Politechnika Śląska
Wydział Matematyki Stosowanej
Informatyka

DOKUMENTACJA PROJEKTU

Ocena poziomu stresu na podstawie analizy snu

Systemy sztucznej inteligencji

Michał Pawełek, Michał Pawlak, Jan Walica

Gliwice, 10 czerwca 2022

Spis treści

Część I

1 Opis projektu

Celem projektu jest porównanie i analiza dokładności i wydajności wybranych algorytmów klasyfikujących poziom stresu w oparciu o zbiór danych zawierających parametry snu. Pod uwagę wzięto różne implementacje danych algorytmów – zrealizowane własnoręcznie oraz te znajdujące się w popularnych bibliotekach.

Do analizy wybrano algorytmy:

- k-najbliższych sąsiadów (KNN),
- naiwny klasyfikator Bayesa,
- Perceptron

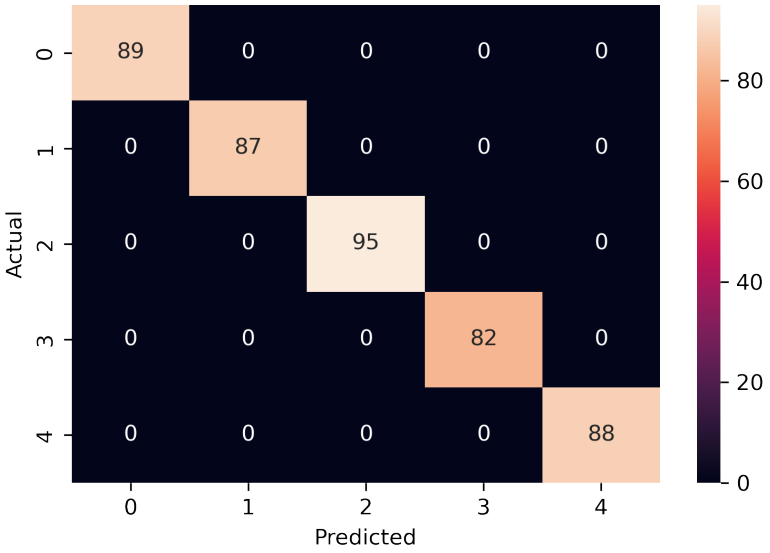
2 Instrukcja obsługi

Projekt zrealizowano przy użyciu środowiska *Jupyter Notebook* i podzielono na pliki wg. algorytmów, które zostały w nich zaimplementowane – każdy plik odpowiada jednemu algorytmowi. W ramach pliku znajdują się wszystkie zastosowane implementacje danego algorytmu.

W wyniku przeprowadzenia klasyfikacji program zwraca wartości metryk i macierz błędów w postaci tekstowej (Listing 1) oraz zapisuje jej graficzną reprezentację do pliku (Rysunek 1)

Listing 1: Przykładowe wyniki oceny algorytmu

```
Predicted   0   1   2   3   4
Actual
0          89   0   0   0   0
1           0  87   0   0   0
2           0   0  95   0   0
3           0   0   0  82   0
4           0   0   0   0  88
 Accuracy: 1.000
Precision: 1.000
   Recall: 1.000
       F1: 1.000
```



Rysunek 1: Przykładowa macierz błędów

Część II

3 Opis działania

3.1 Wspólne

Przed rozpoczęciem klasyfikacji zbiór danych należało podzielić na zbiór treningowy i testowy. Pierwszy z nich, traktowany jako znane dane, służy jako punkt odniesiena dla algorytmów, uczenia ich. Drugi zbiór traktowany był jako dane wprowadzane do programu w celu ich sklasyfikowania. Dzięki posiadaniu wiedzy o klasach, jakie przypisane były do tych próbek można było porównać je do wyników klasyfikacji i przeprowadzić analizę skuteczności algorytmów. Zwykle stosowane proporcje podziału to 7 : 3, jednak testy przeprowadzane były także dla innych wartości.

W przypadku, gdy dane w zbiorze są upporządkowane (tzn. poszczególne klasy występują w grupach), należy przed podziałęm zbioru przemieszać go w celu uzyskania wiarygodnych wyników. Pozwala to uniknąć sytuacji, w której w zbiorze testowym pojawią się klasy niewystępujące w zbiorze treningowym, których algorytm nie znał, a wieć nie miał możliwości poprawnie sklasyfikować.

Kolejnym etapem była normalizacja wartości pojawiających się w zbiorze. Proces ten polega na przekształceniu wartości znajdujących się w pewnym przedziale na takie z przedziału $\langle 0; 1\rangle$, co w praktyce polega na wyszukaniu dla każdej cechy z osobna wartości największej i najmniejszej, a następnie przekształcenie każdej wartości zgodnie ze wzorem:

$$x_{new} = \frac{x_{old} - min}{max - min}$$

Po przeprowadzeniu klasyfikacji poszczególnymi algorytmami obliczone zostały miary wydajności pozwalające określić jakość algorytmów. W tym celu należy wprowadzić pojęcia (przy założeniu, że operujemy na dwóch klasach – pozytywnej i negatywnej) wartości *Prawdziwie Negatywnej* (*TN* – True Negative), czyli właściwie sklasyfikowanej jako negatywna, *Prawdziwie Pozytywnej* (*TP* – True Positive), czyli właściwie sklasyfikowanej jako pozytywna oraz *Fałszywie Negatywnej* (*FN* – False Negative) i *Fałszywie Pozytywnej* (*FP* – False positive), czyli sklasyfikowane niepoprawnie, jako klasy przeciwne.

		Sklasyfikowane	
		Negatywne	Pozytywne
Rzeczywiste	Negatywne	TN	FP
	Pozytywne	FN	TP

Tablica 1: Macierz błędów

Na podstawie tych wartości obliczone zostały miary:

- **Dokładność (Accuracy)** – jest to najbardziej intuicyjna miara określająca stosunek poprawnie sklasyfikowanych próbek do wszystkich próbek

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precyzja (Precision)** – stosunek prawidłowo skalsyfikowanych próbek pozytywnych do wszystkich próbek sklasyfikowanych jako pozytywne

$$Precision = \frac{TP}{TP + FP}$$

- **Czułość (Recall)** – stosunek próbek prawidłowo sklasyfikowanych jako pozytywne do wszystkich pozytywnych

$$Accuracy = \frac{TP}{TP + FN}$$

- **F1** – średnia ważona precyzji i czułości. W przypadku nierównomiernego rozkładu klas daje ona bardziej miarodajne wyniki przy założeniu, że wartości Fałszywe są na podobnym poziomie.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

3.2 algorytm KNN

Algorytm k-najbliższych sąsiadów jest jednym z najprostszych algorytmów uczenia maszynowego. Opiera się na założeniu, że cechy rozpatrywanego przypadku są podobne do tych już znanych (ze zbioru treningowego). Na podstawie oceny odległości danej cechy od innych wartości, nadaje próbce taką samą klasę, jaką posiada największa liczba spośród zadanej ilości (k) sąsiadów.

We własnej implementacji algorytmu, pierwszą wykonywaną czynnością jest określenie listy klas, jakie posiadają dane w zbiorze treningowym. Następnie obliacza jest odległość klasyfikowanej próbki od każdej pozycji w zbiorze treningowym. Do tego celu wykorzystano Odległość Minkowskiego. Przyjmuje się wtedy, że próbki są punktami w n-wymiarowej przestrzeni euklidesowej, gdzie n, to liczba branych pod uwagę cech:

$$X(x_1, x_2, \ldots, x_n), Y(y_1, y_2, \ldots, y_n) \in \mathbb{R}^n$$

Wtedy odległość między takimi punktami w przestrzeni oblicza się wg. wzoru:

$$D(X, Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^m \right)^{\frac{1}{m}}$$

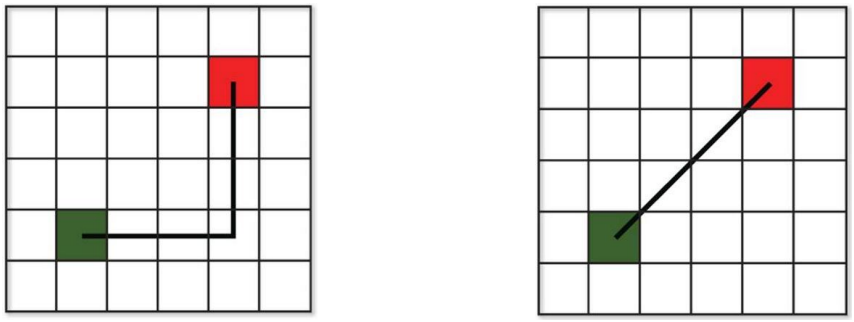
W zależności od parametru m uzyskuje się różne odmiany Odległości Minkowskiego:

- dla $m = 1$ będzie to odległość typu miejska/taksówkowa/Manhattan, liczona jako suma bezwzględnych różnich pomiędzy współrzędnymi kartezjańskimi punktów.

$$D_1(X, Y) = \sum_{i=1}^{n} |x_i - y_i|$$

- dla $m = 2$ będzie to odległość euklidesowa, która jest odległością w lini prostej między punktami w przestrzeni euklidesowej.

$$D_2(X, Y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$$



(a) odległość Manhattan (b) odległość euklidesowa

Rysunek 2: Przedstawienie omawianych metryk w przestrzeni 2-wymiarowej

Gdy zostaną wyznaczone wszystkie odległości, przeprowadzone zostaje głosowanie – należy zliczyć ile spośród zadanych k punktów, które leżą najbliżej próbki należy do każdej z klas, a następnie przypisać rozpatrywanej próbce klasę najliczniejszej grupy sąsiadów.

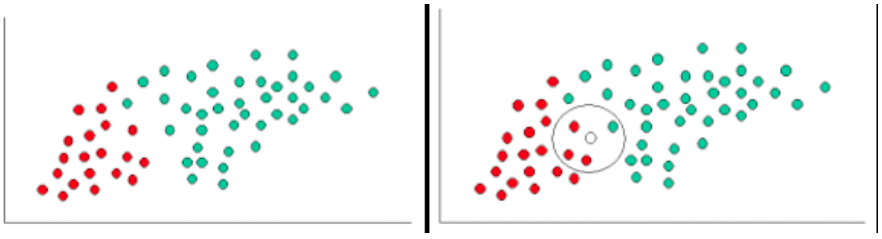
Algorytm KNN jest algorytmem leniwego uczenia się – nie generalizuje on danych, za każdym razem przeprowadza operacje na danych z zadanego zbioru treningowego. Algorytm ten wykorzystuje nadzorowaną technikę uczenia się, co oznacza, że dane treningowe są już sklasyfikowane i algoryt korzysta z tej klasyfikacji.

3.3 naiwny klasyfikator Bayesa

Naiwny klasyfikator Bayesowski, bazujący na twierdzeniu Bayesa, nadaje się szczególnie do problemów o bardzo wielu wymiarach na wejściu. Mimo prostoty metody, często działa ona lepiej od innych, bardzo skomplikowanych metod klasyfikujących. Opisuje się go wzorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

gdzie:
$P(A|B)$ - Jak prawdopodobna jest hipoteza, mając skutek,
$P(B|A)$ - Jak prawdopodobny jest dowód, zakładając że nasza hipoteza jest prawdziwa,
$P(A)$ - Jak prawdopodobna była nasza hipoteza przed zaobserwowaniem dowodów,
P(B) - Suma prawdopodobieństw wszystkich potencjalnych skutków zdarzenia:
$P(B) = \sum P(B|A)P(B)$



Rysunek 3: Sposób działania klasyfikatora Bayesa

Dla ilustracji koncepcji naiwnej metody Bayesa, rozpatrzmy przykład z powyższego rysunku. Jak widać, mamy tu obiekty zielone i czerwone. Naszym zadaniem będzie zaklasyfikowanie nowego obiektu, który może się tu pojawić.

Ponieważ zielonych kółek jest dwa razy więcej niż czerwonych, rozsądnie będzie przyjąć, że nowy obiekt (którego jeszcze nie mamy) będzie miał dwa razy większe prawdopodobieństwo bycia zielonym niż czerwonym.

W analizie Bayesowskiej, takie prawdopodobieństwa nazywane są prawdopodobieństwami a priori. Prawdopodobieństwa a priori wynikają z posiadanych, wcześniejszych (a priori) obserwacji. W tym wypadku, chodzi o procent zielonych względem czerwonych. Prawdopodobieństwa a priori często służą do przewidywania klasy nieznanych przypadków, zanim one się pojawią.

Mając obliczone prawdopodobieństwa a priori, jesteśmy gotowi do zaklasyfikowania nowego obiektu (kółko białe). Ponieważ obiekty są dobrze pogrupowane sensownie będzie założyć, że im więcej jest zielonych (albo czerwonych) obiektów w pobliżu nowego obiektu, tym bardziej prawdopodobne jest, że obiekt ten ma kolor zielony (czerwony). Narysujmy więc okrąg wokół nowego obiektu, taki by obejmował, wstępnie zadaną liczbę obiektów (niezależnie od ich klasy). Teraz będziemy mogli policzyć, ile wewnątrz okręgu jest zielonych, a ile czerwonych kółek. Skąd obliczymy wielkość, którą można nazwać szansą.

Jasne jest, że w powyższym przykładzie szansa, że X będzie czerwone jest większa niż szansa, że X będzie zielone.

Mimo, że prawdopodobieństwo a priori wskazuje, że X raczej będzie zielone (bo zielonych jest dwa razy więcej niż czerwonych), to szanse są odwrotne, ze względu na bliskość czerwonych. Końcowa klasyfikacja w analizie Bayesowskiej bazuje na obu informacjach, wg reguły Bayesa.

W rezultacie klasyfikujemy X jako czerwone, gdyż większe jest prawdopodobieństwo a posteriori takiej właśnie przynależności.

Różne naiwne klasyfikatory Bayesa różnią się głównie założeniami, jakie przyjmują w odniesieniu do rozkładu $P(x_i|y)$. Uczący i klasyfikujący naiwni Bayes'i mogą być niezwykle szybcy w porównaniu z bardziej wyrafinowanymi metodami. Oddzielenie rozkładów cech warunkowych klas oznacza, że każdy rozkład może być niezależnie estymowany jako rozkład jednowymiarowy. To z kolei pomaga złagodzić problemy wynikające z przekleństwa wymiarowości. Z drugiej strony, chociaż naiwny Bayes jest znany jako przyzwoity klasyfikator, wiadomo, że jest złym estymatorem, więc wyników prawdopodobieństwa z przewidywanych próba nie należy traktować zbyt poważnie.

W projekcie są używane trzy sposoby klasyfikacji dla naiwnego Bayesa:

1. Algorytm Gaussa - jest używany szczególnie wtedy, gdy cechy mają wartości ciągłe. Zakłada się również, że wszystkie cechy mają rozkład Gaussa, tj. rozkład normalny.

$$P(x_i|y) = \frac{exp(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2})}{\sqrt{2\pi\sigma_y^2}}$$

   gdzie μ to średnia arytmetyczna a σ to odchylenie standardowe

2. Algorytm Bernolliego - implementuje naiwne algorytmy uczenia i klasyfikacji Bayesa dla danych, które są dystrybuowane zgodnie z wielowymiarowymi rozkładami Bernoulliego

tzn. może istnieć wiele cech, ale zakłada się, że każda z nich jest zmienną o wartości binarnej (Bernoulli, boolean). Dlatego ta klasa wymaga, aby próbki były reprezentowane jako wektory cech o wartościach binarnych; w przypadku przekazania innego rodzaju danych, instancja BernoulliNB może zbinaryzować swoje dane wejściowe (w zależności od parametru binaryzacji).
Zasada decyzyjna dla Bernoulliego naiwnego Bayesa opiera się na:

$$P(x_i|y) = P(i|y)x_i + (1 - P(i|y))(1 - x_i)$$

3. Algorytm nienazwany - był on podany na kolokwium i wstępnie nie posiadał żadnej nazwy. Wzór jego to:

$$f(x) = \begin{cases} \frac{x-\mu}{6\sigma^2} + \frac{1}{\sqrt{6}\sigma} & \text{dla} \quad \mu - \sqrt{6}\sigma \leq x \leq \mu, \\ \frac{-(x-\mu)}{6\sigma^2} + \frac{1}{\sqrt{6}\sigma} & \text{dla} \quad \mu < x \leq \mu + \sqrt{6}\sigma, \\ 0 & \text{dla pozostałych} \end{cases}$$

3.4 Perceptron

Perceptron jest prostym algorytmem uczenia maszynowego. Miał on odwzorowywać pracę pojedynczego neuronu. Perceptron otrzymuje próbkę danych wejściowych, która jest klasyfikowana, poprzez przypisanie wag jej cechom. Aby to osiągnąć perceptron przechodzi przez fazy treningu i testowania. W fazie treningu wagi są inicjowane dowolną wartością. Następnie perceptron ocenia próbkę oraz porównuje swoją decyzję z rzeczywistą klasą próbki.

Jeśli algorytm dokonał złej klasyfikacji, wagi są dostosowywane, aby lepiej pasowały do danej próbki. Proces jest powtarzany w kółko, by precyzyjnie zoptymalizować błędy systematyczne. Po tym algorytm jest gotowy do przetestowania z zestawem nieznanych mu próbek, by sprawdzić, czy wytrenowany model jest wystarczająco ogólny, aby poradzić sobie z innymi próbkami.

Ten prosty opis od razu podsuwa potencjalne problemy:

- Model musi być przetrenowany przez dużą liczbę już sklasyfikowanych próbek. Czasem może wystarczyć ich kilkadziesiąt, a czasem potrzeba będzie nawet tysięcy próbek.
- Faza przetwarzania zająć się musi brakiem cech, nieskorelowanymi danymi oraz skalowaniem.
- Perceptron potrzebuje próbek dających się oddzielić liniowo, w celu osiągnięcia zbieżności.

Reprezentując próbki jako wektory o rozmiarze n, gdzie n jest liczbą cech, perceptron może być modelowany przez złożenie dwóch funkcji. Pierwsza z nich, niech będzie to f(x), mapuje wektor cech wejściowych x na wartość skalarną przesuniętą o wartość b:

$$f(x) = \begin{cases} \mathbb{R}^n \to \mathbb{R} \\ w \cdot x + b \end{cases}$$

,gdzie $w \cdot x$ jest iloczynem skalarnym między w i x zdefiniowanym jako: $\sum_{i=1}^{n} w_i x_i$

Drugą funkcją predict(x), funkcja skokowa, zazwyczaj wykorzystuje się funkcje heavyside'a lub signum, mapuje wartość skalarną na wyjście binarne.

$$predict(x) = sgn(f(x)) = \begin{cases} 1 & \text{jeśli} \quad f(x) \geq 0 \\ -1 & \text{dla pozostałych} \end{cases}$$

4 Algorytmy

4.1 Wspólne

Normalizacja zbioru

Data: X
Result: $XNorm$

// ustalenie wartości maksymalnej i minimalnej w kolumnie
for $i \leftarrow 0$ **to** $len(X[0])$ **do**
  $max = max(X[:, i])$
  $min = min(X[:, i])$

  // obliczenie nowej wartości w kolumnie dla wszystkich próbek
  for $j \leftarrow 0$ **to** $len(X)$ **do**
    $XNorm[j, i] = (X[j, i] - min)/(max - min)$
  end
end

Algorithm 1: Algorytm normalizacji zbioru

4.2 algorytm KNN

Odległość Minkowskiego

Data: $v1$, $v2$, m
Result: D

$D = 0$
for $i \leftarrow 0$ **to** $len(v1)$ **do**
  $D+ = abs(v1[i] - v2[i]) * *m$
end
$D = D * *(1/m)$

Algorithm 2: Algorytm Odległości Minkowskiego

Właściwy algorytm KNN

Data: $sample$, X, C, k, m
Result: $result$

// utworzenie slownika na podstawie nazw klas
$classes = \{\ \}$
for $i \leftarrow 0$ **to** $len(C)$ **do**
  $classes[C[i]] = 0$
end

// obliczenie odleglosci probki od kazdego rekordu w zbiorze
$distances = [\,]$
for $i \leftarrow 0$ **to** $len(X)$ **do**
| $distances \mathrel{+}= D(sample, X[i], m)$
end

// sortowanie (stogowe) zbioru wzgledem odleglosci
$heap = [\,]$
for $i \leftarrow 0$ **to** $len(X)$ **do**
| $heap \mathrel{+}= (distances[i], X[i])$
end

// glosowanie
for $i \leftarrow 0$ **to** k **do**
| $classes[heap.pop()] \mathrel{+}= 1$
end
$result = max(classes)$

Algorithm 3: Algorytm KNN

4.3 naiwny klasyfikator Bayesa

Data: $XTrain, XTest$
Result: $result$

$s = XTrain$
$\mu = XTest$
// pętla po każdym elemencie modelu treningowego
for $q \leftarrow 1$ **to** s **do**
| // inicjalizacja elementów modelu treningowego
| $\mu[q] = 0$
end
// pętla po każdym wektorze
for $j \leftarrow 1$ **to** μ **do**
| // zwiększanie liczby wektorów dla wartości $x_{j.p}$ z wektora x_j
| $\mu[d[j,p]] + +$
| // pętla po każdym atrybucie
| **for** $k \leftarrow 1$ **to** $p - 1$ **do**
| | // zwiększanie liczby wektorów z wartościami $x_{j.k}$ oraz $x_{j.p}$
| | $z = \mu[\phi(k-1) + (d[j,k] - 1) * \phi(0) + d[j,p]]$
| | $z = z + 1$
| **end**
end
$result = z$

Algorithm 4: Algorytm Bayesa

4.4 Perceptron

Data: X, Y, N_iter
Result: Brak //algorytm modyfikuje wartości w obiekcie
//przyjęcie domyślnej wartości przesunięcia jako 0
$self.b = 0$;
//utworzenie tablicy wag równych zero
$self.w = np.zeros(x.shape[1])$;
//utworzenie pustej listy ilości źle sklasyfikowanych próbek
$self.misclassified\_samples = [\ ]$;
//pętle dla wybranej ilości iteracji **for** $i \leftarrow 0$ **to** N_iter **do**
  //utworzenie licznika błędów dla bierzącej iteracji
  errors=0
  //pętla po każdej próbce
  for xi, yi ***in*** $zip(X, Y)$ **do**
    // obliczenie różnicy między przewidzianą wartością t(x),
    //a prawdziwą wartością, przeskalowane przez learning rate(domyślnie= 0.1)
    $update = self.learning\_rate * (y_i - self.predict(x_i))$
    //aktualizacja wartości przesunięcia oraz tablicy wag
    $self.b += update$
    $self.w += update * x_i$
    //jeśli przewidywana wartość jest poprawna, update będzie równy 0
    $errors += int(update! = 0)$
  end
  //wpisanie do listy liczbę błędów w bierzącej iteracji
  $self.missclasified\_samples.append(errors)$
end

Algorithm 5: Algorytm Perceptronu

5 Zbiór danych

Wykorzystany zbiór danych opiera się na informacjach zbieranych przez urządzenie *Smart-Yoga Pillow* (*SaYoPillow*). Ma ono na celu pomóc w zrozumieniu zależności pomiędzy parametrami snu, a poziomem stresu

Zbiór danych posiada 630 unikatowych próbek oraz składa się z poszczególnych cech:

- Pomiar chrapania - ilość wykonywania czynności chrapania podczas snu, im większe tym gorsze,
- Pomiar oddechu - ilość wdychanego oraz wydychanego powietrza podczas snu, im większe tym gorsze,
- Temperatura ciała - wysokość temperatury ciała podczas snu, im mniejsze tym gorsze,
- Ruch kończyn - ilość samowolnych ruchów części ciała podczas snu, im większe tym gorsze,
- Pomiar tlenu we krwi - ilość komórek tlenu we krwi, im mniejsze tym gorsze,

- Ruch gałki ocznej - ilość samowolnego poruszania się gałek ocznych podczas snu, im większe tym gorsze,
- Ilość snu - ilość godzin spędzonych podczas snu, im mniejsze tym gorsze,
- Tętno pracy serca - szybkość pracy serca podczas snu, im większe tym gorsze,
- Poziom stresu - ogólny wyznacznik poziomu stresu na podstawie powyższych zmiennych, wyznacza się poprzez wystawienia jednej liczby z zakresu od 0 do 4, gdzie 0 to najlepszy wynik – minimalny poziom stresu, a 4 to najgorszy – maksymalny poziom stresu.

```
dataset = pd.read_csv('SaYoPillow.csv')
dataset
```

	snoring	respiration	temperature	limb	oxygen	eye	hours	heart	stress
0	93.800	25.680	91.840	16.600	89.840	99.60	1.840	74.20	3
1	91.640	25.104	91.552	15.880	89.552	98.88	1.552	72.76	3
2	60.000	20.000	96.000	10.000	95.000	85.00	7.000	60.00	1
3	85.760	23.536	90.768	13.920	88.768	96.92	0.768	68.84	3
4	48.120	17.248	97.872	6.496	96.248	72.48	8.248	53.12	0
...	...	...	...	...	...	...	...	...	...
625	69.600	20.960	92.960	10.960	90.960	89.80	3.440	62.40	2
626	48.440	17.376	98.064	6.752	96.376	73.76	8.376	53.44	0
627	97.504	27.504	86.880	17.752	84.256	101.88	0.000	78.76	4
628	58.640	19.728	95.728	9.728	94.592	84.32	6.728	59.32	1
629	73.920	21.392	93.392	11.392	91.392	91.96	4.088	63.48	2

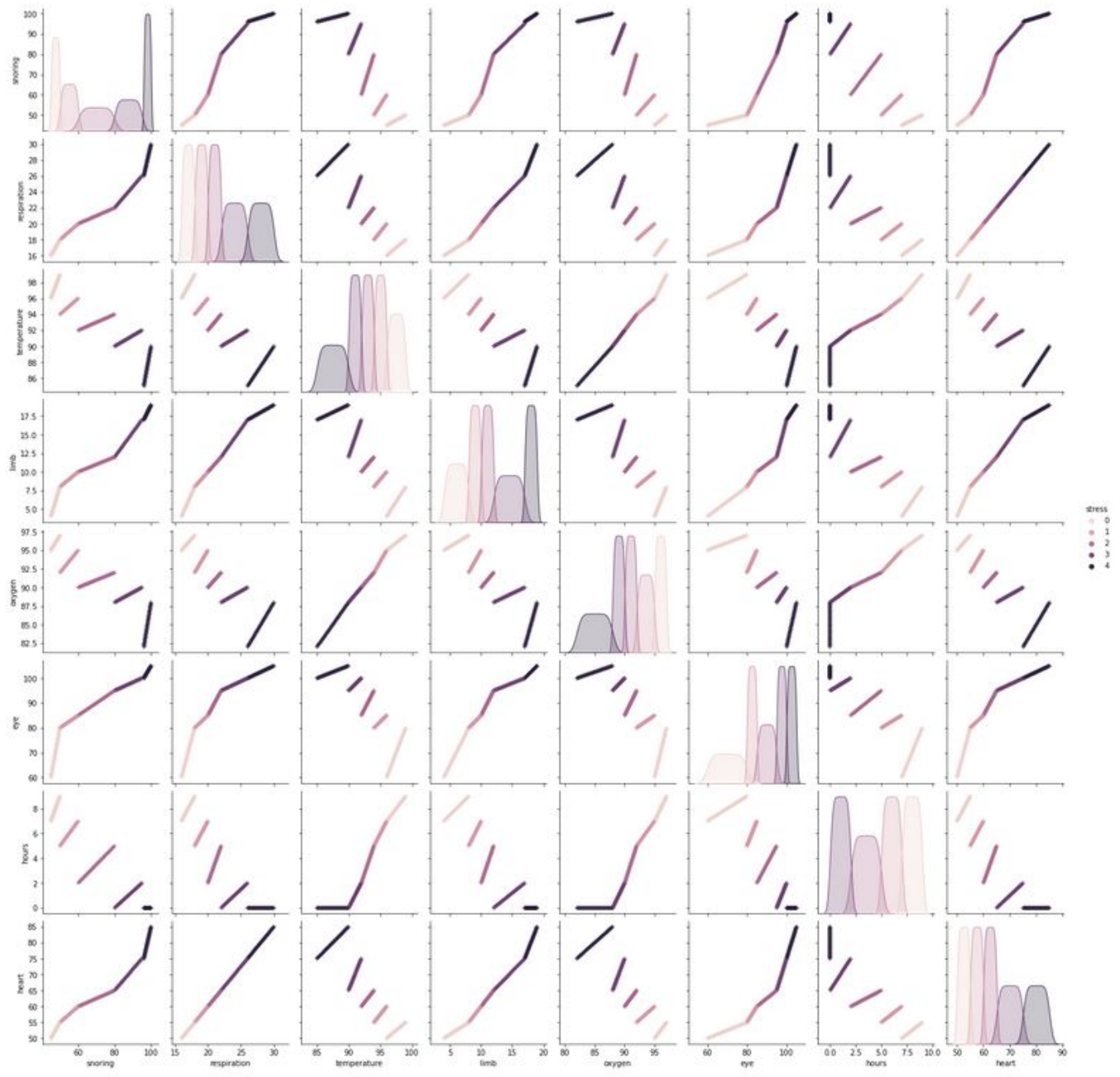
630 rows × 9 columns

Rysunek 4: Przykładowe dane w bazie

Jak widać powyżej dane są już potasowane, więc użycie metody tasowania nie jest tu potrzebne. Za to potrzebują one jak najbardziej normalizacji do lepszego i łatwiejszego odczytu.

Natomiast wykresy poniżej to wynik użycia funkcji 'pairplot'. Wykreśla on relacje parami w zestawie danych. Domyślnie ta funkcja tworzy siatkę osi w taki sposób, że każda zmienna liczbowa w danych będzie współdzielona przez osie y w jednym wierszu i osie x w jednej kolumnie. Wykresy ukośne są traktowane inaczej: wykres rozkładu jednowymiarowego jest rysowany, aby pokazać marginalny rozkład danych w każdej kolumnie.

Dzięki temu możemy zauważyć dużą kompatybilność danych między sobą oraz możemy wysnuć dobre wyniki względem dalszych etapów tego projektu.



Rysunek 5: Pairplot

6 Implementacja

6.1 Wspólne

W implementacji algorytmów przygotowujących zbiór danych do klasyfikacji stworzono klasę *DataProcessing* zawierającą wszystkie przydatne do tego metody:

- metoda *shuffle()* przyjmująca jako argument zbiór danych. Zamienia ona pozycje w zbiorze miejscami przy użyciu generatora liczb pseudolosowych i zwraca przetasowany

zbiór.

- metoda *splitSet()* dzieląca zbiór na zbiory treningowy i testowy według ustalonej z góry proporcji
- metoda *normalize()* normalizująca wartości w zbiorze zgodnie z algorytmem 1.

Listing 2: metoda *normalize()*

```
def normalize(x):
    x = x.copy()
    values = x.select_dtypes(exclude="object")
    columnNames = values.columns.tolist()
    columnNames.remove("stress")

    # znalezienie wartosci skrajnych w kolumnie
    for column in columnNames:
        data = x.loc[:,column]
        max1 = max(data)
        min1 = min(data)

        # normalizacja wartosci w kolumnie
        for row in range(len(x)):
            new Value=((x.at[row,column]-min1)/(max1-min1))
            x.at[row, column] = newValue
    return x
```

Do importu oraz analizy zbioru użyto metod biblioteki *pandas* oraz *seaborn*. Wykorzystano także bibliotekę *SKLearn* do wyznaczenia wskaźników wydajności oraz zaimplementowanych w niej klasyfikatorów w celu porównania ich z własnymi implementacjami.

6.2 algorytm KNN

W ramach implementacji stoworzono klasę *KNN* zawerającą metody obliczające Odległość Minkowskiego oraz klasyfikujące.

- metoda *metric()* – oblicza Odległość Minkowskiego dla dwóch zadanych wektorów. W zależności od parametru m otrzymywana jest odległość odpowiedniego typu.

Listing 3: metoda *metric()*

```
def metric(v1, v2, m):
    tmp = 0
    for i in range(len(v1)-1):
        tmp += abs(v1[i] - v2[i])**m
    return tmp**(1/m)
```

- metoda *classify()* – klasyfikuje zadaną próbkę w oparciu o zbiór treningowy i podane klasy, liczbę sąsiadów i typ metryki zgodnie z algorytmem 3

Listing 4: metoda *classify()* w KNN

```
def classify(sample, X, C, k, m):
```

```
    # utworzenie slownika na podstawie nazw klas
    classes = {}
    for cls in C:
        classes[cls] = 0

    # obliczenie odleglosci probki od kazdego rekordu w zbiorze
    distances = []
    for i in range(len(X)):
        distances.append(KNN.metric(sample, X.iloc[i], m))

    # sortowanie (stogowe) zbioru wzgledem odleglosci
    heap = []
    for i in range(len(X)):
        heappush(heap, (distances[i], X.iloc[i].stress))

    # glosowanie
    for i in range(0, k):
        classes[heappop(heap)[1]] += 1

    return max(classes, key = classes.get)
```

6.3 naiwny klasyfikator Bayesa

Do obliczeń w tym klasyfikatorze utworzono klasę *NaiveBayes* posiadającą metody:

- *mean()* przyjmująca pewne dane ze zbioru danych i obliczająca średnią arytmetyczną
- *std()* przyjmująca pewne dane ze zbioru danych i obliczająca odchylenie standardowe
- *fun()* przyjmująca argumenty ze zbioru danych oraz wcześniej obliczoną średnią jak także odchylenie standardowe i obliczająca klasyfikację z wybranego wzoru
- *classify()* klasyfikująca wartości w zbiorze względem algorytmu 5

Listing 5: metoda *classify()* w Bayesie

```
def classify(train,sample):
#separacja klas z bazy X
    names = train.stress.unique()
    classes = []
    for name in names:
        classes += [train[train['stress'] == name]]
        del classes[-1]['stress']
    #obl sred i odch dla kazdego atrybutu i klasy
    #obl skladowych prawdopodobienstw
    classes_fun = []
    for classy in classes:
        attrs_mean = []
        attrs_std = []
        attrs_fun = []
        for (name, data) in classy.iteritems():
            attrs_mean += [NaiveBayes.mean(data.values)]
            attrs_std += [NaiveBayes.std(data.values)]
```

```
18                 attrs_fun += [NaiveBayes.fun(sample[name],attrs_mean
                       [-1],attrs_std[-1])]
19             classes_fun += [np.prod(attrs_fun)]
20         return names[classes_fun.index(max(classes_fun))]
```

6.4 Perceptron

Do implementacji Perceptronu utworzono klasę *Perceptron* posiadającą metody:

- *konstruktor* tworzy nowy perceptron, przyjmujący liczbę odpowiadającą za szybkość uczenia się, domyślnie przyjmuje 0.1
- metoda *f()* przyjmująca wektor cech, mapuje wektor na wartość skalarną
- metoda *predict()* przyjmuje wektor cech i wykorzystuje metodę *f()*, konwertuje wartość skalarną na wyjście binarne
- metoda *fit()* dopasowuje model Perceptronu do treningowego zbioru danych

Listing 6: metoda *fit()*

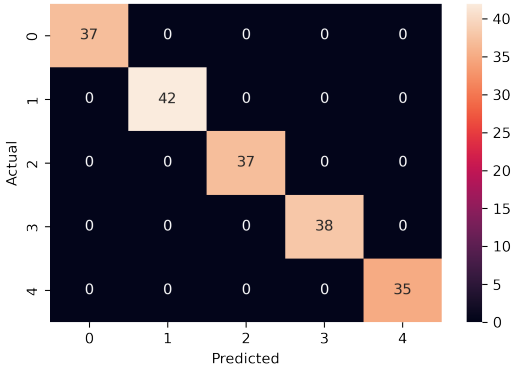
```
1  def fit(self, x: np.array, y: np.array, n_iter=100):
2      self._b = 0.0
3      self._w = np.zeros(x.shape[1])
4      self.misclassified_samples = []
5
6      for _ in range(n_iter):
7          # licznik bledow w aktualnej iteracji
8          errors = 0
9          for xi, yi in zip(x, y):
10         # dla kazdej probki obliczana wartosc aktualizacji
11             update = self.learning_rate * (yi - self.predict(xi))
12             # dodanie jej do przesuniecia b oraz do tablicy wag
13             self._b += update
14             self._w += update * xi
15             errors += int(update != 0.0)
16
17         self.misclassified_samples.append(errors)
```

7 Testy

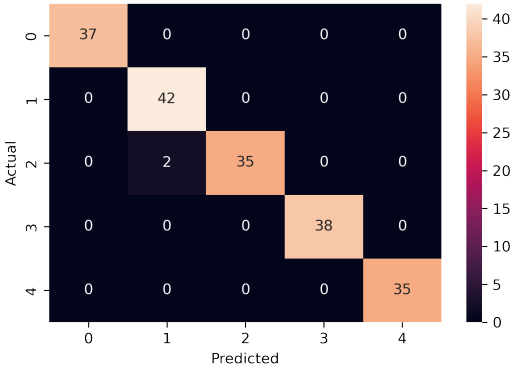
7.1 algorytm KNN

Przeprowadzono testy dla dwóch metod obliczania odległości (dla $p = 1$ oraz $p = 2$) oraz kilku wartości k. Wykorzystano zarówno zbiór znormalizowany, jak i nieznormalizowany.

Standardowo przyjmwaną liczbą sąsiadów k jest 5, jednak dla tej liczby algorytm okazał się mieć 100% dokładność (Rysunek 6) niezależnie od pozostałych parametrów, dlatego przetestowano także większe wartości. Dopiero dla liczby sąsiadów 100 klasyfikacja nieznormalizowanego zbioru przestaje dawać 100% dokładność, jednak nadal są to róznice rzędu 1%.



(a) $k = 5$



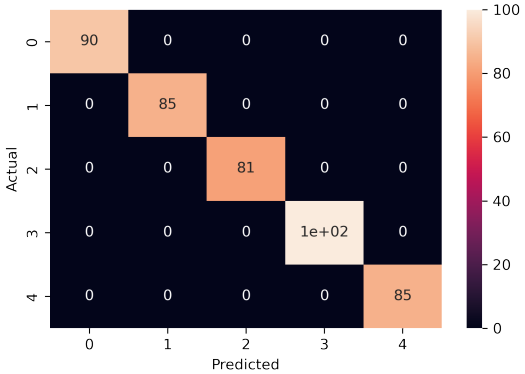
(b) $k = 100$, zbiór nieznormalizowany

Rysunek 6: Macierze błędów dla podziału zbioru 70 : 30

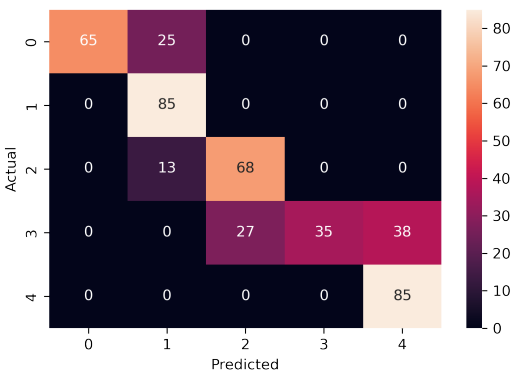
Metryki dla podziału 70 : 30, $k = 100$, zbioru nieznormalizowanego:

```
 Accuracy: 0.989
Precision: 0.990
   Recall: 0.989
       F1: 0.989
```

Ze względu na wysoką dokładność przeprowadzono także testy dla niestandardowego podziału zbioru – 30 : 70. W tym przypadku, ze względu na znacząco mniejszy zbiór treningowy jakość wyników spada znacznie szybciej, jednak nadal dla 5 sąsiadów klasyfikacja okazuje się bezbłędna.



(a) $k = 5$

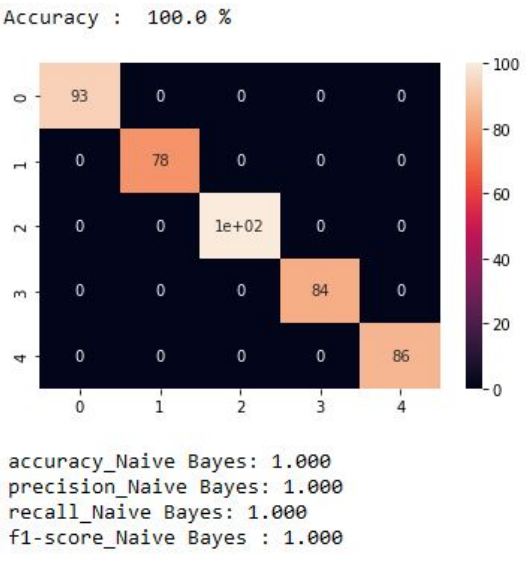


(b) $k = 60$

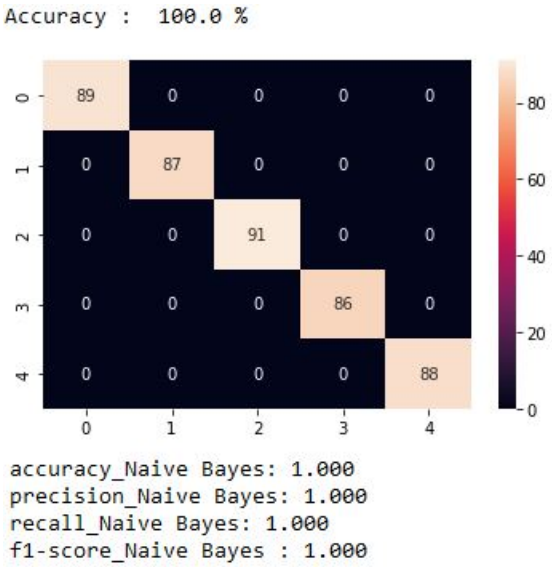
Rysunek 7: Macierze błędów dla podziału zbioru 30 : 70

W przypadku dużej liczby sąsidów wydajność spada, jednak otzymywane wyniki nie różnią się od siebie znacząco bez względu na użyty algorytm, metrykę, czy normalizację wzoru.

7.2 naiwny klasyfikator Bayesa

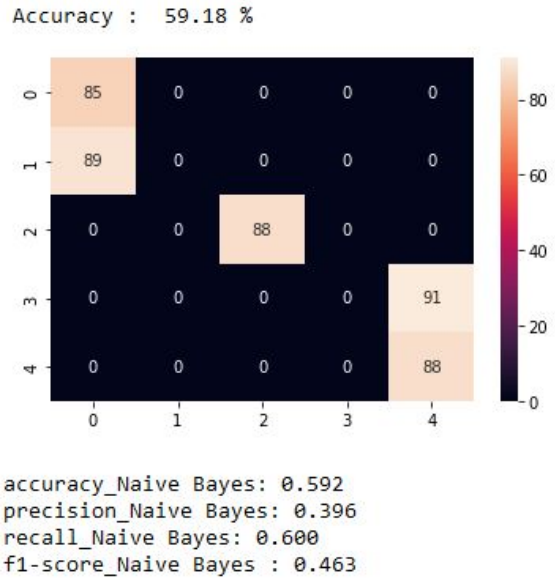


(a) Bayes przy użyciu Gaussa znormalizowanego

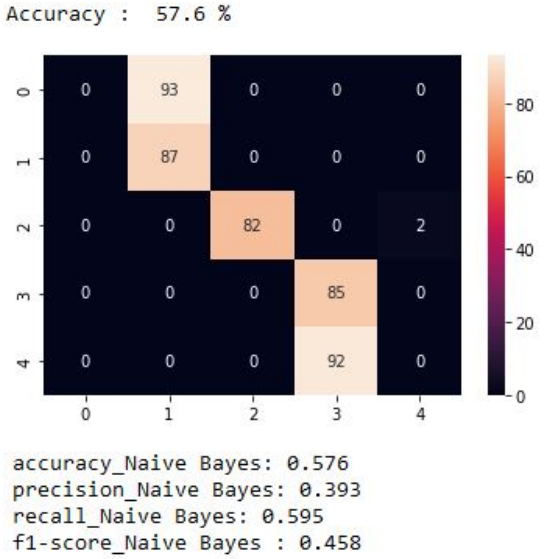


(b) Bayes przy użyciu Gaussa nieznormalizowanego

Rysunek 8: Bayes z algorytmem Gaussa



(a) Bayes przy użyciu Bernolliego znormalizowanego



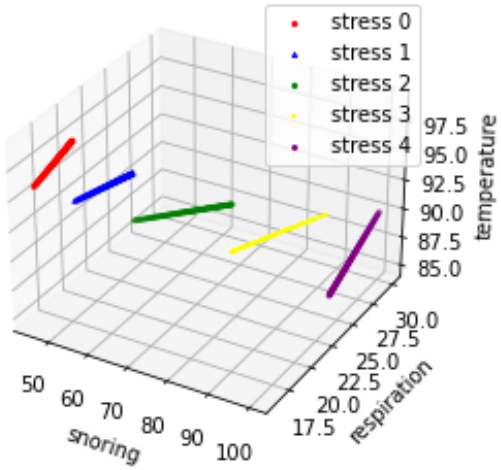
(b) Bayes przy użyciu Bernolliego nieznormalizowanego

Rysunek 9: Bayes z algorytmem Bernolliego

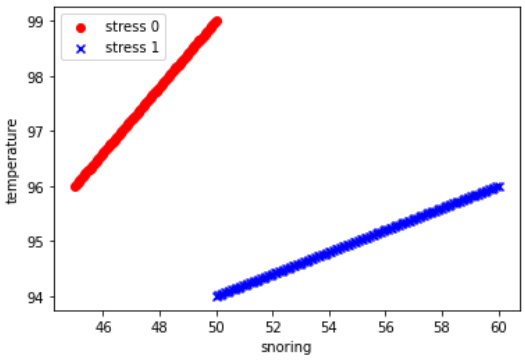
Jak możemy zauważyć przy użyciu algorytmu Gaussa wyniki otrzymujemy perfekcyjne z wynikiem 100%. Jednak podczas użycia Bernolliego wyniki te spadają o niemalże połowę. Dowodzi to temu, iż lepszy oraz skuteczniejszy w przypadku tej bazy będzie wzór Gaussa w klasyfikatorze Bayesa.

7.3 Perceptron

Przeprowadzono testy na próbce wybranej z przykładowego wykresu



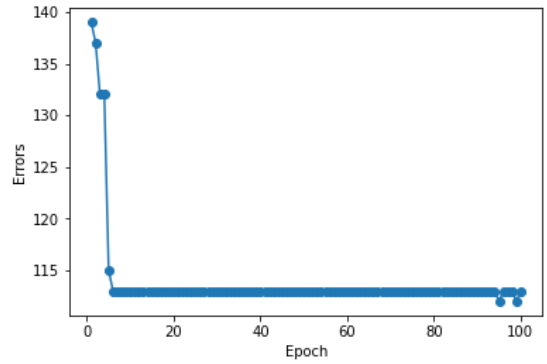
(a) Podgląd rozkładu danych na podstawie trzech klas



(b) Ograniczenie wykresu do dwóch klas

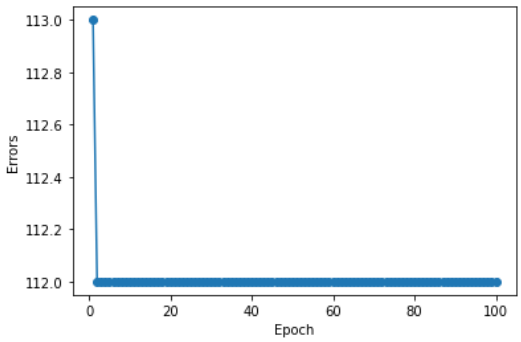
Rysunek 10: Wybór próbki

Trenowanie modelu:



accuracy 0.631579

(a) liczba wystąpień błędów przed standaryzacją



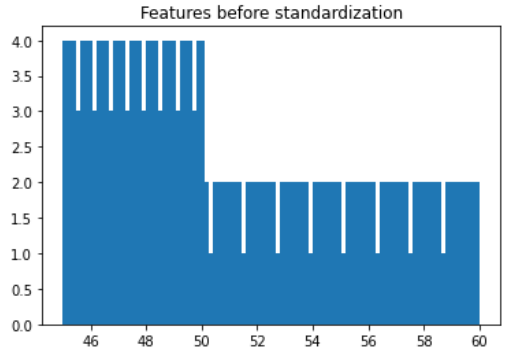
accuracy 0.631579

(b) liczba wystąpień błędów po standaryzacji

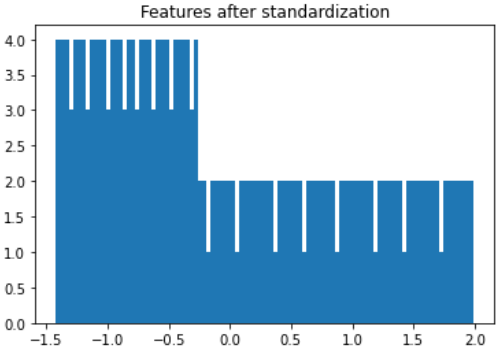
Rysunek 11: Ilość błędów w trenowanym modelu

Pomimo standaryzacji, wykresy różnią się od siebie w głównej mierze zakresem występowanych wartości. W obu przypadkach dokładność modelu jest taka sama. Prawdopodobnie jest to spowodowane zbyt małą próbką danych.

Na podstawie poniższych obrazów widać, że najoptymalniejszym podziałem danych wejściowych jest 85:15. Przetestowano wpływ zwiększenia ilości iteracji oraz wpływu różnych prędkości uczenia. W obu przypadkach wpływ był niezauważalny na rezultat końcowy, co podtrzymuje teorię o zbyt małej próbce danych, jako że jest to jedyny element którego nie dało się zwiększyć.

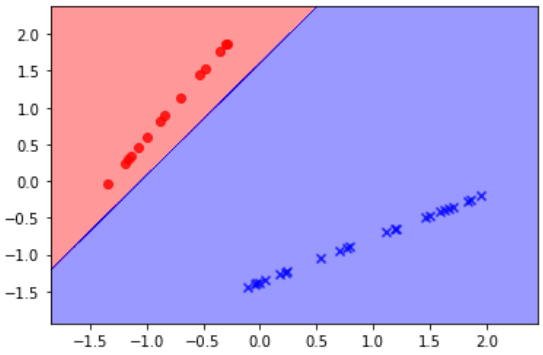


(a) klasy przed standaryzacją

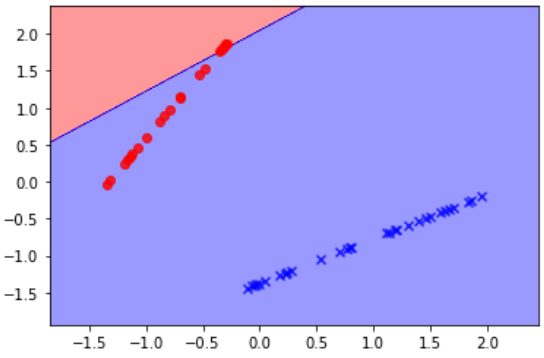


(b) klasy po standaryzacji

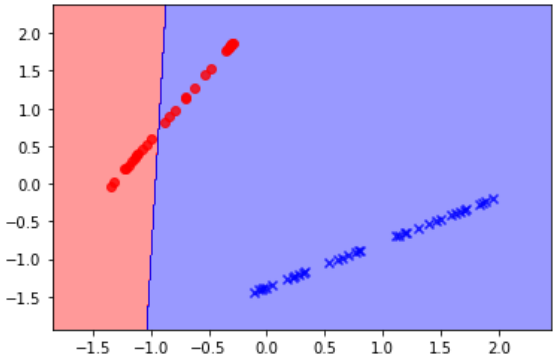
Rysunek 12: Skalowanie modelu



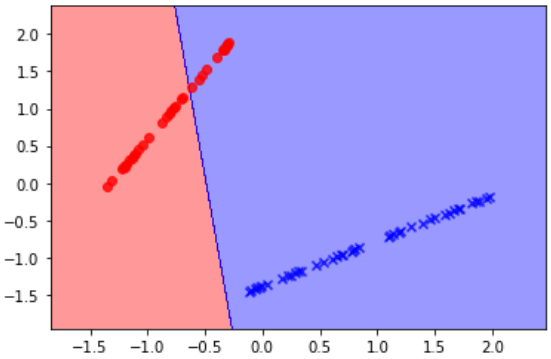
(a) Perceptron przy splicie 85:15



(b) Perceptron przy splicie 80:20



(c) Perceptron przy splicie 75:25



(d) Perceptron przy splicie 70:30

Rysunek 13: Wizualizacja przewidywań modelu

8 Eksperymenty

8.1 algorytm KNN

W Tabeli 2 przedstawiono wskaźniki wydajności uzyskane przy podziale zbioru w niestandardowej proporcji 30 : 70 oraz przy użyciu metryki euklidesowej.

	nNorm	norm	SKL nNorm	SKL norm
	$k = 5$			
Accuracy	1.00	1.00	1.00	1.00
Precision	1.00	1.00	1.00	1.00
Recall	1.00	1.00	1.00	1.00
F1	1.00	1.00	1.00	1.00
	$k = 20$			
Accuracy	0.984	1.000	0.982	1.000
Precision	0.985	1.000	0.983	1.000
Recall	0.984	1.000	0.982	1.000
F1	0.984	1.000	0.982	1.000
	$k = 40$			
Accuracy	0.930	1.000	0.930	0.995
Precision	0.937	1.000	0.937	0.995
Recall	0.930	1.000	0.930	0.995
F1	0.929	1.000	0.929	0.995
	$k = 60$			
Accuracy	0.766	0.787	0.782	0.814
Precision	0.829	0.855	0.837	0.879
Recall	0.766	0.787	0.782	0.814
F1	0.746	0.742	0.760	0.769

Tablica 2: wskaźniki wydajności algorytmu KNN

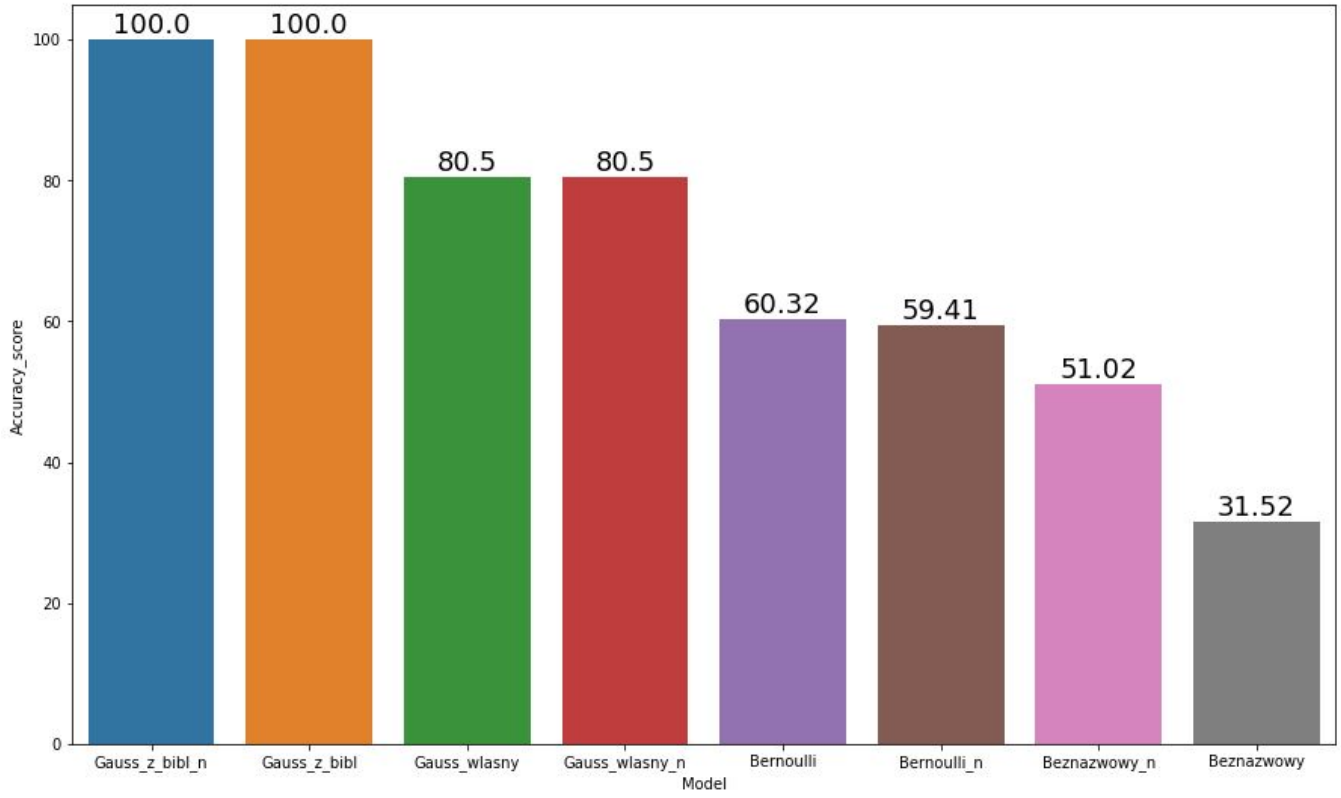
Wdajność algorytmów przy danej liczbie sąsiadów nie różni się znacząco. Różnice na poziomie 0.2% można uznać za pomijanie małe.

Przy porównaniu klasyfikacji zbbioru nieznormalizowanego ze znormalizowanym, algorytm zawsze lepiej radził sobie z danymi znormalizowanymi, różnice na podobnym poziomie ok. 2%.

Otrzymane wyniki świadczą o bardzo dużej przydatności algorytmu KNN do klasyfikacji próbek w użytym zbiorze. Na ich otrzymanie z pewnością wpływ ma rozkład danych, które ostro przechodzą z jednej kalsy w drugą, co ułatwia prawidłową klasyfikację.

Zauważalną różnicą własnej implementacji od gotowego rozwiązania jest czas trwania obliczeń przy własnym algorytmie wynoszący nawet kilkanaście sekund, podczas gdy algorytm biblioteki *SKLearn* kończy pracę po ok. sekundzie.

8.2 naiwny klasyfikator Bayesa



Rysunek 14: Wyniki zbiorcze wielu metod Bayesa

Do wykonania powyższego wykresu zostały użyte 4 różne wzory do obliczenia klasyfikacji Bayesa: Gauss własnoręcznie zaimplementowany, Gauss zapożyczony z zewnętrznej biblioteki, Bernolli również zapożyczony z zewnętrznej biblioteki oraz ostatni bez nadanej mu nazwy (w tym przypadku nazwałem go "Beznazwowy"). Każdy wzór był poddany danymi znormalizowanymi oraz nieznormalizowanymi (znormalizowane mają dopisek '_n' w nazwie.

Owe wyniki jasno sugerują, które metody działają najlepiej w tej konkretnej bazie danych. Oczywiście nie można brać pod poważną uwagę pierwszych dwóch wyników, gdyż są one zbyt "idealne". Jak wiadomo jest bardzo mała szansa, że dostaniem maksymalną precyzję pomiarów danych, lecz to nie uświadamia o fałszywości owych wyników.

Dalsze oceny precyzji są już na zadowalającym poziomie dla prawidłowej analizy. Można zauważyć, że poza jednym przypadkiem znormalizowane oraz nieznormalizowane dane praktycznie się od siebie nie różnią. Może to wynikać z dobrej jakości danych w samej bazie.

Część III

9 Pełny kod programu

9.1 Algorytm KNN

Listing 7: Kod algorytmu KNN

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
import random as rd
import math
import seaborn as sns
from matplotlib import pyplot
from heapq import heappush, heappop


# In[2]:


pillow = pd.read_csv("SaYoPillow.csv")
pillow.head()


# In[3]:


# sns.pairplot(pillow, hue='stress', markers='+')


# In[4]:


pillow.describe()


# In[5]:


class DataProcessing:
    @staticmethod
    def shuffle(x): # metoda tasujaca zbior
        x=x.copy()
        for i in range(len(x)-1):
            j=rd.randint(0,i)
            x.iloc[i], x.iloc[j]=x.iloc[j], x.iloc[i]
        return x

```

```
47      @staticmethod
48      def splitSet(x): # metoda dzielaca zbior
49          x=x.copy()
50          n=int(len(x)*0.7)
51          xTrain=x[:n]
52          xVal=x[n:]
53          return xTrain, xVal
54
55      @staticmethod
56      def normalize(x): # metoda normalizujaca dane
57          x=x.copy()
58          values=x.select_dtypes(exclude="object")
59          columnNames=values.columns.tolist()
60          columnNames.remove("stress")
61
62          # znalezienie wartosci skrajnych w kolumnie
63          for column in columnNames:
64              data=x.loc[:,column]
65              max1=max(data)
66              min1=min(data)
67
68              # normalizacja wartosci w kolumnie
69              for row in range(len(x)):
70                  newValue=((x.at[row,column]-min1)/(max1-min1))
71                  x.at[row, column]=newValue
72          return x
73
74
75  # In[6]:
76
77
78  pillow.head()
79
80
81  # In[7]:
82
83
84  # pillow = DataProcessing.shuffle(pillow)
85
86
87  # In[8]:
88
89
90  pillow.head()
91
92
93  # In[9]:
94
95
96  pillowNorm=DataProcessing.normalize(pillow)
97
98
99  # In[10]:
100
101
```

```
102 pillowNorm.head()
103
104
105 # In[11]:
106
107
108 pillowTrain, pillowTest = DataProcessing.splitSet(pillow)
109
110
111 # In[12]:
112
113
114 pillowNormTrain, pillowNormTest = DataProcessing.splitSet(pillowNorm)
115
116
117 # In[13]:
118
119
120 pillowTrain.head(),pillowTrain.head()
121
122
123 # In[14]:
124
125
126 pillowNorm.describe()
127
128
129 # In[15]:
130
131
132 class KNN:
133     @staticmethod
134     def metric(v1, v2, m): # metoda obliczaj[U+FFFD]ca odleglosc
135         tmp = 0
136         for i in range(len(v1)-1):
137             tmp += abs(v1[i] - v2[i])**m
138         return tmp**(1/m)
139
140     @staticmethod
141     def classify(sample, X, C, k, m):
142         # utworzenie slownika na podstawie nazw klas
143         classes = {}
144         for cls in C:
145             classes[cls] = 0
146
147         # obliczenie odleglo[U+FFFD]ci probki od kazdego rekordu w zbiorze
148         distances = []
149         for i in range(len(X)):
150             distances.append(KNN.metric(sample, X.iloc[i], m))
151
152         # sortowanie (stogowe) zbioru wzgledem odleglosci
153         heap = []
154         for i in range(len(X)):
155             heappush(heap, (distances[i], X.iloc[i].stress))
156
```

```
157         # g[U+FFFD]osowanie
158         for i in range(0, k):
159             classes[heappop(heap)[1]] += 1
160 
161         return max(classes, key = classes.get)
162 
163 
164 # In[16]:
165 
166 
167 NEIGHBORS_NUMBER = 100
168 DISTANCE_PARAMETER = 1
169 
170 
171 # In[17]:
172 
173 
174 actual = pillowTest["stress"].copy()
175 predicted = [] # wyniki klasyfikacji nieznormalizowanego zbioru
    testowego
176 for i in range(len(pillowTest)):
177     predicted.append(KNN.classify(pillowTest.iloc[i], pillowTrain,
        pillowTrain["stress"], NEIGHBORS_NUMBER, DISTANCE_PARAMETER))
178 
179 
180 # In[18]:
181 
182 
183 data = {'y_Actual':    actual,
184         'y_Predicted': predicted
185         }
186 
187 df = pd.DataFrame(data, columns=['y_Actual','y_Predicted'])
188 confusion_matrix = pd.crosstab(df['y_Actual'], df['y_Predicted'],
    rownames=['Actual'], colnames=['Predicted'])
189 print (confusion_matrix)
190 plot = sns.heatmap(confusion_matrix, annot=True)
191 fig = plot.get_figure()
192 fig.savefig('KNNplot.png', dpi=300)
193 
194 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
195 accuracy = accuracy_score(actual, predicted)
196 precision = precision_score(actual, predicted, average="weighted")
197 recall = recall_score(actual, predicted, average="weighted")
198 f1 = f1_score(actual, predicted, average="weighted")
199 
200 print(" Accuracy: %.3f" %accuracy)
201 print("Precision: %.3f" %precision)
202 print("   Recall: %.3f" %recall)
203 print("       F1: %.3f" %f1)
204 
205 
206 # In[19]:
207 
```

```
208
209 actualN = pillowNormTest["stress"].copy()
210 predictedN = [] # wyniki klasyfikacji znormalizowanego zbioru testowego
211 for i in range(len(pillowTest)):
212     predictedN.append(KNN.classify(pillowNormTest.iloc[i],
          pillowNormTrain, pillowNormTrain["stress"], NEIGHBORS_NUMBER,
          DISTANCE_PARAMETER))
213
214
215 # In[20]:
216
217
218 dataN = {'y_Actual':    actualN,
219         'y_Predicted': predictedN
220         }
221
222 dfN = pd.DataFrame(dataN, columns=['y_Actual','y_Predicted'])
223 confusion_matrixN = pd.crosstab(dfN['y_Actual'], dfN['y_Predicted'],
      rownames=['Actual'], colnames=['Predicted'])
224 print (confusion_matrixN)
225 plotN = sns.heatmap(confusion_matrixN, annot=True)
226 figN = plotN.get_figure()
227 figN.savefig('KNNplotNorm.png', dpi=300)
228
229 from sklearn.metrics import accuracy_score, precision_score,
      recall_score, f1_score
230 accuracyN = accuracy_score(actualN, predictedN)
231 precisionN = precision_score(actualN, predictedN, average="weighted")
232 recallN = recall_score(actualN, predictedN, average="weighted")
233 f1N = f1_score(actualN, predictedN, average="weighted")
234
235 print(" Accuracy: %.3f" %accuracyN)
236 print("Precision: %.3f" %precisionN)
237 print("   Recall: %.3f" %recallN)
238 print("       F1: %.3f" %f1N)
239
240
241 # In[21]:
242
243
244 #
245 # KNN from sklearn
246 #
247
248
249 # In[22]:
250
251
252 pillowTrainX = pillowTrain.iloc[:, :-1].values
253 pillowTrainY = pillowTrain.iloc[:, 8].values
254
255 pillowTestX = pillowTest.iloc[:, :-1].values
256 pillowTestY = pillowTest.iloc[:, 8].values
257
258 pillowNormTrainX = pillowNormTrain.iloc[:, :-1].values
```

```
pillowNormTrainY = pillowNormTrain.iloc[:, 8].values

pillowNormTestX = pillowNormTest.iloc[:, :-1].values
pillowNormTestY = pillowNormTest.iloc[:, 8].values


# In[23]:


from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = NEIGHBORS_NUMBER, p =
    DISTANCE_PARAMETER)
classifier.fit(pillowTrainX, pillowTrainY)

y_pred = classifier.predict(pillowTestX)


# In[24]:


dataSK = {'y_Actual':  pillowTestY,
        'y_Predicted': y_pred
        }

dfSK = pd.DataFrame(dataSK, columns=['y_Actual','y_Predicted'])
confusion_matrixSK = pd.crosstab(dfSK['y_Actual'], dfSK['y_Predicted'],
    rownames=['Actual'], colnames=['Predicted'])
print (confusion_matrixSK)
plotSK = sns.heatmap(confusion_matrixSK, annot=True)
figSK = plotSK.get_figure()
figSK.savefig('KNNplot-SKLearn.png', dpi=300)

from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
accuracySK = accuracy_score(pillowTestY, y_pred)
precisionSK = precision_score(pillowTestY, y_pred, average="weighted")
recallSK = recall_score(pillowTestY, y_pred, average="weighted")
f1SK = f1_score(pillowTestY, y_pred, average="weighted")

print(" Accuracy: %.3f" %accuracySK)
print("Precision: %.3f" %precisionSK)
print("   Recall: %.3f" %recallSK)
print("       F1: %.3f" %f1SK)


# In[25]:


from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = NEIGHBORS_NUMBER, p =
    DISTANCE_PARAMETER)
classifier.fit(pillowNormTrainX, pillowNormTrainY)

y_pred_norm = classifier.predict(pillowNormTestX)

```

```
310
311 # In[26]:
312
313
314 dataSKN = {'y_Actual': pillowNormTestY,
315         'y_Predicted': y_pred_norm
316         }
317
318 dfSKN = pd.DataFrame(dataSKN, columns=['y_Actual','y_Predicted'])
319 confusion_matrixSKN = pd.crosstab(dfSKN['y_Actual'], dfSKN['y_Predicted'
    ], rownames=['Actual'], colnames=['Predicted'])
320 print (confusion_matrixSKN)
321 plotSKN = sns.heatmap(confusion_matrixSKN, annot=True)
322 figSKN = plotSKN.get_figure()
323 figSKN.savefig('KNNplotNorm-SKLearn.png', dpi=300)
324
325 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
326 accuracySKN = accuracy_score(pillowNormTestY, y_pred_norm)
327 precisionSKN = precision_score(pillowNormTestY, y_pred_norm, average="
    weighted")
328 recallSKN = recall_score(pillowNormTestY, y_pred_norm, average="weighted
    ")
329 f1SKN = f1_score(pillowNormTestY, y_pred_norm, average="weighted")
330
331 print(" Accuracy: %.3f" %accuracySKN)
332 print("Precision: %.3f" %precisionSKN)
333 print("   Recall: %.3f" %recallSKN)
334 print("       F1: %.3f" %f1SKN)
335
336
337 # In[ ]:
```

9.2 Naiwny klasyfkator Bayesa

Listing 8: Kod klasyfikatora Bayesa

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
4 # In[1]:
5
6
7 import numpy as np
8 import matplotlib.pyplot as plt
9 import pandas as pd
10 import seaborn as sns
11 import math
12
13
14 # In[47]:
15
16
17 dataset = pd.read_csv('SaYoPillow.csv')
18 dataset
```

```
19
20
21 # In[50]:
22
23
24 sns.pairplot(dataset, hue='stress', markers='+')
25
26
27 # In[3]:
28
29
30 dataset.describe()
31
32
33 # In[4]:
34
35
36 #$$ Gauss $$#
37
38
39 # In[5]:
40
41
42
43
44 #### ZNORMALIZOWANE ####
45
46
47 # In[6]:
48
49
50 class DataProcessing:
51     @staticmethod
52     def normalize(x):
53         x=x.copy()
54         values=x.select_dtypes(exclude="object")
55         columnNames=values.columns.tolist()
56         columnNames.remove("stress")
57         for column in columnNames:
58             data=x.loc[:,column]
59             max1=max(data)
60             min1=min(data)
61             for row in range(len(x)):
62                 newValue=((x.at[row,column]-min1)/(max1-min1))
63                 x.at[row, column]=newValue
64         return x
65
66     @staticmethod
67     def SplitData(X,x):
68         #y to x'
69         if x >= 10 or x < 0:
70             raise ValueError('Zły zakres (musi < 1)')
71         return X[:math.ceil(len(X)*x)],X[math.ceil(len(X)*(1-x)):]
72
73
```

```
# In[7]:


pillowNorm=DataProcessing.normalize(dataset)


# In[8]:


X = pillowNorm.iloc[:,:8].values
y = pillowNorm['stress'].values
pillowNorm.head()


# In[9]:


pillowNorm.describe()


# In[10]:


from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
    0.7)


# In[11]:


from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)


# In[12]:


from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)


# In[13]:


y_pred = classifier.predict(X_test)


# In[14]:


from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import accuracy_score
cm = confusion_matrix(y_test, y_pred)
accuracy1=round(accuracy_score(y_test, y_pred)*100 ,2)
print ("Accuracy : ", accuracy1, "%")
sns.heatmap(cm, annot=True)
plt.show()


# In[15]:


from sklearn.metrics import make_scorer,precision_score,recall_score,
    f1_score
accuracy = accuracy_score(y_test,y_pred)
precision =precision_score(y_test, y_pred,average='macro')
recall =  recall_score(y_test, y_pred,average='macro')
f1 = f1_score(y_test,y_pred,average='macro')
print('accuracy_Naive Bayes: %.3f' %accuracy)
print('precision_Naive Bayes: %.3f' %precision)
print('recall_Naive Bayes: %.3f' %recall)
print('f1-score_Naive Bayes : %.3f' %f1)


# In[16]:




#### NIEZNORMALIZOWANE ####


# In[17]:


X1 = dataset.iloc[:,:8].values
y1 = dataset['stress'].values
dataset.head()


# In[18]:


X_train, X_test, y_train, y_test = train_test_split(X1, y1, test_size =
    0.7)
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)


# In[19]:


cm = confusion_matrix(y_test, y_pred)
accuracy2=round(accuracy_score(y_test, y_pred)*100 ,2)
```

```
181 print ("Accuracy : ", accuracy2, "%")
182 sns.heatmap(cm, annot=True)
183 plt.show()
184
185
186 # In[20]:
187
188
189 accuracy = accuracy_score(y_test,y_pred)
190 precision =precision_score(y_test, y_pred,average='micro')
191 recall =  recall_score(y_test, y_pred,average='micro')
192 f1 = f1_score(y_test,y_pred,average='micro')
193 print('accuracy_Naive Bayes: %.3f' %accuracy)
194 print('precision_Naive Bayes: %.3f' %precision)
195 print('recall_Naive Bayes: %.3f' %recall)
196 print('f1-score_Naive Bayes : %.3f' %f1)
197
198
199 # In[21]:
200
201
202 #$$$$ Bernoulli $$$$#
203
204
205 # In[22]:
206
207
208
209
210 #### ZNORMALIZOWANE ####
211
212
213 # In[23]:
214
215
216 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
        0.7)
217 X_train = sc.fit_transform(X_train)
218 X_test = sc.transform(X_test)
219 from sklearn.naive_bayes import BernoulliNB
220 classifier = BernoulliNB()
221 classifier.fit(X_train, y_train)
222 y_pred = classifier.predict(X_test)
223
224
225 # In[24]:
226
227
228 cm = confusion_matrix(y_test, y_pred)
229 accuracy3=round(accuracy_score(y_test, y_pred)*100 ,2)
230 print ("Accuracy : ", accuracy3, "%")
231 sns.heatmap(cm, annot=True)
232 plt.show()
233
234
```

```
# In[45]:


accuracy = accuracy_score(y_test,y_pred)
precision =precision_score(y_test, y_pred,average='weighted')
recall =  recall_score(y_test, y_pred,average='weighted')
f1 = f1_score(y_test,y_pred,average='weighted')
print('accuracy_Naive Bayes: %.3f' %accuracy)
print('precision_Naive Bayes: %.3f' %precision)
print('recall_Naive Bayes: %.3f' %recall)
print('f1-score_Naive Bayes : %.3f' %f1)


# In[26]:




#### NIEZNORMALIZOWANE ####


# In[27]:


X_train, X_test, y_train, y_test = train_test_split(X1, y1, test_size =
    0.7)
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)


# In[28]:


cm = confusion_matrix(y_test, y_pred)
accuracy4=round(accuracy_score(y_test, y_pred)*100 ,2)
print ("Accuracy : ", accuracy4, "%")
sns.heatmap(cm, annot=True)
plt.show()


# In[46]:


accuracy = accuracy_score(y_test,y_pred)
precision =precision_score(y_test, y_pred,average='weighted')
recall =  recall_score(y_test, y_pred,average='weighted')
f1 = f1_score(y_test,y_pred,average='weighted')
print('accuracy_Naive Bayes: %.3f' %accuracy)
print('precision_Naive Bayes: %.3f' %precision)
print('recall_Naive Bayes: %.3f' %recall)
print('f1-score_Naive Bayes : %.3f' %f1)


```

```
# In[30]:


#### Bayes z kolowkium - nienazwany


# In[31]:



#### NIEZNORMALIZOWANE ####


# In[32]:


class NaiveBayes:
    #srednia
    @staticmethod
    def mean(attr):
        try:
            return sum(attr)/len(attr)
        except TypeError:
            print(attr)
            return sum(attr)/len(attr)

    #odch stand
    def std(attr):
        mean = NaiveBayes.mean(attr)
        sumelem = 0
        for i in attr:
            sumelem += (i-mean)**2
        return math.sqrt(sumelem/len(attr))

    #funkcja
    @staticmethod
    def fun(x, mean, std):
        if mean-math.sqrt(6)*std<=x and x<=std:
            a = x-mean/6*std**2 + 1/math.sqrt(6)*std
        elif mean<x and x<=mean+math.sqrt(6)*std:
            a = -(x-mean/6*std**2) + 1/math.sqrt(6)*std
        else:
            a=0
        return a

    #klasyfikacja
    def classify(train,sample):
        #separacja klas z bazy X
        names = train.stress.unique()
        classes = []
        for name in names:
            classes += [train[train['stress'] == name]]
            del classes[-1]['stress']
        #obl sred i odch dla kazdego atrybutu i klasy
```

```
344         #obl skladowych prawdopodobienstw
345         classes_fun = []
346         for classy in classes:
347             attrs_mean = []
348             attrs_std = []
349             attrs_fun = []
350             for (name, data) in classy.iteritems():
351                 attrs_mean += [NaiveBayes.mean(data.values)]
352                 attrs_std += [NaiveBayes.std(data.values)]
353                 attrs_fun += [NaiveBayes.fun(sample[name],attrs_mean
                        [-1],attrs_std[-1])]
354             classes_fun += [np.prod(attrs_fun)]
355         return names[classes_fun.index(max(classes_fun))]
356
357
358 # In[33]:
359
360
361 dataset = pd.read_csv('SaYoPillow.csv')
362 X_train, X_test = DataProcessing.SplitData(dataset,0.7)
363 correct = 0
364 for i in range(0,len(X_test)):
365     sample = X_test.iloc[i].drop('stress').to_dict()
366     if X_test.iloc[i].stress == NaiveBayes.classify(X_train,sample):
367         correct += 1
368 accuracy = correct/len(X_train.index)*100
369 accuracy5=round(accuracy,2)
370 print("Accuracy (regul) -",accuracy5,"%")
371
372
373 # In[34]:
374
375
376
377
378 #### ZNORMALIZOWANE ####
379
380
381 # In[35]:
382
383
384 dataset = pd.read_csv('SaYoPillow.csv')
385 datasetnorm = DataProcessing.normalize(dataset)
386 X_train, X_test = DataProcessing.SplitData(datasetnorm,0.7)
387 correct = 0
388 for i in range(0,len(X_test)):
389     sample = X_test.iloc[i].drop('stress').to_dict()
390     if X_test.iloc[i].stress == NaiveBayes.classify(X_train,sample):
391         correct += 1
392 accuracy = correct/len(X_train.index)*100
393 accuracy6=round(accuracy,2)
394 print("Accuracy (regul) -",accuracy6,"%")
395
396
397 # In[36]:
```

```
398
399
400 # Bayes zaimplementowany
401
402
403 # In[37]:
404
405
406
407
408 #### NIEZNORMALIZOWANE ####
409
410
411 # In[38]:
412
413
414 class NaiveBayes2:
415     @staticmethod
416     def mean(attr):
417         try:
418             return sum(attr)/len(attr)
419         except TypeError:
420             print(attr)
421             return sum(attr)/len(attr)
422
423     def std(attr):
424         mean = NaiveBayes.mean(attr)
425         sumelem = 0
426         for i in attr:
427             sumelem += (i-mean)**2
428         return math.sqrt(sumelem/len(attr))
429
430     @staticmethod
431     def fun2(x, mean, std):
432         exponet = np.exp(-(x - mean) ** 2 / (2 * std ** 2))
433         return 1 / np.sqrt(2 * np.pi * std ** 2) * exponet
434
435     def classify(train,sample):
436         names = train.stress.unique()
437         classes = []
438         for name in names:
439             classes += [train[train['stress'] == name]]
440             del classes[-1]['stress']
441
442         classes_fun = []
443         for classy in classes:
444             attrs_mean = []
445             attrs_std = []
446             attrs_fun = []
447             for (name, data) in classy.iteritems():
448                 attrs_mean += [NaiveBayes2.mean(data.values)]
449                 attrs_std += [NaiveBayes2.std(data.values)]
450                 attrs_fun += [NaiveBayes2.fun2(sample[name],attrs_mean
                        [-1],attrs_std[-1])]
451             classes_fun += [np.prod(attrs_fun)]
```

```
        return names[classes_fun.index(max(classes_fun))]


# In[39]:


dataset = pd.read_csv('SaYoPillow.csv')
X_train, X_test = DataProcessing.SplitData(dataset,0.7)
correct = 0
for i in range(0,len(X_test)):
    sample = X_test.iloc[i].drop('stress').to_dict()
    if X_test.iloc[i].stress == NaiveBayes2.classify(X_train,sample):
        correct += 1
accuracy = correct/len(X_train.index)*100
accuracy7=round(accuracy,2)
print("Accuracy (regul) -",accuracy7,"%")


# In[40]:



#### ZNORMALIZOWANE ####


# In[41]:


dataset = pd.read_csv('SaYoPillow.csv')
datasetnorm = DataProcessing.normalize(dataset)
X_train, X_test = DataProcessing.SplitData(datasetnorm,0.7)
correct = 0
for i in range(0,len(X_test)):
    sample = X_test.iloc[i].drop('stress').to_dict()
    if X_test.iloc[i].stress == NaiveBayes2.classify(X_train,sample):
        correct += 1
accuracy = correct/len(X_train.index)*100
accuracy8=round(accuracy,2)
print("Accuracy (regul) -",accuracy8,"%")


# In[42]:


# Porownywanie modeli


# In[43]:


results = pd.DataFrame({
   'Model': [ 'Gauss_z_bibl_n',
              'Gauss_z_bibl',
              'Bernoulli_n',
```

```
                'Bernoulli',
                'Beznazwowy',
                'Beznazwowy_n',
                'Gauss_wlasny',
                'Gauss_wlasny_n'],
    "Accuracy_score":[accuracy1,
                      accuracy2,
                      accuracy3,
                      accuracy4,
                      accuracy5,
                      accuracy6,
                      accuracy7,
                      accuracy8
                     ]})
result_df = results.sort_values(by='Accuracy_score', ascending=False)
result_df = result_df.reset_index(drop=True)
result_df.head(9)


# In[44]:


plt.subplots(figsize=(15,9))
ax=sns.barplot(x='Model',y="Accuracy_score",data=result_df)
labels = (result_df["Accuracy_score"])
 #add result numbers on barchart
for i, v in enumerate(labels):
    ax.text(i, v+1, str(v), horizontalalignment = 'center', size = 18,
        color = 'black')


# In[ ]:
```

9.3 Perceptron

Listing 9: Kod perceptronu

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
import random as rd
import math
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib.colors import ListedColormap


# In[2]:

```

```
19
20 pillow = pd.read_csv("SaYoPillow.csv")
21 pillow.sort_values(
22 ['stress'],
23 axis=0,
24 inplace=True,
25 na_position='first')
26
27
28 # In[3]:
29
30
31 class Perceptron:
32     """
33     Perceptron neuron
34     """
35
36     def __init__(self, learning_rate=0.1):
37         """
38         instantiate a new Perceptron
39
40         :param learning_rate: coefficient used to tune the model
41         response to training data
42         """
43         self.learning_rate = learning_rate
44         self._b = 0.0  # y-intercept
45         self._w = None  # weights assigned to input features
46         # count of errors during each iteration
47         self.misclassified_samples = []
48     #n_iter=10
49     def fit(self, x: np.array, y: np.array, n_iter=100):
50         """
51         fit the Perceptron model on the training data
52
53         :param x: samples to fit the model on
54         :param y: labels of the training samples
55         :param n_iter: number of training iterations
56         """
57         self._b = 0.0
58         self._w = np.zeros(x.shape[1])
59         self.misclassified_samples = []
60
61         for _ in range(n_iter):
62             # counter of the errors during this training iteration
63             errors = 0
64             for xi, yi in zip(x, y):
65                 # for each sample compute the update value
66                 update = self.learning_rate * (yi - self.predict(xi))
67                 # and apply it to the y-intercept and weights array
68                 self._b += update
69                 self._w += update * xi
70                 errors += int(update != 0.0)
71
72             self.misclassified_samples.append(errors)
73
```

```
74      def f(self, x: np.array) -> float:
75          """
76          compute the output of the neuron
77          :param x: input features
78          :return: the output of the neuron
79          """
80          return np.dot(x, self._w) + self._b
81
82      def predict(self, x: np.array):
83          """
84          convert the output of the neuron to a binary output
85          :param x: input features
86          :return: 1 if the output for the sample is positive (or zero),
87          -1 otherwise
88          """
89          return np.where(self.f(x) >= 0, 1, -1)
90
91
92  # In[4]:
93
94
95  pillow.head()
96
97
98  # In[5]:
99
100
101 # extract the label column
102 y=pillow.iloc[:,8].values
103 # extract features
104 x=pillow.iloc[:,0:7].values
105
106 fig = plt.figure()
107 ax = plt.axes(projection='3d')
108
109 ax.set_title('Spanko set')
110 ax.set_xlabel("snoring")
111 ax.set_ylabel("respiration")
112 ax.set_zlabel("temperature")
113
114 # plot the samples
115 ax.scatter(x[:126, 0], x[:126, 1], x[:126,2], color='red',
116            marker='o', s=4, edgecolor='red', label="stress 0")
117 ax.scatter(x[126:252, 0], x[126:252, 1], x[126:252, 2], color='blue',
118            marker='^', s=4, edgecolor='blue', label="stress 1")
119 ax.scatter(x[252:378, 0], x[252:378, 1], x[252:378, 2], color='green',
120            marker='x', s=4, edgecolor='green', label="stress 2")
121 ax.scatter(x[378:504, 0], x[378:504, 1], x[378:504, 2], color='yellow',
122            marker='+', s=4, edgecolor='green', label="stress 3")
123 ax.scatter(x[504:630, 0], x[504:630, 1], x[504:630, 2], color='purple',
124            marker='x', s=4, edgecolor='green', label="stress 4")
125
126 plt.legend(loc='upper right')
127 plt.show()
128
```

```
129
130 # In[6]:
131
132
133 x = x[0:252, 0:3:2] # reduce the dimensionality of the data
134 y = y[0:252]
135
136 # plot stress0 samples
137 plt.scatter(x[:126, 0], x[:126, 1], color='red', marker='o', label='
    stress 0')
138 # plot stress1 samples
139 plt.scatter(x[126:252, 0], x[126:252, 1], color='blue', marker='x',
    label='stress 1')
140
141 # show the legend
142 plt.xlabel("snoring")
143 plt.ylabel("temperature")
144 plt.legend(loc='upper left')
145
146 # show the plot
147 plt.show()
148
149
150 # In[7]:
151
152
153 from sklearn.model_selection import train_test_split
154
155
156 # split the data
157 x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
    =0.15,
158                                                     random_state=0)
159 #when test_size>0.15, accuracy is around 50% which may cause problems
    later on
160
161 # train the model
162 classifier = Perceptron(learning_rate=0.1)
163 classifier.fit(x_train, y_train)
164
165 # plot the number of errors during each iteration
166 plt.plot(range(1, len(classifier.misclassified_samples) + 1),
167          classifier.misclassified_samples, marker='o')
168 plt.xlabel('Epoch')
169 plt.ylabel('Errors')
170 plt.show()
171 from sklearn.metrics import accuracy_score
172 print("accuracy %f" % accuracy_score(classifier.predict(x_test), y_test)
    )
173
174
175 # In[8]:
176
177
178 # plot the first feature before standardization
```

```
plt.hist(x[:, 0], bins=100)
plt.title("Features before standardization")
plt.show()

# standardization of the two features
x[:, 0] = (x[:, 0] - x[:, 0].mean()) / x[:, 0].std()
x[:, 1] = (x[:, 1] - x[:, 1].mean()) / x[:, 1].std()

# features after standardization
plt.hist(x[:, 0], bins=100)
plt.title("Features after standardization")
plt.show()

# split the data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
    =0.15,
                                                    random_state=0)
#when test_size>0.15 the graph does not show values separated correctly

# train the model
classifier = Perceptron(learning_rate=0.1)
classifier.fit(x_train, y_train)

# plot the number of errors during each iteration
plt.plot(range(1, len(classifier.misclassified_samples) + 1),
         classifier.misclassified_samples, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Errors')
plt.show()

from sklearn.metrics import accuracy_score
print("accuracy %f" % accuracy_score(classifier.predict(x_test), y_test)
    )


# In[9]:


def plot_decision_regions(x, y):
    resolution = 0.001

    # define a set of markers
    markers = ('o', 'x')
    # define available colors
    cmap = ListedColormap(('red', 'blue'))

    # select a range of x containing the scaled test set
    x1_min, x1_max = x[:, 0].min() - 0.5, x[:, 0].max() + 0.5
    x2_min, x2_max = x[:, 1].min() - 0.5, x[:, 1].max() + 0.5

    # create a grid of values to test the classifier on
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))

    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
232     Z = Z.reshape(xx1.shape)
233
234     # plot the decision region...
235     plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
236     plt.xlim(xx1.min(), xx1.max())
237     plt.ylim(xx2.min(), xx2.max())
238
239     # ...and the points from the test set
240     for idx, c1 in enumerate(np.unique(y)):
241         plt.scatter(x=x[y == c1, 0],
242                     y=x[y == c1, 1],
243                     alpha=0.8,
244                     c=cmap(idx),
245                     marker=markers[idx],
246                     label=c1)
247     plt.show()
248
249
250 plot_decision_regions(x_test, y_test)
251
252
253 # In[ ]:
```

Literatura

[1] L. Rachakonda, A. K. Bapatla, S. P. Mohanty, and E. Kougianos, *SaYoPillow: Blockchain-Integrated Privacy-Assured IoMT Framework for Stress Management Considering Sleeping Habits*. IEEE Transactions on Consumer Electronics (TCE), Vol. 67, No. 1, Feb 2021, pp. 20-29.

[2] L. Rachakonda, S. P. Mohanty, E. Kougianos, K. Karunakaran, and M. Ganapathiraju, *Smart-Pillow: An IoT based Device for Stress Detection Considering Sleeping Habits*. in Proceedings of the 4th IEEE International Symposium on Smart Electronic Systems (iSES), 2018, pp. 161–166.