

# zumz4b

April 19, 2018

## 0.1 Uczenie maszynowe UMZ 2017/2018

### 1 4. Algorytm KNN, uczenie nienadzorowane

#### 1.0.1 Cz 2

#### 1.1 4.2. Uczenie nienadzorowane – Algorytm $k$ rednich

In [1]: # Przydatne importy

```
import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np
import pandas
import random
import seaborn
```

```
%matplotlib inline
```

In [2]: # Wczytanie danych (gatunki kosaców)

```
data_iris = pandas.read_csv('iris.csv', header=0, usecols=['od.d.', 'od.sz.', 'p.d.',
data_iris.columns=['x1', 'x2', 'x3', 'x4']
```

```
X = data_iris.values
Xs = data_iris.values[:, 2:4]
```

In [3]: # Wykres danych

```
def plot_unlabeled_data(X, col1=0, col2=1, x1label=r'$x_1$', x2label=r'$x_2$'):
    fig = plt.figure(figsize=(16*.7, 9*.7))
    ax = fig.add_subplot(111)
    fig.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9)
    X1 = X[:, col1].tolist()
    X2 = X[:, col2].tolist()
    ax.scatter(X1, X2, c='k', marker='o', s=50, label='Dane')
    ax.set_xlabel(x1label)
    ax.set_ylabel(x2label)
    ax.margins(.05, .05)
    return fig
```

```
In [4]: # Przygotowanie interaktywnego wykresu
```

```
dropdown_arg1 = widgets Dropdown(options=[0, 1, 2, 3], value=2, description='arg1')  
dropdown_arg2 = widgets Dropdown(options=[0, 1, 2, 3], value=3, description='arg2')
```

```
def interactive_unlabeled_data(arg1, arg2):  
    fig = plot_unlabeled_data(  
        X, col1=arg1, col2=arg2, x1label='$x_{}$'.format(arg1), x2label='$x_{}$'.format
```

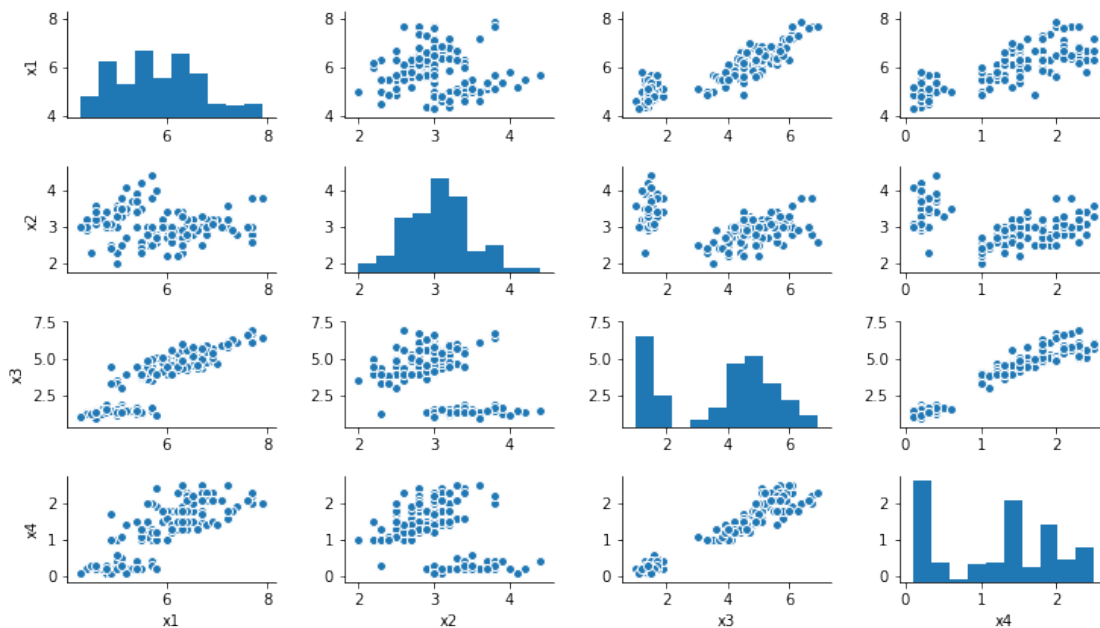
```
In [5]: widgets.interact(interactive_unlabeled_data, arg1=dropdown_arg1, arg2=dropdown_arg2)
```

```
interactive(children=(Dropdown(description=u'arg1', index=2, options=(0, 1, 2, 3), value=2), D
```

```
Out [5]: <function __main__.interactive_unlabeled_data>
```

```
In [6]: seaborn.pairplot(data_iris, vars=data_iris.columns, size=1.5, aspect=1.75)
```

```
Out [6]: <seaborn.axisgrid.PairGrid at 0x7f0eafdaf8d0>
```



```
In [7]: # Odlego euklidesowa
```

```
def euclidean_distance(x1, x2):  
    return np.linalg.norm(x1 - x2)
```

```
In [8]: # Algorytm k rednich
```

```
def k_means(X, k, distance=euclidean_distance):  
    history = []
```

```

Y = []

# Wylosuj centroid dla kadej klasy
centroids = [[random.uniform(X.min(axis=0)[f], X.max(axis=0)[f])
              for f in range(X.shape[1])]
             for c in range(k)]

# Powtarzaj, dopóki klasy si zmieniaj
while True:
    distances = [[distance(centroids[c], x) for c in range(k)] for x in X]
    Y_new = [d.index(min(d)) for d in distances]
    if Y_new == Y:
        break
    Y = Y_new
    XY = np.asarray(np.concatenate((X, np.matrix(Y).T), axis=1))
    Xc = [XY[XY[:, 2] == c][:, :-1] for c in range(k)]
    centroids = [[Xc[c].mean(axis=0)[f] for f in range(X.shape[1])]
                 for c in range(k)]
    history.append((centroids, Y))

result = history[-1][1]
return result, history

```

In [9]: # Wykres danych - klastrowanie

```

def plot_clusters(X, Y, k, centroids=None):
    color = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
    fig = plt.figure(figsize=(16*.7, 9*.7))
    ax = fig.add_subplot(111)
    fig.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9)

    X1 = X[:, 0].tolist()
    X2 = X[:, 1].tolist()
    X1 = [[x for x, y in zip(X1, Y) if y == c] for c in range(k)]
    X2 = [[x for x, y in zip(X2, Y) if y == c] for c in range(k)]

    for c in range(k):
        ax.scatter(X1[c], X2[c], c=color[c], marker='o', s=25, label='Dane')
        if centroids:
            ax.scatter([centroids[c][0]], [centroids[c][1]], c=color[c], marker='+', s=25)

    ax.set_xlabel(r'$x_1$')
    ax.set_ylabel(r'$x_2$')
    ax.margins(.05, .05)
    return fig

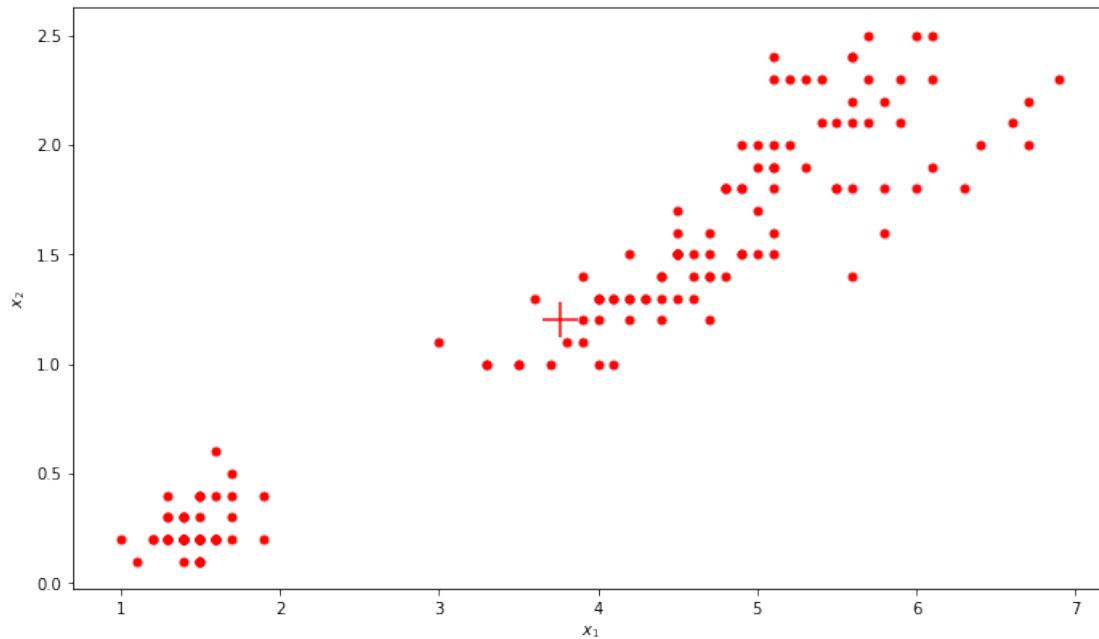
```

In [10]: Ys, history = k\_means(Xs, 2)

```
fig = plot_clusters(Xs, Ys, 2, centroids=history[-1][0])
```

/home/pawel/.local/lib/python2.7/site-packages/ipykernel\_launcher.py:21: RuntimeWarning: Mean of empty slice

```
/home/pawel/.local/lib/python2.7/site-packages/numpy/core/_methods.py:73: RuntimeWarning: invalid value encountered in divide
ret, rcount, out=ret, casting='unsafe', subok=False)
```



```
In [11]: # Przygotowanie interaktywnego wykresu
```

```
slider_k = widgets.IntSlider(min=1, max=7, step=1, value=2, description=r'$k$', width=100)
```

```
def interactive_kmeans_k(steps, history, k):
    if steps >= len(history) or steps == 10:
        steps = len(history) - 1
    fig = plot_clusters(Xs, history[steps][1], k, centroids=history[steps][0])
```

```
def interactive_kmeans(k):
    slider_steps = widgets.IntSlider(min=1, max=10, step=1, value=1, description=r'steps', width=100)
    _, history = k_means(Xs, k)
    widgets.interact(interactive_kmeans_k, steps=slider_steps,
                    history=widgets.fixed(history), k=widgets.fixed(k))
```

```
In [12]: widgets.interact_manual(interactive_kmeans, k=slider_k)
```

```
interactive(children=(IntSlider(value=2, description=u'$k$', max=7, min=1), Button(description='Interact')))
```

```
Out[12]: <function __main__.interactive_kmeans>
```

### 1.1.1 Algorytm $k$ rednich – dane wejciowe

- $k$  – liczba klastrów
- zbiór uczycy  $X = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ ,  $x^{(i)} \in \mathbb{R}^n$

Na wejciu nie ma zbioru  $Y$ , poniewa jest to uczenie nienadzorowane!

### 1.1.2 Algorytm $k$ rednich – pseudokod

1. Zainicjalizuj losowo  $k$  centroidów (rodków cikoci klastrów):  $\mu_1, \dots, \mu_k$ .
2. Powtarzaj dopóki przyporzkowania klastrów si zmieniaj:
3. Dla  $i = 1$  do  $m$ : za  $y^{(i)}$  przyjmij klas najbliszego centroidu.
4. Dla  $c = 1$  do  $k$ : za  $\mu_c$  przyjmij redni wszystkich punktów  $x^{(i)}$  takich, e  $y^{(i)} = c$ .

In [13]: # Algorytm  $k$  rednich

```
def k_means(X, k, distance=euclidean_distance):
    Y = []
    centroids = [[random.uniform(X.min(axis=0)[f], X.max(axis=0)[f])
                  for f in range(X.shape[1])]
                 for c in range(k)] # Wylusuj centroidy
    while True:
        distances = [[distance(centroids[c], x) for c in range(k)]
                     for x in X] # Oblicz odlegosci
        Y_new = [d.index(min(d)) for d in distances]
        if Y_new == Y:
            break # Jeli nic si nie zmienia, przerwij
        Y = Y_new
        XY = np.asarray(np.concatenate((X, np.matrix(Y).T), axis=1))
        Xc = [XY[XY[:, 2] == c][:, :-1] for c in range(k)]
        centroids = [[Xc[c].mean(axis=0)[f]
                     for f in range(X.shape[1])]
                    for c in range(k)] # Przesu centroidy
    return Y
```

- Liczba klastrów jest okrelona z góry i wynosi  $k$ .
- Jeeli w którym kroku algorytmu jedna z klas nie zostanie przyporzkowana adnemu z przykadów, pomija si  $j$  – w ten sposób wynikiem dziaania algorytmu moe by mniej ni  $k$  klastrów.

### 1.1.3 Funkcja kosztu dla problemu klastrowania

$$J(y^{(1)}, \dots, y^{(m)}, \mu_1, \dots, \mu_k) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{y^{(i)}}\|^2$$

- Zauwamy, e z kadym krokiem algorytmu  $k$  rednich koszt si zmniejsza (lub ewentualnie pozostaje taki sam).

### 1.1.4 Wielokrotna inicjalizacja

- Algorytm  $k$  rednich zawsze znajdzie lokalne minimum funkcji kosztu  $J$ , ale nie zawsze będzie to globalne minimum.
- Aby temu zaradzić, można uruchomić algorytm  $k$  rednich wiele razy, za każdym razem z innym losowym poaniem centroidów (tzw. **wielokrotna losowa inicjalizacja** – *multiple random initialization*).
- Za każdym razem obliczamy koszt  $J$ . Wybieramy ten wynik, który ma najniższy koszt.

### 1.1.5 Wybór liczby klastrów $k$

- Najlepiej wybrać  $k$  rznie w zależności od kształtu danych i celu, który chcemy osiągnąć.

## 1.2 4.3. Metryki

```
In [16]: def powerme(x1,x2,n):
        X = []
        for m in range(n+1):
            for i in range(m+1):
                X.append(np.multiply(np.power(x1,i),np.power(x2,(m-i))))
        return np.hstack(X)
```

```
In [17]: # Wykres danych
def plot_data_for_classification(X, Y, xlabel=None, ylabel=None, Y_predicted=[], highlight=None):
    fig = plt.figure(figsize=(16*6, 9*6))
    ax = fig.add_subplot(111)
    fig.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9)
    X = X.tolist()
    Y = Y.tolist()
    X1n = [x[1] for x, y in zip(X, Y) if y[0] == 0]
    X1p = [x[1] for x, y in zip(X, Y) if y[0] == 1]
    X2n = [x[2] for x, y in zip(X, Y) if y[0] == 0]
    X2p = [x[2] for x, y in zip(X, Y) if y[0] == 1]

    if Y_predicted != []:
        Y_predicted = Y_predicted.tolist()
        X1tn = [x[1] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 0 and yp[0] == 0]
        X1fn = [x[1] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 1 and yp[0] == 0]
        X1tp = [x[1] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 1 and yp[0] == 1]
        X1fp = [x[1] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 0 and yp[0] == 1]
        X2tn = [x[2] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 0 and yp[0] == 0]
        X2fn = [x[2] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 1 and yp[0] == 0]
        X2tp = [x[2] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 1 and yp[0] == 1]
        X2fp = [x[2] for x, y, yp in zip(X, Y, Y_predicted) if y[0] == 0 and yp[0] == 1]

    if Y_predicted != []:
        if highlight == 'tn':
            ax.scatter(X1tn, X2tn, c='r', marker='x', s=50, label='Dane')
```

```

        ax.scatter(X1fn, X2fn, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1tp, X2tp, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1fp, X2fp, c='k', marker='x', s=50, label='Dane')
    elif highlight == 'fn':
        ax.scatter(X1tn, X2tn, c='k', marker='x', s=50, label='Dane')
        ax.scatter(X1fn, X2fn, c='r', marker='o', s=50, label='Dane')
        ax.scatter(X1tp, X2tp, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1fp, X2fp, c='k', marker='x', s=50, label='Dane')
    elif highlight == 'tp':
        ax.scatter(X1tn, X2tn, c='k', marker='x', s=50, label='Dane')
        ax.scatter(X1fn, X2fn, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1tp, X2tp, c='g', marker='o', s=50, label='Dane')
        ax.scatter(X1fp, X2fp, c='k', marker='x', s=50, label='Dane')
    elif highlight == 'fp':
        ax.scatter(X1tn, X2tn, c='k', marker='x', s=50, label='Dane')
        ax.scatter(X1fn, X2fn, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1tp, X2tp, c='k', marker='o', s=50, label='Dane')
        ax.scatter(X1fp, X2fp, c='g', marker='x', s=50, label='Dane')
    else:
        ax.scatter(X1tn, X2tn, c='r', marker='x', s=50, label='Dane')
        ax.scatter(X1fn, X2fn, c='r', marker='o', s=50, label='Dane')
        ax.scatter(X1tp, X2tp, c='g', marker='o', s=50, label='Dane')
        ax.scatter(X1fp, X2fp, c='g', marker='x', s=50, label='Dane')

else:
    ax.scatter(X1n, X2n, c='r', marker='x', s=50, label='Dane')
    ax.scatter(X1p, X2p, c='g', marker='o', s=50, label='Dane')

if xlabel:
    ax.set_xlabel(xlabel)
if ylabel:
    ax.set_ylabel(ylabel)

ax.margins(.05, .05)
return fig

```

In [18]: # Wczytanie danych

```

import pandas
import numpy as np

alldata = pandas.read_csv('data.tsv', sep='\t')
data = np.matrix(alldata)

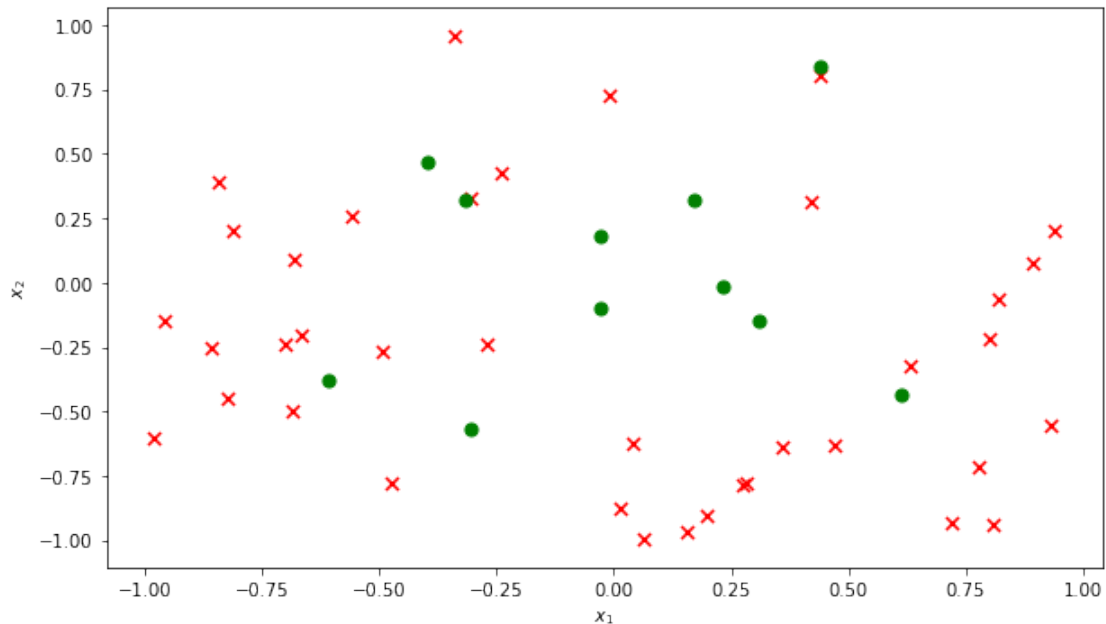
m, n_plus_1 = data.shape
n = n_plus_1 - 1
Xn = data[:, 1:]

X2 = powerme(data[:, 1], data[:, 2], n)

```

```
Y2 = np.matrix(data[:, 0]).reshape(m, 1)
```

```
In [19]: fig = plot_data_for_classification(X2, Y2, xlabel=r'$x_1$', ylabel=r'$x_2$')
```



```
In [20]: def safeSigmoid(x, eps=0):  
    y = 1.0/(1.0 + np.exp(-x))  
    if eps > 0:  
        y[y < eps] = eps  
        y[y > 1 - eps] = 1 - eps  
    return y  
  
def h(theta, X, eps=0.0):  
    return safeSigmoid(X*theta, eps)  
  
def J(h,theta,X,y, lamb=0):  
    m = len(y)  
    f = h(theta, X, eps=10**-7)  
    j = -np.sum(np.multiply(y, np.log(f)) +  
              np.multiply(1 - y, np.log(1 - f)), axis=0)/m  
    if lamb > 0:  
        j += lamb/(2*m) * np.sum(np.power(theta[1:],2))  
    return j  
  
def dJ(h,theta,X,y,lamb=0):  
    g = 1.0/y.shape[0]*(X.T*(h(theta,X)-y))  
    if lamb > 0:
```



```

        g[1:] += lamb/float(y.shape[0]) * theta[1:]
    return g

def classifyBi(theta, X):
    prob = h(theta, X)
    return prob

In [21]: # Metoda gradientu prostego dla regresji logistycznej
def GD(h, fJ, fdJ, theta, X, y, alpha=0.01, eps=10**-3, maxSteps=10000):
    errorCurr = fJ(h, theta, X, y)
    errors = [[errorCurr, theta]]
    while True:
        # oblicz nowe theta
        theta = theta - alpha * fdJ(h, theta, X, y)
        # raportuj poziom bdu
        errorCurr, errorPrev = fJ(h, theta, X, y), errorCurr
        # kryteria stopu
        if abs(errorPrev - errorCurr) <= eps:
            break
        if len(errors) > maxSteps:
            break
        errors.append([errorCurr, theta])
    return theta, errors

In [22]: # Uruchomienie metody gradientu prostego dla regresji logistycznej
theta_start = np.matrix(np.zeros(X2.shape[1])).reshape(X2.shape[1],1)
theta, errors = GD(h, J, dJ, theta_start, X2, Y2,
                  alpha=0.1, eps=10**-7, maxSteps=10000)
print('theta = {}'.format(theta))

theta = [[ 1.37136167]
 [ 0.90128948]
 [ 0.54708112]
 [-5.9929264 ]
 [ 2.64435168]
 [-4.27978238]]

In [23]: # Wykres granicy klas
def plot_decision_boundary(fig, theta, X):
    ax = fig.axes[0]
    xx, yy = np.meshgrid(np.arange(-1.0, 1.0, 0.02),
                        np.arange(-1.0, 1.0, 0.02))
    l = len(xx.ravel())
    C = powerme(xx.reshape(l, 1), yy.reshape(l, 1), n)
    z = classifyBi(theta, C).reshape(int(np.sqrt(l)), int(np.sqrt(l)))

    plt.contour(xx, yy, z, levels=[0.5], lw=3);

```

```
In [24]: Y_predicted = (classifyBi(theta, X2) > 0.5).astype(int)
        Y_predicted[:10]
```

```
Out[24]: matrix([[0],
                 [1],
                 [0],
                 [0],
                 [0],
                 [1],
                 [1],
                 [1],
                 [1],
                 [0]])
```

```
In [25]: # Przygotowanie interaktywnego wykresu
```

```
dropdown_highlight = widgets Dropdown(options=['all', 'tp', 'fp', 'tn', 'fn'], value=
def interactive_classification(highlight):
    fig = plot_data_for_classification(X2, Y2, xlabel=r'$x_1$', ylabel=r'$x_2$',
                                      Y_predicted=Y_predicted, highlight=highlight)
    plot_decision_boundary(fig, theta, X2)
```

```
In [26]: widgets.interact(interactive_classification, highlight=dropdown_highlight)
```

```
interactive(children=(Dropdown(description=u'highlight', options=('all', 'tp', 'fp', 'tn', 'fn'
```

```
Out[26]: <function __main__.interactive_classification>
```

Dokadno (*accuracy*):

$$accuracy = \frac{tp + tn}{tp + fp + tn + fn}$$

Precyzja (*precision*):

$$precision = \frac{tp}{tp + fp}$$

Pokrycie (*recall*):

$$recall = \frac{tp}{tp + fn}$$

*F*-measure:

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

$F_\beta$ -measure:

$$F_\beta = \frac{(1 + \beta) \cdot precision \cdot recall}{\beta^2 * precision + recall}$$

- $F = F_1$

## 1.3 4.4. Analiza głównych składowych

### 1.3.1 Redukcja liczby wymiarów

Z jakich powodów chcemy zredukować liczbę wymiarów?

- Chcemy pozbyć się nadmiarowych cech, np. „długo w cm” / „długo w calach”, „długo” i „szeroko” / „powierzchnia”.
- Chcemy znaleźć bardziej optymalną kombinację cech.
- Chcemy przyspieszyć działanie algorytmów.
- Chcemy zwizualizować dane.

### 1.3.2 Błąd rzutowania

**Błąd rzutowania** – błąd średniokwadratowy pomiędzy danymi oryginalnymi a danymi rzutowanymi.

### 1.3.3 Sformułowanie problemu

**Analiza głównych składowych** (*Principal Component Analysis, PCA*):

Zredukować liczbę wymiarów z  $n$  do  $k$ , czyli znaleźć  $k$  wektorów  $u^{(1)}, u^{(2)}, \dots, u^{(k)}$  takich, że rzutowanie danych na podprzestrzeń rozpiętą na tych wektorach minimalizuje błąd rzutowania.

- Analiza głównych składowych to zupełnie inne zagadnienie niż regresja liniowa!

### 1.3.4 Algorytm PCA

1. Dany jest zbiór składający się z  $x^{(1)}, x^{(2)}, \dots, x^{(m)} \in \mathbb{R}^n$ .
2. Chcemy zredukować liczbę wymiarów z  $n$  do  $k$  ( $k < n$ ).
3. W ramach wstępnego przetwarzania dokonujemy skalowania i normalizacji danych.
4. Znajdujemy macierz kowariancji:

$$\Sigma = \frac{1}{m} \sum_{i=1}^m \begin{pmatrix} x^{(i)} \end{pmatrix} \begin{pmatrix} x^{(i)} \end{pmatrix}^T$$

5. Znajdujemy wektory własne macierzy  $\Sigma$  (rozkład SVD):

$$(U, S, V) := \text{SVD}(\Sigma)$$

6. Pierwszych  $k$  kolumn macierzy  $U$  to szukane wektory.

In [27]: `from sklearn.preprocessing import StandardScaler`

```
# Algorytm PCA - implementacja
def pca(X, k):
    X_std = StandardScaler().fit_transform(X) # normalizacja
    mean_vec = np.mean(X_std, axis=0)
    cov_mat = np.cov(X_std.T) # macierz kowariancji
    n = cov_mat.shape[0]
```

```
eig_vals, eig_vecs = np.linalg.eig(cov_mat) # wektory wasne
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:, i])
              for i in range(len(eig_vals))]
eig_pairs.sort()
eig_pairs.reverse()
matrix_w = np.hstack([eig_pairs[i][1].reshape(n, 1)
                      for i in range(k)]) # wybór
return X_std.dot(matrix_w) # transformacja
```

```
In [28]: X_pca = pca(X, 2)
fig = plot_unlabeled_data(X_pca)
```

