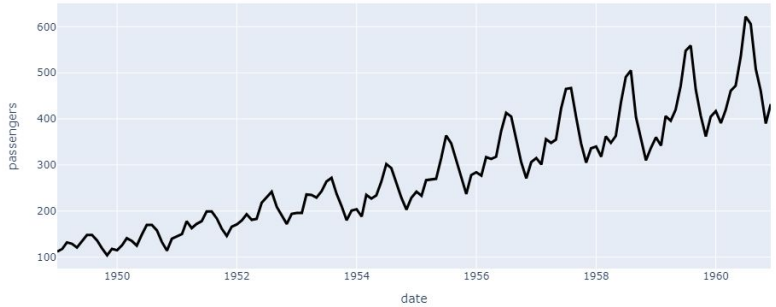
	date	passengers
7	1949-08-01	148
116	1958-09-01	404
36	1952-01-01	171
142	1960-11-01	390
9	1949-10-01	119
39	1952-04-01	181
101	1957-06-01	422
37	1952-02-01	180
109	1958-02-01	318
129	1959-10-01	407

ML Time Series

Implementacja

Time Series Plot

```
import plotly.express as px

# Wykres pasazerow
fig = px.line(AirPassengers, x = "date", y = "passengers")
# Kosmetyka
fig.update_traces(line_color='black', line_width = 3, fillcolor = 'white')
fig.show()
```



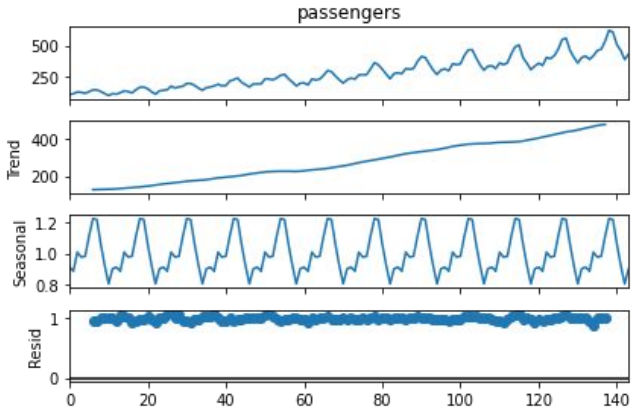
infoShare ACADEMY

ML Time Series

Implementacja

Decomposition

```
res = seasonal_decompose(AirPassengers["passengers"], model='multiplicable', period=12)
res.plot()
```

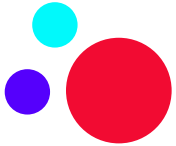


ML Time Series

Implementacja

Sezonowość w poszczególnych miesiącach roku
res._seasonal[0:12]

0	0.910230
1	0.883625
2	1.007366
3	0.975906
4	0.981378
5	1.112776
6	1.226556
7	1.219911
8	1.060492
9	0.921757
10	0.801178
11	0.898824



ML Time Series

Implementacja

```
# Sezonowość jest taka sama z roku na rok
res._seasonal[12:24]
```

12	0.910230
13	0.883625
14	1.007366
15	0.975906
16	0.981378
17	1.112776
18	1.226556
19	1.219911
20	1.060492
21	0.921757
22	0.801178
23	0.898824

ML Time Series

Implementacja

Wartosci pasażerów linii lotniczych wynikające z samego trendu
res._trend[12:24]

12	131.250000
13	133.083333
14	134.916667
15	136.416667
16	137.416667
17	138.750000
18	140.916667
19	143.166667
20	145.708333
21	148.416667
22	151.541667
23	154.708333

Zadanie 16.1 (instrukcja)

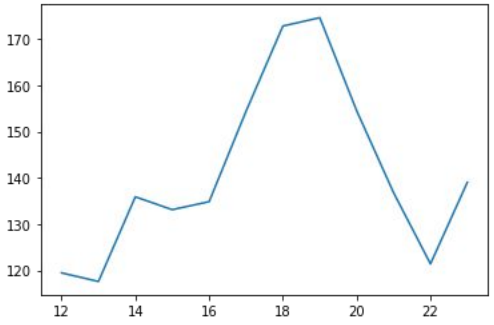
Zbiór danych zawiera dwie kolumny: "Month" (zawierającą daty) oraz "Passengers" (przedstawiającą liczbę pasażerów).

Przeprowadź dekompozycje sezonową zbioru, narysuj wykresy: oryginalnych danych, trendu, sezonowości oraz wykresu reszt po odjęciu sezonowości oraz trendu.

ML Time Series

Implementacja

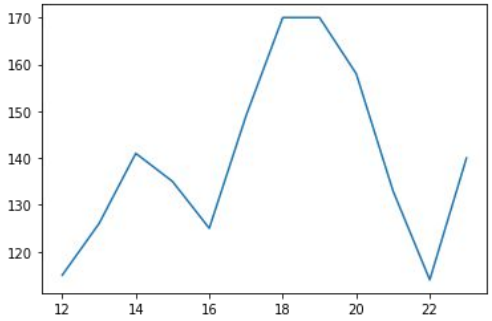
Trend*sezonowość doprowadza nas do wartości bliskich rzeczywistym
(* w przypadku szeregu multiplikatywnego ze słabymi wahaniami losowymi)
(res._trend[12:24] * res._seasonal[12:24]).plot()



ML Time Series

Implementacja

res._observed[12:24].plot()

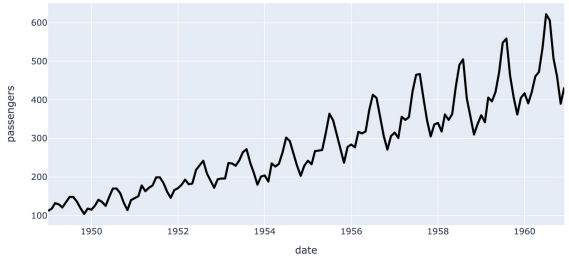


ML Time Series

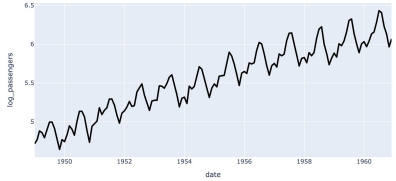
Stacjonarność

- Średnia i wariancja nie są zależne od czasu, czyli są stałe.

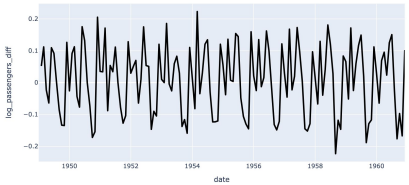
Oryginalny:

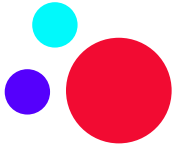


Po logarytmizacji:



Po logarytmizacji i różnicowaniu:





ML Time Series

Dlaczego?

- Używamy modelu, aby opisać rzeczywistość za pomocą prostej JEDNEJ funkcji matematycznej. Jeśli mamy szereg niestacjonarny, to w różnych momentach czasu jest potrzebna inna funkcja matematyczna do opisania danych, co mocno komplikuje sprawę, dlatego większość modeli zakłada, że szereg jest stacjonarny.
- W rzeczywistości biznesowej rzadko się spotyka szeregi stacjonarne. Np.: każdy szereg, który ma jakikolwiek trend już nie jest stacjonarny.

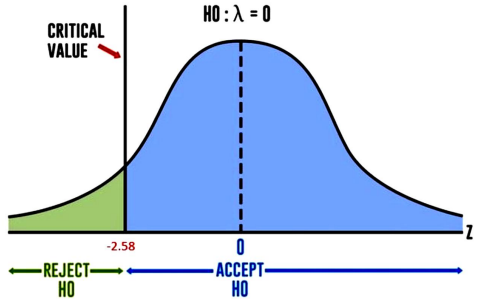
ML Time Series

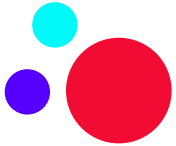
Test Augmented Dickey-Fuller (ADF)

Test ADF:

H0: szereg jest niestacjonarny (p-value > 0.05)

H1: szereg jest stacjonarny (p-value <= 0.05)



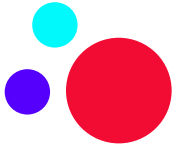


ML Time Series

Implementacja


```
from statsmodels.tsa.stattools import adfuller

    res = adfuller(AirPassengers["passengers"])
    p_value = res[1]
    if p_value > 0.05:
            print("Brak podstaw do odrzucenia hipotezy zerowej ==>
    Dane mają pewną zależność czasowa i nie mogą być uznane za stacjonarne")
    else:
            print("Dane są stacjonarne")
```

Brak podstaw do odrzucenia hipotezy zerowej ==> Dane mają pewną zależność czasowa i nie mogą być uznane za stacjonarne.



ML Time Series

Implementacja

Log to decrease variance

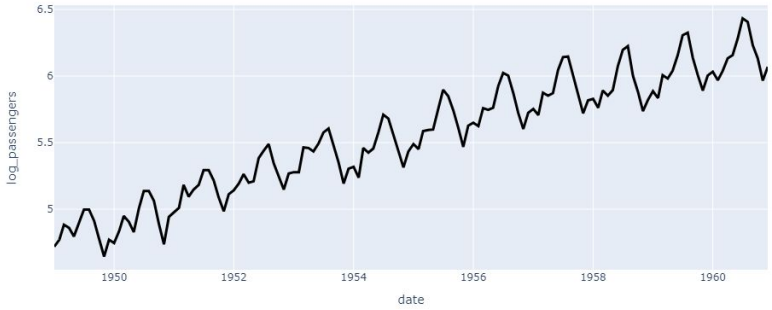
```
import numpy as np
AirPassengers["log_passengers"] = np.log(AirPassengers["passengers"])
```

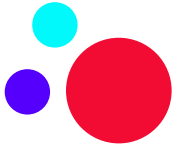
0	4.718499
1	4.770685
2	4.882802
3	4.859812
4	4.795791
	...
139	6.406880
140	6.230481
141	6.133398
142	5.966147
143	6.068426

ML Time Series

Implementacja

```
# Wykres pasażerów
fig = px.line(AirPassengers, x = "date", y = "log_passengers")
# Kosmetyka
fig.update_traces(line_color='black', line_width = 3, fillcolor = 'white')
fig.show()
```





ML Time Series

Implementacja

```
res = adfuller(AirPassengers["log_passengers"], maxlag = 12)
p_value = res[1]
if p_value > 0.05:
        print("Brak podstaw do odrzucenia hipotezy zerowej ⇒
Dane mają pewną zależność czasowa i nie mogą być uznane za
stacjonarne")
else:
        print("Dane sa stacjonarne")
```

Brak podstaw do odrzucenia hipotezy zerowej ==> Dane mają pewną zależność czasowa i nie mogą być uznane za stacjonarne.

ML Time Series

Implementacja

First differencing trying to make TS stationary:

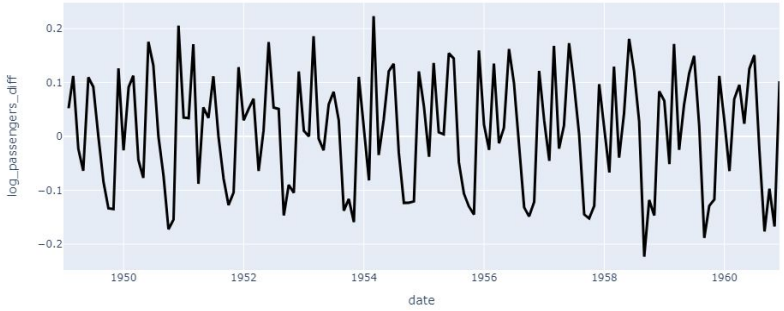
```
AirPassengers["log_passengers_diff"] = AirPassengers["log_passengers"] -
AirPassengers["log_passengers"].shift(1)
```

	log_passengers	log_passengers_diff
0	4.718499	NaN
1	4.770685	0.052186
2	4.882802	0.112117
3	4.859812	-0.022990
4	4.795791	-0.064022
...	...	...
139	6.406880	-0.026060
140	6.230481	-0.176399
141	6.133398	-0.097083
142	5.966147	-0.167251
143	6.068426	0.102279

ML Time Series

Implementacja

```
# Wykres pasażerów
fig = px.line(AirPassengers, x = "date", y = "log_passengers_diff")
# Kosmetyka
fig.update_traces(line_color='black', line_width = 3, fillcolor = 'white')
fig.show()
```



ML Time Series

Implementacja

```
res = adfuller(AirPassengers["log_passengers_diff"].dropna(), maxlag = 12)
p_value = res[1]
if p_value > 0.05:
    print("Brak podstaw do odrzucenia hipotezy zerowej ⇒
Dane mają pewną zależność czasowa i nie mogą być uznane za
stacjonarne")
else:
    print("Dane są stacjonarne")
```

Dane są stacjonarne.

Zadanie 16.2 (instrukcja)

Dla poniższego zbioru danych dotyczących liczby pasażerów linii lotniczych w zależności od czasu należy przeprowadzić proces transformacji, aby doprowadzić dane do stacjonarności.

```
import seaborn as sns
airline_data = sns.load_dataset("flights")
```

ML Time Series

Prognozowanie ARIMA

AR (Autoregressive component) – zważona suma wcześniejszych wartości (parameter p)

MA (Moving Average component) – zważona suma wcześniejszych błędów (parameter q)

I (Integration component) – różnicowanie (parameter d) (od wartości obecnych odejmujemy poprzednie)

ARIMA wymaga stacjonarności szeregu!

ACF - korelacja pomiędzy szeregiem czasowym a szeregiem czasowym przesuniętym o 1,2,3,... wartości do tyłu

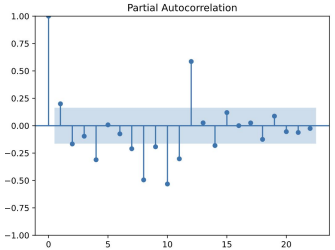
PACF – bezpośrednia korelacja pomiędzy szeregiem czasowym a szeregiem czasowym przesuniętym o 1,2,3,... wartości do tyłu, pomijając wpływ pośrednich przesunięć

ML Time Series

Prognozowanie ARIMA

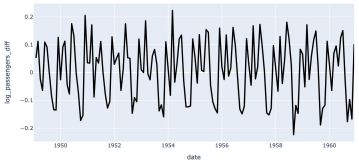
ARIMA(p, d, q)

PACF



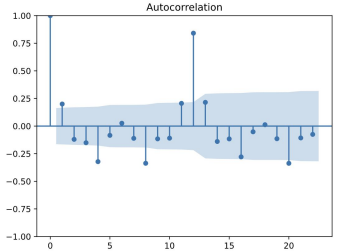
p = 2

Różnicowanie



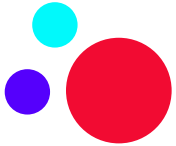
d = 1

ACF



q = 1

infoShareAcademy.com



ML Time Series

Implementacja


```
AirPassengers = AirPassengers.dropna()

from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from statsforecast.models import ARIMA
```

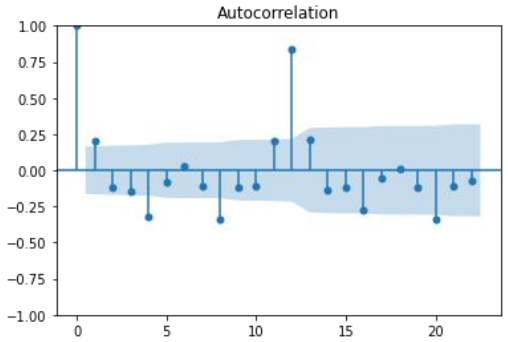
ML Time Series

Implementacja

AR and MA Order => ACF and PACF

MA = 1

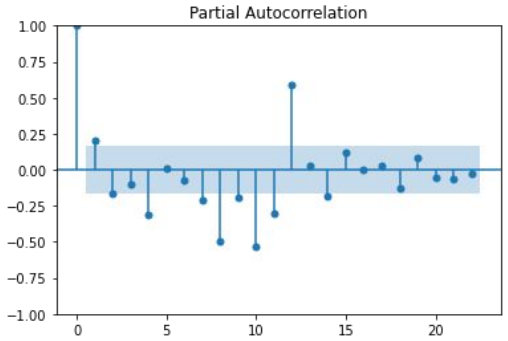
plot_acf(AirPassengers["log_passengers_diff"])

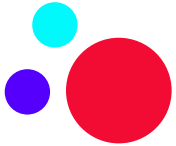


ML Time Series

Implementacja

```
# AR = 2
plot_pacf(AirPassengers["log_passengers_diff"])
```





ML Time Series

Implementacja

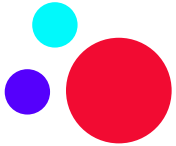
ARIMA modeling

```
def plot_fitted_vs_original_exp(dataframe: pd.DataFrame, date_col_name: str, original_log_col_name: str, fitted_log_col_name: str):
    """
    Plots time series with original and fitted series
    Parameters:
    dataframe (pd.DataFrame): df with fitted and original values
    date_col_name (str): name of date index
    original_log_col_name (str): name of logged original values
    fitted_log_col_name (str): name of logged fitted values
    Returns:
    Plotly chart
    """
    fig = go.Figure()
    fig.add_scatter(x=dataframe[date_col_name], y=np.exp(dataframe[original_log_col_name]), mode='lines', name = 'REAL', line=dict(color='blue'))
    fig.add_scatter(x=dataframe[date_col_name], y=np.exp(dataframe[fitted_log_col_name]), mode='lines', name = 'FITTED', line=dict(color='gray'))
    fig.update_traces(line_width = 3, fillcolor = 'white')
    fig.update_layout(title = "REAL VS FITTED")
    fig.show()
```

ML Time Series

Implementacja

```
def make_prediction_n_steps_ahead(model, orig_df:pd.DataFrame, date_col_name: str, forecast_steps: int, model_family: str):
    """
    Creates fcst_df for defined number of steps ahead
    Parameters:
    model: TS model
    orig_df (pd.DataFrame): original df
    date_col_name (str): name of date index
    forecast_steps (int): number of steps ahead to fcst
    model_family (str): ARIMA or ETS
    Returns:
    Dataframe with columns date_col_name and forecast
    """
    if model_family == "ARIMA":
    forecast = model.predict(h=forecast_steps)["mean"]
    elif model_family == "ETS":
    forecast = model.predict(params = model.params, start = orig_df.shape[0], end = orig_df.shape[0]+forecast_steps-1)

    forecast_dates = [orig_df[date_col_name].iloc[-1] + relativedelta(months=i+1) for i in range(forecast_steps)]
    forecast_df = pd.DataFrame({date_col_name: forecast_dates, 'forecast': forecast})
    return forecast_df
```



ML Time Series

Implementacja

```
def plot_forecast(orig_dataframe: pd.DataFrame, fcst_dataframe: pd.DataFrame, date_col_name: str, original_log_col_name: str, fitted_log_col_name: str, fcst_log_col_name: str):
    """
    Creates df for forecast, makes forecast and then plots fcst vs
    Parameters:
    dataframe (pd.DataFrame): df with fitted and original values
    data_col_name (str): name of date index
    original_log_col_name (str): name of logged original values
    fitted_log_col_name (str): name of logged fitted values
    fcst_log_col_name (str): name of fcst logged values
    Returns:
    Plotly chart
    """
    fig = go.Figure()
    fig.add_scatter(x=orig_dataframe[date_col_name], y=np.exp(orig_dataframe[original_log_col_name]), mode='lines', name = 'REAL', line=dict(color='blue'))
    fig.add_scatter(x=orig_dataframe[date_col_name], y=np.exp(orig_dataframe[fitted_log_col_name]), mode='lines', name = 'FITTED', line=dict(color='gray'))
    fig.add_scatter(x=fcst_dataframe[date_col_name], y=np.exp(fcst_dataframe[fcst_log_col_name]), mode='lines', name = 'FORECAST', line=dict(color='yellow'))
    fig.update_traces(line_width = 3, fillcolor = 'white')
    fig.update_layout(title = "REAL VS FITTED VS FORECAST")
    fig.show()
```

ML Time Series

Implementacja

```
# Modeling
model = ARIMA(order = (2, 1, 1))
res = model.fit(AirPassengers["log_passengers"].values)
```

ML Time Series

Implementacja

```
# Post processing
AirPassengers["FITTED"] = AirPassengers["log_passengers"].values - res.model_["residuals"]
AirPassengers["RESID"] = res.model_["residuals"]
```

```
1      4.765914
2      4.778547
3      4.905208
4      4.831698
5      4.770611
       ...
139    6.418108
140    6.336824
141    6.149232
142    6.110551
143    5.950443
Name: FITTED, Length: 143, dtype: float64
```

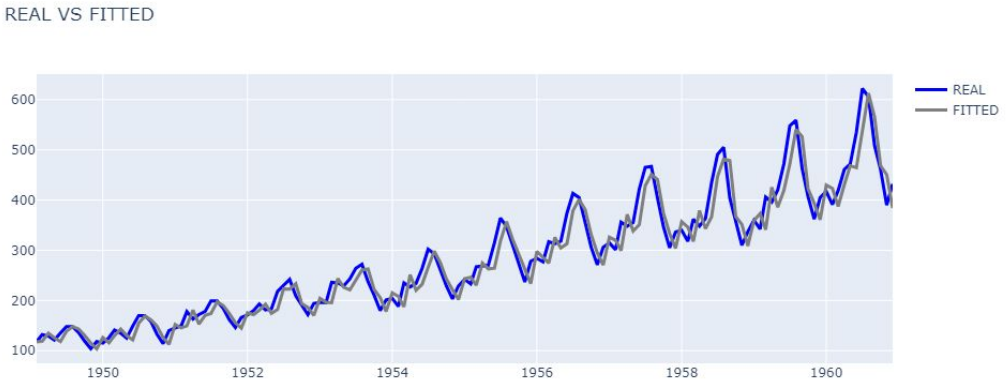
```
1      0.004771
2      0.104255
3     -0.045395
4     -0.035908
5      0.134664
       ...
139   -0.011228
140   -0.106342
141   -0.015834
142   -0.144404
143    0.117982
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

Plot results

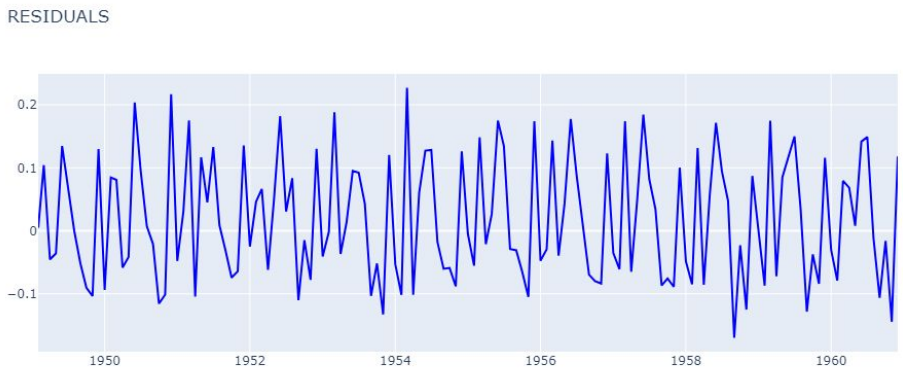
plot_fitted_vs_original_exp(AirPassengers, "date", "log_passengers", "FITTED")

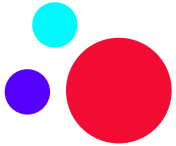


ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers['date'], y = AirPassengers['RESID'], mode = 'lines', name =
'REAL', line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```





ML Time Series

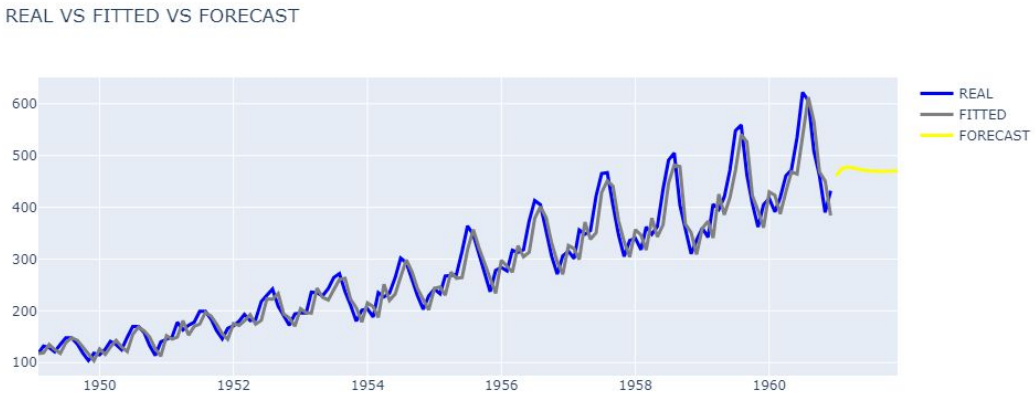
Implementacja


```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model, AirPassengers, date_col_name = "date", forecast_steps
= 12, model_family = "ARIMA")
```

ML Time Series

Implementacja

```
# Plot results
plot_forecast(AirPassengers, forecast_df, "date", "log_passengers", "FITTED", "forecast")
```



ML Time Series

Implementacja

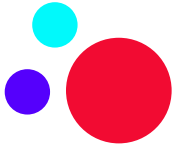
```
print("AIC: ", model.model_["aic"])
```

```
AIC: -247.76290993314
```

Zadanie 16.3 (instrukcja)

Dla poniższego zbioru danych wykonaj prognozowanie z użyciem modelu ARIMA.

```
airline_data = sns.load_dataset("flights")
```



ML Time Series

SARIMA

AR (Autoregressive component) – zważona suma wcześniejszych wartości (parameter p)

MA (Moving Average component) – zważona suma wcześniejszych błędów (parameter q)

I (Integration component) – różnicowanie (parameter d)

Seasonal AR - zważona suma wcześniejszych wartości z tego samego sezonu (parameter P)

Seasonal MA - zważona suma wcześniejszych błędów z tego samego sezonu (parameter Q)

Seasonal I – różnicowanie sezonowe (parameter D) (od wartości obecnych odejmujemy wartości z poprzedniego sezonu)

m – długość sezonu (co jaki czas powtarza się sezon) – w przypadku danych rocznych w ujęciu miesięcznym m = 12, czyli porównujemy ten miesiąc do tego samego miesiąca z roku poprzedniego (12 miesięcy wstecz)

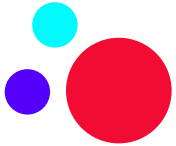
SARIMA wymaga stacjonarności szeregu!

ML Time Series

SARIMA

SARIMA(p, d, q) (P, D, Q, m)

- Tym razem dobierzemy parametry (p, d, q) (P, D, Q) używając GridSearch (Auto-SARIMA).
- Aby zrozumieć jaki model jest najlepszy użyjemy kryterium AIC, który jest miarą błędu dopasowania modelu i używa się go do porównania różnych modeli pomiędzy sobą. Im niższa wartość tym lepiej.

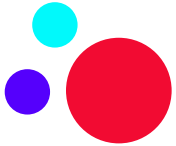


ML Time Series

Implementacja

Finding optimal parameters for p,q,d,P,Q,D

```
my_loop_df = pd.DataFrame(columns = ["sar", "sd", "sma",
"aic"])
for sar in [0,1,2]:
        for sma in [0,1,2]:
        for sd in [0,1,2]:
        model = ARIMA(order = (2, 1, 1), season_length=12,
seasonal_order=(sar,sd,sma))
        model.fit(AirPassengers["log_passengers"].values)
        my_loop_df.loc[len(my_loop_df)] = [sar, sd, sma,
model.model_["aic"]]
```

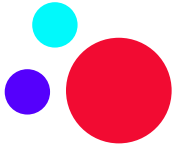


ML Time Series

Implementacja

my_loop_df.sort_values(by="aic")

	sar	sd	sma	aic
4	0.0	1.0	1.0	-475.404169
7	0.0	1.0	2.0	-473.786015
19	2.0	1.0	0.0	-470.734928
13	1.0	1.0	1.0	-469.295326
10	1.0	1.0	0.0	-468.975142
22	2.0	1.0	1.0	-466.382766
25	2.0	1.0	2.0	-465.720692
16	1.0	1.0	2.0	-462.817868
1	0.0	1.0	0.0	-443.640880
9	1.0	0.0	0.0	-421.559979
18	2.0	0.0	0.0	-389.242736
21	2.0	0.0	1.0	-386.498657
24	2.0	0.0	2.0	-371.225698
14	1.0	2.0	1.0	-365.000138



ML Time Series

Implementacja

Modelowanie

```
# Modeling
model = ARIMA(order = (2, 1, 1), season_length=12,
seasonal_order=(0,1,1))
res = model.fit(y = AirPassengers["log_passengers"].values)
```

ML Time Series

Implementacja

```
# Post processing
AirPassengers["FITTED"] = AirPassengers["log_passengers"].values -
res.model_["residuals"]
AirPassengers["RESID"] = res.model_["residuals"]
```

```
1      4.767930
2      4.881483
3      4.858969
4      4.795218
5      4.904719
       ...
139    6.439191
140    6.239405
141    6.103241
142    5.993782
143    6.083684
Name: FITTED, Length: 143, dtype: float64
```

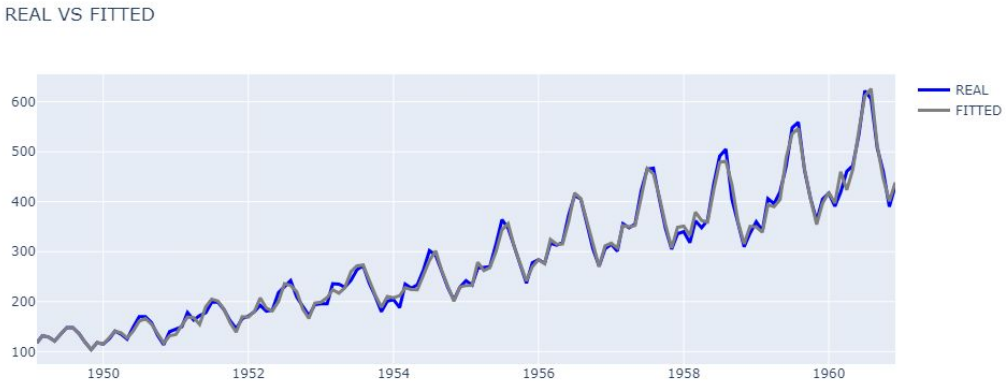
```
1      0.002754
2      0.001319
3      0.000844
4      0.000572
5      0.000556
       ...
139   -0.032311
140   -0.008923
141    0.030157
142   -0.027635
143   -0.015259
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

Plot results

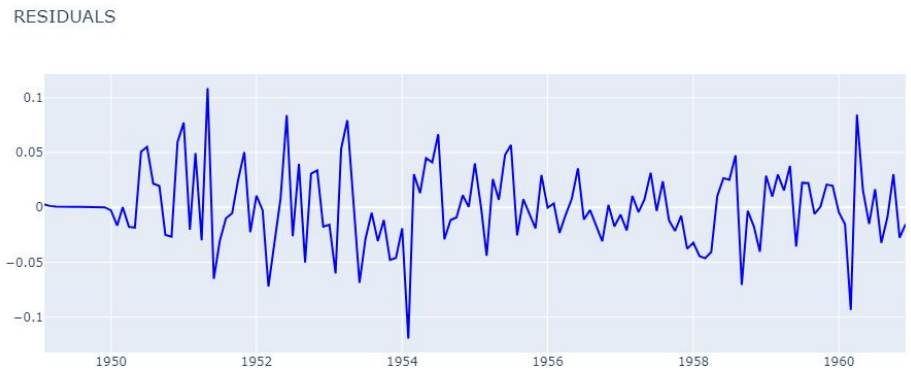
plot_fitted_vs_original_exp(AirPassengers, "date", "log_passengers", "FITTED")



ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers['date'], y = AirPassengers['RESID'], mode = 'lines', name =
'REAL', line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```



ML Time Series

Implementacja

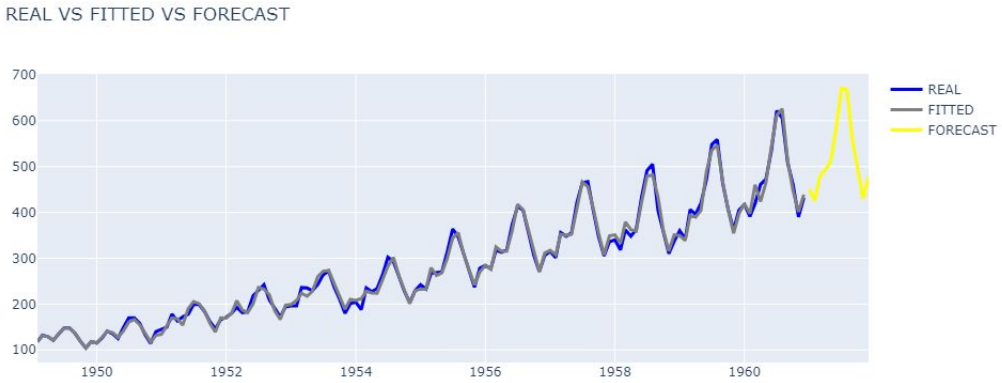
```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model, AirPassengers, date_col_name = "date",
forecast_steps = 12, model_family = "ARIMA")
```

	date	forecast
0	1961-01-01	6.109800
1	1961-02-01	6.053386
2	1961-03-01	6.170493
3	1961-04-01	6.199891
4	1961-05-01	6.233345
5	1961-06-01	6.369128
6	1961-07-01	6.508295
7	1961-08-01	6.503609
8	1961-09-01	6.324857
9	1961-10-01	6.209714
10	1961-11-01	6.063712
11	1961-12-01	6.168045

ML Time Series

Implementacja

```
# Plot results
plot_forecast(AirPassengers, forecast_df, "date", "log_passengers", "FITTED", "forecast")
```



ML Time Series

Implementacja

```
print("AIC: ", model.model_["aic"])
```

```
AIC: -475.40416942195895
```

Zadanie 16.4 (instrukcja)

Przy użyciu zbioru danych sns.load_dataset('flights') z biblioteki seaborn wykonaj prognozowanie przy użyciu modelu SARIMAX().

ML Time Series

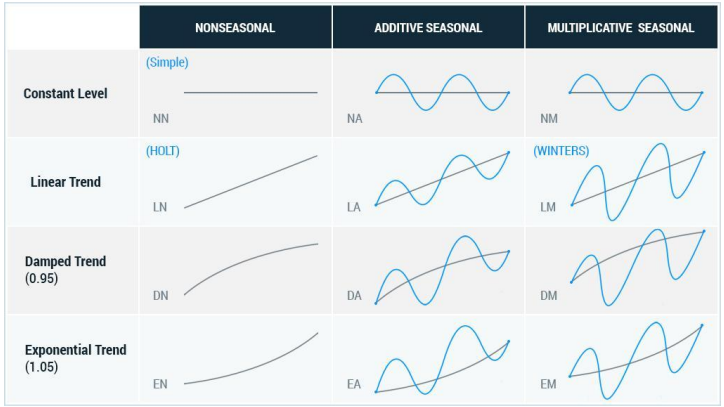
Prognozowanie Exponential Smoothing (ES)

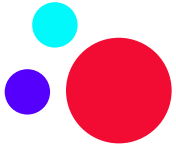
Stacjonarność nie jest wymagana.

Simple ES or Brown model - Jak nie ma sezonowości i trendu.

Double ES or Holt model - Jak mamy trend, ale nie sezonowość.

Triple ES or Winters model - Jak mamy trend i sezonowość.



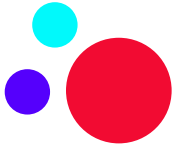


ML Time Series

Implementacja

Simple Exponential Smoothing or Brown model

```
from statsmodels.tsa.holtwinters import SimpleExpSmoothing

# Modeling
model = SimpleExpSmoothing(AirPassengers['log_passengers'])
res = model.fit(optimized=True)
```



ML Time Series

Implementacja


```
# Post processing
AirPassengers["FITTED"] = AirPassengers["log_passengers"].values - res.resid
AirPassengers["RESID"] = res.resid
```

```
1      4.770685
2      4.770685
3      4.882802
4      4.859812
5      4.795791
       ...
139    6.432940
140    6.406880
141    6.230481
142    6.133398
143    5.966147
Name: FITTED, Length: 143, dtype: float64
```

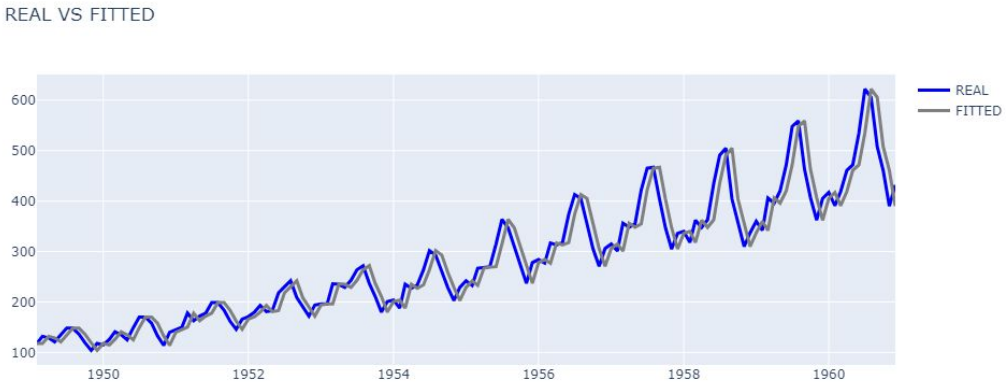
```
1      0.000000
2      0.112117
3     -0.022990
4     -0.064022
5      0.109484
       ...
139   -0.026060
140   -0.176399
141   -0.097083
142   -0.167251
143    0.102279
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

Plot results

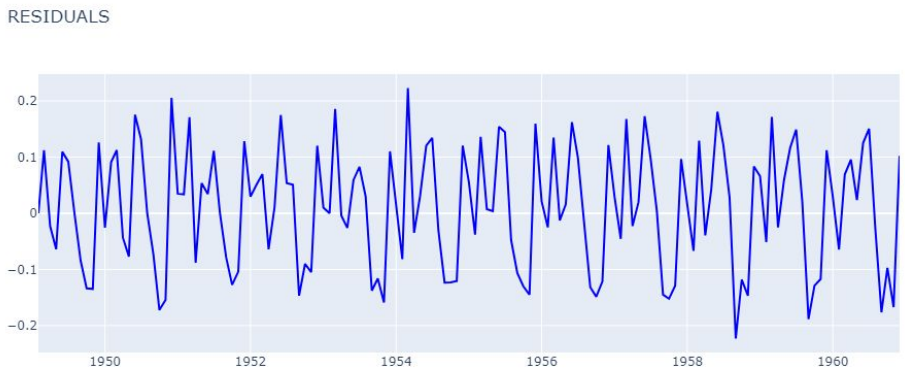
plot_fitted_vs_original_exp(AirPassengers, "date", "log_passengers", "FITTED")

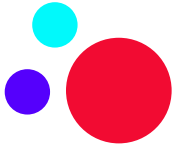


ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers['date'], y = AirPassengers['RESID'], mode = 'lines', name = 'REAL', line =
dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```





ML Time Series

Implementacja

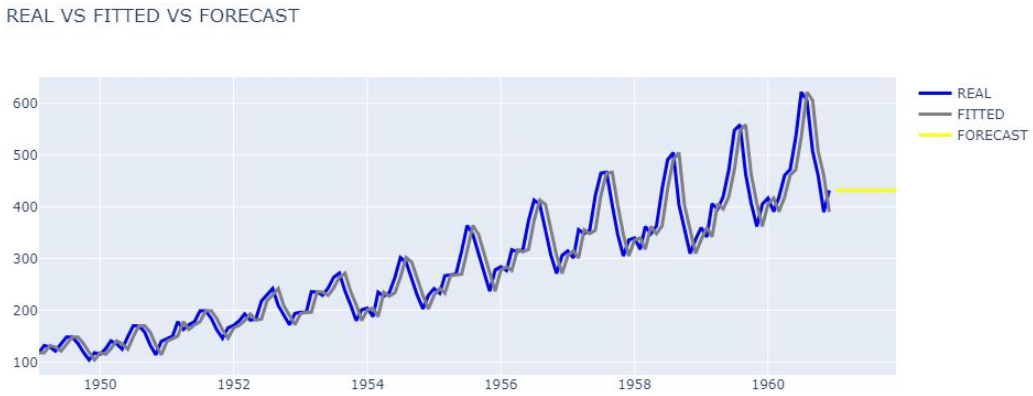

```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model, AirPassengers, date_col_name =
"date", forecast_steps = 12, model_family = "ETS")
```

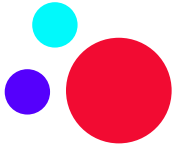
	date	forecast
0	1961-01-01	6.068426
1	1961-02-01	6.068426
2	1961-03-01	6.068426
3	1961-04-01	6.068426
4	1961-05-01	6.068426
5	1961-06-01	6.068426
6	1961-07-01	6.068426
7	1961-08-01	6.068426
8	1961-09-01	6.068426
9	1961-10-01	6.068426
10	1961-11-01	6.068426

ML Time Series

Implementacja

```
# Plot results
plot_forecast(AirPassengers, forecast_df, "date",
"log_passengers", "FITTED", "forecast")
```



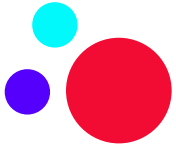


ML Time Series

Implementacja

Double Exponential Smoothing or Holt model

```
from statsmodels.tsa.holtwinters import Holt

# Modeling
model = Holt(AirPassengers['log_passengers'])
# res = model.fit(smoothing_level=0.1, smoothing_trend=0.1)
res = model.fit(smoothing_level=0.1, smoothing_trend=0.5)
# res = model.fit(optimized = True)
```



ML Time Series

Implementacja

```
# Post processing
AirPassengers["FITTED"] = AirPassengers["log_passengers"].values - res.resid
AirPassengers["RESID"] = res.resid
```

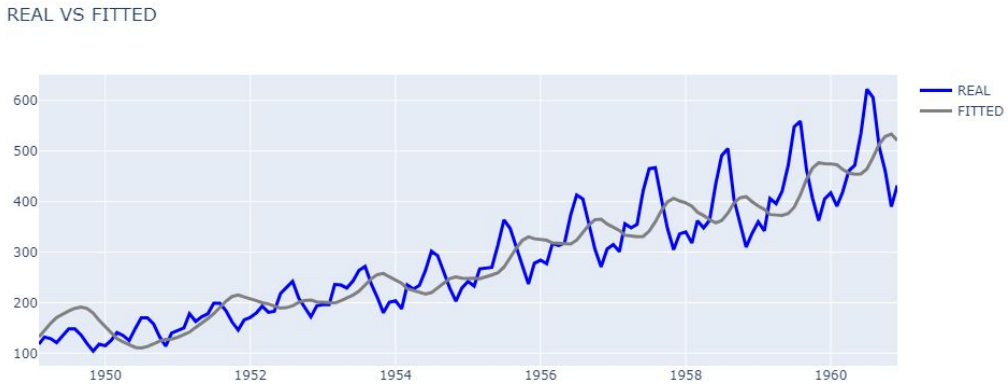
```
1      4.882802
2      4.978102
3      5.070318
4      5.140489
5      5.180005
      ...
139    6.189361
140    6.241477
141    6.270193
142    6.279488
143    6.255462
Name: FITTED, Length: 143, dtype: float64
```

```
1      -0.112117
2      -0.095300
3      -0.210506
4      -0.344698
5      -0.274730
      ...
139     0.217519
140    -0.010996
141    -0.136794
142    -0.313341
143    -0.187036
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

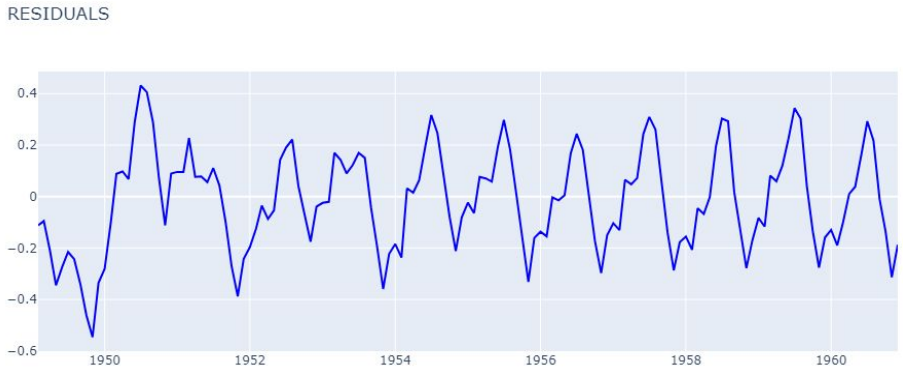
```
# Plot results
plot_fitted_vs_original_exp(AirPassengers, "date", "log_passengers", "FITTED")
```

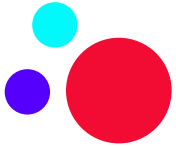


ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers['date'], y = AirPassengers['RESID'], mode = 'lines', name =
'REAL', line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```





ML Time Series

Implementacja

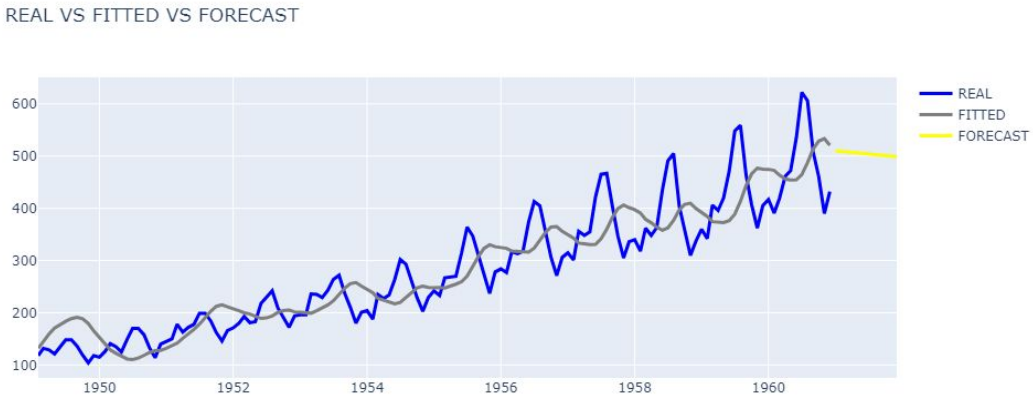

```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model, AirPassengers, date_col_name =
"date", forecast_steps = 12, model_family = "ETS")
```

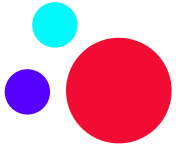
	date	forecast
0	1961-01-01	6.234715
1	1961-02-01	6.232671
2	1961-03-01	6.230627
3	1961-04-01	6.228583
4	1961-05-01	6.226539
5	1961-06-01	6.224495
6	1961-07-01	6.222452
7	1961-08-01	6.220408
8	1961-09-01	6.218364
9	1961-10-01	6.216320
10	1961-11-01	6.214276
11	1961-12-01	6.212232

ML Time Series

Implementacja

# Plot results
plot_forecast(AirPassengers, forecast_df, "date", "log_passengers", "FITTED", "forecast")



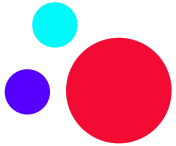


ML Time Series

Implementacja

Triple Exponential Smoothing or Winters model

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Modeling
model = ExponentialSmoothing(AirPassengers['log_passengers'], trend =
'add', seasonal = 'add', seasonal_periods = 12)
res = model.fit()
```



ML Time Series

Implementacja

```
# Post processing
AirPassengers["FITTED"] = AirPassengers["log_passengers"].values - res.resid
AirPassengers["RESID"] = res.resid
```

```
1      4.766726
2      4.895139
3      4.858676
4      4.809498
5      4.915412
       ...
139    6.433156
140    6.226794
141    6.096055
142    5.980444
143    6.078553
Name: FITTED, Length: 143, dtype: float64
```

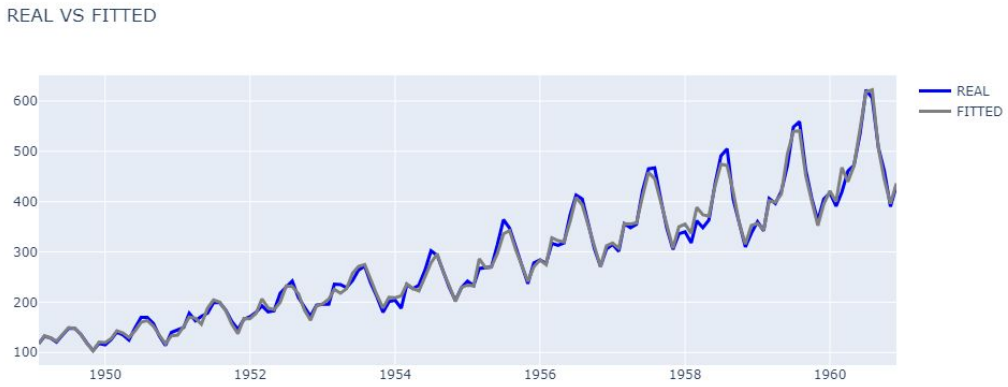
```
1      0.003959
2     -0.012337
3      0.001136
4     -0.013707
5     -0.010137
       ...
139   -0.026276
140    0.003687
141    0.037343
142   -0.014297
143   -0.010127
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

Plot results

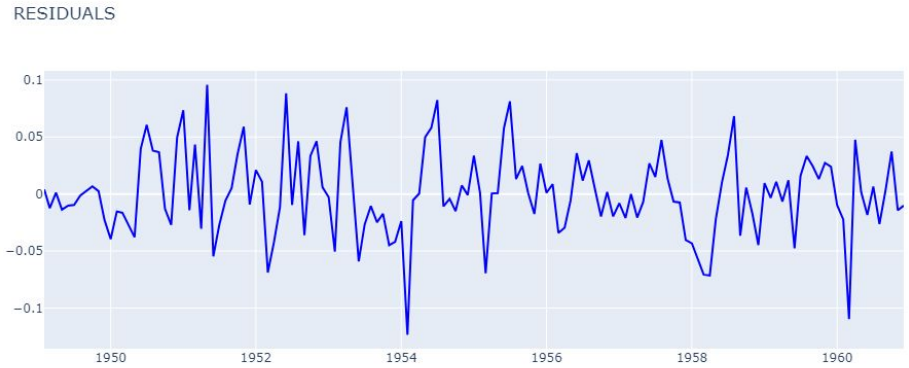
plot_fitted_vs_original_exp(AirPassengers, "date", "log_passengers", "FITTED")



ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers['date'], y = AirPassengers['RESID'], mode = 'lines', name =
'REAL', line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```



ML Time Series

Implementacja

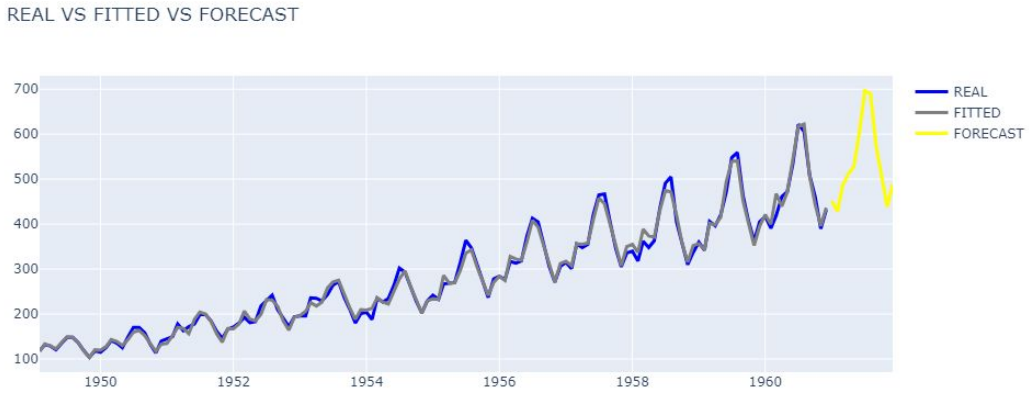
```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model,
AirPassengers, date_col_name = "date", forecast_steps = 12,
model_family = "ETS")
```

	date	forecast
0	1961-01-01	6.112998
1	1961-02-01	6.059285
2	1961-03-01	6.184799
3	1961-04-01	6.236524
4	1961-05-01	6.266887
5	1961-06-01	6.402478
6	1961-07-01	6.546620
7	1961-08-01	6.535975
8	1961-09-01	6.353303
9	1961-10-01	6.236410
10	1961-11-01	6.083023
11	1961-12-01	6.193113

ML Time Series

Implementacja

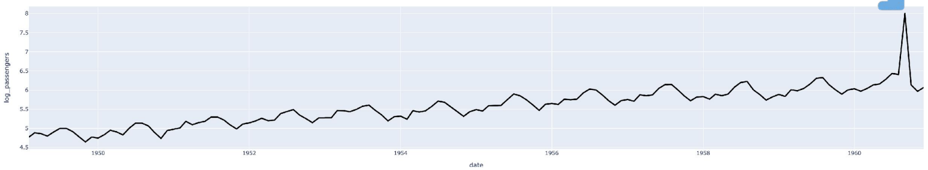
```
# Plot results
plot_forecast(AirPassengers, forecast_df, "date",
"log_passengers", "FITTED", "forecast")
```

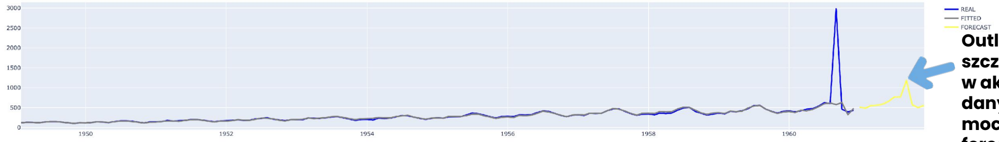


ML Time Series

Outliery i SARIMAX

Czarny łabędź



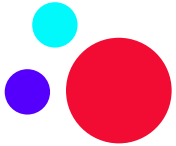


Outlier, szczególnie w aktualnych danych, może mocno zaburzyć forecast.

Oznaczenie outliera daje znaka modelowi o wyjątkowej sytuacji, która ma nie być brana pod uwagę w forecaście.

FLAG 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

info Share ACADEMY



ML Time Series

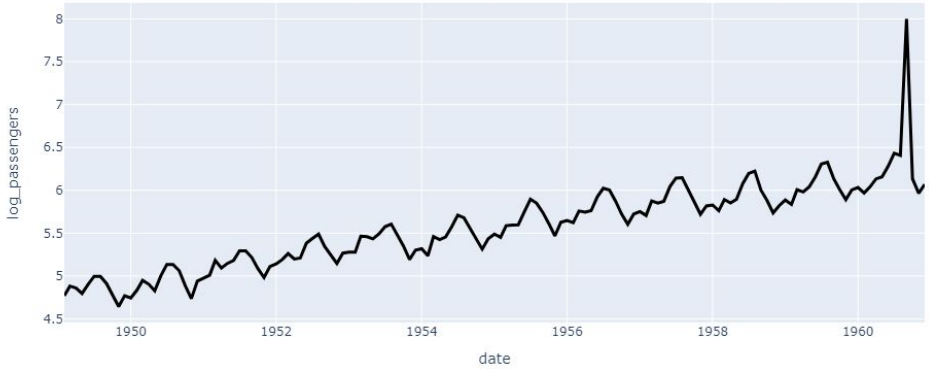
Implementacja

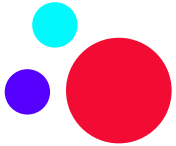
```
AirPassengers_outliers = AirPassengers.drop(columns = ["FITTED", "RESID", "log_passengers_diff", "passengers"]).copy()
AirPassengers_outliers.loc[140, "log_passengers"] = 8
```

ML Time Series

Implementacja

```
# Wykres pasażerów
fig = px.line(AirPassengers_outliers, x = "date", y = "log_passengers")
# Kosmetyka
fig.update_traces(line_color='black', line_width = 3, fillcolor = 'white')
fig.show()
```

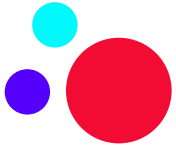




ML Time Series

Implementacja

```
# Modeling
model = ARIMA(order = (2, 1, 1), season_length=12,
seasonal_order=(0,1,1))
res = model.fit(y = AirPassengers_outliers["log_passengers"].values)
```



ML Time Series

Implementacja


```
# Post processing
AirPassengers_outliers["FITTED"] = AirPassengers_outliers["log_passengers"].values - res.model_["residuals"]
AirPassengers_outliers["RESID"] = res.model_["residuals"]
```

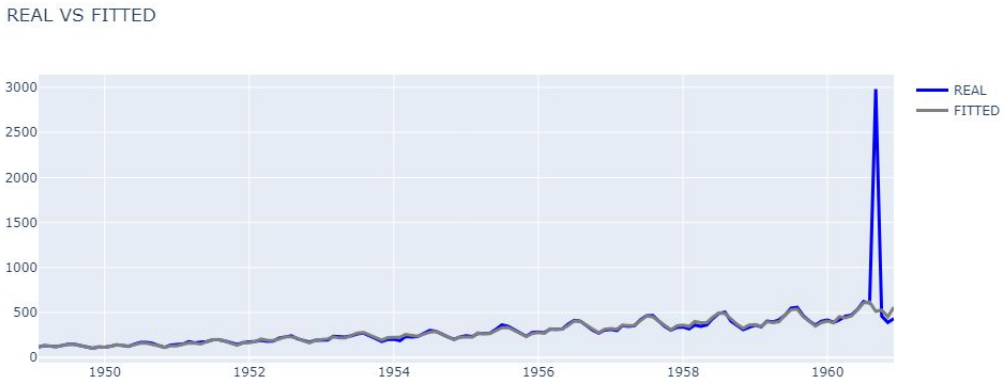
```
1      4.767930
2      4.881483
3      4.858969
4      4.795218
5      4.904719
       ...
139    6.425321
140    6.243641
141    6.267654
142    6.116089
143    6.326839
Name: FITTED, Length: 143, dtype: float64
```

```
1      0.002754
2      0.001319
3      0.000844
4      0.000572
5      0.000556
       ...
139   -0.018441
140    1.756359
141   -0.134256
142   -0.149942
143   -0.258414
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

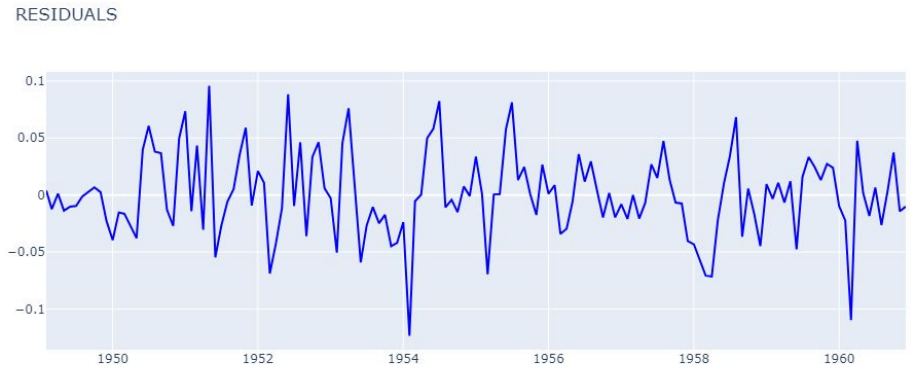
```
# Plot results
plot_fitted_vs_original_exp(AirPassengers_outliers, "date", "log_passengers", "FITTED")
```

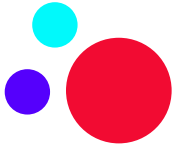


ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers_outliers['date'], y =
AirPassengers_outliers['RESID'], mode = 'lines', name = 'REAL',
line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```





ML Time Series

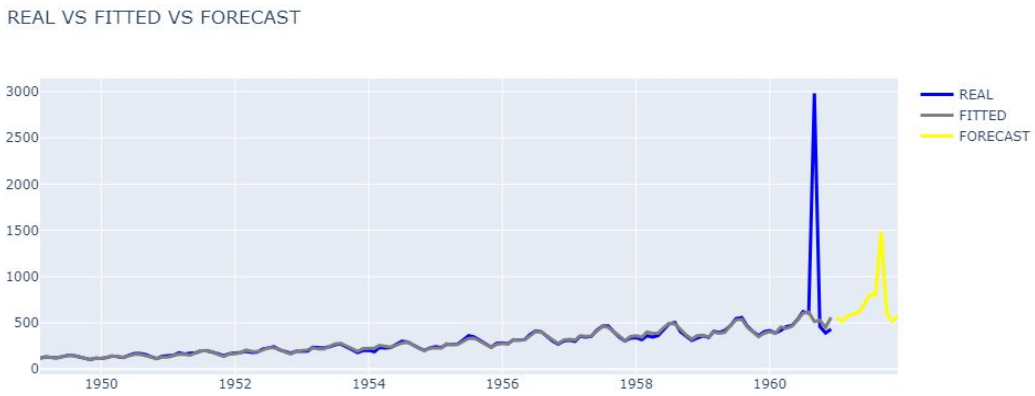
Implementacja

```
# Make forecast
forecast_df = make_prediction_n_steps_ahead(model,
AirPassengers_outliers, date_col_name = "date", forecast_steps
= 12, model_family = "ARIMA")
```

ML Time Series

Implementacja

# Plot results
plot_forecast(AirPassengers_outliers, forecast_df, "date", "log_passengers", "FITTED", "forecast")

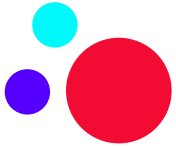


ML Time Series

Implementacja

```
print("AIC: ", model.model_["aic"])
```

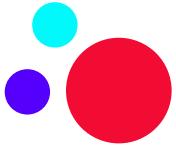
```
AIC: -86.08515467283021
```



ML Time Series

Implementacja

```
# Flagujemy outlier
AirPassengers_outliers["FLAG"] = 0.0
AirPassengers_outliers.loc[140, "FLAG"] = 1.0

# Sztuczna kolumna samych 0, aby zadziałała funkcja w paczce
AirPassengers_outliers["FLAG_2"] = 0.0
```



ML Time Series

Implementacja

```
# Modeling
model = ARIMA(order = (2, 1, 1), season_length=12, seasonal_order=(2,1,0))
res = model.fit(y = AirPassengers_outliers["log_passengers"].values, X =
AirPassengers_outliers[["FLAG", "FLAG_2"]].values)
```

ML Time Series

Implementacja

```
# Post processing
AirPassengers_outliers["FITTED"] = AirPassengers_outliers["log_passengers"].values - res.model_["residuals"]
AirPassengers_outliers["RESID"] = res.model_["residuals"]
```

```
1      4.767930
2      4.881483
3      4.858969
4      4.795218
5      4.904719
       ...
139    6.447823
140    7.989505
141    6.107312
142    5.994620
143    6.076241
Name: FITTED, Length: 143, dtype: float64
```

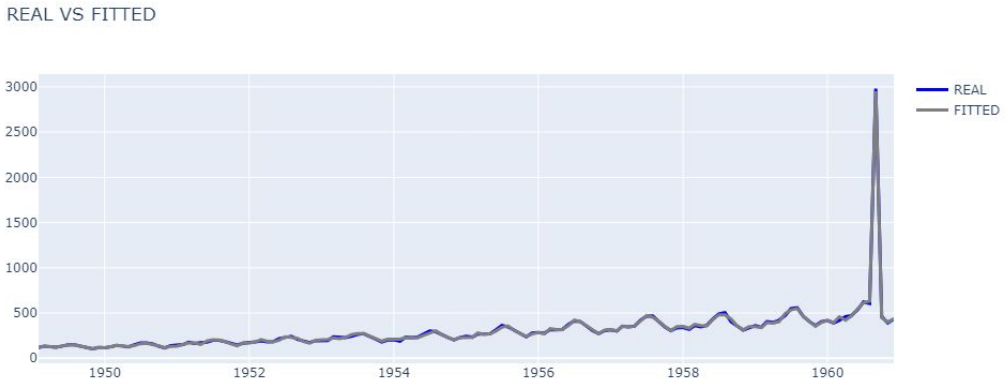
```
1      0.002754
2      0.001319
3      0.000844
4      0.000572
5      0.000556
       ...
139   -0.040943
140    0.010495
141    0.026086
142   -0.028473
143   -0.007816
Name: RESID, Length: 143, dtype: float64
```

ML Time Series

Implementacja

Plot results

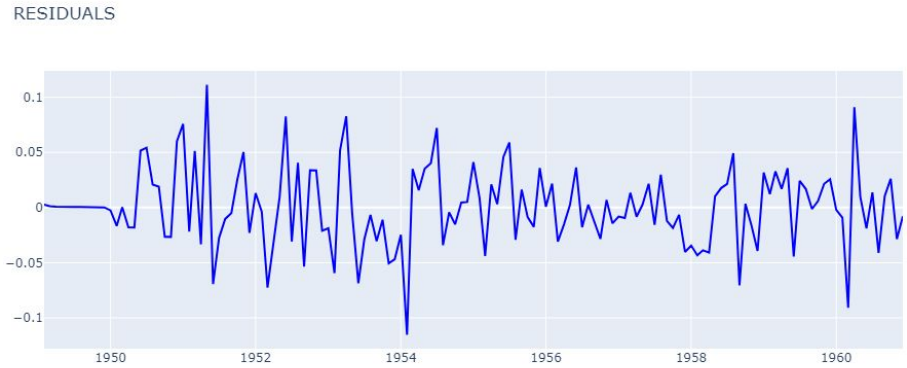
plot_fitted_vs_original_exp(AirPassengers_outliers, "date", "log_passengers", "FITTED")



ML Time Series

Implementacja

```
# Plot residuals
fig = go.Figure()
fig.add_scatter(x = AirPassengers_outliers['date'], y =
AirPassengers_outliers['RESID'], mode = 'lines', name = 'REAL', line = dict(color='blue'))
fig.update_layout(title = "RESIDUALS")
fig.show()
```



ML Time Series

Implementacja

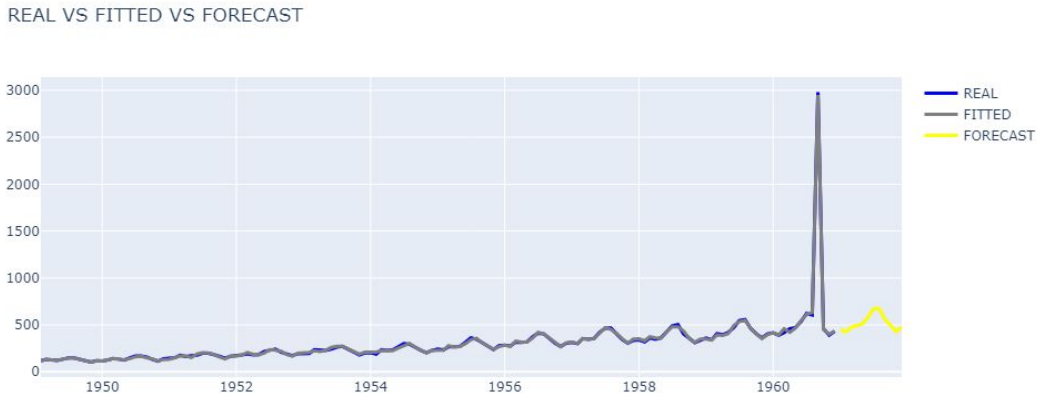
```
# Make forecast
# Powinnismy tez dac informacje modelowo o FLAG i FLAG_2 na przyszlosc, zakładamy, ze w przyszlosci zawsze będzie 0.
forecast = model.predict(h=12, X = AirPassengers_outliers[["FLAG", "FLAG_2"]].values[0:12])["mean"]

forecast_dates = [AirPassengers_outliers['date'].iloc[-1] + relativedelta(months=i+1) for i in range(12)]
forecast_df = pd.DataFrame({'date': forecast_dates, 'forecast': forecast})
```

	date	forecast
0	1961-01-01	6.110131
1	1961-02-01	6.050062
2	1961-03-01	6.164390
3	1961-04-01	6.195786
4	1961-05-01	6.234512
5	1961-06-01	6.366592
6	1961-07-01	6.511655
7	1961-08-01	6.510526
8	1961-09-01	6.326616
9	1961-10-01	6.210453
10	1961-11-01	6.063675
11	1961-12-01	6.166052

ML Time Series

Implementacja

```
# Plot results
plot_forecast(AirPassengers_outliers, forecast_df, "date", "log_passengers", "FITTED", "forecast")
```



ML Time Series

Implementacja

```
print("AIC: ", model.model_["aic"])
```

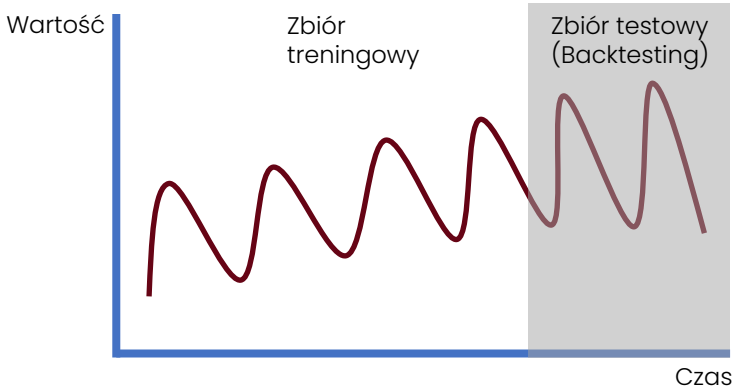
```
AIC: -466.8306865980571
```

Zadanie 16.5 (instrukcja)

Dla zbioru danych sns.load_dataset('flights') z biblioteki seaborn dodaj obserwację odstającą, a następnie wytrenuj model SARIMAX, uwzględniając występowanie tego odstającego punktu, i ocenę wpływu na wyniki modelu.

ML Time Series

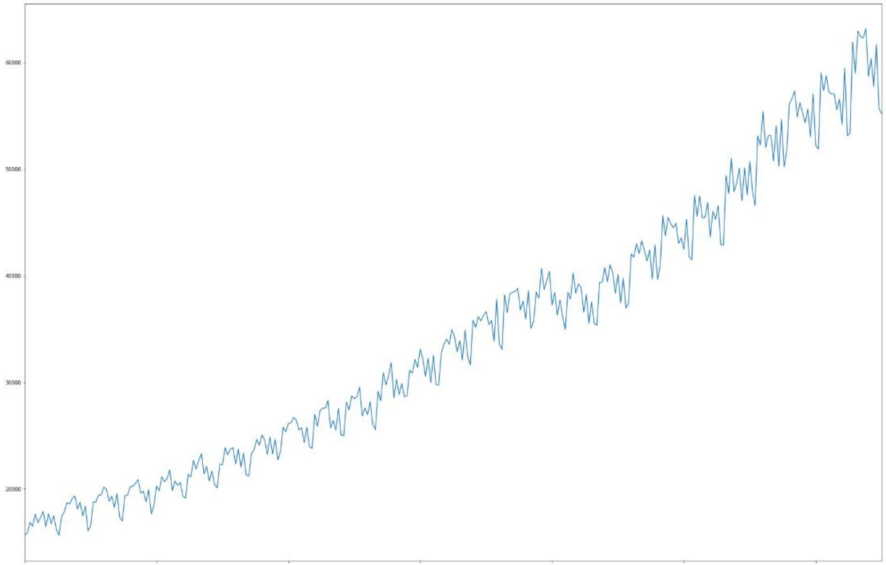
Backtesting



1. Trenujemy różne modele na zbiorze treningowym.
2. Każdy model robi forecast na horyzont testowy.
3. Dla każdego modelu liczymy wartość popełnianego błędu.
4. Wybieramy model, który ma najmniejszy błąd.
5. Przy bardzo zbliżonych wartościach błędów zaleca się wybór prostszego modelu.

ML Time Series

Podsumowanie



infoShareAcademy.com

ML Time Series

Metryki błędu

$$MSE = \frac{1}{n}\sum_{t=1}^{t=n}(y' - y)^2$$

Duże błędy są mocno karane. Lepiej pomylić się wiele razy mało niż jeden raz mocno.

$$MAE = \frac{1}{n}\sum_{t=1}^{t=n}|y' - y|$$

Duże błędy są karane tak samo jak małe. Możemy raz się pomylić mocno, jeśli to pozwoli średnio polepszyć forecast.

$$MAPE = \frac{1}{n}\sum_{t=1}^{t=n}\frac{|y' - y|}{y} * 100\%$$

Wygodna biznesowo interpretacja w postaci średniego błędu procentowego.

$$WAPE = \frac{\sum_{t=1}^{t=n}|y' - y|}{\sum_{t=1}^{t=n}|y|}$$

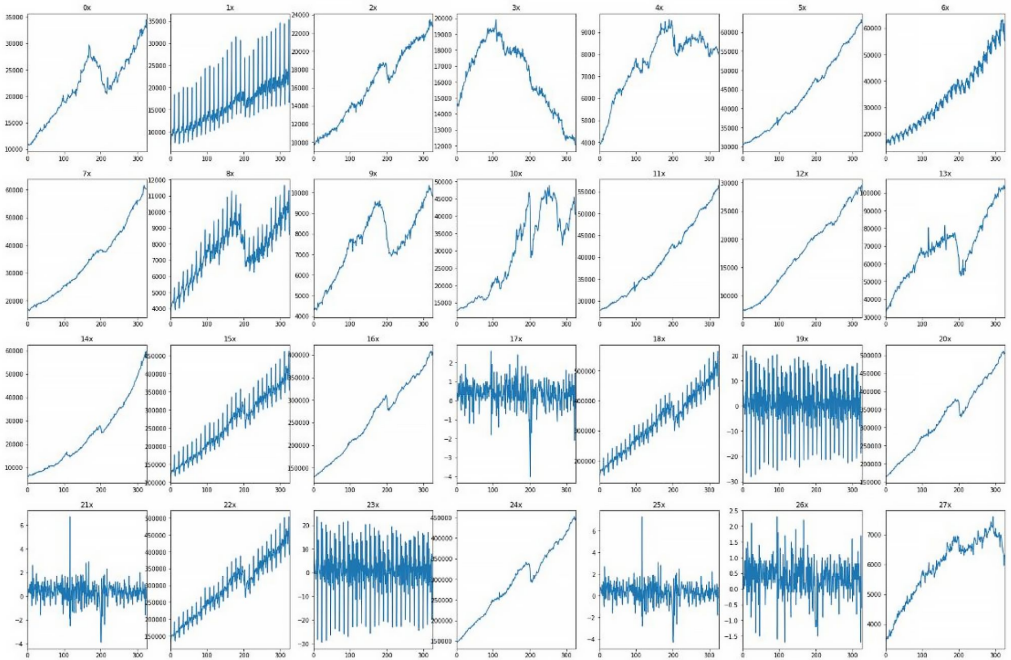
Pozwala uwzględnić, że niektóre produkty są ważniejsze (większa sprzedaż) od innych, a więc model musi bardziej na nich się koncentrować minimalizując błąd.

BIAS % = SUM(Y' – Y) / SUM(Y)

info Share ACADEMY

ML Time Series

Podsumowanie



infoShareAcademy.com