Statystyka w języku Python

Krzysztof Trajkowski

2019-08-19

Spis treści

Rozdział 1

Statystyki rozkładu

1.1 Rozkład dyskretny

Funkcja rozkładu prawdopodobieństwa $f(x)$ przedstawia dowolny rozkład dyskretny gdy:

$$0 \leq f(x_i) \leq 1 \quad \text{oraz} \quad \sum_{i=1}^{n} f(x_i) = 1 \quad \text{dla} \quad i = 1, 2, 3, \ldots, n \tag{1.1}$$

Statystyki dla tego rodzaju rozkładów można obliczyć za pomocą wzorów:

$$E(x) = \sum_{i=1}^{n} x_i \cdot f(x_i) \quad \longrightarrow \quad \text{średnia} \tag{1.2}$$

$$V(x) = \sum_{i=1}^{n} f(x_i) \cdot \left[x_i - E(x)\right]^2 \quad \longrightarrow \quad \text{wariancja} \tag{1.3}$$

$$SK(x) = \frac{1}{D(x)^3} \sum_{i=1}^{n} f(x_i) \cdot \left[x_i - E(x)\right]^3 \quad \longrightarrow \quad \text{skośność} \tag{1.4}$$

$$KU(x) = \frac{1}{D(x)^4} \sum_{i=1}^{n} f(x_i) \cdot \left[x_i - E(x)\right]^4 - 3 \quad \longrightarrow \quad \text{kurtoza} \tag{1.5}$$

Przykładowo, dla rozkładu prawdopodobieństwa (rozkład Poissona) danego wzorem $f(x) = \frac{\lambda^x \exp(-\lambda)}{x!}$ gdzie $x! = \Gamma(x+1)$ można wyprowadzić następujące wzory: $E(x) = \lambda$, $V(x) = \lambda$, $SK(x) = \lambda^{-1/2}$, $KU(x) = \lambda^{-1}$.

```
import scipy.stats as stats
import pandas as pd

mu, var, sk, ku = stats.poisson.stats(mu=3,loc=0, moments='mvsk')
df = pd.DataFrame({'średnia' :[mu], 'wariancja' :[var],\
                   'skośność':[sk.round(4)], 'kurtoza':[ku.round(4)]})
print(df)
```

```
##   średnia wariancja  skośność  kurtoza
## 0     3.0       3.0    0.5774   0.3333
```

1.2 Rozkład ciągły

Funkcja rozkładu prawdopodobieństwa $f(x)$ przedstawia dowolny rozkład ciągły gdy:

$$0 \leq f(x_i) \leq 1 \quad \text{oraz} \quad \int_{-\infty}^{+\infty} f(x)\, dx = 1 \tag{1.6}$$

Statystyki dla tego rodzaju rozkładów można obliczyć za pomocą wzorów:

$$E(x) = \int_{-\infty}^{\infty} x \cdot f(x)\, dx \quad \longrightarrow \quad \text{średnia} \tag{1.7}$$

$$V(x) = \int_{-\infty}^{\infty} f(x) \cdot \left[x - E(x)\right]^2 dx \quad \longrightarrow \quad \text{wariancja} \tag{1.8}$$

$$SK(x) = \frac{1}{D(x)^3} \int_{-\infty}^{\infty} f(x) \cdot \left[x - E(x)\right]^3 dx \quad \longrightarrow \quad \text{skośność} \tag{1.9}$$

$$KU(x) = \frac{1}{D(x)^4} \int_{-\infty}^{\infty} f(x) \cdot \left[x - E(x)\right]^4 dx - 3 \quad \longrightarrow \quad \text{kurtoza} \tag{1.10}$$

Warto zauważyć, że parametry danego rozkładu mogą odpowiadać pewnej kombinacji średniej i wariancj. W przypadku rozkładu gamma (patrz rozdział 3) będziemy mieli: $E(x) = a \cdot s$, $V(x) = a \cdot s^2$, $SK(x) = \sqrt{4/a}$, $KU(x) = 6/a$. Wynika z tego, że $a = E(x)^2/V(x)$ oraz $s = V(x)/E(x)$ to odpowiednio parametr kształtu oraz skali.

```
import scipy.stats as stats

v = stats.gamma.rvs(a=1.3,loc=0,scale=1.36,size=150,random_state=2305)
fit = stats.gamma.fit(v,floc=0)
mu,var,sk,ku = stats.gamma.stats(a=fit[0],loc=fit[1],scale=fit[2], moments='mvsk')

print("MLE:\na= %.4f, loc= %.4f, s= %.4f" % (fit[0],fit[1],fit[2]))
print("\nśrednia= %.2f, wariancja= %.2f, skośność= %.2f, kurtoza= %.2f" % (mu,var,sk,ku))
print("\nMOM:\nśrednia^2/wariancja: a= %.2f, wariancja/średnia: s= %.2f" % (mu**2/var, var/mu))
```

```
## MLE:
## a= 1.1907, loc= 0.0000, s= 1.3814
##
## średnia= 1.64, wariancja= 2.27, skośność= 1.83, kurtoza= 5.04
##
## MOM:
## średnia^2/wariancja: a= 1.19, wariancja/średnia: s= 1.38
```

Ciekawym przypadkiem jest rozkład normalny (patrz rozdział 2) ponieważ średnia oraz odchylenie standardowe czyli pierwiastek kwadratowy z wariancji są jednocześnie parametrami tego rozkładu. Dodajmy, że nie każdy rozkład prawdopodobieństwa musi mieć te statystyki określone. Przykładem może być rozkład Cauchy'ego który nie ma zdefiniowanej średniej, wariancji, skośności oraz kurtozy.

Rozdział 2

Rozkład normalny

2.1 Funkcja gęstości

Do funkcji `scipy.stats.gennorm.pdf` został zaimplementowany uogólniony rozkład normalny $GN(x \mid \beta, m, s)$ który można przedstawić za pomocą wzoru:

$$f(x \mid \beta, m, s) = \frac{\beta}{2\,s\,\Gamma(1/\beta)} \exp\left(-\left|\frac{x-m}{s}\right|^{\beta}\right) \tag{2.1}$$

gdzie β to parametr kształtu (shape), $s > 0$ to parametr skali (scale) oraz m to parametr przesunięcia. Przypadek dla $\beta = 2$ został zaimplementowany do funkcji `scipy.stats.norm.pdf` czyli klasycznej wersji rozkładu normalnego $N(x \mid m, s)$ i jest dana wzorem:

$$f(x \mid m, s) = \frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{(x-m)^2}{2s^2}\right) \tag{2.2}$$

Do oszacowania parametrów rozkładu normalnego można wykorzystać funkcje `scipy.stats.norm.fit` która wykorzystuje metodę największej wiarygodności. Inaczej mówiąc, oszacowanie parametrów w rozkładzie normalnym sprowadza się do wyznaczenia estymatora średniej oraz obciążonego estymatora odchylenia standardowego na podstawie wzorów:

$$\hat{m} = E(X) = \sum_{i=1}^{n} x_i/n, \quad \text{oraz} \quad \hat{s}_{\mathrm{ML}} = \sqrt{V(X)} = \sqrt{\sum_{i=1}^{n}(x_i - \hat{\mu})^2/n}$$

Dodajmy, że dla małych próbek zalecane jest stosowanie nieobciążonego estymatora odchylenia standardowego $\hat{s} = \sqrt{\sum_{i=1}^{n}(x_i - \hat{m})^2/(n-1)}$ ale wraz ze wzrostem liczebności próby różnice między estymatorem obciążonym $\hat{s}_{\mathrm{ML}}$ a nieobciążonym $\hat{s}$ zanikają. Warto zaznaczyć, że estymator parametru skali w uogólnionym rozkładzie normalnym dla $\beta = 2$ będzie równy $\hat{s}\sqrt{2}$.

Gdy założymy, że parametr średniej (przesunięcie funkcji) $m = 0$ i parametr odchylenia standardowego (skalowanie funkcji) $s = 1$ to ogólny wzór funkcji $f(x \mid m, s)$ uprości się do postaci standardowej:

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tag{2.3}$$

Warto zauważyć, że dokonując prostych przekształceń standardowej funkcji prawdopodobieństwa $f(x)$ np. rozkładu normalnego otrzymamy wyniki które będą tożsame z uzyskanymi na podstawie funkcji $g(x \mid m, s)$.

$$g(x \mid m,s) = f\big((x-m)/s\big)/s \quad \longrightarrow \quad \text{funkcja gęstości} \tag{2.4}$$

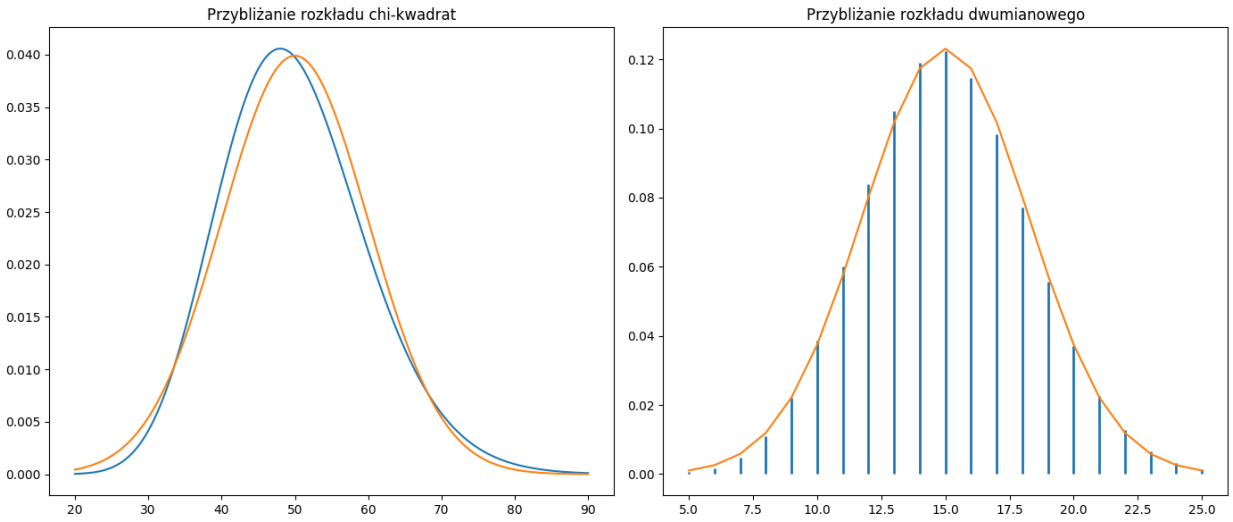
$$\int_{-\infty}^{a} g(x \mid m,s)\,dx = \int_{-\infty}^{a} f\big((x-m)/s\big)\,dx \quad \longrightarrow \quad \text{dystrybuanta} \tag{2.5}$$

$$q_{\alpha \mid m,s} = z_{\alpha \mid 0,1} \cdot s + m \quad \longrightarrow \quad \text{kwantyle} \tag{2.6}$$

$$rv_{i \mid m,s} = rv_{i \mid 0,1} \cdot s + m \quad \longrightarrow \quad \text{liczby losowe} \tag{2.7}$$

Dla dużych prób rozkłady ciągłe oraz dykretne można przybliżać rozkładem normalnym. W przypadku rozkładu chi-kwadrat będziemy mieli $N(df, \sqrt{2\,df})$ gdzie $df = n - 1$ to stopnie swobody. Natomiast dla rozkładu dwumianowego $N(np, \sqrt{np(1-p)})$.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

xc = np.arange(20,90,0.01); xd = np.arange(5,25+1,1)

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.plot(xc,stats.chi2.pdf(xc,df=50))
ax1.plot(xc,stats.norm.pdf(xc,loc=50,scale=(2*50)**0.5))
ax2.vlines(xd, [0], stats.binom.pmf(xd, n= 50, p= 0.3),lw=2,color='C0')
ax2.plot(xd,stats.norm.pdf(xd,loc=50*0.3,scale=(50*0.3*0.7)**0.5),color='C1')
ax1.set_title("Przybliżanie rozkładu chi-kwadrat")
ax2.set_title("Przybliżanie rozkładu dwumianowego")
fig.tight_layout()
plt.savefig('plt01.png')
```



Rysunek 2.1: Przybliżanie rozkładów.

2.2 Liniowy model regresji

Współczynniki modelu liniowego $Y = \hat{\beta}X + \epsilon$ można znaleźć w stosunkowo prosty sposób wykonując działania na macierzach:

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{2.8}$$

gdzie Y to wektor zmiennej zależnej pochodzącej z rozkładu normalnego natomiast X to macierz zmiennych niezależnych.

Błędy standardowe oszacowanych parametrów to pierwiastki kwadratowe elementów na głównej przekątnej macierzy wariancji i kowariancji:

$$D^2(\hat{\beta}) = (X^T X)^{-1} S_e^2 \tag{2.9}$$

gdzie $S_e^2 = e^T e/(n-k-1)$ to wariancja reszt i $e = Y - \hat{\beta}X$ to wektor reszt.

Stopień wyjaśnienia przez model zmiennej zależnej można ocenić za pomocą współczynnika determinacji:

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y} - \bar{y})^2}{\sum_{i=1}^{n}(y - \bar{y})^2} \tag{2.10}$$

Siłę związku dwóch zmiennych można ocenić na podstawie współczynnika korelacji liniowej Pearsona który może być równy pierwiastkowi kwadratowemu współczynnika determinacji ponieważ $|r| = R$.

$$r = \mathrm{cov}(X, Y)/S_X S_Y \tag{2.11}$$

gdzie: $\mathrm{cov}(X, Y)$ to kowariancja dwóch zmiennych natomiast S_X i S_Y to odchylenia standardowe zmiennych.

Błąd standardowy korelacji Pearsona dla dużej próby wyznaczamy według wzoru:

$$SE_r = \sqrt{(1 - r^2)/n} \tag{2.12}$$

Wszystkie wyniki można uzyskać za pomocą funkcji `scipy.stats.linregress` ale tylko dla jednej zmiennej objaśniającej.

```
from scipy import stats
import numpy as np

x = np.sort(stats.norm.rvs(size=300,loc=3,random_state=2305))
y = np.sort(stats.norm.rvs(size=300,scale=3,random_state=4101))

beta, const, r_value, p_value, SE_beta = stats.linregress(x, y)
SE_r = ((1-r_value**2)/(len(x)))**0.5
print("beta: %f, SE_beta: %f" % (beta,SE_beta))
print("cor: %f, SE_cor: %f" % (r_value,SE_r))
```

```
## beta: 2.820229, SE_beta: 0.009932
## cor: 0.998157, SE_cor: 0.003503
```

Przedstawiona powyżej procedura szacowania parametrów to metoda najmniejszych kwadratów która minimalizuje sumę kwadratów reszt:

$$RSS = e^T e \quad \longrightarrow \quad \min \tag{2.13}$$

Innym kryterium optymalizacji możne być maksymalizacja logarytmu wiarygodności:

$$LL = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{e^T e}{2\sigma^2} \quad \longrightarrow \quad \max \tag{2.14}$$

Obie procedury: metoda najmniejszych kwadratów (2.13) i metoda największej wiarygodności (2.14) dla dowolnej liczby zmiennych objaśniających zostały udostępnione w pakiecie `statsmodels`. Ciekawą alternatywą jest wykorzystanie algorytmów optymalizacyjnych ogólnego przeznaczenia z pakietu `scipy.optimize.minimize`. Takie rozwiązanie (patrz podrozdział 3.3) umożliwia szacowanie parametrów modeli o dowolnej postaci analitycznej po uprzednim zdefiniowaniu funkcji logarytmu wiarygodności.

```
from scipy import stats
import numpy as np
import pandas as pd

df = pd.DataFrame(data={'x':np.sort(stats.norm.rvs(size=300,loc=3,random_state=2305)),
                        'y':np.sort(stats.norm.rvs(size=300,scale=3,random_state=4101))})

import statsmodels.formula.api as smf

m = smf.ols("y~x", data=df).fit()
print(m.summary())
```

```
##                             OLS Regression Results
## ==============================================================================
## Dep. Variable:                      y   R-squared:                       0.996
## Model:                            OLS   Adj. R-squared:                  0.996
## Method:                 Least Squares   F-statistic:                 8.064e+04
## Date:                Mon, 19 Aug 2019   Prob (F-statistic):               0.00
## Time:                        19:22:53   Log-Likelihood:                 99.901
## No. Observations:                 300   AIC:                            -195.8
## Df Residuals:                     298   BIC:                            -188.4
## Df Model:                           1
## Covariance Type:            nonrobust
## ==============================================================================
##                  coef    std err          t      P>|t|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept     -8.2571      0.032   -260.743      0.000      -8.319      -8.195
## x              2.8202      0.010    283.964      0.000       2.801       2.840
## ==============================================================================
## Omnibus:                       29.693   Durbin-Watson:                   0.280
## Prob(Omnibus):                  0.000   Jarque-Bera (JB):              127.091
## Skew:                           0.201   Prob(JB):                     2.53e-28
## Kurtosis:                       6.163   Cond. No.                         10.9
## ==============================================================================
##
## Warnings:
## [1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

2.3 Nieliniowy model regresji

Jeśli badana zależność ma charakter nieliniowy a zmienna objaśniana pochodzi z rozkładu normalnego to można zastosować nieliniową metodę najmniejszych kwadratów np. algorytm Levenberga-Marquardta lub Trust Region Reflective jeśli chcemy dodać ograniczenia przedziałowe na parametry. Obie procedury zostały zaimplementowane do funkcji `scipy.optimize.curve_fit`. Dodajmy jeszcze, że są to procedury iteracyjne które wymagają określenia parametrów startowych. W pewnych sytuacjach można je wyznaczyć za pomocą tzw. linearyzacji czyli po sprowadzeniu modelu nieliniowego do postaci liniowej ale nie zawsze jest to możliwe. Poniżej przykłady linearyzacji wybranych modeli nieliniowych:

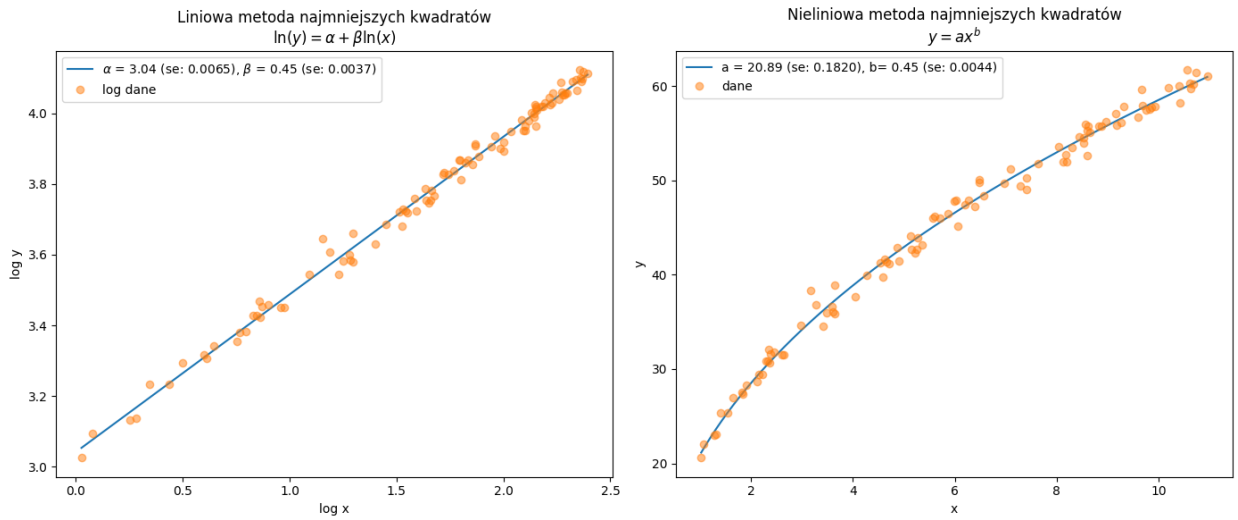
- model potęgowy:

$$y = a \cdot x^b \quad \longrightarrow \quad \ln(y) = \alpha + \beta \cdot \ln(x) \quad \longrightarrow \quad \exp(\alpha) = a, \quad \beta = b \tag{2.15}$$

- model Tornquista 1:

$$y = \frac{ax}{x+b} \quad \longrightarrow \quad \frac{1}{y} = \alpha + \beta \cdot \frac{1}{x} \quad \longrightarrow \quad \frac{1}{\alpha} = a, \quad \frac{\beta}{\alpha} = b \tag{2.16}$$

```
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
import numpy as np
import pandas as pd
from scipy.optimize import curve_fit

x = stats.uniform.rvs(1,10,size=100, random_state=2305)
mu = 20.8 * x**0.45
y = stats.norm.rvs(loc=mu,scale=1,size=100,random_state=2305)
lx = np.log(x)
ly = np.log(y)
df = pd.DataFrame({'x':x,'y':y,'lx':lx,'ly':ly})

modLog = smf.glm('ly~lx', data=df).fit()
p = modLog.params.values
pse = np.diag(modLog.cov_params())**0.5

def modNLS(x,a,b):
    return a*x**b

sol, pcov = curve_fit(modNLS, x, y, p0=(np.exp(p[0]),p[1]))
se = np.diag(pcov)**0.5

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
lX = np.linspace(np.min(lx),np.max(lx), 500)
ax1.plot(lX,p[0]+p[1]*lX,
         label='$\\alpha$ = %.2f (se: %.4f), $\\beta$ = %.2f (se: %.4f)' % (p[0],pse[0],p[1],pse[1]))
ax1.plot(lx,ly,'o',alpha=0.5,label='log dane')
ax1.set_xlabel("log x")
ax1.set_ylabel("log y")
ax1.legend()
ax1.set_title("Liniowa metoda najmniejszych kwadratów\n$\\ln(y)=\\alpha+\\beta\\ln(x)$")
X = np.linspace(np.min(x),np.max(x), 500)
ax2.plot(X,modNLS(X,sol[0],sol[1]),
         label='a = %.2f (se: %.4f), b= %.2f (se: %.4f)' % (sol[0],se[0],sol[1],se[1]))
ax2.plot(x,y,'o',alpha=0.5,label='dane')
ax2.set_title("Nieliniowa metoda najmniejszych kwadratów\n$y=ax^b$")
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.legend()
fig.tight_layout()
plt.savefig('modlin01.png')
```



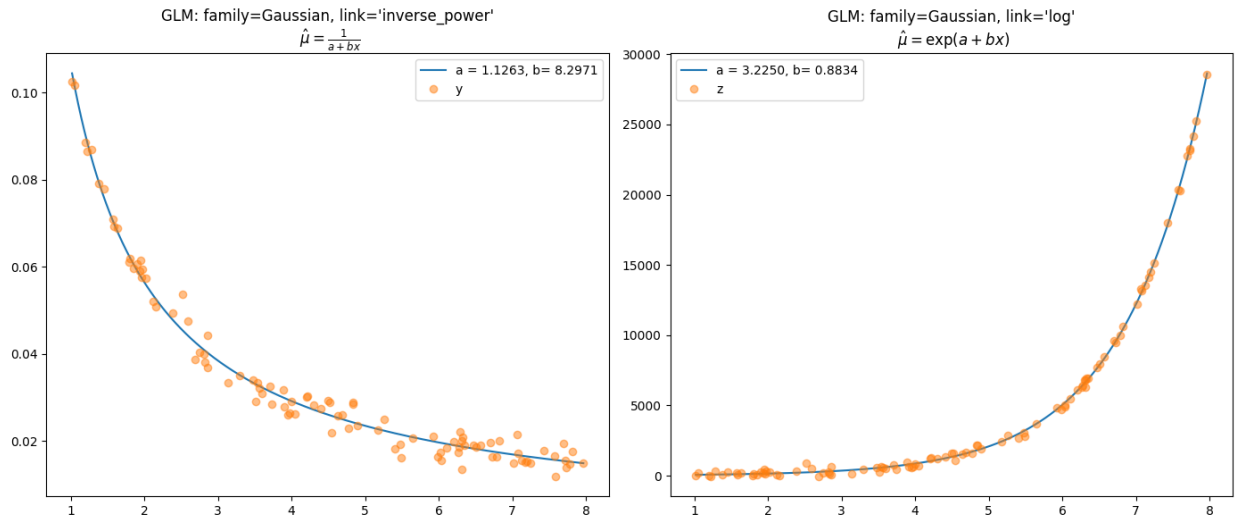
Rysunek 2.2: Graficzna prezentacja linearyzacji funkcji nieliniowej.

Alternatywnym rozwiązaniem jest metoda największej wiarygodności w której można założyć dowolny rozkład prawdopodobieństwa dla zmiennej zależnej. Ta metoda jest stosowana do estymacji parametrów uogólnionych modeli liniowych w których trzeba określić rozkład z rodziny rozkładów wykładniczych dla zmiennej objaśnianej. Dodatkowo dzięki funkcji wiążącej można rozpatrywać szególne przypadki powiązania zmiennej objaśniającej z predyktorem. Przykładowo dla rozkładu normalnego domyślnie jest estymowany model liniowy - opcja `identity`: $\hat{\mu} = \hat{y}$ ale możliwe są też takie przypadki jak `inverse_power`: $\hat{\mu} = 1/\hat{y}$ oraz `log`: $\hat{\mu} = \exp(\hat{y})$ gdzie $\hat{y} = \beta_0 + \sum_{j=1}^{n} \beta_j x_{ij}$. Warto zaznaczyć, że średnia na skali logarytmicznej nie jest równa logarytmowi średniej na oryginalnej skali tzn. $E(\ln Y_i) \neq \ln E(Y_i)$.

```
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
import statsmodels.api as sm
import numpy as np
import pandas as pd
from scipy.optimize import curve_fit

x = stats.uniform.rvs(1,7,size=100, random_state=2305)
mu = 1/(1.25+8.25*x)
MU = np.exp(3.25+0.88*x)
y = stats.norm.rvs(loc=mu,scale=0.0025,size=100,random_state=2305)
z = MU+stats.norm.rvs(scale=200,size=100,random_state=2305)
X = np.linspace(np.min(x),np.max(x), 500)

gaus1 = smf.glm('y~x', data=pd.DataFrame({'x':x,'y':y}),\
        family=sm.families.Gaussian(sm.families.links.inverse_power)).fit()
p = gaus1.params.values
pSE = np.diag(gaus1.cov_params())**0.5
gaus2 = smf.glm('z~x', data=pd.DataFrame({'x':x,'z':z}),\
        family=sm.families.Gaussian(sm.families.links.log)).fit()
P = gaus2.params.values
Pse = np.diag(gaus2.cov_params())**0.5
print("GLM: family=Gaussian, link='inverse_power'")
```

```
print(gaus1.summary().tables[1])
print("\nGLM: family=Gaussian, link='log'")
print(gaus2.summary().tables[1])

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
ax1.plot(X, 1/(p[0]+p[1]*X),label='a = %.4f, b= %.4f' % (p[0],p[1]))
ax1.plot(x,y,'o',alpha=0.5,label="y")
ax1.set_title("GLM: family=Gaussian, link='inverse_power'\n $\\hat{\\mu}=\\frac{1}{a+bx}$")
ax1.legend()
ax2.plot(X, np.exp(P[0]+P[1]*X),label='a = %.4f, b= %.4f' % (P[0],P[1]))
ax2.plot(x,z,'o',alpha=0.5,label='z')
ax2.set_title("GLM: family=Gaussian, link='log'\n $\\hat{\\mu}=\\exp(a+bx)$")
ax2.legend()
fig.tight_layout()
plt.savefig('modlin02.png')
```

```
## GLM: family=Gaussian, link='inverse_power'
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      1.1263      0.214      5.275      0.000       0.708       1.545
## x              8.2971      0.127     65.083      0.000       8.047       8.547
## ==============================================================================
##
## GLM: family=Gaussian, link='log'
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      3.2250      0.030    106.185      0.000       3.165       3.285
## x              0.8834      0.004    216.194      0.000       0.875       0.891
## ==============================================================================
```



Rysunek 2.3: Graficzna prezentacja dwóch funkcji wiążących z wykorzystaniem rozkładu Gaussa.

Rozdział 3

Rozkład gamma

3.1 Funkcja gęstości

Uogólniony rozkład gamma zaimplementowany do funkcji `scipy.stats.gengamma.pdf` można przedstawić za pomocą wzoru:

$$f(x \mid a,k,m,s) = \frac{k(x-m)^{ka-1}}{s^{ka}\Gamma(a)} \exp\left(-\left(\frac{x-m}{s}\right)^{k}\right) \tag{3.1}$$

gdzie $k \neq 0$ i $a > 0$ to parametry kształtu (shape), $s > 0$ to parametr skali (scale) oraz m to parametr przesunięcia. Przypadek dla $k = 1$ został zaimplementowany do funkcji `scipy.stats.gamma.pdf` jako trójparametrowa wersja rozkładu gamma która jest dana wzorem:

$$f(x \mid a,m,s) = \frac{(x-m)^{a-1}}{s^{a}\Gamma(a)} \exp\left(-\frac{x-m}{s}\right) \tag{3.2}$$

Jeśli będziemy rozważać dwuparametrowy rozkład gamma tzn. z pominięciem parametru przesunięcia czyli $m = 0$ to wtedy wzór rozkładu uprości się do postaci:

$$f(x \mid a,s) = \frac{x^{a-1}}{s^{a}\Gamma(a)} \exp\left(-\frac{x}{s}\right) \quad \text{gdzie} \quad E(X) = as,\ V(X) = as^2 \tag{3.3}$$

W innej implementacji tego rozkładu zamiast parametru skali s (scale) jest stosowany parametr r (rate) gdzie $r = 1/s$:

$$f(x \mid a,r) = \frac{r^{a}x^{a-1}}{\Gamma(a)} \exp(-rx) \quad \text{gdzie} \quad E(X) = a/r,\ V(X) = a/r^2 \tag{3.4}$$

Do estymacji parametrów rozkładu można wykorzystać metodę największej wiarygodności która polega na optymalizacji zlogarytmowanej funkcji wiarygodności:

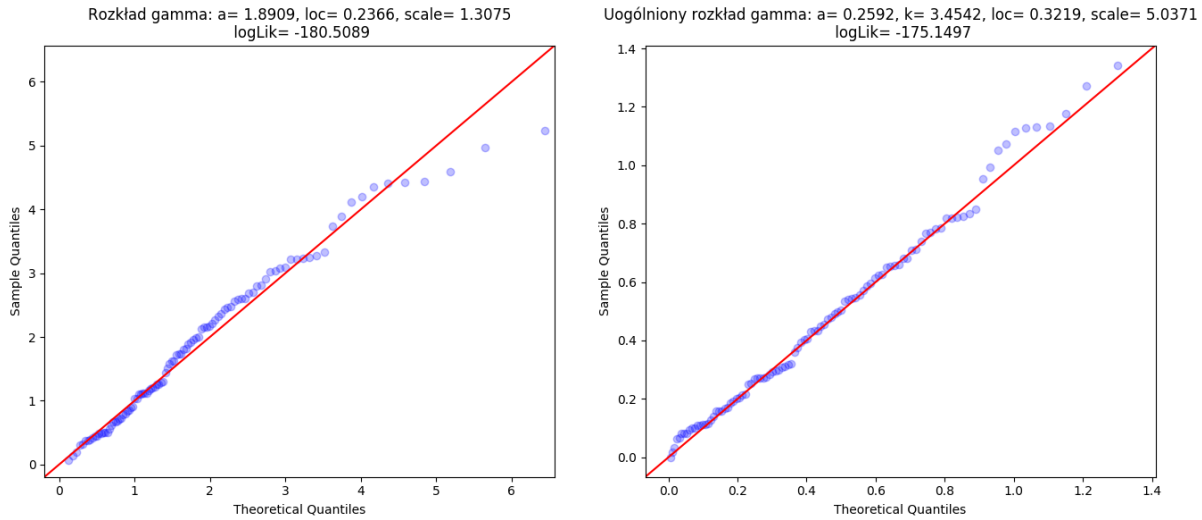
$$LL_{scale} = (a-1)\ln(y) - (y/s) - a\ln(s) - \ln\Gamma(a) \tag{3.5}$$

$$LL_{rate} = a\ln(ry) - \ln\Gamma(a) - \ln(y) - ry \tag{3.6}$$

Funkcja `scipy.stats.gamma.fit` wykonuje estymację parametrów: a, m oraz s. Dzięki argumentom `floc`, `fscale` i `fa` możemy założyć stałą wartość dwóch lub jednego parametru. Jeśli założymy, że $m = 0$ to będziemy optymalizować funkcję (3.5) czyli oszacujemy parametr kształtu a oraz skali s.

```
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.api as sm
```

```
y = stats.gamma.rvs(2.83, size=100, random_state=2305)
f = stats.gamma.fit(y)
F = stats.gengamma.fit(y)
logLik_f = sum(stats.gamma.logpdf(y, a=f[0], loc=f[1], scale=f[2]))
logLik_F = sum(stats.gengamma.logpdf(y, a=F[0], c=F[1], loc=F[2], scale=F[3]))

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
sm.qqplot(y, stats.gamma, fit=True, line='45', alpha=0.25, ax=ax1)
sm.qqplot(y, stats.gengamma, fit=True, line='45', alpha=0.25, ax=ax2)
ax1.set_title("Rozkład gamma: a= %.4f, loc= %.4f, scale= %.4f\n logLik= %.4f" \
% (f[0],f[1],f[2],logLik_f))
ax2.set_title("Uogólniony rozkład gamma: a= %.4f, k= %.4f, loc= %.4f, scale= %.4f\n logLik= %.4f" \
% (F[0],F[1],F[2],F[3],logLik_F))
fig.tight_layout()
plt.savefig('gamma01.png')
```



Rysunek 3.1: Wykresy kwantylowe.

3.2 Liniowy model gamma regresji

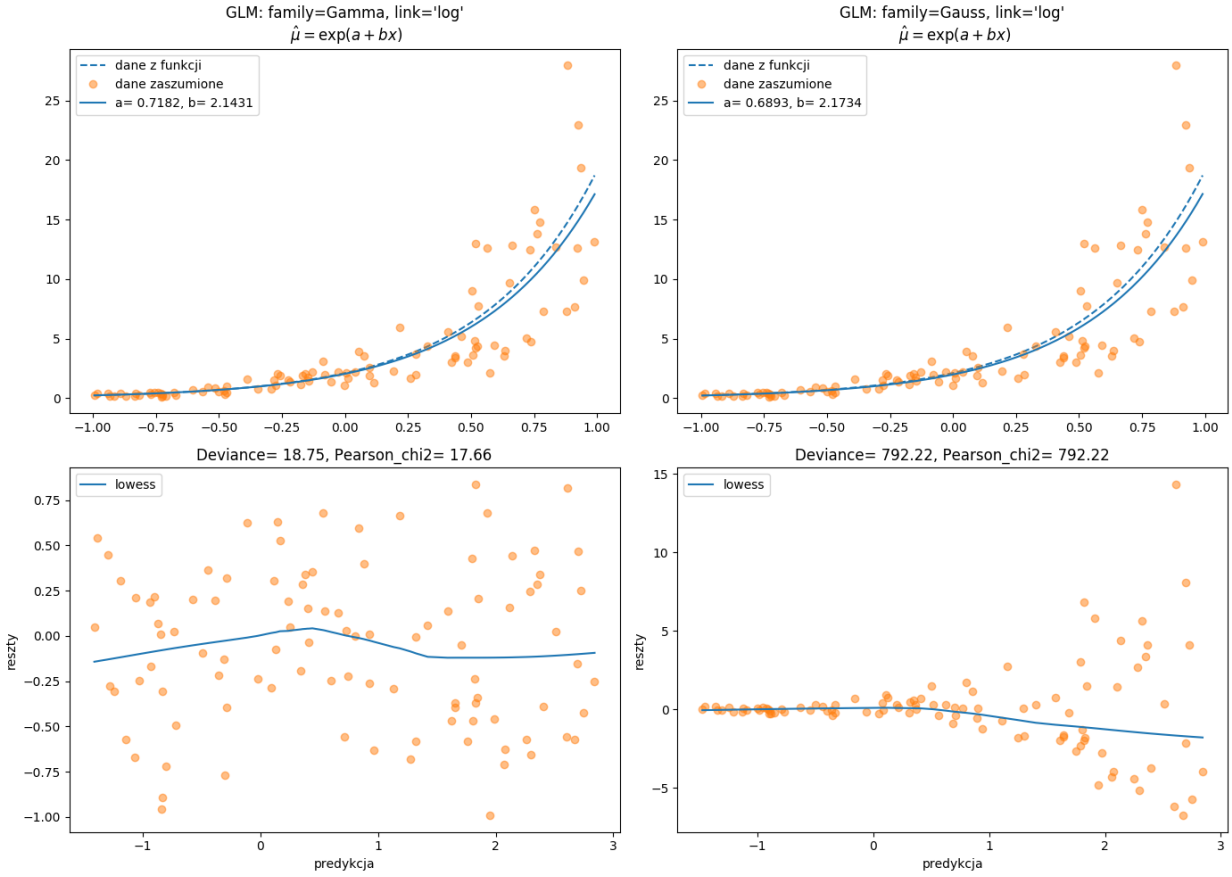
Metoda najmniejszych kwadratów ma zastosowanie w modelowaniu zmiennej objaśnianej która pochodzi z rozkładu normalnego. Zatem gdy zmienna zależna przyjmuje tylko nieujemne wartości z rozkładu ciągłego prawostronnie skośnego to warto rozważyć zastosowanie uogólnionego modelu liniowego z rozkładem gamma i logarytmiczną funkcją wiążącą.

```
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
import numpy as np
import pandas as pd
```

```
import patsy

x = stats.uniform.rvs(-1,2,size=100, random_state=2305)
mu = np.exp(0.75 + 2.2 * x)
y = stats.gamma.rvs(a=6,scale=mu/6,size=100,random_state=2305)
df = pd.DataFrame()
df['x'] = x
df['y'] = y
model = 'y ~ x'
Y, X = patsy.dmatrices(model, df, return_type='dataframe')

modGam = sm.GLM(Y,X, family=sm.families.Gamma(sm.families.links.log)).fit()
p = modGam.params.values
print("GLM: family=Gamma, link='log', logLik= %.2f, scale= %.2f\n" % (modGam.llf, modGam.scale))
print(modGam.summary().tables[1])
modGaus = sm.GLM(Y,X, family=sm.families.Gaussian(sm.families.links.log)).fit()
P = modGaus.params.values
print("\nGLM: family=Gaussian, link='log', logLik= %.2f, scale= %.2f\n" % (modGaus.llf,modGaus.scale))
print(modGaus.summary().tables[1])

fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)
ax4 = fig.add_subplot(2,2,4)
predGam = modGam.predict(linear=True)
resGam = modGam.resid_deviance
lowessGam = sm.nonparametric.lowess(resGam, predGam, frac=2/3)
predGaus = modGaus.predict(linear=True)
resGaus = modGaus.resid_deviance
lowessGaus = sm.nonparametric.lowess(resGaus, predGaus, frac=2/3)
Xg = np.linspace(np.min(x),np.max(x), 500)
ax1.plot(Xg,np.exp(0.75 + 2.2 * Xg),ls='--',color='C0',label='dane z funkcji')
ax1.plot(x,y,'o',alpha=0.5,color='C1',label='dane zaszumione')
ax1.plot(Xg,np.exp(p[0]+p[1]*Xg),color='C0',label='a= %.4f, b= %.4f' % (p[0],p[1]))
ax1.set_title("GLM: family=Gamma, link='log'\n $\\hat{\\mu}=\\exp(a+bx)$")
ax1.legend()
ax2.plot(Xg,np.exp(0.75 + 2.2 * Xg),ls='--',color='C0',label='dane z funkcji')
ax2.plot(x,y,'o',alpha=0.5,color='C1',label='dane zaszumione')
ax2.plot(Xg,np.exp(P[0]+P[1]*Xg),color='C0',label='a= %.4f, b= %.4f' % (P[0],P[1]))
ax2.set_title("GLM: family=Gauss, link='log'\n $\\hat{\\mu}=\\exp(a+bx)$")
ax2.legend()
ax3.plot(predGam,resGam,'o',alpha=0.5,color='C1')
ax3.plot(lowessGam[:, 0], lowessGam[:, 1],label='lowess',color='C0')
ax3.set_xlabel("predykcja")
ax3.set_ylabel("reszty")
ax3.set_title("Deviance= %.2f, Pearson_chi2= %.2f" % (modGam.deviance,modGam.pearson_chi2))
ax3.legend()
ax4.plot(predGaus,resGaus,'o',alpha=0.5,color='C1')
ax4.plot(lowessGaus[:, 0], lowessGaus[:, 1],label='lowess',color='C0')
ax4.set_xlabel("predykcja")
ax4.set_ylabel("reszty")
ax4.set_title("Deviance= %.2f, Pearson_chi2= %.2f" % (modGaus.deviance,modGaus.pearson_chi2))
```

```
ax4.legend()
fig.tight_layout()
plt.savefig('modgam01.png')
```

```
## GLM: family=Gamma, link='log', logLik= -123.79, scale= 0.18
##
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      0.7182      0.042     16.917      0.000       0.635       0.801
## x              2.1431      0.072     29.861      0.000       2.002       2.284
## ==============================================================================
##
## GLM: family=Gaussian, link='log', logLik= -245.39, scale= 8.08
##
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      0.6893      0.170      4.050      0.000       0.356       1.023
## x              2.1734      0.213     10.197      0.000       1.756       2.591
## ==============================================================================
```



Rysunek 3.2: Graficzna prezentacja tej samej funkcji wiążącej z wykorzystaniem dwóch rozkładów.

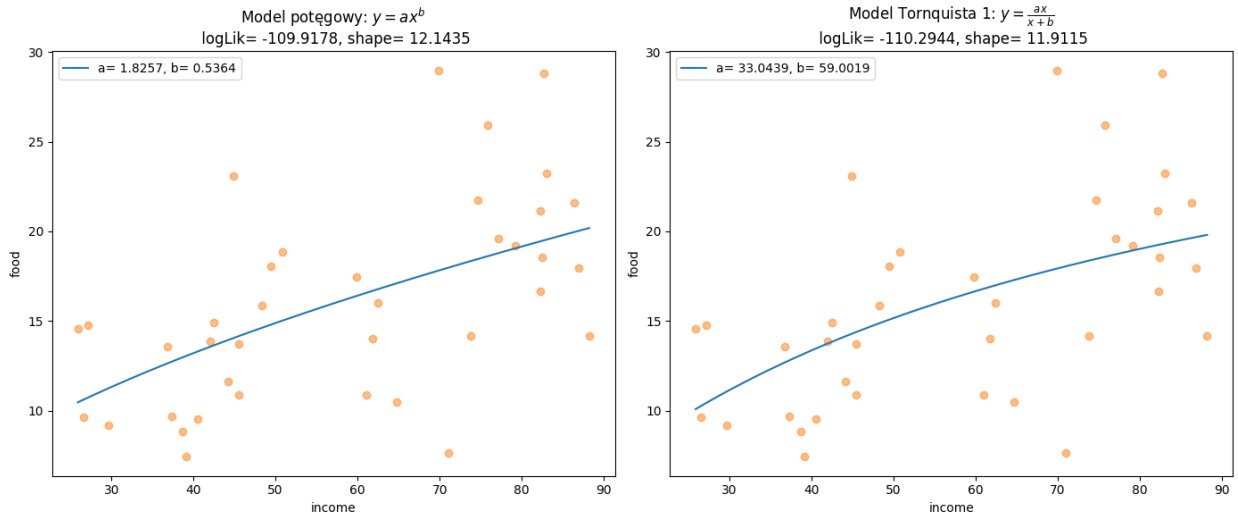
3.3 Nieliniowy model gamma regresji

Zastosowanie nieliniowego modelu gamma regresji zostanie zaprezentowane na przykładzie zestawu danych FoodExpenditure. Są w nim zawarte informacje na temat przychodów i wydatków na żywność z uwzględnieniem liczby osób w gospodarstwie domowym. Wyniki dotyczą próby losowej 38 gospodarstw domowych w dużym amerykańskim mieście.

Wykorzystamy parametryzację funkcji (3.5) z parametrami shape oraz scale. Parametry startowe dla modelu potęgowego oraz Tornquista 1 wyznaczymy odpowiednio na podstawie wzorów (2.15) oraz (2.16).

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd
import statsmodels.api as sm
import patsy
from scipy.optimize import minimize

df = pd.read_csv("https://raw.githubusercontent.com/krzysiektr/datacsv/master/FoodExpenditure.csv")

ly, lx = patsy.dmatrices('np.log(food) ~ np.log(income)', df, return_type='dataframe')
m1 = sm.OLS(ly,lx).fit()
p1 = m1.params.values
iy, ix = patsy.dmatrices('food ~ income', 1/df, return_type='dataframe')
m2 = sm.OLS(iy,ix).fit()
p2 = m2.params.values

def L_gamma_power(par):
    mod = par[0]*df["income"]**par[1]
    mu = mod
    shape = par[2]
    scale = mu/shape
    logLik = -np.sum( stats.gamma.logpdf(df["food"], a=shape, scale=scale) )
    return(logLik)

def L_gamma_torn1(par):
    mod = (par[0]*df["income"])/(df["income"]+par[1])
    mu = mod
    shape = par[2]
    scale = mu/shape
    logLik = -np.sum( stats.gamma.logpdf(df["food"], a=shape, scale=scale) )
    return(logLik)

initPower = [p1[0],p1[1],1]
initTorn1 = [1/p2[0],p2[1]/p2[0],1]
res = minimize(L_gamma_power, initPower, method= "Nelder-Mead")
sol = minimize(L_gamma_torn1, initTorn1, method= "Nelder-Mead")

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
Xg = np.linspace(np.min(df["income"]),np.max(df["income"]), 500)
ax1.plot(df["income"],df["food"],'o',alpha=0.5,color='C1')
ax1.plot(Xg,res.x[0]*Xg**res.x[1],color='C0',
         label='a= %.4f, b= %.4f' % (res.x[0],res.x[1]))
```

```
ax1.set_xlabel("income")
ax1.set_ylabel("food")
ax1.set_title("Model potęgowy: $y=ax^b$\n logLik= %.4f, shape= %.4f" % (-1*res.fun,res.x[2]))
ax1.legend()
ax2.plot(df["income"],df["food"],'o',alpha=0.5,color='C1')
ax2.plot(Xg,(sol.x[0]*Xg)/(Xg+sol.x[1]),color='C0',
         label='a= %.4f, b= %.4f' % (sol.x[0],sol.x[1]))
ax2.set_xlabel("income")
ax2.set_ylabel("food")
ax2.set_title("Model Tornquista 1: $y=\\frac{ax}{x+b}$\n logLik= %.4f, shape= %.4f" % (-1*sol.fun,sol.x
ax2.legend()
fig.tight_layout()
plt.savefig('nlsGamma01.png')
```



Rysunek 3.3: Graficzna prezentacja nieliniowej zależności wydatków na żywność i dochodów.

Rozdział 4

Rozkład beta

4.1 Funkcja gęstości

Rozkład beta na przedziale $[a, b]$ został zaimplementowany do funkcji `scipy.stats.beta` gdzie: `loc`$= a$ oraz `scale`$= b - a$. Funkcję gęstości prawdopodobieństwa tego rozkładu można zapisać za pomocą wzoru:

$$f(x \mid a, b, \alpha, \beta) = \frac{1}{\mathrm{B}(\alpha, \beta)(b-a)^{\alpha+\beta-1}}(x-a)^{\alpha-1}(b-x)^{\beta-1} \tag{4.1}$$

po podstawieniu:

$$\mathrm{B}(\alpha, \beta)(b-a)^{\alpha+\beta-1} = \int_a^b u^{\alpha-1}(1-u)^{\beta-1}du = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}(b-a)^{\alpha+\beta-1}$$

otrzymamy:

$$f(x \mid a, b, \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)(b-a)^{\alpha+\beta-1}}(x-a)^{\alpha-1}(b-x)^{\beta-1} \tag{4.2}$$

gdzie: $E(x) = a + \frac{\alpha}{\alpha+\beta}(b-a)$ oraz $V(x) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}(b-a)^2$.

Standardowa wersja rozkładu beta jest rozpatrywana na przedziale $[0; 1]$ a więc wzór (4.2) upraszcza się do postaci:

$$f(x \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{\alpha-1}(1-x)^{\beta-1} \tag{4.3}$$

gdzie: $E(x) = \frac{\alpha}{\alpha+\beta}$ oraz $V(x) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$.

Jeśli do wzoru (4.3) podstawimy $\alpha = \mu\phi$ oraz $\beta = (1-\mu)\phi$ dla $\alpha + \beta = \phi$ to otrzymamy:

$$f(x \mid \mu, \phi) = \frac{\Gamma(\phi)}{\Gamma(\mu\phi)\Gamma\big((1-\mu)\phi\big)}x^{\mu\phi-1}(1-x)^{(1-\mu)\phi-1} \tag{4.4}$$

gdzie: $E(X) = \mu$ oraz $V(X) = \frac{\mu(1-\mu)}{1+\phi}$.

Po zlogarytmowaniu funkcji prawdopodobieństwa (4.4) otrzymamy funkcję logarytmu wiarygodności o postaci:

$$LL_{beta} = \ln\Gamma(\phi) - \ln\Gamma(\mu\phi) - \ln\Gamma\big((1-\mu)\phi\big) + (\mu\phi - 1)\ln(x) + \big((1-\mu)\phi - 1\big)\ln(1-x) \tag{4.5}$$

```
import scipy.stats as stats

y = stats.beta.rvs(1.78, 2.34, size=100, random_state=2305)

f = stats.beta.fit(y)
logLik = sum(stats.beta.logpdf(y, a=f[0], b=f[1], loc=f[2], scale=f[3]))

print("alpha= %.4f, beta= %.4f, loc= %.4f, scale= %.4f" \
      % (f[0],f[1],f[2],f[3]))
print("\nlogLik= %.4f" % (logLik))
```

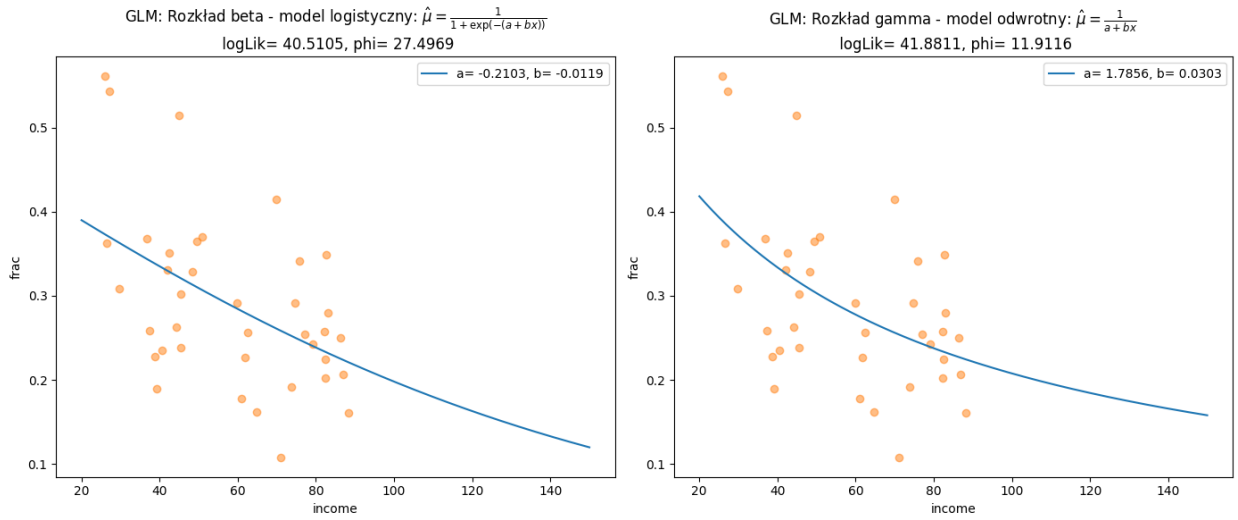
```
## alpha= 0.9859, beta= 0.9528, loc= 0.0152, scale= 0.8953
##
## logLik= 12.6281
```

4.2 Liniowy model beta regresji

Rozkład beta zdefiniowany za pomocą wzoru (4.4) jest wykorzystywany do budowy liniowego modelu regresji dla proporcji. Inaczej mówiąc, w regresji beta wartości zmiennej zależnej mogą określać np. pewną frakcję dochodów gospodarstw domowych wydawanych na żywność. Po podstawieniu do wzoru (4.5) $\hat{\mu} = \frac{1}{1+\exp(-\hat{y})}$ gdzie $\hat{y} = \beta_0 + \sum_{j=1}^{n} \beta_j x_{ij}$ otrzymamy parametry modelu beta regresji. Dodatkowo parametr precyzji ϕ może być uważany za stały lub można go rozszerzyć o dodatkowy zestaw regresorów tzn. $\phi = \exp(\hat{y})$. Zastosowanie liniowego modelu beta regresji zostanie zaprezentowane na przykładzie zestawu danych `FoodExpenditure`.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd
import statsmodels.api as sm
import patsy
from scipy.optimize import minimize

df = pd.read_csv("https://raw.githubusercontent.com/krzysiektr/datacsv/master/FoodExpenditure.csv")
df["frac"] = df["food"]/df["income"]

y, x = patsy.dmatrices('frac ~ income', df, return_type='dataframe')
ols = sm.OLS(y,x).fit()
b = ols.params.values

def L_beta(par):
    mod = par[0]+ par[1]*df["income"]
    mu = stats.logistic.cdf(mod)
    phi = par[2]
    shape1 = mu*phi
    shape2 = (1-mu)*phi
    logLik = -np.sum( stats.beta.logpdf(df["frac"], a=shape1, b=shape2) )
    return(logLik)

def L_gamma(par):
    mod = par[0]+ par[1]*df["income"]
    mu = 1/mod
```

```
    shape = par[2]
    scale = mu/shape
    logLik = -np.sum( stats.gamma.logpdf(df["frac"], a=shape, scale=scale) )
    return(logLik)

initParams = [b[0],b[1],1]
res = [minimize(i, initParams, method= "Nelder-Mead") for i in [L_beta,L_gamma]]

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
Xg = np.linspace(20,150, 1000)
ax1.plot(df["income"],df["frac"],'o',alpha=0.5,color='C1')
ax1.plot(Xg,1/(1+np.exp(-1*(res[0].x[0]+res[0].x[1]*Xg))),color='C0',
         label='a= %.4f, b= %.4f' % (res[0].x[0],res[0].x[1]))
ax1.set_xlabel("income")
ax1.set_ylabel("frac")
ax1.set_title("GLM: Rozkład beta - model logistyczny: $\\hat{\\mu}=\\frac{1}{1+\\exp(-(a+bx))}$\n logLik
ax1.legend()
ax2.plot(df["income"],df["frac"],'o',alpha=0.5,color='C1')
ax2.plot(Xg,1/(res[1].x[0]+res[1].x[1]*Xg),color='C0',
         label='a= %.4f, b= %.4f' % (res[1].x[0],res[1].x[1]))
ax2.set_xlabel("income")
ax2.set_ylabel("frac")
ax2.set_title("GLM: Rozkład gamma - model odwrotny: $\\hat{\\mu}=\\frac{1}{a+bx}$\n logLik= %.4f, phi= %
ax2.legend()
fig.tight_layout()
plt.savefig('prop01.png')
```



Rysunek 4.1: Graficzna prezentacja nieliniowej zależności frakcji wydatków na żywność i dochodów.

Rozdział 5

Rozkład beta dwumianowy

5.1 Funkcja gęstości

Złożenie dwóch rozkładów: dwumianowego oraz beta na przedziale $(0,1)$ tworzy rozkład beta dwumianowy o postaci:

$$f(x \mid n,\alpha,\beta) = \underbrace{\binom{n}{x} p^x (1-p)^{n-x}}_{\text{rozkład dwumianowy}} \cdot \underbrace{\frac{1}{\mathrm{B}(\alpha,\beta)} p^{\alpha-1}(1-p)^{\beta-1}}_{\text{rozkład beta}} \tag{5.1}$$

w którym po podstawieniu:

$$\binom{n}{x} = \frac{\Gamma(n+1)}{\Gamma(x+1)\Gamma(n-x+1)} \quad \text{oraz} \quad p^{x+\alpha-1}(1-p)^{n-x+\beta-1} = \mathrm{B}(x+\alpha,\ n-x+\beta)$$

otrzymamy:

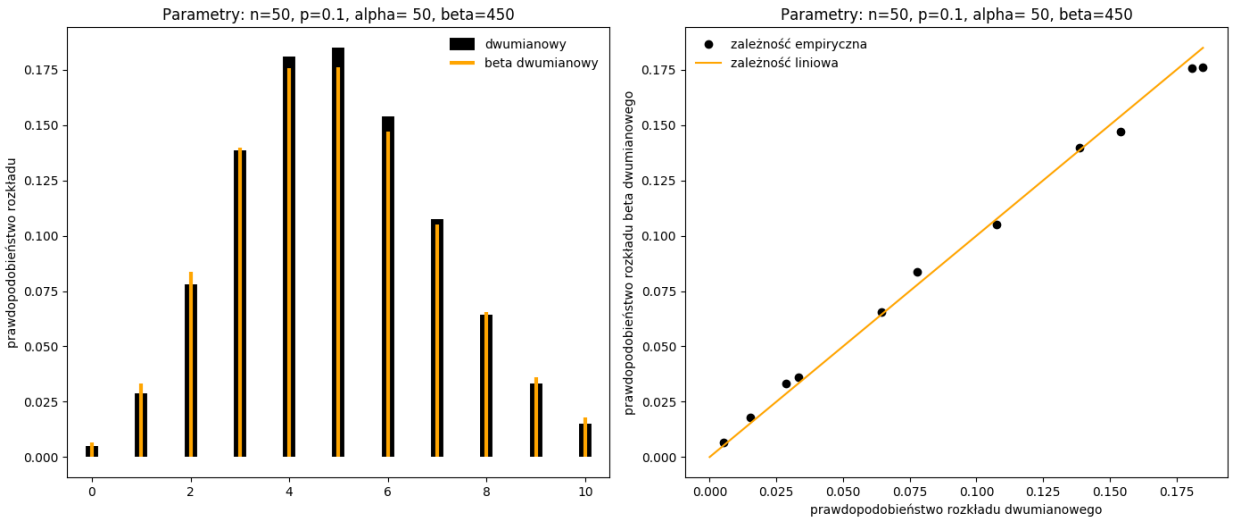
$$f(x \mid n,\alpha,\beta) = \frac{\Gamma(n+1)}{\Gamma(x+1)\Gamma(n-x+1)} \cdot \frac{1}{\mathrm{B}(\alpha,\beta)} \cdot \mathrm{B}(x+\alpha,\ n-x+\beta) \tag{5.2}$$

gdzie: $E(X) = n\frac{\alpha}{\alpha+\beta}$ oraz $V(X) = n\frac{\alpha\beta(\alpha+\beta+n)}{(\alpha+\beta)^2(\alpha+\beta+1)}$.

Za pomocą rozkładu beta dwumianowego $f(x \mid n,\alpha,\beta)$ można przybliżać rozkład dwumianowy gdzie: $\alpha = n$ oraz $\beta = n(1-p)/p$.

```
import numpy as np
import scipy.stats as stats
import scipy.special as spec
import matplotlib.pyplot as plt

def bb(x,n,alpha,beta):
    B1 = (spec.gamma(n+1)*spec.beta(x+alpha,n-x+beta))
    B2 = (spec.gamma(x+1)*spec.gamma(n-x+1)*spec.beta(alpha,beta))
    return(B1/B2)

x = [0,1,2,3,4,5,6,7,8,9,10]
n = 50
p = 0.1
BB = [bb(i,n,n,n*(1-p)/p).round(7) for i in x]
B  = [stats.binom.pmf(i,n,p).round(7) for i in x]
fig = plt.figure(figsize=(14,6))
```

```
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
ax1.vlines(x, 0, B, colors='k', linestyles='-', lw=10,\
           label='dwumianowy')
ax1.vlines(x, 0, BB, colors='orange', linestyles='-', lw=3,\
           label='beta dwumianowy')
ax1.set_title('Parametry: n=%.0f, p=%.1f, alpha= %.0f, beta=%.0f' % (n,p,n,n*(1-p)/p))
ax1.set_ylabel('prawdopodobieństwo rozkładu')
ax1.legend(loc='best', frameon=False)
ax2.plot(B,BB,'o',color='k',label='zależność empiryczna')
ax2.plot([0,max(B)],[0,max(B)],color='orange',label='zależność liniowa')
ax2.set_title('Parametry: n=%.0f, p=%.1f, alpha= %.0f, beta=%.0f' % (n,p,n,n*(1-p)/p))
ax2.set_xlabel('prawdopodobieństwo rozkładu dwumianowego')
ax2.set_ylabel('prawdopodobieństwo rozkładu beta dwumianowego')
ax2.legend(loc='best', frameon=False)
plt.tight_layout()
plt.savefig("betabinom.png")
```



Rysunek 5.1: Przybliżanie rozkładu dwumianowego rozkładem beta dwumianowym.

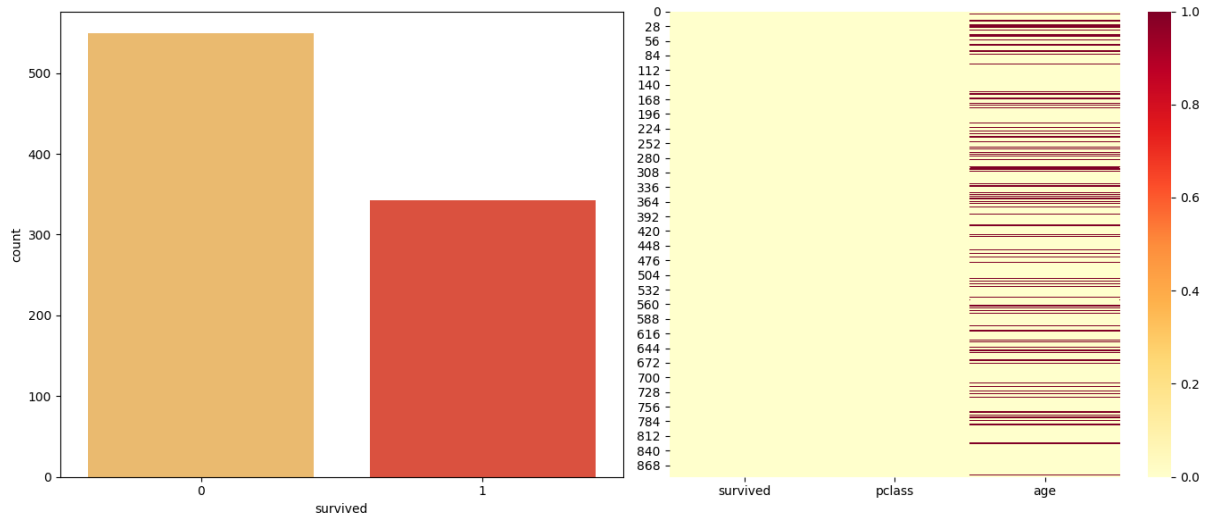
Po uwzględnieniu parametryzacji: $\alpha = \mu\rho^{-1}(1-\rho)$ i $\beta = (1-\mu)\rho^{-1}(1-\rho)$ lub $\alpha = \mu/\sigma$ i $\beta = (1-\mu)/\sigma$ otrzymamy zmodyfikowany rozkład beta dwumianowy w którym funkcję logarytmu wiarygodności można zapisać za pomocą wzoru:

$$LL = \ln\Gamma(n+1) + \ln \mathrm{B}\left(x + \frac{\mu}{\sigma}, n - x + \frac{1-\mu}{\sigma}\right) - \ln\Gamma(x+1) - \ln\Gamma(n-x+1) - \ln \mathrm{B}\left(\frac{\mu}{\sigma}, \frac{1-\mu}{\sigma}\right) \quad (5.3)$$

5.2 Liniowy model beta dwumianowej regresji

Zastosowanie liniowego modelu beta dwumianowej regresji zostanie zaprezentowane na przykładzie zestawu danych `titanic`. Na ich podstawie możemy obliczyć prawdopodobieństwo ocalenia życia w katastrofie brytyjskiego transatlantyka Titanic w zależności od wieku pasażera oraz klasy podróżowania.

```
import warnings
warnings.filterwarnings("ignore")

import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = sns.load_dataset("titanic")

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
sns.countplot(x = 'survived', data = df, palette = 'YlOrRd',ax=ax1)
sns.heatmap(df[['survived','pclass','age']].isnull(),cmap='YlOrRd',ax=ax2)
plt.tight_layout()
plt.savefig("titanic.png")
```



Rysunek 5.2: Graficzna prezentacja danych Titanic.

Rozkład beta dwumianowy jest wykorzystywany do budowy liniowego modelu regresji dla danych binarnych jako alternatywa np. dla regresji logitowej/probitowej. W tym zastosowaniu optymalizujemy (maksymalizacja) funkcję daną wzorem (5.3) w której $\hat{\mu} = \frac{1}{1+\exp(-\hat{y})}$ dla $\hat{y} = \beta_0 + \sum_{j=1}^{n} \beta_j x_{ij}$ otrzymamy parametry modelu beta dwumianowej regresji.

```
import seaborn as sns
import numpy as np
import scipy.stats as stats
import pandas as pd
import scipy.special as spec
from scipy.optimize import minimize
import patsy

df = sns.load_dataset("titanic").dropna()
survived = df['survived']
```

```
pclass = df['pclass']
age = df['age']
yObs, xObs = patsy.dmatrices('survived ~ pclass + age', df, return_type='dataframe')
b0, b1, b2 = np.linalg.lstsq(xObs, yObs, rcond=None)[0]

def Lbb(x,mu,sigma,n):
    return spec.loggamma(n+1)+spec.betaln(x+mu/sigma,n-x+((1-mu)/sigma))-\
           spec.loggamma(x+1)-spec.loggamma(n-x+1)-spec.betaln(mu/sigma,(1-mu)/sigma)

def h(par):
    mod = par[0] +par[1]*pclass +par[2]*age
    MU  = stats.logistic.cdf(mod)
    SIGMA = par[3]
    logLik = -np.sum( Lbb(survived, mu=MU, sigma=SIGMA, n=1) )
    return(logLik)

initParams = [b0[0],b1[0],b2[0],1]
bnds = ((None, None), (None,None),(None, None), (0,None))
results = minimize(h, initParams, method= "L-BFGS-B", bounds=bnds)
print("GLM: family= beta-binomial, link= logistic\n")
print("const= %.4f, pclass= %.4f, age= %.4f, phi= %.4f" % (results.x[0],results.x[1],results.x[2],result
print("\nlogLik= %.4f" % (-1*results.fun))
```

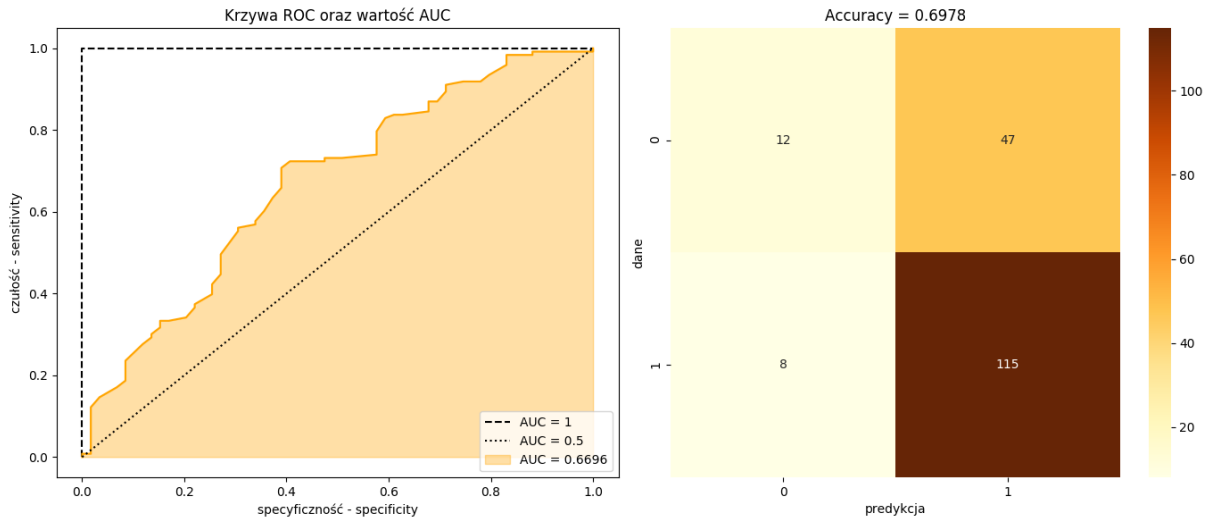
```
## GLM: family= beta-binomial, link= logistic
##
## const= 2.9762, pclass= -0.5559, age= -0.0424, phi= 1.0000
##
## logLik= -107.3699
```

Wyniki można porównać ze standardowym rozwiązaniem tzn. liniowym modelem regresji logitowej lub probitowej. W optymalizacji funkcji logarytmu wiarygodności rozkładu dwumianowego wykorzystujemy dystrybuantę rozkładu logistycznego (logit) lub normalnego (probit). Instnieje pewna zależność pomiędzy parametrami z tych dwóch modeli tzn. $\beta_{probit} \approx 1.6 \cdot \beta_{logit}$. Dodatkowo można wyznaczyć efekty krańcowe za pomocą funkcji `get_margeff`. Krzywa ROC informuje nas o jakości modelu tzn. im wykres bardziej "wypukły", tym lepszy model. Zatm pole pod krzywą (AUC - Area Under ROC Curve) powinno być większe od 0.5.

```
import pandas as pd
import statsmodels.api as sm
import patsy
import seaborn as sns
from sklearn.metrics import roc_curve, auc, confusion_matrix
import numpy as np
import matplotlib.pyplot as plt

df = sns.load_dataset("titanic").dropna()
y, x = patsy.dmatrices('survived ~ pclass + age', df, return_type='dataframe')

logit = sm.Logit(y,x).fit()
print(logit.params)

yp = logit.predict()
fpr, tpr, thresholds = roc_curve(y, yp)
aur = auc(fpr,tpr)
```

```
sens = tpr[thresholds > 0.5][-1]      # czułość
spec = 1 - fpr[thresholds > 0.5][-1] # specyficzność
xx = np.arange(101) / float(100)
ypp = (yp>0.5).astype(int)
tab = confusion_matrix(y,ypp)
tn, fp, fn, tp = confusion_matrix(y,ypp).ravel()

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
ax1.plot([0,0,1],[0,1,1], '--', color='k', label='AUC = 1')
ax1.plot(fpr,tpr, color='orange')
ax1.plot(xx,xx, ':', color='k', label='AUC = 0.5')
ax1.fill_between(x=fpr, y1=tpr, color='orange', alpha=0.35, label='AUC = %.4f' % aur)
ax1.set_title("Krzywa ROC oraz wartość AUC")
ax1.set_xlabel('specyficzność - specificity')
ax1.set_ylabel('czułość - sensitivity')
ax1.legend()
sns.heatmap(tab,annot=True, fmt="d",cmap="YlOrBr",ax=ax2)
ax2.set_title("Accuracy = %.4f" % ((tp+tn)/(tp+tn+fp+fn)))
ax2.set_xlabel('predykcja')
ax2.set_ylabel('dane')
plt.tight_layout()
plt.savefig("roc.png")
```

```
## Optimization terminated successfully.
##          Current function value: 0.589944
##          Iterations 5
## Intercept    2.976216
## pclass      -0.555920
## age         -0.042442
## dtype: float64
```



Rysunek 5.3: Krzywa ROC i tablica trafności prognoz.

Rozdział 6

Rozkład ujemny dwumianowy

6.1 Funkcja gęstości

Rozkład ujemny dwumianowy (zwany też rozkładem Pascala) został zaimplementowany do funkcji `scipy.stats.nbinom` i można go przedstawić za pomocą wzoru:

$$f(x \mid r,p) = \binom{x+r-1}{r-1} p^r (1-p)^x, \quad x = 0,1,..., \quad r \in N \tag{6.1}$$

gdzie: r jest liczbą sukcesów, x jest liczbą niepowodzeń tj. liczba zdarzeń poprzedzających r sukcesów, a p jest prawdopodobieństwem niepowodzeń.

Do powyższego wzoru (6.1) można zastosować alternatywny zapis współczynnika dwumianu:

$$f(x \mid r,p) = \binom{x+r-1}{x} p^r (1-p)^x, \quad x = 0,1,..., \quad r > 0 \tag{6.2}$$

dzięki któremu w rozkładzie ujemnym dwumianowym (zwanym też rozkładem Polya) można przyjąć, że parametr $r > 0$. Dodatkowo współczynnik dwumianowy można zapisać w oparciu o funkcję gamma:

$$f(x \mid r,p) = \frac{\Gamma(x+r)}{\Gamma(r)\Gamma(x+1)} p^r (1-p)^x, \quad x = 0,1,..., \quad r > 0 \tag{6.3}$$

gdzie: $E(X) = r(1-p)/p$ oraz $V(X) = r(1-p)/p^2$.

```
import scipy.stats as stats
import numpy as np

x = stats.nbinom.rvs(n=5.7, p=0.3, size=10000, random_state=2305)
r = np.mean(x)**2/(np.var(x,ddof=1)-np.mean(x))
p = np.mean(x)/np.var(x,ddof=1)
print("MOM: r= %.4f, p= %.4f" % (r,p))
```

```
## MOM: r= 5.5556, p= 0.2948
```

Jeżeli przyjmiemy, że parametr $r = \phi$ oraz $p = \frac{\phi}{\mu+\phi}$ to mieszanka rozkładu Poissona-Gamma będzie miała postać:

$$f(x \mid \mu,\phi) = \frac{\Gamma(x+\phi)}{\Gamma(\phi)\Gamma(x+1)} \left(\frac{\phi}{\mu+\phi}\right)^{\phi} \left(\frac{\mu}{\mu+\phi}\right)^{x}, \quad x = 0,1,..., \quad \phi > 0 \tag{6.4}$$

gdzie: $E(X) = \mu$ oraz $V(X) = \mu + \phi^{-1}\mu^2$.

```
import scipy.stats as stats
import numpy as np
from scipy.optimize import minimize

def rn(mu, phi, n, rand):
    r = phi # phi_nb2
    p = r/(mu+r)
    return stats.nbinom.rvs(n=r, p=p, size=n, random_state=rand)

x = rn(mu = 1.5, phi = 5, n = 10000, rand = 2305)
mu = np.mean(x)
phi = np.mean(x)**2/(np.var(x,ddof=1)-np.mean(x))
print("MOM: mean= %.4f, phi_NB2= %.4f" % (mu,phi))

def L_nb2(par):
    phi = par[0]
    mu = par[1]
    logLik = -np.sum( stats.nbinom.logpmf(x, n=phi, p=phi/(phi+mu)) )
    return(logLik)

initParams = [1,1]
res = minimize(L_nb2, initParams, method= "Nelder-Mead")
print("MLE: mean= %.4f, phi_NB2= %.4f, logLik= %.2f" % (res.x[1],res.x[0],L_nb2(res.x)))
```

```
## MOM: mean= 1.5083, phi_NB2= 5.1984
## MLE: mean= 1.5083, phi_NB2= 5.1768, logLik= 16106.60
```

Gdy do wzoru (6.4) podstawimy $\phi = \alpha^{-1}$ i dokonamy prostych przekształceń to otrzymamy:

$$f(x \mid \mu, \alpha) = \frac{\Gamma(x+\alpha^{-1})}{\Gamma(\alpha^{-1})\Gamma(x+1)} \left(\frac{1}{\alpha\mu+1}\right)^{\alpha^{-1}} \left(\frac{\alpha\mu}{\alpha\mu+1}\right)^{x} \tag{6.5}$$

gdzie: $E(X) = \mu$ oraz $V(X) = \mu + \alpha\mu^2$.

```
import scipy.stats as stats
import numpy as np

def rn(mu, alpha, n, rand):
    r = 1/alpha # phi_nb2
    p = r/(mu+r)
    return stats.nbinom.rvs(n=r, p=p, size=n, random_state=rand)

x = rn(mu = 1.5, alpha = 9, n = 10000, rand = 2305)
mu = np.mean(x)
alpha = (np.var(x,ddof=1)-np.mean(x))/np.mean(x)**2
print("MOM: mean= %.4f, alpha_NB2= %.4f" % (mu,alpha))
```

```
## MOM: mean= 1.5355, alpha_NB2= 9.1284
```

Po zlogarytmowaniu wyrażenia (6.5) otrzymamy funkcję logarytmu wiarygodności o postaci:

$$L(x \mid \mu, \alpha) = \ln\Gamma(x+\alpha^{-1}) - \ln\Gamma(\alpha^{-1}) - \ln\Gamma(x+1) - \alpha^{-1}\ln(1+\alpha\mu) - x\ln(1+\alpha\mu) + x\ln(\alpha\mu) \tag{6.6}$$

6.2 Liniowy model ujemnej dwumianowej regresji

Do modelowania zmiennych licznikowych można stosować regresję Poissona jeśli zostało spełnione założenie równości średniej i wariancji. W przypadku wystąpienia zjawiska zawyżonej dyspersji tj. $E(X) \leq V(X)$ nie można prawidłowo wyznaczyć błędów standardowych ocen parametrów. Rozwiązaniem tego problemu może być zastosowanie takich rozkładów które są przeznaczone do modelowania nadmiernie rozproszonych danych. Jedną z wielu propozycji (Coly et al., 2016) jest rozkład ujemny dwumianowy z liniową (NB1) lub kwadratową (NB2) zależnością między średnią a wariancją (Cameron and Trivedi, 1998):

$$V(X) = \mu + \alpha\mu^p \tag{6.7}$$

- model NB1 dla $p = 1$:

$$E[(y_i - \mu_i)^2] = \phi\mu_i \quad \longrightarrow \quad \phi = E[(y_i - \mu_i)^2/\mu_i] \tag{6.8}$$

  Estymator parametru ϕ po zastosowaniu korekty:

$$\hat{\phi}_{\text{NB1}} = \frac{1}{n-k}\sum_{i=1}^{n}\frac{(y_i - \hat{\mu}_i)^2}{\hat{\mu}_i} \quad \text{gdzie} \quad \hat{\alpha}_{\text{NB1}} = \hat{\phi}_{\text{NB1}} - 1 \tag{6.9}$$

- model NB2 dla $p = 2$:

$$E[(y_i - \mu_i)^2 - \mu_i] = \alpha\mu_i^2 \quad \longrightarrow \quad \alpha = E[\{(y_i - \mu_i)^2 - \mu_i\}/\mu_i^2] \tag{6.10}$$

  Estymator parametru α po zastosowaniu korekty:

$$\hat{\alpha}_{\text{NB2}} = \frac{1}{n-k}\sum_{i=1}^{n}\frac{(y_i - \hat{\mu}_i)^2 - \hat{\mu}_i}{\hat{\mu}_i^2} \quad \text{gdzie} \quad \hat{\phi}_{\text{NB2}} = 1/\hat{\alpha}_{\text{NB2}} \tag{6.11}$$

W modelu który uwzględnia nadmierną dyspersje (model quasi-Poissona) błędy standardowe są modyfikowane w oparciu o wzór:

$$SE_Q(\beta) = SE_{Pois}(\beta) \cdot \sqrt{\hat{\phi}_{\text{NB1}}} \tag{6.12}$$

Warto podkreślić, że w równaniu (6.12) jest wykorzystany estymator $\hat{\phi}_{\text{NB1}}$ który jest powszechnie stosowany do szacowania nadmiernej dyspersji w modelu Poissona (McCullagh and Nelder, 1989). Dodajmy jeszcze, że estymator dyspersji (6.9) można zapisać w alternatywny sposób:

$$\hat{\phi}_{\text{NB1}} = \frac{\chi^2_P}{n-k} \tag{6.13}$$

gdzie: χ^2_P to suma kwadratów reszt Pearsona która jest często stosowana do oceny dobroci dopasowania modelu. Reszty Pearsona wyznaczamy na bazie regresji Poissona za pomocą wzoru:

$$r_i^P = \frac{y_i - \hat{u}_i}{\sqrt{\hat{u}_i}} \tag{6.14}$$

Wykorzystanie funkcji `statsmodels.discrete.discrete_model.NegativeBinomial` umożliwia oszacowanie modelu NB1 lub NB2 (opcja domyślna) oraz parametru α. Zastosowanie liniowego modelu ujemnej dwumianowej regresji zostanie zaprezentowane na przykładzie zestawu danych `nb_data`.

```
import pandas as pd
import statsmodels.api as sm
import patsy

df = pd.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/nb_data.dta')
```

```
df['prog'].replace([1.0,2.0,3.0],['General','Academic','Vocational'],inplace=True)
model = 'daysabs ~ math + C(prog, Treatment(reference="General"))'
y, x = patsy.dmatrices(model, df, return_type='dataframe')
x.columns = ['Intercept', 'Academic', 'Vocational','math']

nb = sm.NegativeBinomial(y,x,loglike_method='nb2').fit(disp=0)
a = nb.params.values[4]
print(nb.summary())
print("\nphi: ",round(1/a, 8), ", sqrt_phi: ", round((1/a)**0.5, 8))
```

```
##                     NegativeBinomial Regression Results
## ==============================================================================
## Dep. Variable:                 daysabs   No. Observations:                  314
## Model:               NegativeBinomial   Df Residuals:                      310
## Method:                           MLE   Df Model:                            3
## Date:                Mon, 19 Aug 2019   Pseudo R-squ.:                 0.03441
## Time:                        19:23:11   Log-Likelihood:                -865.63
## converged:                       True   LL-Null:                       -896.47
## Covariance Type:            nonrobust   LLR p-value:                 2.563e-13
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      2.6153      0.196     13.319      0.000       2.230       3.000
## Academic      -0.4408      0.183     -2.414      0.016      -0.799      -0.083
## Vocational    -1.2787      0.202     -6.331      0.000      -1.675      -0.883
## math          -0.0060      0.003     -2.390      0.017      -0.011      -0.001
## alpha          0.9683      0.100      9.729      0.000       0.773       1.163
## ==============================================================================
##
## phi:  1.03271369 , sqrt_phi:  1.01622522
```

Za pomocą wybranej pomocniczej regresji liniowej:

$$w_i = \hat{\alpha}_{\text{NB1}} + \epsilon_i \tag{6.15}$$

$$w_i = \hat{\alpha}_{\text{NB2}}\, \hat{u}_i + \epsilon_i \tag{6.16}$$

można weryfikować hipotezy statystyczne:

$$H_0 : \alpha_{\text{NB1}} = 0 \quad \text{vs} \quad H_1 : \alpha_{\text{NB1}} \neq 0 \tag{6.17}$$

$$H_0 : \alpha_{\text{NB2}} = 0 \quad \text{vs} \quad H_1 : \alpha_{\text{NB2}} \neq 0 \tag{6.18}$$

Warto podkreślić, że $\phi_{\text{NB1}} = \alpha_{\text{NB1}} + 1$ więc hipotezę (6.17) można przedstawić jako:

$$H_0 : \phi_{\text{NB1}} = 1 \quad \text{vs} \quad H_1 : \phi_{\text{NB1}} \neq 1 \tag{6.19}$$

Dodatkowo wyniki uzyskane za pomocą regresji (6.15) są tożsame wynikami uzyskanymi na podstawie wzorów:

$$\hat{\alpha}_{\text{NB1}} = E(w_i) \quad \text{oraz} \quad SE_{\hat{\alpha}_{1NB}} = \sqrt{V(w_i)/n} \tag{6.20}$$

gdzie:

$$w_i = \frac{(y_i - \hat{u}_i)^2 - y_i}{\hat{u}_i} \tag{6.21}$$

```
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import statsmodels.api as sm
import patsy

df = pd.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/nb_data.dta')
df['prog'].replace([1.0,2.0,3.0],['General','Academic','Vocational'],inplace=True)
model = 'daysabs ~ math + C(prog, Treatment(reference="General"))'
y, x = patsy.dmatrices(model, df, return_type='dataframe')
x.columns = ['Intercept', 'Academic', 'Vocational','math']

yp = sm.Poisson(y,x).fit(disp=0).predict()
w = ((y['daysabs']-yp)**2-y['daysabs'])/yp
n1 = sm.OLS(w,yp*0+1).fit(use_t=1) # regresja OLS dla alpha_NB1
n1.model.data.xnames = ['alpha_NB1']
n2 = sm.OLS(w,yp).fit(use_t=1)     # regresja OLS dla alpha_NB2
n2.model.data.xnames = ['alpha_NB2']
print(n1.summary().tables[1])
print("phi_NB1: ",n1.params.values[0]+1,'\n')
print(n2.summary().tables[1])
print("phi_NB2: ",1/n2.params.values[0])
```

```
## ==============================================================================
##                  coef    std err          t      P>|t|      [0.025      0.975]
## ------------------------------------------------------------------------------
## alpha_NB1      5.5105      0.768      7.178      0.000       4.000       7.021
## ==============================================================================
## phi_NB1:  6.51052636765949
##
## ==============================================================================
##                  coef    std err          t      P>|t|      [0.025      0.975]
## ------------------------------------------------------------------------------
## alpha_NB2      0.7986      0.117      6.835      0.000       0.569       1.028
## ==============================================================================
## phi_NB2:  1.2522191233195814
```

Funkcja `statsmodels.genmod.families.family.NegativeBinomial` umożliwia estymację modelu NB2 dla ustalonej wartości α. Warto zwrócić uwagę, że za pomocą tej funkcji możemy w sposób symulacyjny dobrać odpowienik parametr α. Dodajmy jeszcze, że do modelowania danych licznikowych można wykorzystać złożony rozkład Poissona–gamma który jest szczególnym przypadkiem rozkładu Tweedie:

$$E(X) = \mu \quad \text{oraz} \quad Var(X) = \phi\mu^p \tag{6.22}$$

W zależności od wartości parametru kształtu p można otrzymać kilka znanych rozkładów jako szczególne przypadki dystrybucji Tweedie:

- $p = 0$ - rozkład normalny,
- $0 < p < 1$ - rozkład nie jest zdefiniowany,
- $p = 1$ - rozkład Poissona,
- $1 < p < 2$ - rozkład Poissona–gamma,

- $p = 2$ - rozkład gamma,
- $2 < p < 3$ - dodatnie rozkłady stabilne,
- $p = 3$ - odwrotny rozkład Gaussa / rozkład Walda,
- $p > 3$ - dodatnie rozkłady stabilne,
- $p = \infty$ - ekstremalne stabilne rozkłady.

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

df = pd.read_stata('https://stats.idre.ucla.edu/stat/stata/dae/nb_data.dta')
df['prog'].replace([1.0,2.0,3.0],['General','Academic','Vocational'],inplace=True)
model = 'daysabs ~ math + C(prog, Treatment(reference="General"))'
B = 100
X = np.linspace(1.0001,1.9999,B)
n = [smf.glm(model, data=df,\
     family = sm.families.Tweedie(link = sm.families.links.log, var_power=i)).fit() for i in X]
res = [n[i].deviance for i in range(B)]
sol = n[np.argmin(res)]
sol.model.data.xnames = ['Inercept','Academic','Vocational','math']
print(sol.summary())
print("\np: ",X[np.argmin(res)])
```

```
##                  Generalized Linear Model Regression Results
## ==============================================================================
## Dep. Variable:                daysabs   No. Observations:                  314
## Model:                            GLM   Df Residuals:                      310
## Model Family:                 Tweedie   Df Model:                            3
## Link Function:                    log   Scale:                          2.3160
## Method:                          IRLS   Log-Likelihood:                    nan
## Date:                Mon, 19 Aug 2019   Deviance:                       902.82
## Time:                        19:23:15   Pearson chi2:                     718.
## No. Iterations:                    10
## Covariance Type:            nonrobust
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Inercept       2.6258      0.192     13.699      0.000       2.250       3.001
## Academic      -0.4410      0.178     -2.484      0.013      -0.789      -0.093
## Vocational    -1.2779      0.203     -6.305      0.000      -1.675      -0.881
## math          -0.0062      0.003     -2.423      0.015      -0.011      -0.001
## ==============================================================================
##
## p:  1.6363636363636363
```

Rozdział 7

Błąd standardowy estymatora

7.1 Średnia

Średnia jest parametrem który ma swoje zastosowanie dla rozkładów symetrycznych. Błąd standardowy estymatora średniej $\bar{x}$ można zapisać za pomocą wzoru:

$$SE_{\bar{x}} = \sqrt{s^2/n} \tag{7.1}$$

gdzie: s^2 to nieobciążony estymator wariancji: $\sum_{i=1}^{n}(x_i - \bar{x})^2/(n-1)$ oraz $\bar{x}$ to estymator średniej czyli $\sum_{i=1}^{n} x_i/n$. Dodatkowo x_i to kolejne elementy próby a n to liczebność próby.

```
import numpy as np
import scipy.stats as stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

MU = np.mean(y)
SE_mu = np.sqrt(np.var(y, ddof=1)/len(y))
conf = [ stats.norm.ppf(i, loc=MU, scale=SE_mu) for i in [0.025,0.975] ]
p = stats.norm.cdf(0, MU, SE_mu)

print("średnia:",MU,", błąd:",SE_mu)
print("95% przedział ufności:",conf)
print("\nH0: mu = 0 vs. H1: mu != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## średnia: 0.2707762688190597 , błąd: 0.16529559742815514
## 95% przedział ufności: [-0.053197148943156025, 0.5947496865812754]
##
## H0: mu = 0 vs. H1: mu != 0
## p-wartość: 0.1013938313336142
```

Ponieważ badamy hipotezę zerową $H_0 : \mu = 0$ więc takie same wyniki można uzyskać za pomocą funkcji regresji liniowej.

```
from scipy import stats
import pandas as pd

df = pd.DataFrame(data={'y':stats.norm.rvs(size=300,scale=3,random_state=4101)})
```

```
import statsmodels.formula.api as smf

m = smf.glm('y ~ 1', data=df).fit().summary().tables[1]
print(m)
```

```
## ==============================================================================
##                  coef    std err          z      P>|z|      [0.025      0.975]
## ------------------------------------------------------------------------------
## Intercept      0.2708      0.165      1.638      0.101      -0.053       0.595
## ==============================================================================
```

7.2 Proporcja

Obliczenie proporcji na podstawie próby binarnej czyli zawierającej tylko zera (porażka) i jedynki (sukces) sprowadza się do wyznaczenia średniej. Przykładem może być zmienna `plec` gdzie: `0` - kobieta, `1` - mężczyzna. Błąd standardowy dla oszacowanej frakcji $\hat{p}$ jest dany wzorem:

$$SE_{\hat{p}} = \sqrt{\left(\hat{p}(1-\hat{p})\right)/n} \quad \text{gdzie} \quad \hat{p} \in (0,1) \tag{7.2}$$

```
from scipy import stats
import numpy as np

y = stats.binom.rvs(n=1, p=0.3, loc=0, size=50, random_state=2305)

P = np.mean(y)
SE_pw = np.sqrt((P*(1-P))/len(y))
conf = [ stats.norm.ppf(i, loc=P, scale=SE_pw) for i in [0.025,0.975] ]
p = stats.norm.cdf(0.5, P, SE_pw)

print("proporcja:",P,", błąd:",SE_pw)
print("95% przedział ufności:",conf)
print("\nH0: p = 0.5 vs. H1: p != 0.5")
print("p-wartość:",2*min(p,1-p))
```

```
## proporcja: 0.28 , błąd: 0.06349803146555018
## 95% przedział ufności: [0.15554614523833055, 0.4044538547616695]
##
## H0: p = 0.5 vs. H1: p != 0.5
## p-wartość: 0.0005308739249216821
```

Więcej procedur budowy przedziału ufności dla proporcji jest zaimplementowanych do funkcji `statsmodels.stats.proportion.proportion_confint` w której metoda Walda (7.2) jest rozwiązaniem domyślnym.

```
from statsmodels.stats.proportion import proportion_confint
from scipy import stats

y = stats.binom.rvs(n=1, p=0.3, loc=0, size=50, random_state=2305)

print(proportion_confint(sum(y), 50, alpha=0.05, method='normal'))
```

```
## (0.15554614523833055, 0.4044538547616695)
```

7.3 Mediana

Jednym z bardziej popularnych odpornych estymatorów średniej jest mediana czyli drugi kwartyl który dzieli uporządkowany rosnąco zbiór obserwacji x na połowę. Wartość środkowa to $x_{(n+1)/2}$ lub $(x_{n/2} + x_{(n/2)+1})/2$ odpowiednio dla nieparzystej i parzystej liczebności danych. Do estymacji przedziałowej mediany często jest wykorzystywany błąd standardowy:

$$SE_{np_0} = \sqrt{p_0(1-p_0)\cdot n} \quad \text{gdzie} \quad p_0 = 0.5 \tag{7.3}$$

Po uporządkowaniu rosnąco danych wybieramy dwa elementy o indeksach v_1 i v_2 które wyznaczają dolną x_{v_1} i górną x_{v_2} granicę przedziału ufności gdzie:

$$v_1 = \lceil \Phi^{-1}(\alpha/2,\ np_0, SE_{np_0}) \rceil, \quad v_2 = \lceil \Phi^{-1}(1-\alpha/2,\ np_0, SE_{np_0}) \rceil \tag{7.4}$$

Dodajmy, że dla mediany czyli $p_0 = 0.5$ otrzymamy $np_0 = n/2$ oraz $SE_{np_0} = \sqrt{n/4}$.

W innym rozwiązaniu (McKean and Schrader, 1984) wykorzystywany jest błąd standardowy mediany dany wzorem:

$$SE_{\hat{m}} = \frac{x_{n-k+1} - x_k}{2 \cdot z_{0.995}} \quad \text{dla} \quad k = \frac{n+1}{2} - z_{0.995} \cdot \sqrt{\frac{n}{4}} \tag{7.5}$$

gdzie kwantyle z rozkładu normalnego: $\Phi^{-1}(\alpha/2,\ \hat{m}, SE_{\hat{m}})$ oraz $\Phi^{-1}(1-\alpha/2,\ \hat{m}, SE_{\hat{m}})$ to odpowiednio dolna i górna granica przedziału ufności.

Zwróćmy uwagę, że powyższe metody nie uwzględniają wartości wiązanych a więc są przeznaczone dla danych ciągłych. W artykule (Iwasaki Manabu, 2005) jest przedstawiona metoda która uwzględnia występowanie duplikatów:

$$v = \left[n/2 - z_{1-\alpha/2} \cdot \sqrt{n/4} \right] + 1, \quad v' = \left[n/2 - z_{1-\alpha/2} \cdot \sqrt{n/4} + 0{,}5 \right] + 1 \tag{7.6}$$

$$\begin{array}{rl} x_v\,,\ x_{n+1-v} & \text{dla} \quad v = v' \\ (x_{v-1} + x_v)/2\,,\ (x_{n+1-v} + x_{(n+1-v)+1})/2 & \text{dla} \quad v = v' - 1 \end{array} \tag{7.7}$$

W literaturze (Wilcox Rand, 2017) można znaleźć jeszcze wiele innych propozycji. Za przykład może posłużyć estymator Harrella–Davisa (Harrel and Davis, 1982) dla mediany lub wybranych kwantyli:

$$HD_q = \sum_{i=1}^{n} \left[\Big(B(i/n,\ a,\ b) - B((i-1)/n,\ a,\ b) \Big) x_i \right] \tag{7.8}$$

z wykorzystaniem rozkładu beta (patrz rozdział 4):

$$B(t, a, b) = \int_0^t \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}\, dx \tag{7.9}$$

gdzie: $a = (n+1)q$ oraz $b = (n+1)(q-1)$ dla $q = 0.5$ w przypadku drugiego kwartyla czyli mediany. Błąd standardowy estymatora mediany Harrella–Davisa można obliczyć za pomocą funkcji `scipy.stats.mstats.hdquantiles_sd` do której zaimplementowano metodę jackknife. Inne rozwiązanie to wykorzystanie metody bootstrap (patrz podrozdział 7.8) w której można wykorzystać estymator Harrella–Davisa zaimplementowany w funkcji `scipy.stats.mstats.hdquantiles`.

```
from scipy import stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

MD = stats.mstats.hdquantiles(y,[0.5])[0]
SE_md = stats.mstats.hdquantiles_sd(y,[0.5])[0]
conf = [ stats.norm.ppf(i, loc=MD, scale=SE_md) for i in [0.025,0.975] ]
```

```
p = stats.norm.cdf(0, MD, SE_md)

print("medianaHD:",MD,", błąd:",SE_md)
print("95% przedział ufności:",conf)
print("\nH0: md = 0 vs. H1: md != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## medianaHD: 0.30109081321402076 , błąd: 0.1584119409852424
## 95% przedział ufności: [-0.009390885838138907, 0.6115725122661804]
##
## H0: md = 0 vs. H1: md != 0
## p-wartość: 0.05734360449353051
```

7.4 Wariancja

Błąd standardowy estymatora wariancji (Coeurjolly et al., 2009) można obliczyć za pomocą wzoru:

$$SE_{s^2} = \sqrt{S^2/n} \tag{7.10}$$

gdzie S^2 to nieobciążony estymator wariancji dla zmiennej $z_i = (x_i - \bar{x})^2$.

```
from scipy import stats
import numpy as np

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

V = np.var(y,ddof=1)
z = (y-np.mean(y))**2
SE_v = np.sqrt(np.var(z,ddof=1)/len(z))
conf = [ stats.norm.ppf(i, loc=V, scale=SE_v) for i in [0.025,0.975] ]
p = stats.norm.cdf(10, V, SE_v)

print("wariancja:",V,", błąd:",SE_v)
print("95% przedział ufności:",conf)
print("\nH0: var = 10 vs. H1: var != 10")
print("p-wartość:",2*min(p,1-p))
```

```
## wariancja: 8.19679035873922 , błąd: 0.6734837212813637
## 95% przedział ufności: [6.876786520853734, 9.516794196624705]
##
## H0: var = 10 vs. H1: var != 10
## p-wartość: 0.007418799795601672
```

Inne rozwiązanie (Fuchs and Krautenbacher, 2016) jest przedstawione poniżej:

$$SE_{var} = \sqrt{\left(\frac{1}{2(n-2)} + \frac{1}{2n} - \frac{2}{n-3}\right) v^2 + \left(\frac{3}{n-3} - \frac{2}{n-2}\right) m_4} \tag{7.11}$$

gdzie: $m_4 = \sum_{i=1}^{n}(x_i - \bar{x})^4/n$ to czwarty moment centralny oraz $v = \sum_{i-1}^{n}(x_i - \bar{x})^2/(n-1)$ to nieobciążony estymator wariancji.

```
from scipy import stats
import numpy as np

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

n = len(y)
v2 = np.var(y,ddof=1)**2
m4 = stats.moment(y, moment=4)
SE_v = np.sqrt((1/(2*n-4)+1/(2*n)-2/(n-3))*v2+(3/(n-3)-2/(n-2))*m4)
print("wariancja:",v2**0.5,", błąd:",SE_v)
```

```
## wariancja: 8.19679035873922 , błąd: 0.6768982673783448
```

7.5 Średnia ucięta

W przypadku gdy dane nie pochodzą z rozkładu normalnego (np. gdy rozkład jest skośny, w danych występują obserwacje odstające itp.) wnioskowanie o populacji z wykorzystaniem średniej nie jest dobrym wyborem. Jednym z możliwych rozwiązań jest zweryfikowanie hipotezy statystycznej dotyczącej średniej uciętej (Tukey and McLaughlin, 1963) w oparciu o rozkład t-Studenta ze stopniami swobody $df = n - 2\lfloor nG \rfloor - 1$. Estymator średniej uciętej jest obliczany na podstawie próbki z której została usunięta pewna frakcja skrajnych obserwacji a jego błąd standardowy jest dany wzorem:

$$SE_{\bar{x}_t} = s_w/(1-2G)\sqrt{n} \tag{7.12}$$

gdzie: sw to odchylenie standardowe obliczone na podstawie próbki poddanej procesowi winsoryzacji.

```
from scipy import stats
import numpy as np

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

tMU = stats.mstats.trimmed_mean(y,limits=(0.2,0.2))
SE_tmu = stats.mstats.trimmed_stde(y,limits=(0.2,0.2))
conf = stats.mstats.trimmed_mean_ci(y, limits=(0.2, 0.2))
df = len(y) - 2 * np.floor(0.2 * len(y)) - 1
p = stats.t.cdf(0, df, tMU, SE_tmu)

print("średnia ucięta:",tMU,", błąd:",SE_tmu)
print("95% przedział ufności:",conf)
print("\nH0: tmu = 0 vs. H1: tmu != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## średnia ucięta: 0.2929142000036129 , błąd: 0.17589712092651877
## 95% przedział ufności: [-0.05418454  0.64001294]
##
## H0: tmu = 0 vs. H1: tmu != 0
## p-wartość: 0.09761041405364616
```

Warto dodać, że dla parametru ucięcia G równego 0 wynik będzie tożsamy z testem t-Studenta który został zaimplementowany do funkcji `scipy.stats.ttest_1samp`.

7.6 Skośność

W rozkładzie normalnym (rozkład symetryczny) parametr skośności jest równy zero. Wartość tego parametru może być większa lub mniejsza od zera dla rozkładu odpowiednio prawostronnie skośnego lub lewostronnie skośnego. Dla podstawowej wersji parametru skośności $g1 = m_3/m_2^{3/2}$ błąd standardowy można zapisać za pomocą wzoru:

$$SE_{g1} = \sqrt{\frac{6(n-2)}{(n+1)(n+3)}} \tag{7.13}$$

gdzie: m_2 i m_3 to odpowiednio drugi i trzeci moment centralny, n to liczebność próby.

```
from scipy import stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

g1 = stats.skew(y)
n = len(y)
SE_g1 = ((6*(n-2))/((n+1)*(n+3)))**0.5
conf = [ stats.norm.ppf(i, loc=g1, scale=SE_g1) for i in [0.025,0.975] ]
p = stats.norm.cdf(0, g1, SE_g1)

print("skośność:",g1,", błąd:",SE_g1)
print("95% przedział ufności:",conf)
print("\nH0: skew = 0 vs. H1: skew != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## skośność: 0.043243631316632496 , błąd: 0.14001649284686388
## 95% przedział ufności: [-0.23118365190483087, 0.3176709145380958]
##
## H0: skew = 0 vs. H1: skew != 0
## p-wartość: 0.7574381454853335
```

Warto dodać, że zostały opracowane również inne estymatory skośności (Wright and Herrington, 2011) które są modyfikacjami parametru $g1$. Ciekawe rozwiązanie na bazie transformacji skośności (D'Agostino, 1970) zostało zaimplementowane do funkcji `scipy.stats.skewtest` i bada hipotezę zerową $H_0 : S = 0$.

```
from scipy import stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

print(stats.skewtest(y))
```

```
## SkewtestResult(statistic=0.3128664753170487, pvalue=0.7543821086322899)
```

7.7 Kurtoza

Podstawowa wersja parametru kurtozy jest dana wzorem $g2 = m_4/m_2^2 - 3$ i dla rozkładu normalnego przyjmuje wartość zero. Jeśli kurtoza jest większa od zera to rozkład jest spiczasty tzw. rozkład leptokurtyczny. Natomiast gdy kurtoza jest mniejsza od zera to rozkład jest spłaszczony tzw. platykurtyczny. Te określenia są formułowane w stosunku do rozkładu normalnego tzw. rozkładu mezokurtycznego. Błąd standardowy dla estymatora $g2$ można przedstawić za pomocą wzoru:

$$SE_{g2} = \sqrt{\frac{24n(n-2)(n-3)}{(n+1)^2(n+3)(n+5)}} \tag{7.14}$$

gdzie: m_2 i m_4 to odpowiednio drugi i czwarty moment centralny, n to liczebność próby.

```
from scipy import stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

g2 = stats.kurtosis(y)
n = len(y)
SE_g2 = ((24*n*(n-2)*(n-3))/((n+1)**2*(n+3)*(n+5)))**0.5
conf = [ stats.norm.ppf(i, loc=g2, scale=SE_g2) for i in [0.025,0.975] ]
p = stats.norm.cdf(0, g2, SE_g2)

print("kurtoza:",g2,", błąd:",SE_g2)
print("95% przedział ufności:",conf)
print("\nH0: kurt = 0 vs. H1: kurt != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## kurtoza: 0.03206629047789944 , błąd: 0.27587660663283947
## 95% przedział ufności: [-0.5086419226995899, 0.5727745036553886]
##
## H0: kurt = 0 vs. H1: kurt != 0
## p-wartość: 0.9074669505499613
```

Podobnie jak w przypadku parametru skośności (patrz podrozdział 7.6) zostały wprowadzone pewne modyfikacje parametru $g2$ (Wright and Herrington, 2011). Istotność statystyczną parametru $g2 + 3$ można badać za pomocą transformacji (Anscombe and Glynn, 1983). To rozwiązanie jest dostępne dzięki funkcji `scipy.stats.kurtosistest` i bada hipotezę zerową $H_0 : \ K = 3$.

```
from scipy import stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

print(stats.kurtosistest(y))
```

```
## KurtosistestResult(statistic=0.3158947534000299, pvalue=0.7520823943589993)
```

7.8 Bootstrap

Metoda bootstrap jest często stosowane gdy nie wiemy z jakiego rozkładu pochodzi zmienna losowa. Polega ona na wielokrotnym losowaniu ze zwracaniem z próby w celu wyznaczenia B estymatorów $\hat{\theta}$. Inaczej mówiąc, tworzymy wektor $\hat{\theta}_i^* = [\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_B]$ i na jego podstawie obliczamy średnią (szacunek estymatora) oraz odchylenie standardowe (błąd estymatora). Jednym z możliwych rozwiązań jest zastosowanie pętli `for` w połączeniu z funkcją `numpy.random.choice` a następnie obliczenie wartości estymatora i jego błąd standardowy odpowiednio z wykorzystaniem funkcji `numpy.mean` oraz `numpy.std`. Poniżej przykład dla parametru skośności (patrz podrozdział 7.6) dla metody bootstrap.

```
import numpy as np
import scipy.stats as stats

y = stats.norm.rvs(size=300,scale=3,random_state=4101)

B = 10000
S = [stats.skew(np.random.choice(y,size=len(y))) for i in range(B)]
```

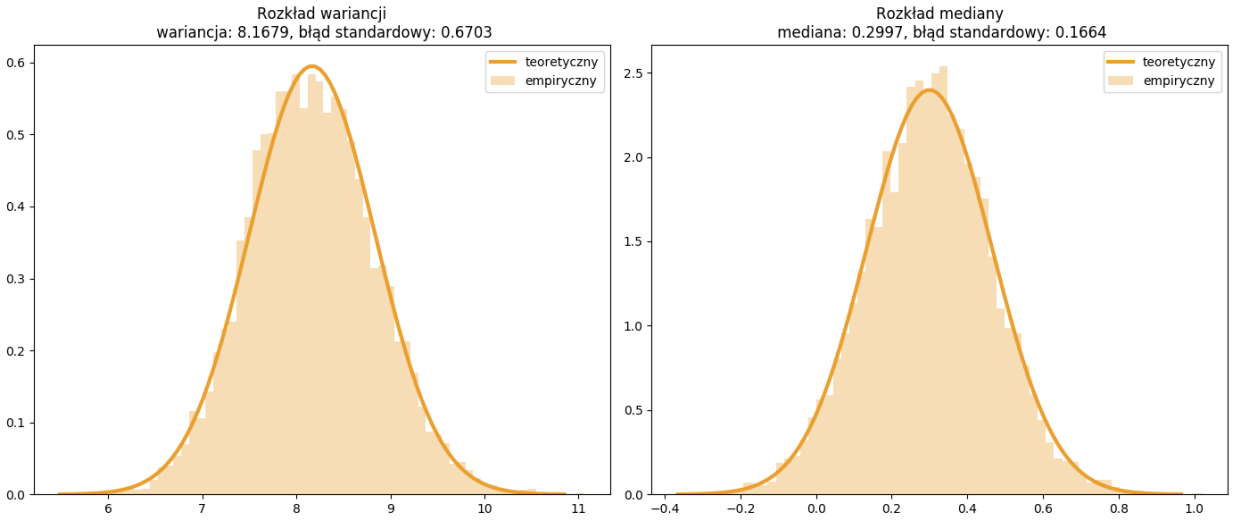
```
conf = np.percentile(S, [2.5, 97.5])
p = np.mean(np.less(S,[0]))

print("skośność:",np.mean(S),", błąd:",np.std(S,ddof=1))
print("95% przedział ufności:",conf)
print("\nH0: skew = 0 vs. H1: skew != 0")
print("p-wartość:",2*min(p,1-p))
```

```
## skośność: 0.038253619242244886 , błąd: 0.15344657596149805
## 95% przedział ufności: [-0.24559301  0.35166342]
##
## H0: skew = 0 vs. H1: skew != 0
## p-wartość: 0.8326
```

Do wyznaczenia wektora $\hat{\theta}_i^*$ można wykorzystać funkcję `astropy.stats.bootstrap` która jest przeznaczona do generowania próbek bootstrap. Poniżej przykład dla wariancji (patrz podrozdział 7.4) oraz mediany (patrz podrozdzial 7.3).

```
import numpy as np
import scipy.stats as stats
from astropy.stats import bootstrap
import matplotlib.pyplot as plt

y = stats.norm.rvs(size=300,scale=3,random_state=4101)
B = 10000
stat = lambda x: (np.var(x,ddof=1), stats.mstats.hdquantiles(x,[0.5])[0])
boot = bootstrap(np.array(y), bootnum=B, bootfunc=stat)
V  = boot[:,0] # próbka bootstrap dla wariancji
MD = boot[:,1] # próbka bootstrap dla mediany

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.hist(V, density=True,alpha=0.35,bins=60,color="#ea9f2f",label="empiryczny")
xV = np.linspace(np.mean(V)-4*np.std(V,ddof=1),np.mean(V)+4*np.std(V,ddof=1),100)
ax1.plot(xV,stats.norm.pdf(xV,loc=np.mean(V),scale=np.std(V,ddof=1)),
         lw=3,color="#ea9f2f",label="teoretyczny")
ax2.hist(MD, density=True,alpha=0.35,bins=60,color="#ea9f2f",label="empiryczny")
xMD = np.linspace(np.mean(MD)-4*np.std(MD,ddof=1),np.mean(MD)+4*np.std(MD,ddof=1),100)
ax2.plot(xMD,stats.norm.pdf(xMD,loc=np.mean(MD),scale=np.std(MD,ddof=1)),\
         lw=3,color="#ea9f2f",label="teoretyczny")
ax1.set_title("Rozkład wariancji\n wariancja: %.4f, błąd standardowy: %.4f" \
% (np.mean(V),np.std(V,ddof=1)))
ax2.set_title("Rozkład mediany\n mediana: %.4f, błąd standardowy: %.4f" \
% (np.mean(MD),np.std(MD,ddof=1)))
ax1.legend(); ax2.legend()
fig.tight_layout()
plt.savefig('boot01.png')
```

W metodach symulacyjnych można również wykorzystać generatory liczb losowych jeśli wiemy z jakiego rozkładu pochodzi próba. Przykładowo dla proporcji będzie to rozkład dwumianowy o parametrach np. $p = 0,3$ oraz $n = 50$ (patrz podrozdział 7.2).



Rysunek 7.1: Rozkład empiryczny (bootstrap) vs rozkład teoretyczny (normalny).

```
import numpy as np
import scipy.stats as stats

mc = [np.mean(stats.binom.rvs(n=1, p=0.3, loc=0, size=50)) for i in range(10000)]
P_mc = np.mean(mc)
SE_p_mc = np.std(mc,ddof=1)
conf = np.percentile(mc, [2.5, 97.5])
p = np.mean(np.less(mc,[0.5]))

print("proporcja:",P_mc,", błąd:",SE_p_mc,"\n95% przedział ufności:",conf)
print("\nH0: p = 0.5 vs. H1: p != 0.5","\np-wartość:",2*min(p,1-p))
```

```
## proporcja: 0.299728 , błąd: 0.06472607637186306
## 95% przedział ufności: [0.18 0.42]
##
## H0: p = 0.5 vs. H1: p != 0.5
## p-wartość: 0.0027999999999999137
```

Rozdział 8

Porównanie zmiennych niezależnych

8.1 Porównanie średnich

Test t-Studenta / Test t-Welcha. Do porównania dwóch średnich tj. do zweryfikowania hipotezy $H_0 : \mu_1 = \mu_2$ najczęściej proponowany jest test t-Studenta Wymaga on spełnienia dwóch warunków: normalność rozkładu oraz jednorodności wariancji. Statystyka klasycznego testu dla dwóch średnich: $(\bar{x}_1 - \bar{x}_2)/\sqrt{d_1 + d_2}$ ma rozkład t-Studenta ze stopniami swobody $df = n_1 + n_2 - 2$. Jeśli wariancje w próbkach nie są równe to zalecane jest stosowanie poprawki Welcha (Derrick and White, 2016) która polega na modyfikacji stopni swobody:

$$df_{\text{Welch}} = \frac{(d_1 + d_2)^2}{\frac{d_1}{n_1 - 1} + \frac{d_2}{n_2 - 1}} \tag{8.1}$$

gdzie: s_k^2 to wariancja, n_k to liczebność próby dla $k = 1, 2$ oraz $d_k = s_k^2/n_k$.

Dzięki funkcji `pingouin.ttest` jest dostępna klasyczna wersja testu t-Studenta oraz z poprawką Welcha.

```
import scipy.stats as stats
import pingouin as pg

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

print(pg.ttest(x,y, correction=True)[['T','dof','CI95%','p-val']])
```

```
##             T    dof           CI95%     p-val
## T-test -3.448  23.26  [-2.95, -0.74]  0.002164
```

Anova / Welch-Anova. Klasyczna analiza wariancji - inaczej ANOVA to rozwinięcie testu t-Studenta dla więcej niż dwóch zmiennych niezależnych. Inaczej mówiąc w przypadku porównania średnich z dwóch grup wyniki z obu procedur są tożsame. Funkcja `pingouin.anova` realizuje jedno lub dwuczynnikową analizę wariancji z interakcją. Natomiast do funkcji `pingouin.welch_anova` jest zaimplementowana metoda Welcha dla jednej zmiennej grupującej.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg
```

```
y = np.concatenate((stats.norm.rvs(0, 1, size=20, random_state=2305),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4101),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4026)))
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)

print(pg.welch_anova(dv='y', between='g', data=pd.DataFrame({'y':y,'g':g})))
```

```
##   Source  ddof1   ddof2      F     p-unc
## 0      g      2  30.937  6.448  0.004562
```

Dalsza analiza. Po odrzuceniu hipotezy zerowej w analizie wariancji stosujemy testy do porównań wielokrotnych. Jednym z bardziej popularnych tzw. testów po fakcie dla grup niezależnych jest procedura Tukeya lub seria testów t-Studenta z odpowiednią korektą p-wartości. Są one zaimplementowane odpowiednio do funkcji `pingouin.pairwise_tukey` oraz `pingouin.pairwise_ttests`. Natomiast w warunkach heteroskedastyczności można wykonać serię testów t-Welcha z odpowiednią korektą p-wartości lub test Gamesa-Howella. Są one dostępne odpowiednio w funkcji `pingouin.pairwise_ttests` oraz `pingouin.pairwise_gameshowell`. Wiele ciekawych rozwiązań np. test Tamhane T2 zostało zaimplementowanych do pakietu `scikit-posthocs` który bazuje na bibliotece `PMCMRplus`/`PMCMR` dla programu R.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.norm.rvs(0, 1, size=20, random_state=2305),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4101),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4026)))
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)

print(pg.pairwise_gameshowell(dv='y', between='g',
                              data=pd.DataFrame({'y':y,'g':g})))
```

```
##      A    B  mean(A)  mean(B)   diff  ...      T      df      pval  efsize  eftype
## 0  1.0  2.0    0.155    2.001 -1.846  ... -3.448  23.261  0.002070  -1.069  hedges
## 1  1.0  3.0    0.155    0.857 -0.702  ... -1.539  25.039  0.275781  -0.477  hedges
## 2  2.0  3.0    2.001    0.857  1.143  ...  1.730  36.818  0.196416   0.536  hedges
##
## [3 rows x 12 columns]
```

8.2 Porównanie rang

Test Manna-Whitneya. Przy założeniu, że dwa badane rozkłady mają ten sam kształt (takie same wariancje, skośność itp.) można zweryfikować hipotezę zerową o postaci $H_0: F(x) = G(y + \Delta)$ w której parametr Δ określa przesunięcie dystrybuanty $G(y)$ względem dystrybuanty $F(x)$ (Divine et al., 2018). Inaczej mówiąc rozmieszczenie rozkładów $F(x)$ i $G(y)$ różni się w zależności od Δ. Parametr przesunięcia można oszacować za pomocą estymatorora Hodgesa-Lehmanna:

$$\hat{\Delta} = \text{mediana}\{x_i - y_j \ : \ i = 1, \ \ldots n_1 \ ; \ j = 1, \ \ldots n_2\} \tag{8.2}$$

Warto zaznaczyć, że parametr Δ w funkcji `scipy.stats.mannwhitneyu` oraz `pingouin.mwu` ma stałą wartość równą zero. Zatem rozważana hipoteza zerowa ma postać:

$$H_0: \ \Delta = 0 \quad \text{vs.} \quad H_1: \ \Delta \neq 0 \tag{8.3}$$

Równoważnym zapisem może być:

$$H_0 : F(x) = G(y) \quad \text{vs.} \quad H_1 : F(x) \neq G(y) \tag{8.4}$$

Statystyka testowa:

$$Z = \frac{|W - \frac{n_1 n_2}{2}| - 0,5}{\sqrt{\frac{n_1 n_2 (n_1+n_2+1)}{12} - \frac{n_1 n_2 \sum_{i=1}^{c}(t^3-t)}{12(n_1+n_2)(n_1+n_2-1)}}} \tag{8.5}$$

gdzie: c to liczba grup pomiarów wiązanych, t_i to liczba pomiarów wiązanych w i-tej grupie pomiarów wiązanych.

```
import scipy.stats as stats
import pingouin as pg
import numpy as np

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

hl = np.median(x[:, None] - y)
df = pg.mwu(x,y)
df['LH-median'] = hl
print(df)
```

```
##      U-val     p-val   RBC  CLES  LH-median
## MWU   96.0  0.005115  0.52  0.76  -2.003235
```

Test Brunera-Munzela. Dobrą alternatywną dla testu sumy rang Wilcoxona w warunakch heteroskedastyczności może być test Brunera-Munzela (Brunner and Munzel, 2000) dostępny dzięki funkcji `scipy.stats.brunnermunzel`. Wersja permutacyjna tego testu (Neubert and Brunner, 2007) jest zalecana dla przypadku małolicznych próbek o nierównych liczebnościach. W tym teście hipoteza zerowa dla równości stochastycznej ma postać:

$$H_0 : p = 0,5 \quad \text{vs.} \quad H_1 : p \neq 0,5 \tag{8.6}$$

gdzie p określa prawdopodobieństwo tego, że obserwacje w grupie pierwszej są zazwyczaj mniejsze niż w grupie drugiej.

Wynika z tego, że prawdopodobieństwo zdarzenia przeciwnego (obserwacje w grupie pierwszej x są zazwyczaj większe niż w grupie drugiej y) jest także równe $0,5$. Zatem w hipotezie zerowej zakładamy, że wartości w obu próbkach mają porównywalne wartości tzn. wartości z pierwszej próbki nie mają tendencji do mniejszych/większych wartości niż w próbce drugiej. Estymację tego prawdopodobieństwa można dokonać w dwojaki sposób:

$$\hat{p} = P(x < y) + 0,5 \cdot P(x = y) \quad \text{lub} \quad \hat{p} = \frac{\bar{r}_2 - (n_2 + 1) \cdot 0,5}{n_1} \tag{8.7}$$

gdzie $\bar{r}_2$ to średnia ranga dla drugiej zmiennej a rangi są liczone na podstawie próbki zbiorczej.

Warto dodać, że na podstawie estymatora prawdopodobieństwa $\hat{p}$ można obliczyć statystykę testu sumy rang Wilcoxona na podstawie wzoru:

$$W = (1 - \hat{p}) n_1 n_2 \tag{8.8}$$

Statystyka testu Brunera-Munzela:

$$BM = \frac{n_1 n_2 (\bar{r}_1 - \bar{r}_2)}{(n_1 + n_2)\sqrt{n_1 s_1^2 + n_2 s_2^2}} \tag{8.9}$$

ma rozkład t-Studenta ze stopniami swobody według formuły:

$$df_{\text{Satterthwaite}} = \frac{(d_1 + d_2)^2}{\frac{d_1}{n_1 - 1} + \frac{d_2}{n_2 - 1}} \tag{8.10}$$

gdzie $d_k = n_k \cdot s_k^2$ to iloczyn liczebności próby n_k oraz wariancji s_k^2 dla każdej k-tej grupy.

Wariancja jest zdefiniowana w następujący sposób:

$$s_k^2 = \frac{1}{n_k - 1} \sum_{i=1}^{n_k} \left(r_{ki} - w_{ki} - \bar{r}_k + \frac{n_k + 1}{2} \right)^2 \tag{8.11}$$

gdzie: $\bar{r}_k$ oznacza średnią rangę k-tej grupy z próbki zbiorczej, r_k to rangi dla k-tej grupy z próbki zbiorczej, w_k to rangi dla k-tej grupy.

```
import warnings
warnings.filterwarnings("ignore")
import scipy.stats as stats
import PyNonpar
from PyNonpar import *

x = stats.norm.rvs(0, 1, size=20, random_state=2305).tolist()
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101).tolist()

res = PyNonpar.twosample.brunner_munzel_test(x,y)
print("BM= %.4f, df= %.4f, pvalue= %.4f" % (res[1],res[2],res[3]))
```

```
## BM= 2.9380, df= 20.6352, pvalue= 0.0080
```

Test Kruskala-Wallisa. Nieparametrycznym odpowiednikiem analizy wariancji jest test Kruskala-Walisa jako rozszerzenie testu sumy rang Wilcoxona na kilka grup. W tej metodzie zakładamy, że próbki pochodzą z tego samego rozkładu o dowolnym kształcie. Oznacza to, że rozkład w grupach nie musi być normalny ale w dalszym ciągu zakładamy homoskedastyczność wariancji. Dokładny rozkład statystyki Kruskala-Wallisa można przybliżać za pomocą metod permutacyjnych lub takich dystrybuant jak: chi-kwadrat, F-Snedecora oraz beta (Meyer and Seaman, 2013).

Statystyka testowa:

$$\chi^2_{KW} = \left(1 - \frac{\sum_{i=1}^{c} (t_i^3 - t_i)}{n^3 - n} \right)^{-1} \left[\frac{12}{n(n+1)} \left(\sum_{j=1}^{k} \frac{R_j^2}{n_j} \right) - 3(n+1) \right] \tag{8.12}$$

gdzie: n to liczebność z wszystkich k grup, n_j to liczebność w j-tej grupie, R_j to suma rang w j-tej grupie, c to liczba grup pomiarów wiązanych, t_i to liczba pomiarów wiązanych w i-tej grupie pomiarów wiązanych.

Poniżej implementacja wersji permutacyjnej testu Kruskala-Wallisa:

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.expon.rvs(0, 1, size=20, random_state=2305),
                    stats.expon.rvs(0.5, 1, size=20, random_state=4101),
                    stats.expon.rvs(1, 1, size=20, random_state=4026)))
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)
```

```
kw = pg.kruskal(dv='y', between='g', data=pd.DataFrame({'y':y,'g':g}))

H = kw["H"][0]
B = 1000
h = [list(pg.kruskal(dv='y', between='g',\
          data=pd.DataFrame({'y':y,'g':np.random.choice(g,size=60)}))["H"])[0]\
          for i in range(B)]
perm = np.greater(h,[H]).mean()
kw["p-perm"] = perm
print(kw)
```

```
##         Source  ddof1     H     p-unc  p-perm
## Kruskal      g      2  7.61  0.022254   0.017
```

ANOVA-rank. Warto zauważyć, że problem heterogeniczności wariancji można uwzględnić za pomocą testu Brunner-Dette-Munk (Brunner et al., 1997) w którym można także testować interakcję w dwuczynnikowej analizie wariancji. Jednak ta metoda nie jest dostępna w pakietach `scipy.stats` oraz `pingouin`. Alternatywą może być zastosowanie procedury wykorzystującej rozkład F-Snedecora która polega na porangowaniu danych i zastosowaniu klasycznej metody ANOVA. Innym rozwiązaniem może być wykorzystanie ważonej metody najmniejszych kwadratów lub odpornych błędów standardowych z wykorzystaniem funkcji `statsmodels.stats.anova.anova_lm`.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.expon.rvs(0, 1, size=20, random_state=2305),
                    stats.expon.rvs(0.5, 1, size=20, random_state=4101),
                    stats.expon.rvs(1, 1, size=20, random_state=4026)))
r = stats.rankdata(y)
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)
d = pd.DataFrame({"y":y,"g":g,"r":r})
print(pg.anova(dv='r', between='g', data=d))
```

```
##   Source  ddof1  ddof2      F     p-unc    np2
## 0      g      2     57  4.221  0.019527  0.129
```

Dalsza analiza. Po odrzuceniu hipotezy zerowej w teście Kruskala-Wallisa można dokonać bardziej szczegółowej analizy czyli przeprowadzić porównania wielokrotne. Popularnym rozwiązaniem jest zastosowanie serii testów sumy rang Wilcoxona. Ta metoda jest dostępna dzięki funkcji `pingouin.pairwise_ttests` z zaznaczeniem opcji `parametric=False`. Jednak szerszy zestaw testów post hoc dla grup niezależnych znajdziemy w pakiecie `scikit-posthocs`. Poniżej przykład testu Conovera.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg
from scikit_posthocs import posthoc_conover

y = np.concatenate((stats.expon.rvs(0, 1, size=20, random_state=2305),
                    stats.expon.rvs(0.5, 1, size=20, random_state=4101),
                    stats.expon.rvs(1, 1, size=20, random_state=4026)))
r = stats.rankdata(y)
```

```
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)
d = pd.DataFrame({"y":y,"g":g,"r":r})
print(posthoc_conover(d, val_col='y', group_col='g', p_adjust='holm'))
```

```
##           1.0       2.0       3.0
## 1.0 -1.000000  0.221096  0.015936
## 2.0  0.221096 -1.000000  0.221096
## 3.0  0.015936  0.221096 -1.000000
```

8.3 Porównanie wariancji

Test Z-diff / Test Z-ratio. Jeśli chcemy porównać dwie wariancje to rozważamy hipotezy statystyczne o postaci:

$$H_0: \sigma_1^2 = \sigma_2^2 \quad \text{vs.} \quad H_1: \sigma_1^2 \neq \sigma_2^2 \tag{8.13}$$

Zauważmy, że powyższą hipotezę statystyczną można sprowadzić do zapisu:

$$H_0: \sigma_1^2/\sigma_2^2 = 1 \quad \text{vs.} \quad H_1: \sigma_1^2/\sigma_2^2 \neq 1 \tag{8.14}$$

Statystyka testowa:

$$Z_{ratio} = \frac{(s_1^2/s_2^2) - 1}{SE_{ratio}} \tag{8.15}$$

gdzie: $SE_{ratio} = \frac{1}{s_2^2}\sqrt{SE_1^2 + r_0^2 \cdot SE_2^2}$ to błąd standardowy ilorazu dwóch wariancji oraz $SE = \sqrt{s^2/n}$ to błąd standardowy wariancji s^2 dla przekształconej zmiennej $(x_i - \bar{x})^2$, r_0^2 to iloraz wariancji podniesiony do drugiej potęgi.

```
import scipy.stats as stats
import numpy as np

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

z1 = (x-np.mean(x))**2
z2 = (y-np.mean(y))**2
ratV = np.var(x,ddof=1)/np.var(y,ddof=1)
SE = np.sqrt(np.var(z1,ddof=1)/len(x)+ratV**2*np.var(z2,ddof=1)/len(y))/np.var(y,ddof=1)
conf = [stats.norm.ppf(i,ratV,SE) for i in [0.025,0.975]]
h0 = 1
p = stats.norm.cdf(h0,ratV,SE)

print("iloraz wariancji:",ratV,", błąd:",SE)
print("95% przedział ufności:",conf)
print("\nH0: rVar = %.0f vs. H1: rVar != %.0f" % (h0,h0))
print("p-wartość:",2*min(p,1-p))
```

```
## iloraz wariancji: 0.2555508988591833 , błąd: 0.0975515172257403
## 95% przedział ufności: [0.06435343845949357, 0.446748359258873]
##
## H0: rVar = 1 vs. H1: rVar != 1
## p-wartość: 2.3314683517128287e-14
```

Równoważnym zapisem powyższych hipotez statystycznych (8.13) oraz (8.14) będzie zapis:

$$H_0: \sigma_1^2 - \sigma_2^2 = 0 \quad \text{vs.} \quad H_1: \sigma_1^2 - \sigma_2^2 \neq 0 \tag{8.16}$$

Statystyka testowa:

$$Z_{diff} = \frac{(s_1^2 - s_2^2) - 0}{SE_{diff}} \tag{8.17}$$

gdzie: $SE_{diff} = \sqrt{SE_1^2 + \rho^2 \cdot SE_2^2}$ to błąd standardowy różnicy dwóch wariancji oraz $SE = \sqrt{s^2/n}$ to błąd standardowy wariancji s^2 dla przekształconej zmiennej $(x_i - \bar{x})^2$, ρ^2 to opcjonalny parametr do osłabienia/wzmocnienia udziału drugiej wariancji.

```
import scipy.stats as stats
import numpy as np

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

z1 = (x-np.mean(x))**2
z2 = (y-np.mean(y))**2
difV = np.var(x,ddof=1)-np.var(y,ddof=1)
SE = np.sqrt(np.var(z1,ddof=1)/len(x)+1*np.var(z2,ddof=1)/len(y))
conf = [stats.norm.ppf(i,difV,SE) for i in [0.025,0.975]]
h0 = 0
p = stats.norm.cdf(h0,difV,SE)

print("różnica wariancji:",difV,", błąd:",SE)
print("95% przedział ufności:",conf)
print("\nH0: dVar = %.0f vs. H1: dVar != %.0f" % (h0,h0))
print("p-wartość:",2*min(p,1-p))
```

```
## różnica wariancji: -3.8307446353496024 , błąd: 1.267036271883883
## 95% przedział ufności: [-6.314090095347914, -1.347399175351292]
##
## H0: dVar = 0 vs. H1: dVar != 0
## p-wartość: 0.0024995998590693347
```

Test Bartletta / Test Levene. Badanie równości wariancji można wykonać również za pomocą testu Fligner-Killen lub testu Levene które w przeciwieństwie do testu Bartletta są mało wrażliwe na odchylenia od rozkładu normalnego w próbkach. Przeważnie są one stosowane do badania równości kilku wariancji ale nic nie stoi na przeszkodzie aby wykorzystać je do porównania dwóch wariancji na podstawie hipotezy (8.13). Dodajmy, że test Levene i Fligner-Killeen mogą występować w trzech wariantach tzn. za parametr lokalizacji można przyjąć średnią, średnią uciętą lub medianę. Taki wybór oferują funkcje `scipy.stats.levene` oraz `scipy.stats.fligner`. Natomiast funkcja `pingouin.homoscedasticity` jako parametr lokalizacji stosuje medianę. Jeśli zmienne mają rozkład normalny to podawany jest wynik testu Bartletta.

```
import scipy.stats as stats
import numpy as np
import pingouin as pg

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

print(stats.levene(x,y))
```

```
print(stats.fligner(x,y))
print(stats.bartlett(x,y))
print(pg.homoscedasticity(x,y))
```

```
## LeveneResult(statistic=8.994663638939507, pvalue=0.0047575161444811925)
## FlignerResult(statistic=6.954492770953616, pvalue=0.008360900023288296)
## BartlettResult(statistic=8.019535289470467, pvalue=0.004627544281207727)
## (False, 0.005)
```

8.4 Porównanie rozkładów

Test normalności Andersona-Darlinga. Założenie normalności zmiennych to jedno z głównych założeń w klasycznej statystyce. W związku z tym zostało opracowanych wiele metod porównywania dystrybuanty empirycznej z rozkładem normalnym. Jednym z bardziej popularnych rozwiązań jest test Shapiro-Wilka który wymaga aby liczebność próby nie przekraczała 5000 elementów. Inne metody jak np. test Jarque-Bera, D'Agostino-Pearsona czy Andersona-Darlinga nie mają tego ograniczenia. Dodajmy jeszcze, że wysoka moc testu może być dobrym uzasadnieniem wyboru konkretnej metody (Biecek, 2013). Przykładowo test normalności Andersona-Darlinga może być ciekawą alternatywą dla testu Shapiro-Wilka w przypadku wielomodalności lub występowania grubych ogonów (Biecek, 2017, str. 244-246).

Statystyka testu Andersona-Darlinga ma postać:

$$AD = -n - \frac{1}{n}\sum_{i=1}^{n}(2i-1)\big(\ln(z_i) + \ln(1 - z_{n+1-i})\big) \tag{8.18}$$

gdzie: z_i to wartości wyznaczone na podstawie dystrybuanty rozkładu normalnego $\Phi(x_i, \bar{x}, s)$ dla posortowanych rosnąco elementów próby x_i.

W przypadku badania normalności o nieznanych parametrach μ oraz σ jest stosowana poprawka:

$$A1 = AD\left(1 + \frac{0,75}{n} + \frac{2,25}{n^2}\right) \tag{8.19}$$

Weryfikację hipotezy zerowej można wykonać w oparciu o otrzymaną p-wartość która jest uzależniona od wartości statystyki testu (8.19).

- jeżeli $A1 < 0,2$ to:

$$p - value = 1 - \exp(-13,436 + 101,14\,A1 - 223,73\,A1^2) \tag{8.20}$$

- jeżeli $0,2 \leq A1 < 0,34$ to:

$$p - value = 1 - \exp(-8,318 + 42,796\,A1 - 59,938\,A1^2) \tag{8.21}$$

- jeżeli $0,34 \leq A1 < 0,6$ to:

$$p - value = \exp(0,9177 - 4,279\,A1 - 1,38\,A1^2) \tag{8.22}$$

- jeżeli $A1 \geq 0,6$ to:

$$p - value = \exp(1,2937 - 5,709\,A1 + 0,0186\,A1^2) \tag{8.23}$$

```
import scipy.stats as stats
from statsmodels.stats.diagnostic import normal_ad

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)
ad = normal_ad(x)
print('AD = %.4f, p-value = %.4f' % (ad[0],ad[1]))
```

```
## AD = 0.2568, p-value = 0.6848
```

W stosunkowo prosty sposób można wygenerować wartości krytyczne na podstawie wzoru:

$$A_{crit} = a\left(1 - \frac{b}{n} - \frac{d}{n^2}\right) \tag{8.24}$$

gdzie n to liczebności próby oraz a, b i d to parametry które zależą od poziomu istotności α:

α	0.005	0.01	0.025	0.05	0.10	0.20
a	1.1578	1.0348	0.8728	0.7514	0.6305	0.5091
b	1.063	1.013	0.881	0.795	0.750	0.756
d	1.34	0.93	0.94	0.89	0.80	0.39

(8.25)

Poniżej przykład wygenerowania różnych wartości krytycznych testu normalności Andersona-Darlinga.

```
import numpy as np
import scipy.stats as stats
import pandas as pd

def q(alpha=0.05,n=10):
    if alpha == 0.005:\
    return 1.1578*(1-1.063/n-1.34/n**2)
    elif alpha == 0.01:\
    return 1.0348*(1-1.013/n-0.93/n**2)
    elif alpha == 0.025:\
    return 0.8728*(1-0.881/n-0.94/n**2)
    elif alpha==0.05:\
    return 0.7514*(1-0.795/n-0.89/n**2)
    elif alpha == 0.1:\
    return 0.6305*(1-0.750/n-0.80/n**2)
    elif alpha == 0.2:\
    return 0.5091*(1-0.756/n-0.39/n**2)

n = [20,50,100,150,300,900,1500]
q0_01  = [q(alpha=0.01, n=i) for i in n]
q0_025 = [q(alpha=0.025,n=i) for i in n]
q0_05  = [q(alpha=0.05, n=i) for i in n]
q0_1   = [q(alpha=0.1,  n=i) for i in n]
print(pd.DataFrame({'1%':q0_01,'2.5%':q0_025,'5%':q0_05,'10%':q0_1},index=n))
```

```
##             1%      2.5%        5%       10%
## 20    0.979981  0.832302  0.719860  0.605595
## 50    1.013450  0.857093  0.739185  0.620841
## 100   1.024221  0.865029  0.745359  0.625721
## 150   1.027769  0.867637  0.747388  0.627325
## 300   1.031295  0.870228  0.749401  0.628918
## 900   1.033634  0.871945  0.750735  0.629974
## 1500  1.034101  0.872287  0.751001  0.630185
```

Poniżej wygenerujemy w sposób symulacyjny wartości krytyczne dla $n = 20$:

```
from statsmodels.stats.diagnostic import normal_ad
import numpy as np
import scipy.stats as stats
import pandas as pd

res = [normal_ad(stats.norm.rvs(0, 1.5, size=20))[0] for i in range(10000)]
q = np.percentile(res,[99,97.5,95,90])
print(pd.DataFrame({'1%':q[0],'2.5%':q[1],'5%':q[2],'10%':q[3]},index=['20']))
```

```
##           1%      2.5%       5%       10%
## 20  1.001288  0.845457  0.73237  0.604666
```

Test zgodności Andersona-Darlinga. Oprócz rozkładu normalnego dystrybuantę empiryczną można porównywać również z innymi dystrybuantami teoretycznymi. Do badania zgodności z rozkładami ciągłymi można wykorzystać test Kołmogorowa który został zaimplementowany do funkcji `scipy.stats.kstest`. Alternatywą do tego rozwiązania jest test Andersona-Darlinga dostępny dzięki funkcji `scipy.stats.anderson`. W tej implementacji zamiast p-wartości są podawane wartości krytyczne A_{crit} które określają granicę prawostronnego obszaru odrzucenia. Inaczej mówiąc jeśli $AD > A_{crit}$ to hipotezę zerową o zadanej dystrybuancie należy odrzucić. Dodajmy jeszcze, że wartości krytyczne zależą od liczebności próby n, poziomu istotności α oraz roważanego rozkładu. W zaimplementowanej funkcji można założyć rozkład np. normalny, wykładniczy, logistyczny, gumbela.

W przypadku rozkładu normalnego wartości krytyczne są obliczane za pomocą wzoru:

$$A_{crit} = k(\alpha) / \left(1 + \frac{4}{n} - \frac{25}{n^2}\right) \tag{8.26}$$

gdzie wartość współczynnika $k(\alpha)$ jest uzależniona od tego czy znane są parametry rozkładu. Poniżej wykaz współczynników dla różnych wariantów.

wariant	α	0.15	0.10	0.05	0.025	0.01
$N(\mu,\sigma)$	$k(\alpha)$	1.610	1.993	2.492	3.070	3.857
$N(?,?)$	$k(\alpha)$	0.576	0.656	0.787	0.918	1.092

(8.27)

Wartości krytyczne dla dwóch pozostałych rozkładów np. wykładniczegi oraz logistycznego obliczamy za pomocą wzoru:

$$A_{crit} = k(\alpha) / \left(1 + \frac{v}{n}\right) \tag{8.28}$$

gdzie odpowiednie współczynniki $k(\alpha)$ dla danej liczebności próby n są przedstawione poniżej:

wariant	v	α	0.10	0.05	0.025	0.01
$Expon$	0.6	$k(\alpha)$	1.065	1.325	1.587	1.934
$Logist$	0.25	$k(\alpha)$	0.56	0.657	0.765	0.901

(8.29)

Poniżej przykład jak wygenerować tablicę z wartościami krytycznymi dla rozkładu normalnego, wykładniczego i logistycznego przy założonym $\alpha = 0,05$ oraz różnych liczebności próby n. Dodajmy jeszcze, że funkcja podaje wartości krytyczne dla przypadku gdy parametry rozkładu nie są znane i trzeba je oszacować.

```
import scipy.stats as stats
import numpy as np
import pandas as pd

n = [10,20,30,50,70,100,150,300]
nor = [stats.anderson(stats.norm.rvs(size=i), dist='norm')[1][2] for i in n]
```

```
exp = [stats.anderson(stats.norm.rvs(size=i), dist='expon')[1][2] for i in n]
logis = [stats.anderson(stats.norm.rvs(size=i), dist='logistic')[1][2] for i in n]
gumbel = [stats.anderson(stats.norm.rvs(size=i), dist='gumbel')[1][2] for i in n]
print(pd.DataFrame({'nor_0.05':nor,'exp_0.05':exp,
                    'logis_0.05':logis,'gumbel_0.05':gumbel},index=n))
```

```
##      nor_0.05  exp_0.05  logis_0.05  gumbel_0.05
## 10      0.684     1.265       0.644        0.712
## 20      0.692     1.302       0.652        0.725
## 30      0.712     1.315       0.655        0.730
## 50      0.736     1.325       0.657        0.736
## 70      0.748     1.330       0.658        0.739
## 100     0.759     1.333       0.658        0.742
## 150     0.767     1.336       0.659        0.745
## 300     0.777     1.338       0.659        0.748
```

Poniżej przykład wywołania funkcji `scipy.stats.anderson` w celu zbadania zgodności rozkładu empirycznego z rozkładem normalnym.

```
import scipy.stats as stats
import pandas as pd

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)

ad = stats.anderson(x, dist='norm')
print(pd.DataFrame({'20':ad[1]},index=ad[2]/100).T)
print('\nad: %.4f'% (ad[0]))
```

```
##     0.150  0.100  0.050  0.025  0.010
## 20  0.506  0.577  0.692  0.807   0.96
##
## ad: 0.2568
```

Otrzymana wartość statystyki testu Andersona-Darlinga nie przekracza wartości krytycznej nawet dla $\alpha = 0.15$ więc brak jest podstaw do odrzucenia hipotezy zerowej. Warto dodać, że w tym teście można zweryfikować hipotezę zerową w oparciu o p-wartość wyznaczoną w sposób analityczny (Jäntschi and Bolboacă, 2018) lub symulacyjny. Jedna z propozycji (Marsaglia and Marsaglia, 2004) została zaimplementowana do pakietu `ADGofTest` dla środowiska R.

```
import scipy.stats as stats
import numpy as np

x = stats.norm.rvs(0, 1.5, size=20, random_state=2305)

A = stats.anderson(x,dist='norm')
ad = [stats.anderson(np.random.choice(x,size=len(x),replace=True), dist='norm')[0] \
      for i in range(1000)]
print('AD: %.4f, p-value: %.4f' % (A[0], np.mean(np.greater(ad,[A[0]]))))
```

```
## AD: 0.2568, p-value: 0.9810
```

Test zgodności Cressie-Read. Badanie zgodności rozkładu empirycznego z założonym rozkładem teoretycznym (ciągłym lub dyskretnym) o zdefiniowanych parametrach można wykonać za pomocą testu chi-kwadrat lub jego uogólnionej wersji tzn. testu Cressie-Reada. Do tego celu można wykorzystać odpowiednio

funkcje `scipy.stats.chisquare` oraz `scipy.stats.power_divergence` w których argumentami są wartości empiryczne `f_obs=fi` oraz teoretyczne `f_exp=ei`. Jeśli w teście Cressie-Reada ustalimy, że parametr `lambda` będzie równy `"1"` lub przypiszemy mu nazwę `"pearson"` to zostanie wykonany test chi-kwadrat.

```
import scipy.stats as stats
import numpy as np
import pandas as pd

x = stats.poisson.rvs(1.5, size=80, random_state=2305)

def goodfitPois(x):
    t = pd.Series(x).value_counts(sort=False)
    fi = t.values
    xi = list(t.index)
    pi = [stats.poisson.pmf(i,np.mean(x)) for i in xi]
    pi.append(1-sum(pi))
    ei = np.asarray(pi) * len(x)
    e = ei[-1]+ei[-2]
    ei = ei[:-2]
    ei = list(ei)
    ei.append(e)
    return stats.power_divergence(fi, ei, ddof=1, lambda_=2/3)

print(goodfitPois(x))
```

```
## Power_divergenceResult(statistic=0.47455822372954887, pvalue=0.9759300084452401)
```

Test Andersona-Darlinga dla k prób. Test Kołmogorowa-Smirnowa (funkcja `scipy.stats.ks_2samp`) to częsty wybór do weryfikacji hipotezy zerowej w której zakładamy, że dwie dystrybuanty są takie same. Inaczej mówiąc badamy czy dwie zmienne losowe pochodzą z tego samego ciągłego rozkładu o takich samych parametrach. Gdy porównujemy dwie próbki warto zwrócić uwagę także na test Eppsa-Singletona który jest zaimplementowany do funkcji `scipy.stats.epps_singleton_2samp`. Ta metoda charakteryzuje się między innymi tym, że ma większą moc niż test Kołmogorowa-Smirnowa oraz może porównywać także rozkłady dyskretne. Alternatytwnym rozwiązaniem jest test Andersona-Darlinga (funkcja `scipy.stats.anderson_ksamp`) który można stosować dla dwóch lub większej liczby próbek z rozkładu ciągłego. Dodatkowo po odrzuceniu hipotezy zerowej można sprawdzić które zmienne różnią się między sobą za pomocą testów post hoc – porównania wielokrotne.

```
import scipy.stats as stats
import numpy as np
import pandas as pd
from scikit_posthocs import posthoc_anderson

x1 = stats.expon.rvs(0, 1, size=20, random_state=2305)
x2 = stats.expon.rvs(0.5, 1, size=20, random_state=4101)
x3 = stats.expon.rvs(1, 1, size=20, random_state=4026)
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)
d = pd.DataFrame({"y":np.concatenate((x1,x2,x3)),"g":g})
adk = stats.anderson_ksamp([x1,x2,x3])

print('ad = %.4f, p-wartość = %.4f' % (adk[0],adk[2]),'\n')
print(posthoc_anderson(d,val_col='y',group_col='g'))
```

```
## ad = 3.8139, p-wartość = 0.0066
```

```
##
##           1.0       2.0       3.0
## 1.0 -1.000000  0.103108  0.001627
## 2.0  0.103108 -1.000000  0.121164
## 3.0  0.001627  0.121164 -1.000000
```

8.5 Moc testu

Test t-Studenta. Standardowy rozkład t-Studenta ma swój ogólniejszy odpowiednik tzn. niecentralny rozkład t-Studenta z dodatkowym parametrem ncp – non-centrality parameter. Dla $ncp = 0$ niecentralny rozkład t-Studenta jest tożsamy z centralnym rozkładem t-Studenta – takie szczególne przypadki mają także rozkłady chi-kwadrat oraz F-Snedecora. Rozkłady niecentralne są często wykorzystywane do obliczania mocy testów np. funkcja `pingouin.power_ttest` oblicza moc testu t-Studenta dla dwóch niezależnych prób (test dwustronny) według wzoru:

$$\text{moc} = P(T \leq t_{crit}, df, ncp) \tag{8.30}$$

gdzie: t_{crit} to kwantyl rzędu $1 - \alpha/2$ z rozkładu t-Studenta o stopniach swobody $df = 2n - 2$ oraz $ncp = |d| \cdot \sqrt{\frac{n}{2}}$ to non-centrality parameter.

Wielkość efektu d można obliczyć na podstawie wzoru:

$$d = t_{val} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}. \tag{8.31}$$

gdzie: t_{val} to statystyka testu t-Studenta dla dwóch niezależnych prób, n_1 oraz n_2 to liczenbość odpowiednio dla pierwszej i drugiej próby.

```
import scipy.stats as stats
import numpy as np
import pingouin as pg

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)

tval, n1, n2 = stats.ttest_ind(x,y)[0], len(x), len(y)
d = pg.compute_effsize_from_t(tval, nx=n1, ny=n2, eftype='cohen')
power = pg.power_ttest(d=d, n=len(x), contrast='two-samples')
print("Efekt: %.4f, Moc: %.4f" % (d, power))
```

```
## Efekt: -1.0903, Moc: 0.9192
```

ANOVA. Moc testu dla klasycznej wersji jednoczynnikowej ANOVY można obliczyć za pomocą nie centralnego rozkładu F-Snedecora czyli z dodatkowym parametrem ncp:

$$\text{moc} = P(F \geq F_{crit}, df_1, df_2, ncp) \tag{8.32}$$

gdzie: F_{crit} to kwantyl rzędu $1 - \alpha$ z rozkładu F-Snedecora o stopniach swobody $df1 = k - 1, df2 = n - 3$ oraz $npc = f^2 N$ to non-centrality parameter.

Wielkość efektu f można obliczyć według formuły:

$$f = \sqrt{\frac{\sum_{i=1}^{k} p_i(\mu_i - \mu)^2}{\sigma^2}} = \sqrt{\frac{SS_{betveen}}{MS_{residuals} \cdot N}} \tag{8.33}$$

gdzie: $p_i = n_i/N$, n_i to liczba obserwacji w i-tej grupie, N to suma wszystkich obserwacji, μ_i to średnia w i-tej grupie, μ to ogólna średnia, σ^2 to wariancja błędu w obrębie grupy ($MS_{residuals}$ - mean squares for resuduals).

Metoda zaimplementowana do funkcji `pingouin.power_anova` bazuje na obliczeniu wielkości efektu f według wzoru:

$$f = \sqrt{\frac{\eta^2}{1-\eta^2}} \tag{8.34}$$

gdzie: η^2 to wielkość efektu dla jednoczynnikowej analizy wariancji która jest tożsama z współczynnikiem determinacji R^2 dla regresji liniowej.

$$\eta^2 = \frac{df_1 \cdot F}{df_1 \cdot F + df_2} \quad \text{lub} \quad \eta^2 = \frac{SS_{between}}{SS_{residuals} + SS_{between}} \tag{8.35}$$

gdzie: $SS_{between}$ to suma kwadratów dla czynnika, SS_{total} to suma kwadratów dla czynnika oraz reszt.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.expon.rvs(0, 1, size=20, random_state=2305),
                    stats.expon.rvs(0.5, 1, size=20, random_state=4101),
                    stats.expon.rvs(1, 1, size=20, random_state=4026)))
g = np.repeat(np.linspace(1,3,3), [20,20,20], axis=0)
d = pd.DataFrame({"y":y,"g":g})
eta2 = pg.anova(dv='y', between='g', data=d)['np2']
f = np.sqrt(eta2/(1-eta2))
power = pg.power_anova(eta=eta2, k=3, n=20)[0]
print("Efekt: %.4f, Moc: %.4f" % (f[0], power))
```

```
## Efekt: 0.1534, Moc: 0.1636
```

Jeśli metoda analityczna do obliczenia mocy wybranego testu nie jest dostępne to wygodnym rozwiązaniem może być symulacja komputerowa.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 2, size=20, random_state=4101)
z = np.concatenate((x,y))
g = np.repeat(np.linspace(1,2,2), [20,20], axis=0)

def pvalA(x,y):
    nx = len(x)
    ny = len(y)
    z = np.concatenate((np.random.choice(x,nx),np.random.choice(y,ny)))
    g = np.repeat(np.linspace(1,2,2), [nx,ny], axis=0)
    return pg.welch_anova(dv='z', between='g',data=pd.DataFrame({"z":z,"g":g}))['p-unc'][0]

m = [pvalA(x,y) for i in range(1000)]
print("Moc: ", np.less(m,[0.05]).mean(), "dla 1000 symulacji testu Welch-Anova")
```

```
## Moc:  0.909 dla 1000 symulacji testu Welch-Anova
```

Rozdział 9

Porównanie zmiennych zależnych

9.1 Porównanie średnich

Test t-Studenta. Metoda do porównania dwóch zmiennych zależnych sprowadza się do przeprowadzenia testu t-Studenta dla jednej zmiennej tzn. różnic między obserwacjami.

```
import numpy as np
import scipy.stats as stats
import pingouin as pg

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 3, size=20, random_state=4101)

print(pg.ttest(x,y,paired=True))
```

```
##             T  dof       tail     p-val           CI95%  cohen-d   BF10  power
## T-test -2.688   19  two-sided  0.014562  [-3.73, -0.46]    1.006  3.752  0.989
```

Dokładny test znaków. Ta procedura sprowadza się do określenia liczby znaków dla różnic między obserwacjami. Inaczej mówiąc po pominięciu różnic równych zero zliczamy dodatnie (statystyka dokładnego testu T_+) i ujemne różnice. Na podstawie testu dwumianowego `scipy.stats.binom_test` o argunentach: $x = T_+$,$n = T_+ + T_-$ oraz $p = 0,5$ możemy określić dokładną p-wartość. Weryfikowana hipoteza zerowa ma postać:

$$H_0 : \ p = 0,5 \quad \text{vs} \quad H_1 : \ p \neq 0,5 \tag{9.1}$$

gdzie: p to prawdopodobieństwo tego, że $P(x > y) = 0,5$.

```
import numpy as np
import scipy.stats as stats

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
z = np.greater(x-y,[0]).astype(int)

print("S:",sum(z),", p-value:",stats.binom_test(sum(z), len(z)))
```

```
## S: 6 , p-value: 0.11531829833984371
```

RM-Anova / Greenhouse-Geisser. Test Mauchly który został zaimplementowany do funkcji `pingouin.sphericity` określa czy warunek sferyczności jest spełniony. W hipotezie zerowej zakładamy, że wariancje dla różnic pomiędzy parami powtarzanych pomiarów są takie same.

$$H_0: \ \sigma^2_{d1} = \sigma^2_{d2} = \ldots = \sigma^2_{di} \quad \text{vs} \quad H_1: \text{nie wszystkie wariancje są równe} \tag{9.2}$$

gdzie: σ^2_{di} to wariancja dla i-tej różnicy zmiennych.

Jeśli analizowane zmienne nie spełniają tego założenia, to należy dostosować wyniki RM-ANOVA za pomocą jednej z korekt: Greenhouse-Geisser [1958] lub Huynh and Feldt [1976]. Funkcja `pingouin.rm_anova` ma opcję `correction` dzięki której można wykonać test z korektą lub bez. Generalnie współczynnik korekcyjny HF jest używany częściej, ponieważ współczynnik GG jest zbyt konserwatywny tzn. nie zawsze udaje się wykryć prawdziwą różnicę między grupami. Dzięki funkcji `pingouin.epsilon` można otrzymać współczynniki $\epsilon-$epsilon. Określają one odstępstwo od symetrii złożonej dla każdej z dwóch procedur: GG i HF. Im mniejsza wartość ϵ tym większe jest odstępstwo od warunku sferyczności.

$$\epsilon_{HF} = \frac{n(k-1)\epsilon_{GG} - 2}{(k-1)(n-1-(k-1)\epsilon_{GG})} \tag{9.3}$$

Wartości p-value są obliczane na podstawie rozkładu F po skorygowaniu stopni swobody:

$$df_1 = (k-1) \cdot \epsilon_{HF} \quad \text{oraz} \quad df_2 = (k-1) \cdot (n-1) \cdot \epsilon_{HF} \tag{9.4}$$

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.norm.rvs(0, 1, size=20, random_state=2305),
                    stats.norm.rvs(1.5, 3, size=20, random_state=4101),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4026)))
Dpaired = pd.DataFrame({'y': y,
                        'g': np.repeat(np.linspace(1,3,3), [20,20,20], axis=0),
                        'b': np.tile(np.linspace(1,20,20), 3)})

print(pg.rm_anova(dv='y', within='g', subject='b', data=Dpaired,\
                  correction=True).drop(['sphericity','np2'], axis=1),'\n')

dat = Dpaired.pivot(index='b', columns='g', values='y')
hf = pg.epsilon(dat, correction='hf')
df1 = dat.shape[1]-1
df2 = dat.shape[0]-1
p = 1-stats.f.cdf(5.094, hf*df1, hf*df1*df2)

print(pd.DataFrame({'HF':[hf],'df1':[df1],'df2':[df2],'p-HF-corr':[p]}))
```

```
##   Source  ddof1  ddof2      F     p-unc  p-GG-corr   eps  W-spher   p-spher
## 0      g      2     38  5.094  0.010968   0.018047  0.79    0.734  0.061652
##
##          HF  df1  df2  p-HF-corr
## 0  0.849268    2   19   0.015664
```

W większości przypadków lepiej zastosować wielowymiarową analize wariancji tj. MANOVA (O'brien and Kaiser, 1985) lub liniowe modele mieszane (Zieliński, 2010) ponieważ są one odporne na złamanie założenia kulistości. Ta procedura jest dostępna dzięki funkcji `pingouin.mixed_anova`.

9.2 Porównanie rang

Test Wilcoxona / metoda Pratta. Procedura rangowanych znaków dla dwóch zmiennych zależnych polega na obliczeniu d_i czyli różnic między obserwacjami a następnie porangowaniu ich wartości bezwzględnych tzn. $\text{rank}|d_i|$. W metodzie Pratta sumujemy tylko te rangi dla których różnica dwóch zmiennych d_i była mniejsza od zera tzn. $V = \sum_{d_i<0} \text{rank}|d_i|$. Według metody Wilcoxona zanim porangujemy wartości bezwzględnych różnic musimy usunąć różnice równe zero. Następnie obliczamy sumę rang według wzoru $V = \sum_{d_i>0} \text{rank}|d_i|$. Do funkcji `scipy.stats.wilcoxon` zostały zaimplementowane obie metody.

Statystyka testowa dla metody Wilcoxona z poprawką na ciągłość:

$$Z = \frac{V - \frac{1}{4}\left[n(n+1)\right] - 0,5}{\sqrt{\frac{1}{24}\left[n(n+1)(2n+1)\right] - \frac{1}{48}\sum_{i=1}^{c}(t_i^3 - t_i)}} \tag{9.5}$$

gdzie: n to liczba różnic czyli par zmiennych, V to suma rang dla różnic dodatnich, $0,5$ to poprawka na ciągłość, c to liczba grup pomiarów wiązanych, t_i to liczba pomiarów wiązanych w i-tej grupie pomiarów wiązanych.

```
import scipy.stats as stats

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 3, size=20, random_state=4101)

print(stats.wilcoxon(x, y, zero_method='wilcox', correction=True))
```

```
## WilcoxonResult(statistic=43.0, pvalue=0.021678215270968325)
```

Statystyka testowa dla metody Pratta:

$$Z = \frac{V - \frac{1}{4}\left[n(n+1) - t_0(t_0+1)\right]}{\sqrt{\frac{1}{24}\left[n(n+1)(2n+1) - t_0(t_0+1)(2t_0+1)\right] - \frac{1}{48}\sum_{i=1}^{c}(t_i^3 - t_i)}} \tag{9.6}$$

gdzie: n to liczba różnic czyli par zmiennych, V to suma rang dla różnic ujemnych, t_0 to liczba zerowych różnic, c to liczba grup pomiarów wiązanych, t_i to liczba pomiarów wiązanych w i-tej grupie pomiarów wiązanych.

```
import scipy.stats as stats

x = stats.norm.rvs(0, 1, size=20, random_state=2305)
y = stats.norm.rvs(1.5, 3, size=20, random_state=4101)

print(stats.wilcoxon(x, y, zero_method='pratt'))
```

```
## WilcoxonResult(statistic=43.0, pvalue=0.020633435105949553)
```

Test Friedmana. Rozszerzeniem testu znaków na kilka zmiennych sparowanych jest test Friedmana który został zaimplementowany do funkcji `scipy.stats.friedmanchisquare` oraz `pingouin.friedman`.

Statystyka testowa:

$$\chi^2 = \left(1 - \frac{\sum_{i=1}^{c}(t_i^3 - t_i)}{nk(k^2-1)}\right)^{-1} \left[\frac{12}{nk(k+1)}\sum_{j=1}^{k} R_j^2 - 3n(k+1)\right] \tag{9.7}$$

gdzie: k to liczebność grup, n_j to liczebność obserwacji w i-tej grupie, c to liczba grup pomiarów wiązanych, t_i to liczba pomiarów wiązanych w i-tej grupie pomiarów wiązanych.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.norm.rvs(0, 1, size=20, random_state=2305),
                    stats.norm.rvs(1.5, 3, size=20, random_state=4101),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4026)))
Dpaired = pd.DataFrame({'y': y,
                        'g': np.repeat(np.linspace(1,3,3), [20,20,20], axis=0),
                        'b': np.tile(np.linspace(1,20,20), 3)})

print(pg.friedman(dv='y', within='g', subject='b', data=Dpaired))
```

```
##          Source  ddof1    Q     p-unc
## Friedman      g      2  3.9  0.142274
```

Test Imana-Davenporta. Modyfikacją testu Friedmana jest metoda Imana-Davenporta która sprowadza się do przekształcenia statystyki χ^2 według wzoru:

$$F = \frac{(n_j - 1)\chi^2}{n_j(k-1) - \chi^2} \tag{9.8}$$

gdzie: k to liczebność grup, n_j to liczebność obserwacji w j-tej grupie, $df_1 = k - 1$ oraz $df_2 = (k-1)(n_j - 1)$.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import pingouin as pg

y = np.concatenate((stats.norm.rvs(0, 1, size=20, random_state=2305),
                    stats.norm.rvs(1.5, 3, size=20, random_state=4101),
                    stats.norm.rvs(1.5, 2, size=20, random_state=4026)))
Dpaired = pd.DataFrame({'y': y,
                        'g': np.repeat(np.linspace(1,3,3), [20,20,20], axis=0),
                        'b': np.tile(np.linspace(1,20,20), 3)})

F = pg.friedman(dv='y', within='g', subject='b', data=Dpaired)['Q'][0]
n = 20
k = 3
df1 = k-1
df2 = (k-1)*(n-1)
F = ((n-1)*F)/(n*(k-1)-F)
p = 1-stats.f.cdf(F,df1,df2)
print(pd.DataFrame({'F':[F],'df1':[df1],'df2':[df2],'p':[p]}))
```

```
##          F  df1  df2         p
## 0 2.052632    2   38  0.142396
```

Test wyrównanych rang Friedmana. W tej procedurze (ang. Friedman Aligned Ranks) obliczenia wykonujemy na przekształconych danych tj. $x_{ij} - \bar{x}_i$. Otrzymane w ten sposób wartości trzeba porangować bez podziału na grupy i obliczyć sumy kwadratów rang dla k grup (kolumn) $\sum_{j=1}^{k} \hat{R}_j^2$ oraz dla n obserwacji (wierszy) $\sum_{i=1}^{n} \hat{R}_i^2$ aby wyznaczyć statystykę testu.

Statystyka testu:

$$T = \frac{(k-1)[\sum_{j=1}^{k} \hat{R}_j^2 - (kn^2/4)(kn+1)^2]}{([kn(kn+1)(2kn+1)]/6) - (1/k)\sum_{i=1}^{n} \hat{R}_i^2} \tag{9.9}$$

```
import numpy as np
import scipy.stats as stats
import pandas as pd

df = pd.DataFrame()
df['a1'] = stats.norm.rvs(0, 1, size=20, random_state=2305)
df['a2'] = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
df['a3'] = stats.norm.rvs(1.5, 2, size=20, random_state=4026)

mu = df.mean(axis=1)
w = [ df[i]-mu for i in df.columns ]
r = stats.rankdata(w)
rdf = pd.DataFrame({'a1':r[:20],'a2':r[20:40],'a3':r[40:60]})
Sk = sum(rdf.sum(axis=0)**2)
Sn = sum(rdf.sum(axis=1)**2)
n = df.shape[0]
k = df.shape[1]
T = ((k-1)*(Sk-((k*n**2)/4)*(k*n+1)**2))/(((k*n*(k*n+1)*(2*k*n+1))/6)-(1/k)*Sn)
p = 1-stats.chi2.cdf(T,df=k-1)
print(pd.DataFrame({'T':[T],'df':[k-1],'p':[p]}))
```

```
##           T  df         p
## 0  7.448925   2  0.024126
```

Test Quade. Dobrą alternatywą dla testu Friedmana może być również metoda Quade dostępna w funkcji `stac.nonparametric_tests.quade_test`. Jest to rozszerzenie testu rangowanych znaków Wilcoxona na więcej niż dwie sparaowane zmienne.

Statystyka testu:

$$F_Q = \frac{(n-1)SS_{tre}}{SS_{tot} - SS_{tre}} \tag{9.10}$$

gdzie: n to liczba bloków, k to liczba grup, R_{ij} to rangi obliczone oddzielnie dla każdego bloku, Q_i to rangi obliczone dla różnic $x_{max} - x_{min}$ obliczonych dla każdego bloku, S_{ij} to macierz o postaci $S_{ij} = Q_i \left[R_{ij} - (k+1)/2\right]$, $SS_{tot} = \sum_{i=1}^{n}\sum_{j=1}^{k} S_{ij}^2$ to suma wszystkich elementów macierzy S_{ij} które zostały podniesione do kwadratu, $SS_{tre} = \frac{1}{n}\sum_{j=1}^{k} S_i^2$ to suma elementów macierzy S_{ij} dla każdej grupy i podniesionych do kwadratu a następnie te wartości są sumowane i podzielone przez liczbę bloków. Stopnie swobody są obliczane na podstawie wzorów $df_1 = k-1$ oraz $df_2 = (n-1)(k-1)$.

```
import numpy as np
import scipy.stats as stats
import pandas as pd

df = pd.DataFrame()
df['a1'] = stats.norm.rvs(0, 1, size=20, random_state=2305)
df['a2'] = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
df['a3'] = stats.norm.rvs(1.5, 2, size=20, random_state=4026)

def quade_test(x):
    df = x
```

```
    n = df.shape[0]
    k = df.shape[1]
    rdat = df.rank(axis=1)
    minmax = df.apply(lambda x: max(x)-min(x),axis=1)
    Q = pd.DataFrame(stats.rankdata(minmax), columns=['a'])
    m = rdat-(k+1)/2
    S = m.values * Q.values
    SStot = sum(sum(S**2))
    SStre = sum(sum(S)**2)/n
    F = (n-1)*(SStre)/(SStot-SStre)
    df1 = k-1
    df2 = (n-1)*(k-1)
    p = 1-stats.f.cdf(F,df1,df2)
    DF = pd.DataFrame({'F':[F],'df1':[df1],'df2':[df2],'SStot':[SStot],'SStre':[SStre],'p':[p]})
    return DF

print(quade_test(df))
```

```
##           F  df1  df2   SStot   SStre         p
## 0  4.066348    2   38  5740.0  1011.9  0.025103
```

Dalsza analiza. W pakiecie `scikit-posthocs` jest dostępnych wiele testów post hoc dla nieparametrycznej analizy wariancji z powtarzanymi pomiarami. Popularnym wyborem do porównań wielokrotnych po odrzuceniu hipotezy zerowej w teście Friedmana jest przeprowadzenie serii testów znaków lub test Nemenyi. W przypadku wyrównanych rang Friedmana błąd standardowy badanych różnic rang $\hat{R}_i - \hat{R}_j$ jest dany wzorem:

$$SE = \sqrt{\frac{k(kn+1)}{6}} \tag{9.11}$$

Poniżej przykład skryptu dla tego rozwiązania w języku Python:

```
import numpy as np
import scipy.stats as stats
import pandas as pd
from pingouin import multicomp
import itertools

df = pd.DataFrame()
df['a1'] = stats.norm.rvs(0, 1, size=20, random_state=2305)
df['a2'] = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
df['a3'] = stats.norm.rvs(1.5, 2, size=20, random_state=4026)

def post_hoc_far(x):
    df = x
    n = df.shape[0]
    k = df.shape[1]
    mu = df.mean(axis=1)
    w = [ df[i]-mu for i in df.columns ]
    r = stats.rankdata(w)
    rdat = pd.DataFrame(r.reshape((k,n)).T,columns=df.columns)
    SS = rdat.mean(axis=0)
    SE = np.sqrt((k * (n * k + 1))/6)
    stat = SS/SE
    res = list(itertools.combinations(stat, 2))
```

```
    sol = [np.abs(np.diff(res[i])) for i in range(len(res))]
    es = list(itertools.combinations(list(df.columns), 2))
    f = pd.DataFrame({'stat':np.ravel(sol).tolist()},index=es)
    f['p-val'] = [ 2*(1-stats.norm.cdf(i)) for i in f['stat']]
    f['p-val_Holm'] = multicomp(f['p-val'].tolist(), method='holm')[1].tolist()
    f['sign'] = multicomp(f['p-val'].tolist(), method='holm')[0].tolist()
    return f

print(post_hoc_far(df))
```

```
##                stat     p-val  p-val_Holm   sign
## (a1, a2)  3.250233  0.001153    0.003459   True
## (a1, a3)  1.231286  0.218216    0.218216  False
## (a2, a3)  2.018947  0.043493    0.086985  False
```

Z kolei rozwiązaniem dedykowanym dla testu Quade jest seria testów rangowanych znaków Wilcoxona zaimplementowanych do funkcji `scikit_posthocs.posthoc_wilcoxon` oraz metoda dostępna dzięki funkcji `scikit_posthocs.posthoc_quade` która działa z wykorzystaniem rozkładu t-Studenta lub normalnego. W tej metodzie badamy różnice wyznaczone w oparciu o sumy obliczone dla każdej grupy z wykorzystaniem macierzy S_{ij} natomiast błąd standardowy można określić za pomocą wzoru:

$$SE = \sqrt{\frac{2n(SS_{tot} - SS_{tre})}{(n-1)(k-1)}} \tag{9.12}$$

```
import numpy as np
import scipy.stats as stats
import pandas as pd
from pingouin import multicomp
import itertools

df = pd.DataFrame()
df['a1'] = stats.norm.rvs(0, 1, size=20, random_state=2305)
df['a2'] = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
df['a3'] = stats.norm.rvs(1.5, 2, size=20, random_state=4026)

def post_hoc_quade_test_1(x):
    df = x
    n = df.shape[0]
    k = df.shape[1]
    rdat = df.rank(axis=1)
    minmax = df.apply(lambda x: max(x)-min(x),axis=1)
    Q = pd.DataFrame(stats.rankdata(minmax), columns=['a'])
    m = rdat-(k+1)/2
    S = m.values * Q.values
    SStot = sum(sum(S**2))
    SStre = sum(sum(S)**2)/n
    SS = S.sum(axis=0)
    df2 = (n-1)*(k-1)
    SE = np.sqrt((2*n*(SStot-SStre))/df2)
    stat = SS/SE
    res = list(itertools.combinations(stat, 2))
    sol = [np.abs(np.diff(res[i])) for i in range(len(res))]
    es = list(itertools.combinations(list(df.columns), 2))
```

```
    f = pd.DataFrame({'stat':np.ravel(sol).tolist()},index=es)
    f['p-val'] = [ 2*(1-stats.t.cdf(i,df=df2)) for i in f['stat']]
    f['p-val_Holm'] = multicomp(f['p-val'].tolist(), method='holm')[1].tolist()
    f['sign'] = multicomp(f['p-val'].tolist(), method='holm')[0].tolist()
    return f

print(post_hoc_quade_test_1(df))
```

```
##               stat     p-val  p-val_Holm   sign
## (a1, a2)  2.849145  0.007041    0.021122   True
## (a1, a3)  1.318261  0.195307    0.268163  False
## (a2, a3)  1.530884  0.134081    0.268163  False
```

Alternatywą może być badanie różnic w oparciu o sumy elementów macierzy $R_{ij}Q_i$ obliczone dla każdej grupy tzn. $W_j = \frac{\sum_{i=1}^{n} R_{ij}Q_i}{n(n+1)/2}$ a błąd standardowy jest dany wzorem:

$$SE = \sqrt{\frac{k(k+1)(2n+1)(k-1)}{18n(n+1)}} \tag{9.13}$$

```
import numpy as np
import scipy.stats as stats
import pandas as pd
from pingouin import multicomp
import itertools

df = pd.DataFrame()
df['a1'] = stats.norm.rvs(0, 1, size=20, random_state=2305)
df['a2'] = stats.norm.rvs(1.5, 3, size=20, random_state=4101)
df['a3'] = stats.norm.rvs(1.5, 2, size=20, random_state=4026)

def post_hoc_quade_test_2(x):
    df = x; n = df.shape[0]; k = df.shape[1]
    rdat = df.rank(axis=1)
    minmax = df.apply(lambda x: max(x)-min(x),axis=1)
    Q = pd.DataFrame(stats.rankdata(minmax), columns=['a'])
    W = rdat.values * Q.values
    SS = W.sum(axis=0)/(n*(n+1)/2)
    SE = np.sqrt((k*(k+1)*(2*n+1)*(k-1))/(18*n*(n+1)))
    stat = SS/SE
    res = list(itertools.combinations(stat, 2))
    sol = [np.abs(np.diff(res[i])) for i in range(len(res))]
    es = list(itertools.combinations(list(df.columns), 2))
    f = pd.DataFrame({'stat':np.ravel(sol).tolist()},index=es)
    f['p-val'] = [ 2*(1-stats.norm.cdf(i)) for i in f['stat']]
    f['p-val_Holm'] = multicomp(f['p-val'].tolist(), method='holm')[1].tolist()
    f['sign'] = multicomp(f['p-val'].tolist(), method='holm')[0].tolist()
    return f

print(post_hoc_quade_test_2(df))
```

```
##               stat     p-val  p-val_Holm   sign
## (a1, a2)  2.653017  0.007978    0.023933   True
## (a1, a3)  1.227516  0.219629    0.308024  False
## (a2, a3)  1.425502  0.154012    0.308024  False
```

Bibliografia

Anscombe, F. J. and Glynn, W. J. (1983). Distribution of the kurtosis statistic b2 for normal samples. *Biometrika*, 70(1):227–234.

Biecek, P. (2013). Wybrane testy normalności.

Biecek, P. (2017). *Przewodnik po pakiecie R*. Oficyna Wydawnicza "GIS".

Brunner, E., Dette, H., and Munk, A. (1997). Box-type approximations in nonparametric factorial designs. *Journal of the American Statistical Association*, 92(440):1494–1502.

Brunner, E. and Munzel, U. (2000). The nonparametric behrens-fisher problem: Asymptotic theory and a small-sample approximation. *Biometrical Journal*, 42(1):17–25.

Cameron, A. and Trivedi, P. (1998). *Regression Analysis of Count Data 2nd Edition*. Cambridge University Press.

Coeurjolly, J.-F., Drouilhet, R., Lafaye de Micheaux, P., and Robineau, J.-F. (2009). asympTest: A Simple R Package for Classical Parametric Statistical Tests and Confidence Intervals in Large Samples. *The R Journal*, 1(2):26–30.

Coly, S., Yao, A.-F., Abrial, D., and Charras-Garrido, M. (2016). Distributions to model overdispersed count data. *Journal de la Societe Francaise de Statistique*, 157(2):39–64.

D'Agostino, R. B. (1970). Transformation to normality of the null distribution of g1. *Biometrika*, 57(3):679–681.

Derrick, B. and White, P. (2016). Why welch's test is type i error robust. *The Quantitative Methods for Psychology*, 12(1):30–38.

Divine, G. W., Norton, H. J., Barón, A. E., and Juarez-Colunga, E. (2018). The wilcoxon–mann–whitney procedure fails as a test of medians. *The American Statistician*, 0(0):1–9.

Fuchs, M. and Krautenbacher, N. (2016). Minimization and estimation of the variance of prediction errors for cross-validation designs. *Journal of Statistical Theory and Practice*, 10(2):420–443.

Harrel, F. E. and Davis, C. E. (1982). A new distribution-free quantile estimator. *Biometrika*, 69(3):635–640.

Iwasaki Manabu (2005). Less conservative distribution-free confidence intervals and tests for the median. *Japanese Journal of Biometrics*, 26(2):65–80.

Jäntschi, L. and Bolboacă, S. D. (2018). Computation of probability associated with anderson–darling statistic. *Mathematics*, 6(6).

Marsaglia, J. and Marsaglia, G. (2004). Evaluating the anderson-darling distribution. *Journal of Statistical Software*, 09.

McCullagh, P. and Nelder, J. (1989). *Generalized Linear Models, Second Edition*. Chapman and Hall/CRC Monographs on Statistics and Applied Probability Series. Chapman & Hall.

McKean, J. W. and Schrader, R. M. (1984). A comparison of methods for studentizing the sample median. *Communications in Statistics - Simulation and Computation*, 13(6):751–773.

Meyer, J. P. and Seaman, M. A. (2013). A comparison of the exact kruskal-wallis distribution to asymptotic approximations for all sample sizes up to 105. *The Journal of Experimental Education*, 81(2):139–156.

Neubert, K. and Brunner, E. (2007). A studentized permutation test for the non-parametric behrens–fisher problem. *Computational Statistics and Data Analysis*, 51(10):5192 – 5204.

O'brien, R. G. and Kaiser, M. K. (1985). Manova method for analyzing repeated measures designs: an extensive primer. *Psychological bulletin*, 97 2:316–33.

Tukey, J. W. and McLaughlin, D. H. (1963). Less vulnerable confidence and significance procedures for location based on a single sample:trimming/winsorization 1. *Sankhya*, 25:331–352.

Wilcox Rand (2017). *Introduction to Robust Estimation and Hypothesis Testing 4th Edition*. Elsevier.

Wright, D. B. and Herrington, J. A. (2011). Problematic standard errors and confidence intervals for skewness and kurtosis. *Behavior Research Methods*, 43(1):8–17.

Zieliński, P. (2010). Multilevel analysis for repeated measures - hierarchical linear model as an alternative to the analysis of variance. *Psychologia Społeczna*, 5(14):234–259.