

**Uniwersytet Warszawski**  
Wydział Matematyki, Informatyki i Mechaniki

**Krzysztof Rutkowski**

Nr albumu: 319379

**Wybrane metody dyskretyzacji i  
generowania nowych cech z  
zastosowaniem paradygmatu  
MapReduce**

**Praca magisterska  
na kierunku MATEMATYKA**

Praca wykonana pod kierunkiem  
**prof.dr.hab Andrzej Skowron**  
Zakład Logiki Matematycznej

Czerwiec 2016

## **Oświadczenie kierującego pracą**

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data

Podpis kierującego pracą

## **Oświadczenie autora (autorów) pracy**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data

Podpis autora (autorów) pracy

## **Streszczenie**

TODO

## **Słowa kluczowe**

discretization, feature extraction, decision system, mapreduce

## **Dziedzina pracy (kody wg programu Socrates-Erasmus)**

11.1 Matematyka

## **Klasyfikacja tematyczna**

sztuczna inteligencja, systemy decyzyjne

## **Tytuł pracy w języku angielskim**

Discretization and feature extraction methods with application of MapReduce paradigm



# Spis treści

<b>1. Podstawowe informacje</b>	5
<b>2. Skalowalność</b>	9
2.1. Wstępne definicje	9
2.2. Paradygmat MapReduce	9
2.3. Metryki	12
<b>3. Biblioteka mllib-extensions</b>	13
<b>4. Skalowalne algorytmy dyskretyzacji</b>	15
4.1. Sprawdzanie spójności tabeli	15
4.2. Opis algorytmów	15
4.2.1. Sortowanie	16
4.2.2. Algorytm bazowy	16
4.2.3. OneR Discretizer	17
4.2.4. Dyskretyzacja za pomocą entropii	18
4.3. Wyniki eksperymentów	20
4.3.1. Skalowalność	21
<b>5. Generowanie nowych cech na podstawie algorytmów genetycznych</b>	23
5.1. Dyskretyzacja jako generowanie nowych cech oraz generowanie nowych cech przez hiperpłaszczyzny	23
5.2. Algorytmy ewolucyjne	24
5.2.1. Reprezentacja hiperpłaszczyzny	24
5.2.2. Szukanie optymalnej hiperpłaszczyzny za pomocą algorytmu genetycznego	25
5.2.3. Analiza złożoności czasowej algorytmu	30
5.3. Indukcja drzewa decyzyjnego na podstawie cięć przez hiperpłaszczyzny	31
5.3.1. Generowanie lasu losowego	32
5.4. Drzewo SVM	32
5.5. Strategie klasyfikacji bliskich punktów o różnej decyzji oraz warunek oraz przycinanie	33
5.6. Wyniki eksperymentów	35
5.6.1. Generowanie nowych cech	35
5.6.2. Generowanie drzewa decyzyjnego	36
5.6.3. Porównanie wyników	38



# Rozdział 1

## Podstawowe informacje

**Definicja 1.1.** Systemem decyzyjnym (*ang. Decision System*) nazwiemy następującą trójkę obiektów  $\mathbb{S} = (U, A \cup d)$ , gdzie

- $U$  jest skończonym zbiorem, którego elementami są obiekty:

$$U = \{u_1, \dots, u_n\}$$

- $A$  jest skończonym zbiorem atrybutów warunkowych (cech) takim, że dla każdego  $a \in A$  istnieje funkcja

$$a : U \rightarrow V^a$$

gdzie przez  $V^a$  oznaczamy zbiór wartości atrybutów,

- $d$  jest skończonym zbiorem atrybutów, które nazywamy atrybutami decyzyjnymi.

$$d : U \rightarrow \{d_1, \dots, d_k\},$$

gdzie zbiór  $\{d_1, \dots, d_k\}$ , jest to zbiór wszystkich możliwych decyzji.

Jedną z możliwości reprezentacji systemu decyzyjnego 1 jest dwuwymiarowa tabela. W poszczególnych wierszach tabeli znajdują się obiekty  $\{u_1, \dots, u_n\}$ , natomiast w kolumnach znajdują się wartości atrybutów  $\{a_{11}, \dots, a_{nn}\}$ . Dokładniej mówiąc w  $i$ -tym wierszu znajduje się wartość  $j$ -tego atrybutu ( $a_{ij}$ ) dla obiektu  $u_i$ . W ostatniej kolumnie umieszczony jest atrybut decyzyjny  $d_i$  dla obiektu  $u_i$

$U$	$a_1$	$\dots$	$a_n$	$dec$
$u_1$	$a_{11}$	$\dots$	$a_{1n}$	$d_1$
$u_2$	$a_{21}$	$\dots$	$a_{2n}$	$d_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$u_n$	$a_{n1}$	$\dots$	$a_{nn}$	$d_n$

1 : System decyzyjny

Obiektami tablicy decyzyjnej mogą być takie rzeczy jak szeregi czasowe, ludzie, papiery wartościowe i inne.

Dla przykładu rozważmy tablicę decyzyjną, której obiektami są ludzie. Rozważmy pacjentów, którzy zostali przebadani pod kątem białaczki. Każdemu pacjentowi zbadano kilkanaście genów. Wynikiem tego padania jest kilkanaście liczb rzeczywistych opisujących poszczególne geny. Na podstawie wyników badań lekarz diagnozował pacjentów.

<i>Pacjent</i>	gen 1	gen 2	gen 3	Chory
<i>pacjent<sub>1</sub></i>	0.02	0.12	1.2	Tak
<i>pacjent<sub>2</sub></i>	-0.45	4.56	2.3	Nie
<i>pacjent<sub>3</sub></i>	1.32	2.3	0.01	Tak
<i>pacjent<sub>4</sub></i>	1.23	1.2	2.12	Nie

W powyższej sytuacji obiekt (pacjent) tablicy decyzyjnej zawiera ciąg liczb (każda traktowana jako jeden atrybut), które zostały uzyskane podczas przeprowadzonego badania. Ostatnim elementem wiersza jest atrybut decyzyjny(decyzja), stwierdzająca czy pacjent jest chory.

Często zdarza się tak, że w tabeli występują wartości rzeczywiste, np w powyższych danych medycznych, gdzie wyniki badań mają wartości z pewnego przedziału bądź gdy mamy podane pewne ciągłe wielkości fizyczne, np temperaturę. Z różnych względów, np wymagają tego niektóre klasyfikatory, chcielibyśmy mieć tabelę z wartościami nominalnymi. I tak, np poniższą tabelę:

<i>U</i>	Temparatura	Wilgotność	Zachmurzenie	Gra
<i>u<sub>1</sub></i>	0	30	słabe	Nie
<i>u<sub>2</sub></i>	10	90	umiarkowane	Nie
<i>u<sub>3</sub></i>	20	30	słabe	Tak
<i>u<sub>4</sub></i>	15	30	deszcz	Nie

Moglibyśmy zastąpić tabelą:

<i>U</i>	Temparatura	Wilgotność	Zachmurzenie	Gra
<i>u<sub>1</sub></i>	niska	mała	słabe	Nie
<i>u<sub>2</sub></i>	średnia	duża	umiarkowane	Nie
<i>u<sub>3</sub></i>	wysoka	mała	słabe	Tak
<i>u<sub>4</sub></i>	średnia	mała	deszcz	Nie



Wprowadzę teraz wstępne pojęcia dotyczące dyskretyzacji atrybutów rzeczywistych.

**Definicja 1.2** (Niesprzeczna tabela decyzyjna). Tabela  $\mathbb{S} = (U, A, \{d\})$  jest niesprzeczna, jeśli dla dowolnych dwóch obiektów  $u_1$  i  $u_2$ :

$$(a_1(u_1), a_2(u_1), \dots, a_k(u_1)) = (a_1(u_2), a_2(u_2), \dots, a_k(u_2)) \Rightarrow d(u_1) = d(u_2)$$

. Tj jeśli mamy te same wartości na atrybutach musimy mieć te same decyzje.

**Definicja 1.3** (Podział atrybutu). Dla tabeli  $\mathbb{S} = (U, A, \{d\})$  i zbioru cięć dla atrybutu  $a$ :

$$c_0^a < c_1^a < \dots < c_{k_a}^a < c_{k_a+1}^a$$

definiujemy podział (dyskretyzację) atrybutu  $a$  jako:

$$P_a = [c_0^a, c_1^a], \dots, [c_{k_a}^a, c_{k_a+1}^a]$$

Zaś podział całego uniwersum jako:

$$P = \cup_{a \in A} P_a$$

*Uwaga 1.4.* Szukanie niesprzecznego zbioru cięć o najmniejszej mocy jest  $NP$ -zupełny. Dowód poprzez sprowadzenie do problemu **SET COVER**.



## Rozdział 2

# Skalowalność

Większość algorytmów została zaimplementowana w sposób skalowalny za pomocą frameworku do obliczeń równoległych **Apache Spark** bazującego na paradygmacie MapReduce. Na wstępie przytoczę kilka definicji przydatnych w dalszej części omawiania Sparka.

### 2.1. Wstępne definicje

**Definicja 2.1** (Programowanie funkcyjne). Filozofia i metodyka programowania będąca odmianą programowania deklaratywnego, w której funkcje należą do wartości podstawowych, a nacisk kładzie się na wartościowanie (często rekurencyjnych) funkcji, a nie na wykonywanie poleceń. W czystym programowaniu funkcyjnym, raz zdefiniowana funkcja zwraca zawsze tę samą wartość dla danych wartości argumentów, tak jak funkcje matematyczne. W czystych językach funkcyjnych nie występują zmienne ani efekty uboczne, a wartościowanie jest leniwe, tj wykonywanie obliczenia następuje tylko w momencie, gdy jest to potrzebne.

**Definicja 2.2** (Klaster). Grupa połączonych jednostek komputerowych, które współpracują ze sobą w celu udostępnienia zintegrowanego środowiska pracy. Komputery wchodzące w skład klastra (będące członkami klastra) nazywamy węzłami (ang. node).

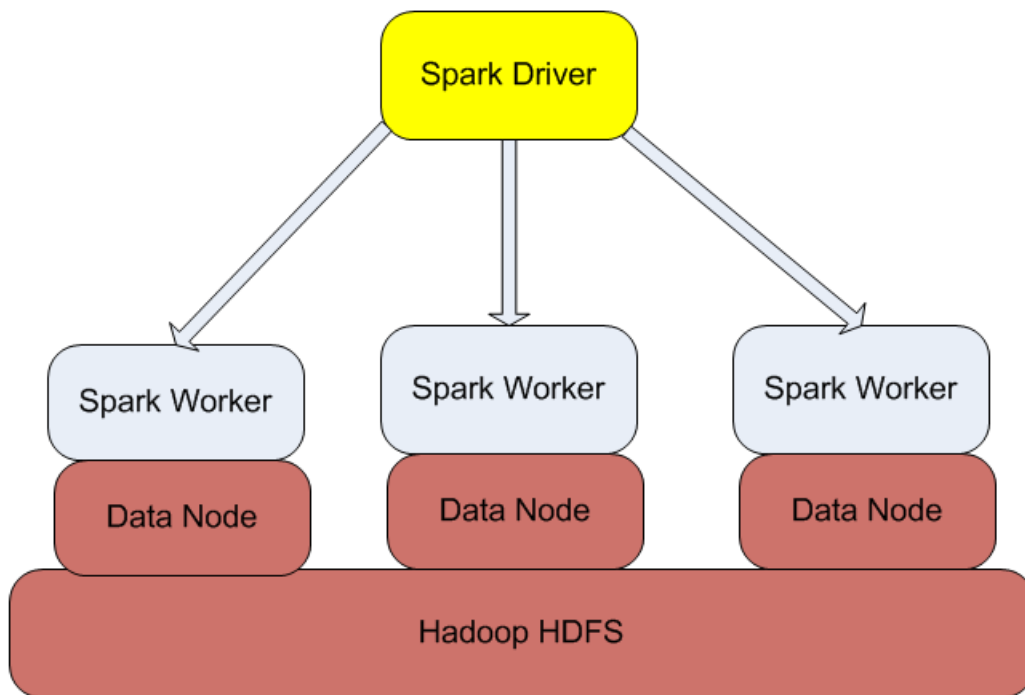
**Definicja 2.3** (HDFS). Hadoop Distributed File System, rozproszony, skalowalny, spójny system plików używany przez framework Apache Spark.

**Definicja 2.4** (RDD). Resilient Distributed Dataset, podstawowa struktura danych w Sparku. RDD jest kolekcją elementów dystrybuowaną pomiędzy węzły klastra, na których operacje mogą być wykonywane równolegle. Po zakończeniu operacji na RDD można je dalej przechowywać w pamięci RAM, co umożliwia szybsze dostępy do następnych obliczeń.

### 2.2. Paradygmat MapReduce

Paradygmat MapReduce został wprowadzony przez firmę **Google** i służy do równoległego przetwarzania dużych danych przy użyciu klastra. Rozproszenie obliczeń pomaga w istotny sposób zmniejszyć czas wykonywanych zadań. Apache Spark jest niejako rozszerzeniem wprowadzonego paradygmatu. Jak każdy framework do obliczeń na klastrze posiada pod sobą rozproszony system plików (HDFS), a także oprogramowanie (dla workerów i mastera), które odpowiada za zarządzanie awariami, przydzielanie zadań i organizację.

Rysunek 2.1: architektura Sparka



Spark jest dedykowany dla języka funkcyjnego - Scali, algorytmy zostały jednak napisane w pythonie, który mimo, iż językiem funkcyjnym nie jest, posiada jednak cechy języków funkcyjnych.

Typowy potok obliczeń w Sparku polega na:

- Stworzeniu RDD bazującego na kolekcji utrzymywanej w programie
- Wykonaniu transformacji na RDD
- Wykonaniu akcji

Funkcje w Sparku dzielą się na transformacje i akcje.

- **Transformacja** jest funkcją działającą na RDD i zwracającą RDD
- **Akcja** jest funkcją działającą na RDD i zwracającą kolekcję do programu po zakończeniu obliczeń

Zarówno akcje, jak i transformacje wykonywane są leniwie. Przykłady transformacji:

- **map(f)** - zwraca nowy RDD aplikując funkcję  $f$  do każdego elementu RDD
- **filter(f)** - zwraca nowy RDD składający się tylko z tych elementów, dla których funkcja  $f$  zwróci True
- **reduceByKey(f)** - dla RDD postaci (klucz, wartość), grupuje według kluczy i zwraca RDD z agregacją wartości za pomocą funkcji  $f$  dla zgrupowanych kluczy

Przykłady akcji:

- **reduce(f)** - zwraca kolekcję będącą agregacją RDD za pomocą funkcji  $f$ , która przyjmuje dwa argumenty i zwraca jeden
- **collect()** - zwraca kolekcję elementów do programu

W powyższych przykładach zamiast funkcji  $f$  można użyć wyrażeń lambda (funkcji anonimowych).

Przykład, zliczanie słów w języku python:

```
#ustawiamy kontekst i konfiguracje
conf = (SparkConf().setMaster("spark://localhost:7077").
        setAppName("entropy"))
sc = SparkContext(conf=conf)
words = ["ala", "ma", "ala", "kota", "ma"]
#tworzenie rdd na podstawie listy
words_rdd = sc.parallelize(words)
words_counts = words_rdd.
                map(lambda x: (x, 1)).
                reduceByKey(lambda x, y: x+y)
```

W powyższym, jak i w następnych listingach algorytmów w języku python, sc zawsze będzie obiektem klasy *SparkContext*, który inicjuje RDD na podstawie konfiguracji klastrowego środowiska dla Sparka. Konfigurację tworzymy na podstawie obiektu klasy *SparkConf*, bądź ustawiamy parametry klastra, takie jak:

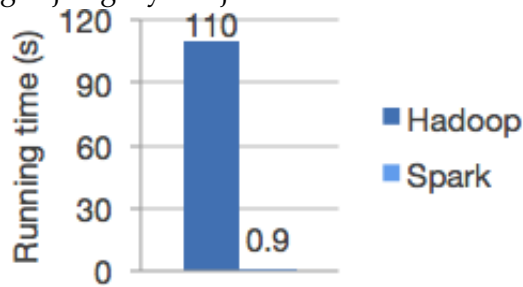
- liczba komputerów
- liczba procesorów
- pamięć RAM na jednostkę obliczeniową
- nazwa aplikacji

w odpowiednim pliku konfiguracyjnym.

Obliczenia wykonywane w Sparku są nie tylko skalowalne, ale również szybkie ze względu na:

- leniwość wykonywanych operacji, obliczenia wykonywane są tylko wtedy gdy są potrzebne (ważna cecha języków funkcyjnych)
- możliwość przechowywania rozproszonych struktur danych (RDD) w pamięci RAM, co powoduje szybszy dostęp do danych podczas ponownego przetwarzania (np w Hadoop MapReduce musieliśmy dane wczytywać z HDFS)

Rysunek 2.2: porównanie szybkości działania Hadoop MapReduce i Spark na przykładzie regresji logistycznej



Na rysunku 2.2 widzimy porównanie szybkości działania Hadoop MapReduce i Sparka. Widzimy jak bardzo idea RDD, która nie występuje w Hadoop MapReduce zwiększa szybkość działania.

## 2.3. Metryki

Do oceniania jakości skalowalnych algorytmów wykorzystywanych jest kilka metryk ocenić czy dany algorytm skaluje się w odpowiedni sposób. W tej sekcji zaprezentowane zostały metryki pozwalające ocenić skalowalność algorytmów.

### Speed-up

Jedną z metryk mierzącą ile razy obliczenia na jednej maszynie są większe od obliczeń wykonywanych na  $n$  maszynach jest metryka zdefiniowana następująco:

$$speed - up(n) = \frac{t_1}{t_n}$$

gdzie  $t_j$  jest czasem obliczeń wykonywanym na  $j$  maszynach.

### Scale-up

Ta metryka z kolei opisuje jak zmienia się czas obliczeń podczas gdy liczba komputerów oraz wielkość zbioru danych rosną taką samą liczbę razy.

$$scale - up(n) = \frac{t_{D_n}}{t_{D_1}}$$

$t_{D_1}$  – czas obliczeń dla ustalonego zbioru danych na jednym komputerze,  $t_{D_n}$  – czas obliczeń dla  $n$ -razy większego zbioru danych na  $n$  komputerach.

## Rozdział 3

# Biblioteka **mlib-extensions**

Algorytmy użyte w dalsze części pracy zostały zaimplementowane w bibliotece **mlib-extensions**, umieszczonej w repozytorium, które znajduje się pod adresem: <https://github.com/cpawols/mlib-extension>. Biblioteka została napisana w języku python, jest skalowalna, wszystkie obliczenia mogą być wykonywane na wielu maszynach, bądź w zwyczajnym trybie, na jednym procesorze, zostały zaimplementowane następujące moduły:

- **discretization**: moduł do wykonywania dyskretyzacji
- **feature-extraction**: moduł do generowania nowych cech
- **decision-tree**: moduł do indukcji drzew decyzyjnych
- **reduct-feature-selection**: moduł do selekcji atrybutów (autor: Paweł Olszewski)





## Rozdział 4

# Skalowalne algorytmy dyskretyzacji

W tym rozdziale omówię zaimplementowane algorytmy dyskretyzacji, jak i przedstawię wyniki eksperymentów i obliczeń. Algorytmy dyskretyzacji dzielimy na:

- **globalne** - szukają cięć na podstawie całej tabeli
- **lokalne** - szukają cięć na podstawie każdego atrybutu niezależnie

Będziemy tu analizować algorytmy lokalne, gdyż w naturalny sposób łatwo je zrównoleglić, zgodnie ze schematem:

```
# tworzymy RDD na podstawie listy kolumn i
# dzielimy na $numer_of_columns$ kawalkow
columns_rdd = sc.parallelize(table, numer_of_columns)
# wykonujemy algorytm dyskretyzacyjny rownolegle dla kazdej kolumny
discretized_columns = columns_rdd.
    mapPartitions(discretizeColumn).
    collect()
```

Gdzie funkcja `discretizeColumn` przyjmuje kolumnę liczb rzeczywistych tabeli decyzyjnej i zwraca jej dyskretyzację. Metody globalne szukają zbioru cięć dla całej tabeli, przez co trudniej je zrównoleglić. Niżej przedstawię 3 lokalne algorytmy dyskretyzacji.

### 4.1. Sprawdzanie spójności tabeli

By dyskretyzacja miała sens, potrzebujemy sprawdzić czy tabela jest spójna, w Sparku możemy to w prosty sposób zrobić za pomocą poniższego algorytmu:

```
# funkcja agregująca, zwraca tuple składająca się z różnych decyzji
# dla danej kombinacji atrybutow
def get_unique_dec(x, y):
    return tuple(set(x + y))
# rows_rdd = (row, (dec,))
rows_rdd = sc.parallelize(table, numer_of_rows)
one_dec = rows_rdd.
    reduceByKey(get_unique_dec).
    map(lambda k, v: len(v) == 1).
    collect()
# tabela jest spojna jesli dla kazdej kombinacji atrybutow
# mamy jedna decyzje
is_consistent = all(one_dec)
```

### 4.2. Opis algorytmów

Wszystkie z poniższych algorytmów są zrównoleglane ze względu na kolumnę, tak jak było to napisane we wstępie. Różnią się metodą `discretize_column`.

### 4.2.1. Sortowanie

Wszystkie niżej wymienione algorytmy operują na posortowanych kolumnach. By posortować całą tabelę najpierw sprowadzam ją do postaci EAV, tj listy trójek (Obiekt, Atrybut, Wartość), a następnie sortuję tę listę najpierw względem wartości, potem obiektów, a następnie atrybutów, otrzymując listę posortowanych wartości zgrupowanych względem atrybutów. W Sparku wykonujemy sortowanie równoległe, za pomocą poniższego algorytmu, będącego uogólnieniem mergesort:

- dzielę listę na *numchunks* kawałków
- sortuję każdy kawałek niezależnie
- scalam posortowane kawałki

Kod algorytmu w języku python:

```
# funkcja porownujaca
# wzgledem atrybutu, potem wartosci, a nastepnie nazwy obiektu
def compare(iterator):
    yield sorted(iterator, key=lambda x: (x[1], x[2], x[0]))

def merge_sort():
    numChunks = 10
    eav_rdd_part = Configuration.sc.parallelize(eav, numChunks)
    self.eav = eav_rdd_part.
        mapPartitions(compare).
        reduce(lambda x, y:
            sorted(x+y, key=lambda x: (x[1], x[2], x[0])))
```

Dla podziału na  $k$  kawałków algorytm ma złożoność czasową  $O(\underbrace{\frac{n}{k} \cdot \log(\frac{n}{k})}_{\text{dzielenie}} + \underbrace{\frac{n}{k} \cdot k}_{\text{scalanie}})$ , czyli  $O(\frac{n}{k} \cdot \log(\frac{n}{k}) + n)$  a więc dla dostatecznie dużych  $k$  otrzymujemy algorytm liniowy.

### 4.2.2. Algorytm bazowy

Jest to prosty algorytm polegający na wykonywaniu cięć w równych odstępach, co  $m$  obiektów, gdzie  $m$  jest parametrem. Jego zaletą jest prostota, jednak wogóle nie patrzymy na decyzję. Kod algorytmu:

```
def discretizeColumn(column, m):
    for ind, row in enumerate(column):
        yield (row[0], row[1], ind / m)
```

**Przykład 4.1** (Przykładowa dyskretyzacja). Dla tabeli:

$U$	Temperatura	Wilgotność	Zachmurzenie	Gra
$u_1$	0	30	słabe	Nie
$u_2$	10	90	umiarkowane	Nie
$u_3$	20	30	słabe	Tak
$u_4$	15	50	deszcz	Nie
$u_5$	15	30	deszcz	Nie
$u_6$	30	70	deszcz	Nie
$u_7$	15	30	deszcz	Nie

Oraz  $m = 3$ , mamy następującą dyskretyzację (dla kolumn Temperatura i Wilgotność):

$U$	<i>Temperatura</i>	<i>Wilgotność</i>	<i>Zachmurzenie</i>	<i>Gra</i>
$u_1$	0	0	<i>słabe</i>	<i>Nie</i>
$u_2$	0	2	<i>umiarkowane</i>	<i>Nie</i>
$u_3$	1	0	<i>słabe</i>	<i>Tak</i>
$u_4$	0	1	<i>deszcz</i>	<i>Nie</i>
$u_5$	1	0	<i>deszcz</i>	<i>Nie</i>
$u_6$	2	1	<i>deszcz</i>	<i>Nie</i>
$u_7$	1	1	<i>deszcz</i>	<i>Nie</i>

### 4.2.3. OneR Discretizer

W tym algorytmie wykonujemy cięcie w momencie, gdy jednej z decyzji jest więcej niż innych, przy czym dany przedział musi zawierać więcej niż  $m$  elementów, gdzie  $m$  jest ustalonym parametrem. Algorytm:

```
def discretize_column(column, m):
    new_dec = 0
    block_elements = 0
    dec_histogram_block = Counter()
    for elem in column:

        if block_elements > m and
            most_common(dec_histogram_block) > block_elements / 2:
            block_elements = 0
            new_dec += 1
            dec_histogram_block = Counter()
        dec_histogram_block[decision(elem)] += 1
        block_elements += 1

    yield (elem[0], elem[1], new_dec)
```

**Przykład 4.2** (Przykładowa dyskretyzacja). Dla tabeli:

$U$	<i>Temperatura</i>	<i>Wilgotność</i>	<i>Zachmurzenie</i>	<i>Gra</i>
$u_1$	0	30	<i>słabe</i>	<i>Nie</i>
$u_2$	10	90	<i>umiarkowane</i>	<i>Nie</i>
$u_3$	20	30	<i>słabe</i>	<i>Tak</i>
$u_4$	15	50	<i>deszcz</i>	<i>Nie</i>
$u_5$	15	30	<i>deszcz</i>	<i>Tak</i>
$u_6$	30	70	<i>deszcz</i>	<i>Nie</i>
$u_7$	15	30	<i>deszcz</i>	<i>Nie</i>

Oraz  $m = 4$ , mamy następującą dyskretyzację (dla kolumn *Temperatura* i *Wilgotność*):

$U$	<i>Temperatura</i>	<i>Wilgotność</i>	<i>Zachmurzenie</i>	<i>Gra</i>
$u_1$	0	0	<i>słabe</i>	<i>Nie</i>
$u_2$	0	1	<i>umiarkowane</i>	<i>Nie</i>
$u_3$	1	0	<i>słabe</i>	<i>Tak</i>
$u_4$	0	0	<i>deszcz</i>	<i>Nie</i>
$u_5$	0	0	<i>deszcz</i>	<i>Tak</i>
$u_6$	1	1	<i>deszcz</i>	<i>Nie</i>
$u_7$	1	0	<i>deszcz</i>	<i>Nie</i>

#### 4.2.4. Dyskretyzacja za pomocą entropii

Entropię dla zbioru obiektów  $U$  definiujemy jako:

$$Ent(U) = - \sum_{d \in dec(U)} p_d \cdot \log(p_d)$$

Gdzie  $dec(U)$  to zbiór decyzji obiektów ze zbioru  $U$ , oraz  $p_d$  jest empirycznym prawdopodobieństwem decyzji  $d$ , liczonym z wzoru:

$$p_d = \frac{|\{u \in U : dec(u) = d\}|}{|U|}$$

Entropia jest miarą informacji jaką mamy o danym zbiorze obiektów. Jest ona zerowa, gdy w danym zbiorze obiektów mamy jedną decyzję, a więc mamy pełną informację, zaś największa gdy wszystkich decyzji jest po równo. Entropię dla danego cięcia  $c$  na danym atrybucie  $a$  możemy zdefiniować jako:

$$E(a, c, U) = \frac{U_1}{n} Ent(U_1) + \frac{U_2}{n} Ent(U_2)$$

Gdzie  $n$  jest liczbą obiektów w  $u$  oraz  $U_1$  i  $U_2$  powstają z  $U$  na podstawie cięcia  $c$ . Dla danego cięcia możemy więc zdefiniować miarę przyrostu informacji:

$$Gain(a, c, U) = Ent(U) - E(a, c, U)$$

Wybieramy takie cięcie, by zmaksymalizować zysk informacji. Cięcia wykonujemy rekurencyjnie, dopóki nie spełniony zostanie pewien warunek stopu, tj gdy zysk informacji, który osiągamy dla danego optymalnego cięcia jest mniejszy niż jakaś wielkość.

#### Szukanie optymalnego cięcia - algorytm liniowy

Możemy łatwo znaleźć optymalne cięcie dla danego zbioru obiektów w czasie  $O(n^2)$  gdzie  $n$  jest liczbą obiektów za pomocą algorytmu (gdzie  $Ent(U)$  jest funkcją obliczającą entropię obiektów ze zbioru  $U$ ):

```
min_cut = 1
# mozliwe ciecia - pomiedzy obiektami od 1 do n
possible_cuts = range(1, n)
# ustawiamy wartosc minimalnego ciecia na nieskonczonosc
min_cut_value = INFINITY
for cut in possible_cuts:
    cut_value = cut / n * Ent(U[cut:]) +
                (n - cut) / n * Ent(U[:cut])
    if cut_value < min_cut_value:
        min_cut_value = cut_value
        min_cut = cut
```

Możemy jednak nie obliczać za każdym razem entropii, lecz liczyć ją na podstawie wcześniejszych wyników w czasie stałym, otrzymując algorytm liniowy ze względu na liczbę obiektów. Algorytm opiera się na obserwacji (zakładamy, że dla obiektów  $1, \dots, k-1$  mamy już obliczone częstości poszczególnych decyzji (słownik **decHist** w którym trzymamy histogram decyzji, za pomocą którego w czasie stałym liczymy  $p_d$ ):

$$\begin{aligned} Ent(U_k) &= Ent(U_{k-1}) + \log(p_{dec(u_k)}) \cdot p_{dec(u_k)} \\ &\quad decHist[dec(u_k)] += 1 \\ Ent(U_k) &= Ent(U_{k-1}) - \log(p_{dec(u_k)}) \cdot p_{dec(u_k)} \end{aligned}$$

Analogicznie trzymamy histogram dla drugiej obiektów  $k, \dots, n$  aktualizując entropię drugiej połowy. Otrzymujemy więc następujący algorytm liniowy:

```

def find_cut(U):
    # decHist1 -> histogram dla 1 po owy
    # decHist2 -> histogram dla 2 po owy
    # p1 -> prawdopodobieństwo liczone na podstawie decHist1
    # p2 -> prawdopodobieństwo liczone na podstawie decHist2
    def updateEnt(U, k):
        Ent[U[k:]] = Ent([U[k-1:]] - (-log(p1[dec(u_k)]) * p1[dec(u_k)]))
        Ent[U[:k]] = Ent[U[:k+1]] - (-log(p2[dec(u_k)]) * p2[dec(u_k)])
        decHist1[dec(U[k])] += 1
        decHist2[dec(U[k])] -= 1
        Ent[U[k:]] = Ent[U[k-1:]] - (-log(p1[dec(u_k)]) * p1[dec(u_k)])
        Ent[U[:k]] = Ent[U[:k+1]] - (-log(p2[dec(u_k)]) * p2[dec(u_k)])
        return Ent[U[k:]], Ent[U[:k]]

    min_cut = 1
    # mozliwe ciecia - pomiedzy obiektami od 1 do n
    possible_cuts = range(1, n)
    # ustawiamy wartosc minimalnego ciecia na nieskonczonosc
    min_cut_value = INFINITY
    Ent[0] = 0
    for cut in possible_cuts:
        Ent1, Ent2 = updateEnt(U, cut)
        cut_value = cut / n * Ent1 +
                    (n - cut) / n * Ent2
        if cut_value < min_cut_value:
            min_cut_value = cut_value
            min_cut = cut
    # jesli nie jest spelnione kryterium stopu zwracamy minimalne ciecie
    # oraz ciecia dla odpowiednich podzbiorow obiektow
    if not stop_criterion_satisfied(U, min_cut_value):
        yield min_cut

    for cut in find_cut(U[k:]):
        yield cut
    for cut in find_cut(U[:k]):
        yield cut

```

### Koncowy Algorytm:

```

# zwracamy tabele z odpowiednimi wartosciami disc_value
def discretize_column(column):
    disc_value = 0
    cuts_set = sorted(find_cut_lin(column))
    cur_cut = cuts_set[0]

    for i, elem in enumerate(column):
        if cur_cut == i:
            disc_value += 1
        if len(cuts_set) > disc_value:
            cur_cut = cuts_set[disc_value]

    yield (elem[0], elem[1], disc_value)

```

### Przykład 4.3 (Przykładowa dyskretyzacja). Dla tabeli:

$U$	Temperatura	Wilgotność	Zachmurzenie	Gra
$u_1$	0	30	słabe	Tak
$u_2$	10	90	umiarkowane	Nie
$u_3$	20	30	słabe	Nie
$u_4$	15	50	deszcz	Tak
$u_5$	15	30	deszcz	Nie
$u_6$	30	70	deszcz	Nie
$u_7$	15	30	deszcz	Nie

mamy następującą dyskretyzację (dla kolumn Temperatura i Wilgotność):

$U$	Temperatura	Wilgotność	Zachmurzenie	Gra
$u_1$	0	0	słabe	Tak
$u_2$	0	2	umiarkowane	Nie
$u_3$	1	1	słabe	Nie
$u_4$	0	2	deszcz	Tak
$u_5$	1	1	deszcz	Nie
$u_6$	1	2	deszcz	Nie
$u_7$	1	1	deszcz	Nie

### 4.3. Wyniki eksperymentów

Do wykonywania eksperymentów wykorzystałem komputery z laboratorium MIMUW. Maksymalnie obliczenia wykonywałem na 48 komputerach o następujących parametrach:

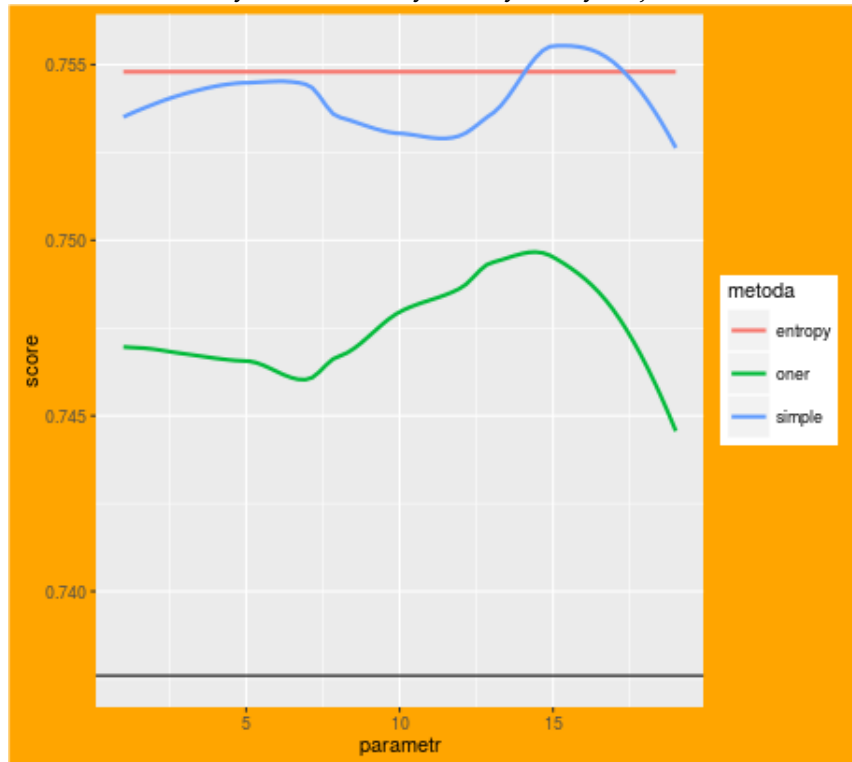
- **pamięć RAM:** 32 komputery 4 GB RAM, 16 komputerów 8 GB RAM
- **liczba jednostek CPU:** 8
- **taktowanie pojedynczego CPU:** 16 komputerów 3.1 GHz, 16 komputerów 3.4 GHz, 16 komputerów 2.8 GHz

Do eksperymentów wykorzystałem zbiór danych WAVEFORM z UCI Machine Learning Repository zawierający informacje na temat parametrów fal dźwiękowych. Zbiór ten zawiera 5000 obiektów i 40 atrybutów oraz 3 wartości decyzyjne - rodzaj fali. Oto jak prezentowały się wyniki predykcji dla tego zbioru danych w zależności od parametrów:

- dla metody simple jest to liczba obiektów dla pojedynczego cięcia
- dla metody OneR jest to minimalna liczba obiektów dla pojedynczego cięcia

Predykcja została wykonana za pomocą drzewa decyzyjnego z pythonowej biblioteki sklearn za pomocą 10-krotnej krosvalidacji metodą accuracy. Brązową kreską został zaznaczony wynik bez wykonywania dyskretyzacji.

Rysunek 4.1: Wyniki dyskretyzacji



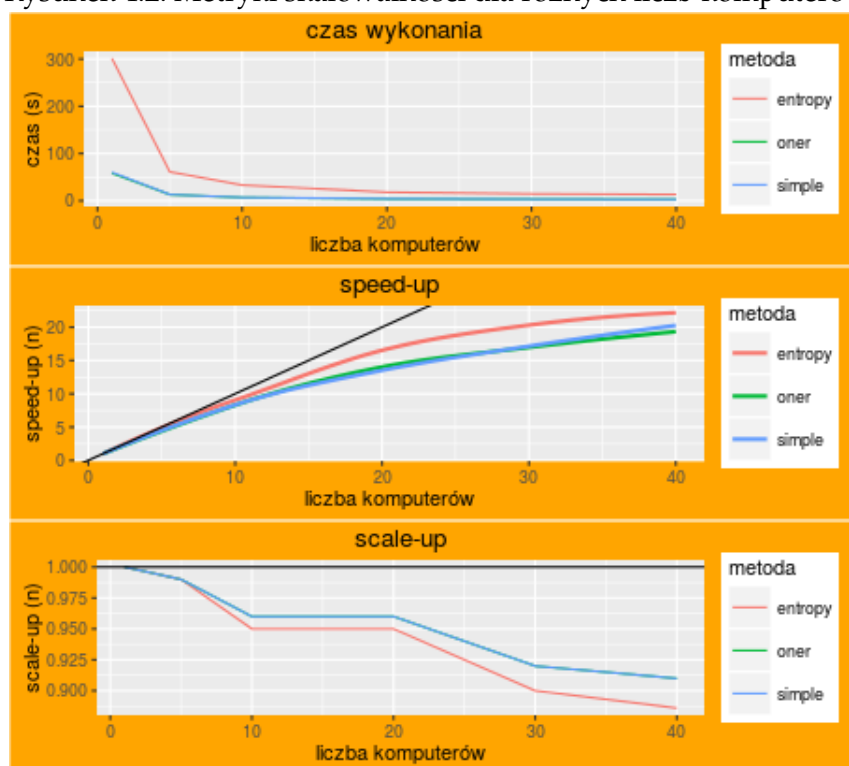
Widzimy, że wyniki dla dyskretyzacji są lepsze od baseline-u, zaś najlepsze dla metody prostej i parametru  $m = 15$ , lecz średnio najlepsza jest metoda entropii.

#### 4.3.1. Skalowalność

Zobaczmy jak wypadły eksperymenty pod kątem skalowalności. Wykonałem eksperymenty dla następujących liczb komputerów: 1, 5, 10, 20, 30, 40. Zbadałem jak rozkładają się poszczególne metryki w zależności od liczby komputerów i użytej metody.

Widzimy, że zarówno speed-up jak i scale-up wraz ze zwiększaniem liczby komputerów odbiega coraz bardziej od idealnej sytuacji (zaznaczonej na wykresie na brązowo). Jest to zgodne z prawem Gustafsona. Gdy liczba komputerów wzrasta, jest coraz większy narzut czasowy ze względu na przesyłanie danych po sieci. Widzimy jednak, że dopiero od ok 20 komputerów speed-up zaczyna mieć coraz wolniejszy przyrost, który zbiega do logarytmu. Powyższe algorytmy są więc całkiem dobrze skalowalne.

Rysunek 4.2: Metryki skalowalności dla różnych liczb komputerów





## Rozdział 5

# Generowanie nowych cech na podstawie algorytmów genetycznych

W tym rozdziale omówię uogólnienie pojęcia dyskretyzacji jako generowanie nowych cech i omówię znajdowanie nowych cech za pomocą hiperpłaszczyzn i algorytmu genetycznego.

### 5.1. Dyskretyzacja jako generowanie nowych cech oraz generowanie nowych cech przez hiperpłaszczyzny

Na dyskretyzację, a konkretnie na cięcie na pewnym atrybucie rzeczywistym możemy też patrzeć jak na nową cechę zdefiniowaną przez funkcję charakterystyczną odpowiedniej półprzestrzeni. Na cięcie cięcia o wartości  $c$  na atrybucie  $a_k$  możemy patrzeć jak na cechę:

$$v(a_1(u), a_2(u), \dots, a_n(u)) = \begin{cases} 1 & \text{dla } a_k(u) \leq c \\ 0 & \text{dla } a_k(u) > c \end{cases}$$

Dyskretyzacja na danym atrybucie definiuje nam jedynie hiperpłaszczyzny równoległe do odpowiednich osi. Możemy zdefiniować nowe cechy jako funkcje charakterystyczne ogólnych półprzestrzeni:

$$v(a_1(u), a_2(u), \dots, a_n(u)) = \begin{cases} 1 & \text{dla } w_1 \cdot a_1(u) + w_2 \cdot a_2(u) + \dots + w_n \cdot a_n(u) \leq c \\ 0 & \text{dla } w_1 \cdot a_1(u) + w_2 \cdot a_2(u) + \dots + w_n \cdot a_n(u) > c \end{cases}$$

Ogólne półprzestrzenie mają o wiele większą moc wyrazu i zwykle wystarczy ich mniej by znaleźć minimalny zbiór cięć dla danej przestrzeni, tj tak podzielić przestrzeń, aby otrzymać spójną tabelę. Jednak przestrzeń poszukiwań odpowiednich parametrów, tj wag  $w_k$  jest nieskończona. Dla danego zbioru wag, tj dla danej półprzestrzeni, możemy jednak ocenić jak dobre jest dane cięcie na podstawie funkcji oceny. Jedną z popularniejszych miar jest miara rozróżnialności, zdefiniujemy  $C_i^R$  oraz  $C_i^L$  jako liczbę obiektów o decyzji  $i$  należących do odpowiednich półprzestrzeni:

$$C_i^R(H) = |\{u \in U : dec(u) = i \wedge w_1 \cdot a_1(u) + w_2 \cdot a_2(u) + \dots + w_n \cdot a_n(u) > c\}|$$
$$C_i^L(H) = |\{u \in U : dec(u) = i \wedge w_1 \cdot a_1(u) + w_2 \cdot a_2(u) + \dots + w_n \cdot a_n(u) \leq c\}|$$

Na podstawie tych wielkości definiujemy miarę rozróżnialności dla hiperpłaszczyzny  $H$ :

$$discMeasure(H) = \sum_{i \neq j} C_i^R(h) \cdot C_j^L(H)$$

Mając funkcję oceny możemy więc z powodzeniem stosować pewne heurystyki doboru wag i algorytmy genetyczne. Konkretny algorytm omówię w następnym rozdziale.

## 5.2. Algorytmy ewolucyjne

Algorytmy ewolucyjne są jedną popularniejszych metod heurystycznych metod optymalizacji, jak np symulowane wyżarzanie. Sprawdzają się, gdy mamy dobrze określoną funkcję celu oraz potrafimy odpowiednio zamodelować nasze uniwersum. Algorytm genetyczny polega na iteracyjnym poprawianiu wyniku poprzez dobieranie do następnej fazy tych osobników, które osiągają najlepsze wartości funkcji celu. Podstawowym pojęciem jest tu populacja i osobnik. Osobnik jest pewnym obiektem matematycznym, najczęściej wektorem zerojedynkowym, który modeluje wielkości, które chcemy optymalizować. W przypadku problemu szukania optymalnej hiperpłaszczyzny będzie to wektor reprezentujący wagi  $w_i$ . Populacja zaś jest zbiorem osobników. Algorytm genetyczny można podzielić na następujące etapy:

1. Inicjowanie populacji początkowej
  - 1.1. Inicjowanie losowe
  - 1.2. Inicjowanie wg wcześniej ustalonego schematu
2. W pętli, aż do spełnienia pewnego warunku stopu wykonuj:
  - 2.1. Liczymy funkcję oceny każdego osobnika w populacji. Najlepsze osobniki poddajemy reprodukcji.
  - 2.2. Proces reprodukcji najlepszych osobników:
    - 2.2.1. Krzyżowanie - łączenie genotypów rodziców
    - 2.2.2. Mutacja - wprowadzanie losowych zmian
  - 2.3. Po procesie reprodukcji wybieramy najlepsze osobniki.
  - 2.4. Jeśli został spełniony warunek stopu wybieramy i zwracamy najlepszego osobnika w populacji.

Podstawowymi zadaniami przy projektowaniu algorytmu genetycznego są:

- ustalenie genomu osobnika jako reprezentanta wyniku
- ustalenie funkcji oceny
- dobór parametrów (warunek stopu, prawdopodobieństwo krzyżowania, mutacji, rozmiar populacji)

Dla problemu znajdowania optymalnej hiperpłaszczyzny funkcją oceny będzie wspomniana wcześniej miara **discMeasure**. Pozostaje ustalić genom osobnika (hiperpłaszczyzny). Będziemy reprezentować hiperpłaszczyznę w przestrzeni  $\mathbb{R}^n$  jako zbiór  $n - 1$  liniowo niezależnych wektorów i przesunięcia.

### 5.2.1. Reprezentacja hiperpłaszczyzny

Ustalmy jedną z osi, niech to będzie  $x_1$  odpowiadająca atrybutowi  $a_1$  i liczbę naturalną  $b$ , będzie to parametr odpowiadający za zbiór, z którego inicjujemy populację początkową. Z każdej dwu-wymiarowej płaszczyzny  $L(x_1, x_i)$  wybieramy wektory  $v_1^i, v_2^i, \dots, v_{2^b}^i$  (nierównoległe do  $x_1$ ) zdefiniowane przez:

$$v_j^i = [\alpha_j^i, 0, \dots, 0, \underbrace{1}_{i\text{-ta pozycja}}, 0, \dots, 0] \text{ dla } i = 2, \dots, n \text{ oraz } j = 1, \dots, 2^b$$

Wektory te będziemy inicjować przez losowy dobór  $2^b$  liczb:  $\alpha_1^i, \alpha_2^i, \dots, \alpha_{2^b}^i$ . Każdy z wektorów  $v_j^i$  może być reprezentowany przez  $b$  bitów odpowiadających reprezentacji binarnej liczby  $j$ . Rozważmy zbiór  $n - 1$  wektorów  $\{v_{j_2}^2, v_{j_3}^3, \dots, v_{j_n}^n\}$ :

$$\begin{aligned} v_{j_2}^2 &= [\alpha_{j_2}^2, 1, 0, 0, \dots, 0] \\ v_{j_3}^3 &= [\alpha_{j_3}^3, 0, 1, 0, \dots, 0] \\ &\dots \\ v_{j_n}^n &= [\alpha_{j_n}^n, 0, 0, 0, \dots, 1] \end{aligned}$$

Łatwo widać, że wektory dla dowolnego doboru indeksów  $j_2, j_3, \dots, j_n$  wektory  $\{v_{j_2}^2, v_{j_3}^3, \dots, v_{j_n}^n\}$  są liniowo niezależne. Wybierając przesunięcie  $P_1 = (p, 0, \dots, 0)$  na osi  $x_1$ , możemy zdefiniować hiperpłaszczyznę  $(P_1, v_{j_2}^2, v_{j_3}^3, \dots, v_{j_n}^n)$  przez:

$$H = \{(x_1, x_2, \dots, x_n) \in \mathbb{R}^n : x_1 - \alpha_{j_2}^2 x_2 - \alpha_{j_3}^3 x_3 - \dots - \alpha_{j_n}^n x_n - p = 0\}$$

Osobniki będziemy reprezentować przez  $n - 1$  liczb  $b$ -bitowych  $\alpha_1, \alpha_2, \dots, \alpha_{n-1}$ , które będą reprezentować wektory (dla osi  $x_1$ ):

$$\begin{aligned} v_1 &= [\alpha_1, 1, 0, 0, \dots, 0] \\ v_2 &= [\alpha_2, 0, 1, 0, \dots, 0] \\ &\dots \\ v_{n-1} &= [\alpha_{n-1}, 0, 0, 0, \dots, 1] \end{aligned}$$

Są one liniowo niezależne i rozpinają  $n - 1$  wymiarową podprzestrzeń liniową. Taka reprezentacja umożliwia nam sprawdzenie w czasie  $O(n)$  po której stronie hiperpłaszczyzny znajduje się dany obiekt  $u \in U$  oraz policzyć w czasie  $O(n)$  rzut wektora  $u$  na  $x_1$  równoległy do  $L = \text{lin}(v_{j_2}^2, v_{j_3}^3, \dots, v_{j_n}^n)$ . Odpowiednie funkcje,  $Test(u)$  licząca, po której stronie hiperpłaszczyzny jest obiekt  $u$ :

$$Test(u) = \begin{cases} 1 & \text{jeśli } a_1(u) - \alpha_{j_2}^2 a_2(u) - \alpha_{j_3}^3 a_3(u) - \dots - \alpha_{j_n}^n a_n(u) \geq p \\ 0 & \text{wpp} \end{cases}$$

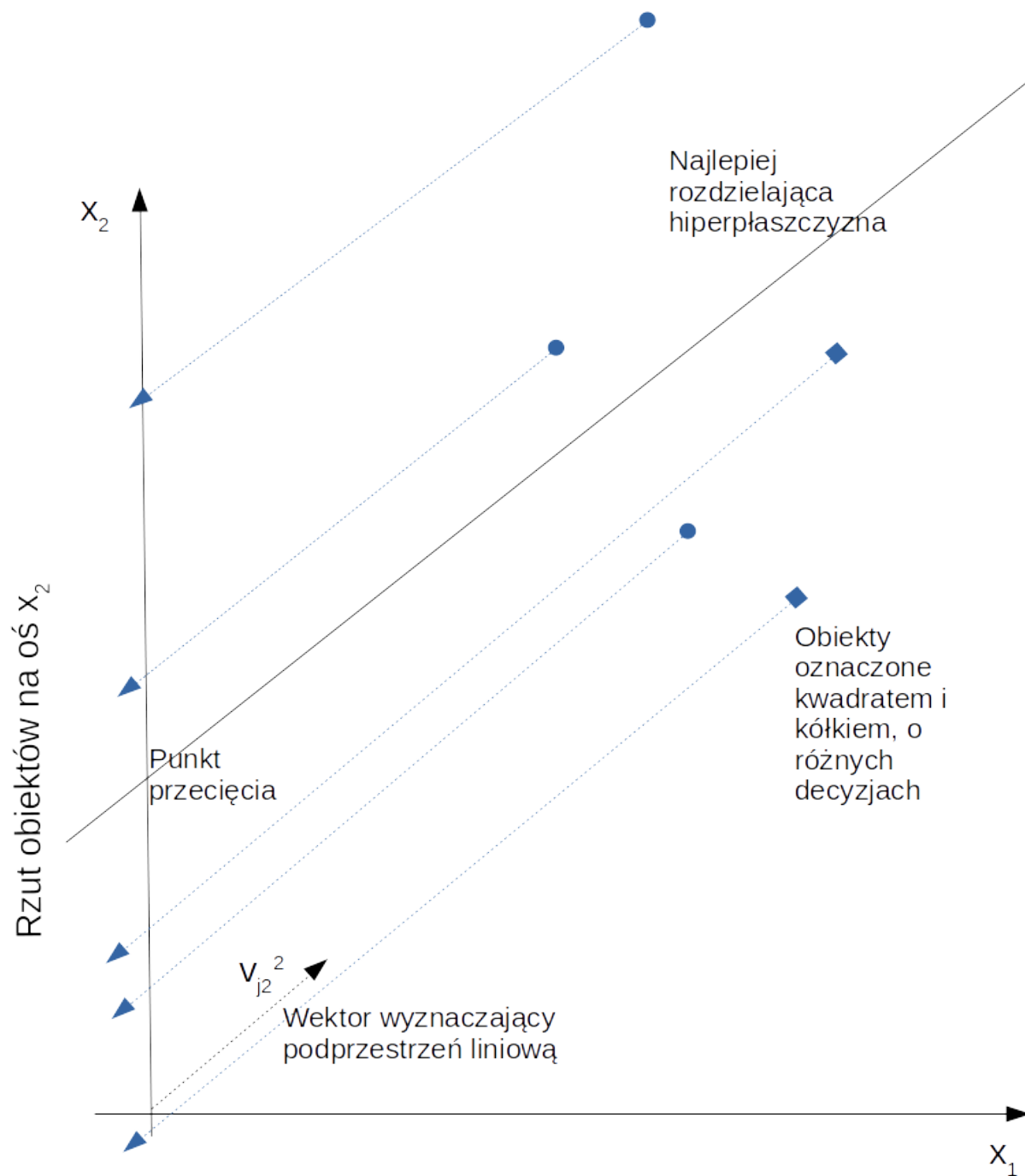
Oraz  $Projection(u)$  licząca rzut obiektu  $u$  na odpowiednią oś (tu  $x_1$ ):

$$Projection(u) = a_1(u) - \alpha_{j_2}^2 a_2(u) - \alpha_{j_3}^3 a_3(u) - \dots - \alpha_{j_n}^n a_n(u)$$

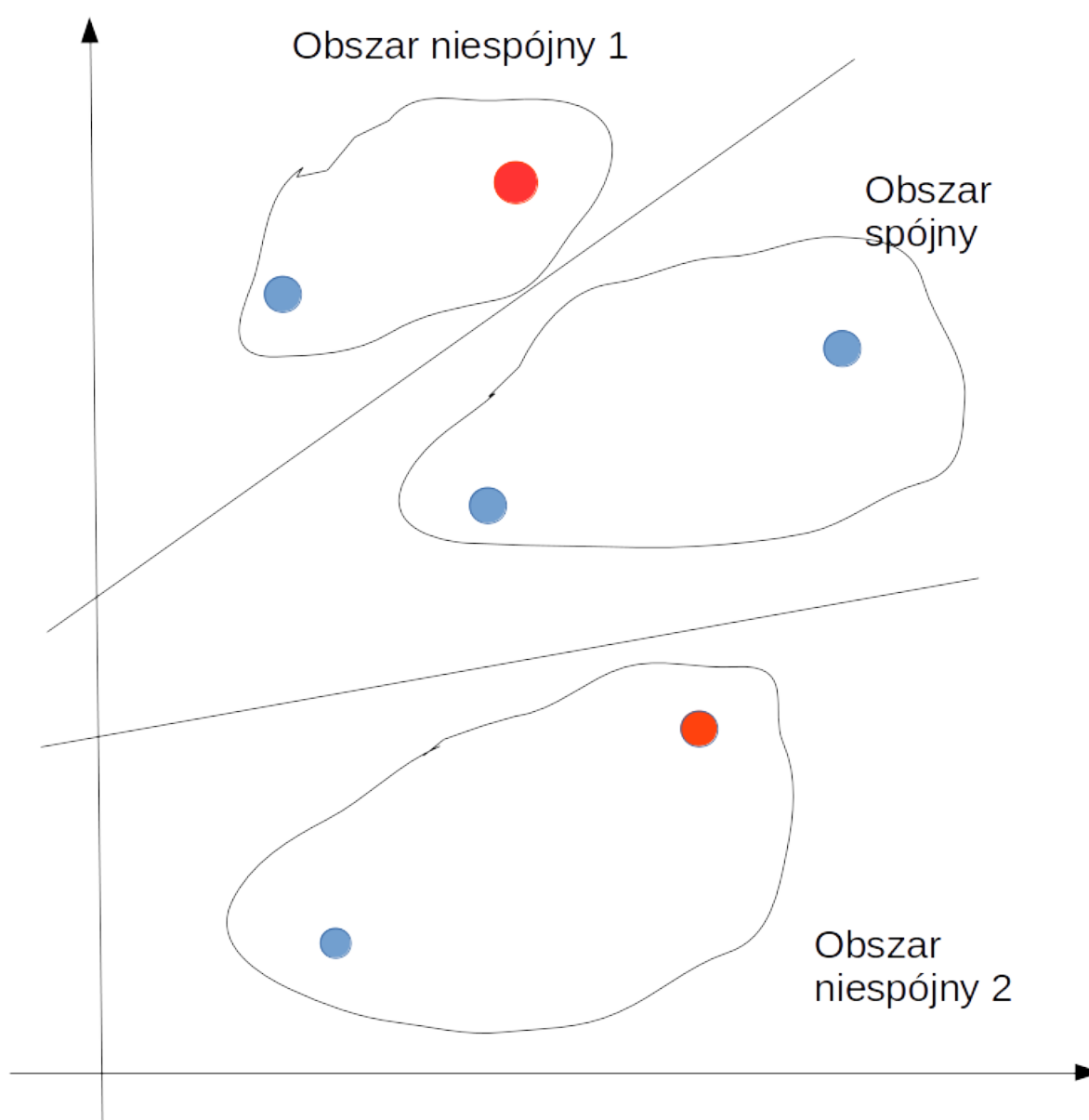
### 5.2.2. Szukanie optymalnej hiperpłaszczyzny za pomocą algorytmu genetycznego

Za pomocą algorytmu genetycznego będziemy chcieli szukać optymalnej hiperpłaszczyzny dla danej osi. Dla każdego osobnika liczymy rzut obiektów na odpowiednią oś za pomocą funkcji **Projection**, a następnie wybieramy najlepszą hiperpłaszczyznę równoległą do podprzestrzeni liniowej reprezentowanej przez osobnika za pomocą funkcji oceny. Na rysunku 4.1 widzimy to dla przestrzeni dwuwymiarowej. Powtarzając algorytm dla każdej z osi, wybierzemy najlepszą hiperpłaszczyznę. Hiperpłaszczyzny szukamy do momentu, w którym tabela otrzymana z nowych cech indyktorów odpowiednich półprzestrzeni będzie spójna, bądź spełniony zostanie warunek stopu, o którym opowiem w następnych rozdziałach. Na rysunku 4.2 widac przykładowy podział obiektów w przestrzeni dwuwymiarowej po znalezieniu dwóch hiperpłaszczyzn. Po znalezieniu kolejnych hiperpłaszczyzn grupy obszarów niespójnych będą się stale zmniejszać. Funkcję liczę jedynie dla obszarów niespójnych, które nie są jeszcze podzielone. Funkcję celu dla listy obszarów niespójnych wyliczam z wzoru (kod w języku python):

Rysunek 5.1: Interpretacja szukania optymalnej hiperpłaszczyzny w przestrzeni dwuwymiarowej



Rysunek 5.2: Podział obiektów przez hiperpłaszczyzny



```
# inconsistent_groups jest lista niespojnych grup obiektow
award = sum([discMeasure(H, U) for U in inconsistent_groups])
```

Kod algorytmu w języku python:

```
# dla wszystkich funkcji:

# dec <- lista decyzji
# table <- tabela
# attrs_list <- lista atrybutow

# ekstrakcja nowych cech, generowanie nowej tabeli
def extract_features(sc=None):
    extracted_table = []
    while True:
        inconsistent_groups = ConsistentChecker.
            count_inconsistent_groups(
                extracted_table, dec, sc)
        # kompresja niespojnych obszarow, np wyrzucanie
        # bliskich punktow o roznych decyzjach
        inconsistent_groups = filter(
            lambda x:
                dpoints_strategy.decision(x) is None,
            inconsistent_groups)
        # jesli sa jeszcze jakies niespojne grupy szukamy hiperplaszczyn
        if inconsistent_groups:
            best_hyperplane = search_best_hyperplane(
                inconsistent_groups, sc)
            extracted_table.append(
                count_objects_positions(best_hyperplane))
        else:
            break

    return np.transpose(np.array(extracted_table))

# szukanie optymalnej hiperplaszczyny
def search_best_hyperplane(inconsistent_groups, sc=None):
    rdd_attributes = sc.parallelize(attrs_list)
    # zrownoleglamy ze wzgledu na osie (atrybuty)
    # i liczymy optymalne hiperplaszczyny dla rzutow
    hyperplanes = rdd_attributes.map(
        lambda x:
            search_best_hyperplane_for_projection(
                x, inconsistent_groups)).collect()

    # zwracamy najlepsza
    return max(hyperplanes, key=lambda x: x[1][0])

# szukanie optymalnej hiperplaszczyny dla rzutu
def search_best_hyperplane_for_projection(
    attr, inconsistent_groups, sc=None):
    projection_axis = table[attr]
    other_axes = [x for x in attrs_list if not x == attr]
    new_table = table[other_axes]

    # szukamy optymalnej hiperplaszczyny algorytmem genetycznym
    gen_search = GeneticSearch(len(other_axes), dec, new_table,
                                projection_axis, inconsistent_groups)
    cand_hyperplane = gen_search.genetic_search(sc)

    return attr, cand_hyperplane
```

```

# inconsistent_groups <- lista niespojnych grup
# population_size <- rozmiar populacji
# przeszukiwanie genetyczne
def genetic_search(sc=None):

# inicjowanie populacji
    population = init_generation()

    current_best_award = 0
    the_same_awards = 0
    for i in range(max_iter):
# liczymy funkcje celu dla populacji
        rdd_population = sc.parallelize(population,
                                         population_size * 10)
        awards = rdd_population.mapPartitions(
                                count_award_for_chunk).collect()

# wybieramy najlepszych osobnik w
        population_awards = select_best_individuals(awards)

        best_individual = population_awards[0]

        population = map(lambda x: x[1], population_awards)

        new_generation = count_new_generation(
                        copy.deepcopy(population))

        population = new_generation + population

    return best_individual

def count_award_for_chunk(self, population):
    for individual in population:
        yield self.count_award(individual)

def count_award(individual):
    all_objects = reduce(add, inconsistent_groups)
# liczymy rzuty wszystkich obiektow
    projections = count_projections(individual, all_objects)
# sortujemy po wartosci rzutu
    objects = sorted(projections, key=lambda x: x[1])

# liczenie funkcji oceny (discMeasure) dla posortowanych obiektow
    object_group_dict = {}
    group_fqs_dict = {}
    act_left_sum = {}
    act_right_sum = {}

    for ind, group in enumerate(inconsistent_groups):
        group_decisions = []
        for obj in group:
            group_decisions.append(self.dec[obj])
            object_group_dict[obj] = ind
            act_left_sum[ind] = 0
            act_right_sum[ind] = len(group_decisions)
            group_fqs_dict[ind] = DiscMeasureCalculator.
                                prepare_hist(group_decisions)

    act_award = 0
    max_award = 0
    for obj, proj in objects:
        group_id = object_group_dict[obj]

```

```

        dec = self.dec[obj]
        act_left_sum[group_id], act_right_sum[group_id], act_award = \
            DiscMeasureCalculator.
            update_award(dec, act_left_sum[group_id],
                        act_right_sum[group_id],
                        group_fqs_dict[group_id], act_award)
        if act_award > max_award:
            max_award = act_award
            good_proj = proj

# zwracamy wartosc najlepszego wyniku, osobnika (wektory) i przesuniecie
return max_award, individual, good_proj

# wyliczanie rzutu dla osobnika i grupy obiektow
def count_projections(individual, objects):
    table = table_as_matrix(objects, :)
    proj = [projection_axis[obj] for obj in objects]
    return zip(objects, proj - table.dot(individual))

```

### 5.2.3. Analiza złożoności czasowej algorytmu

W skrócie algorytm można przedstawić za pomocą pseudokodu:

**Data:** Tabela decyzyjna  $\mathbb{S}$   
 Zbiór obiektów  $\mathbb{U}$   
 Zbiór atrybutów  $\mathbb{A}$   
 Lista niespójnych grup obiektów  $UncGroups$   
 Nowa tabela  $\mathbb{B}$   
**Result:** Nowa tabela  $\mathbb{B}$  z nowymi cechami  
 $B = \emptyset$ ;  
 $UncGroups = \mathbb{U}$ ;  
 $H = \emptyset$ ;  
**while**  $UncGroups \neq \emptyset$  **do**  
     **for**  $atr$  **in**  $\mathbb{A}$  **do**  
          $H = H \cup$  najlepsza hiperpłaszczyzna znaleziona za pomocą algorytmu  
         genetycznego dla osi odpowiadającej atrybutowi  $atr$  i grupom obiektów z  
          $UncGroups$   
     **end**  
**end**  
 $OptH =$  najlepsza hiperpłaszczyzna z zbioru  $H$ .  
 $B = B \cup$  cecha wyznaczona przez  $OptH$ .  
 Aktualizuj  $UncGroups$  na podstawie  $B$

Koszt czasowy zewnętrznej pętli while ciężko jest oszacować, gdyż płaszczyzna wybierana jest w sposób niedeterministyczny. Możemy ustalić pewien próg czasowy, w którym algorytm musi się zatrzymać jeśli  $UncGroups$  wciąż jest niepuste. Koszt czasowy pętli wewnętrznej wynosi  $O(n \cdot gen)$ , gdzie  $gen$  to koszt czasowy algorytmu genetycznego. Przyjrzyjmy się pseudokodowi algorytmu genetycznego oraz wyznaczania funkcji oceny:



**Data:** Populacja  $P$   
maksymalna liczba iteracji  $max\_iter$   $i = 0$ ;  
 $P = init\_generation$ ;  
**while**  $i \leq max\_iter$  **do**  
     $i = i + 1$   
    **for**  $Osobnik \in P$  **do**  
         $Oceny = Oceny \cup$  wartość oceny dla  $Osobnik$   
    **end**  
    Wybierz najlepszych osobników.  
    Skrzyżuj i mutuj najlepsze osobniki.  
     $P =$  Zainicjuj nowe pokolenie.  
**end**  
zwróć najlepszego osobnika.

**Algorytm 1:** przeszukiwanie genetyczne

**Data:** Osobnik  $o$   
Niespójne grupy obiektów  $UncGroups$   
rzuty = oblicz rzuty obiektów z  $UncGroups$  na odpowiednią oś równoległą do podprzestrzeni wyznaczonej przez  $o$ .  
sortuj rzuty.  
oceny = licz wartość  $DiscMeasure$  dla wszystkich rzutów.  
zwróć najlepszą ocenę.

**Algorytm 2:** wyznaczanie funkcji oceny

Złożoność czasowa wyznaczania rzutu dla obiektu wynosi  $O(n)$ , zaś liczenia funkcji oceny dla wszystkich rzutów  $O(k)$ , gdzie  $k$  jest liczbą obiektów (dla posortowanych obiektów aktualizujemy w czasie stałym wartość **DiscMeasure**). A więc koszt czasowy wyznaczania funkcji oceny wynosi  $O(n \cdot k + k \cdot \log(k) + k) = O(k(\log(k) + n))$ . Koszt czasowy algorytmu genetycznego wynosi więc  $O(max\_iter \cdot |P| \cdot k(\log(k) + n))$ . A więc koszt szukania pojedynczej hiperpłaszczyzny wynosi  $O(n \cdot max\_iter \cdot |P| \cdot k(\log(k) + n))$  i zależy od doboru parametrów  $|P|$  i  $max\_iter$ . W idealnym modelu obliczeń równoległych, gdy mamy do dyspozycji  $K$  komputerów koszt czasowy wyniesie:  $O(\frac{n}{K} \cdot max\_iter \cdot \frac{|P|}{K} \cdot k(\log(k) + n))$ , a więc dla dostatecznie dużych  $K$  dostajemy:  $O(max\_iter \cdot k(\log(k) + n))$ .

### 5.3. Indukcja drzewa decyzyjnego na podstawie cięć przez hiperpłaszczyzny

Optymalne hiperpłaszczyzny mogą być użyte nie tylko ekstrakcji nowych cech, ale również do indukcji drzewa decyzyjnego. W standardowym drzewie, w węźle mamy funkcję *test* odpowiadającą atrybutom, dla wartości ciągłych zwykle jest to:

$$test(x) = \begin{cases} 1 & \text{jeśli } a_i(x) > c \\ 0 & \text{wpp} \end{cases}$$

Możemy jednak użyć funkcji testu opartej o hiperpłaszczyznę, wtedy drzewo będzie miało zwykle mniejszą głębokość (hiperpłaszczyzny mogą mieć wyższą wartość funkcji oceny na danym węźle), ale większy jest koszt czasowy predykcji (funkcja testu działa w czasie  $O(n)$  gdzie  $n$  jest liczbą atrybutów) oraz szukania drzewa. Funkcja testu dla uogólnionego drzewa (dla wag  $w_i$ ):

$$test(x) = \begin{cases} 1 & \text{jeśli } w_1 a_1(x) + \dots + w_n a_n(x) > c \\ 0 & \text{wpp} \end{cases}$$

Oraz funkcja dla testu dla reprezentacji hiperplaszczyny z algorytmu genetycznego (gdy  $x_1$  jest najlepszą osią):

$$test(x) = \begin{cases} 1 & \text{jeśli } a_1(x) - \alpha_{j_2}^2 a_2(x) - \alpha_{j_3}^3 a_3(x) - \dots - \alpha_{j_n}^n a_n(x) \geq p \\ 0 & \text{wpp} \end{cases}$$

Algorytm znajdowania drzewa przez algorytm genetyczny w języku python:

```
def count_decision_tree(objects, sc=None):
    # jesli mozemy podjac decyzje dla danego zbioru obiektow
    # zwracamy koncowy wezel
    decision = dpoints_strategy.decision(objects)
    if decision is not None:
        return DecisionTree(decision, 0, 0)

    # szukamy najlepszej hiperplaszczyny algorytmem genetycznym
    # (tak jak dla ekstrakcji cech)
    best_hyperplane = search_best_hyperplane([objects], sc)
    hyperplane_indicator = count_objects_positions(
        best_hyperplane, objects)
    left_son_objects = map(
        lambda x: x[1],
        filter(
            lambda (i, x):
                hyperplane_indicator[i] == 0,
            enumerate(objects)))
    right_son_objects = map(
        lambda x: x[1],
        filter(
            lambda (i, x):
                hyperplane_indicator[i] == 1,
            enumerate(objects)))

    return DecisionTree(best_hyperplane,
        count_decision_tree(left_son_objects),
        count_decision_tree(right_son_objects))
```

### 5.3.1. Generowanie lasu losowego

Jako, że w algorytmie genetycznym początkowe wartości wektorów dobierane są w sposób losowy, drzewa decyzyjne generowane kilkakrotnie przez algorytm genetyczny mogą się od siebie różnić. Przy jednokrotnym generowaniu drzewa istnieje prawdopodobieństwo utknięcia w lokalnym maksimum funkcji celu. Możemy więc stworzyć wiele drzew i podjąć decyzję na podstawie głosowania, tworząc coś na kształt lasu losowego. Algorytm w języku python:

```
# objects <- lista wszystkich obiektow w tabeli
def count_random_forest(size=50):
    return [count_decision_tree(objects) for _ in range(size)]

def predict(object):
    forest = count_random_forest()
    # liczymy predykcje dla kazdego drzewa
    decisions = map(lambda tree: tree.predict(object), forest)
    # zwracamy najczesciej wystpujaca decyzje
    return most_common(Counter(decisions))
```

## 5.4. Drzewo SVM

Dla celów analizy porównawczej możemy stworzyć drzewo, w którym funkcją testu również będzie funkcja charakterystyczna półprzestrzeni, ale odpowiednią hiperplaszczynę będzie-

my szukać nie za pomocą algorytmu genetycznego, ale algorytmu SVM, szukającego hiperpłaszczyzny maksymalizującej sumę odległości od obiektów. Rysunek 4.3 ilustruje działanie algorytmu SVM. Kod algorytmu w języku python:

```
def count_decision_tree(objects, sc=None):
    decision = dpoints_strategy.decision(objects)
    if decision is not None:
        return DecisionTree(decision, 0, 0)

    X = [self.table[i] for i in objects]
    y = [self.dec[i] for i in objects]
    svm = LinearSVC()
    # wyliczanie SVM z zewnętrznej biblioteki
    svm.fit(X, y)
    # wybieranie współczynników optymalnej hiperpłaszczyzny
    coefs = svm.coef_[0]
    intercept = svm.intercept_[0]
    best_hyperplane = (intercept, coefs)
    hyperplane_indicator = map(
        lambda r:
            (np.dot(
                list(r), coefs) + intercept > 0),
        X)

    left_son_objects = map(
        lambda x: x[1],
        filter(
            lambda (i, x):
                not hyperplane_indicator[i],
            enumerate(objects)))

    right_son_objects = map(
        lambda x: x[1],
        filter(lambda (i, x):
            hyperplane_indicator[i],
            enumerate(objects)))

    return DecisionTree(best_hyperplane,
                        count_decision_tree(left_son_objects),
                        count_decision_tree(right_son_objects))
```

## 5.5. Strategie klasyfikacji bliskich punktów o różnej decyzji oraz warunek oraz przycinanie

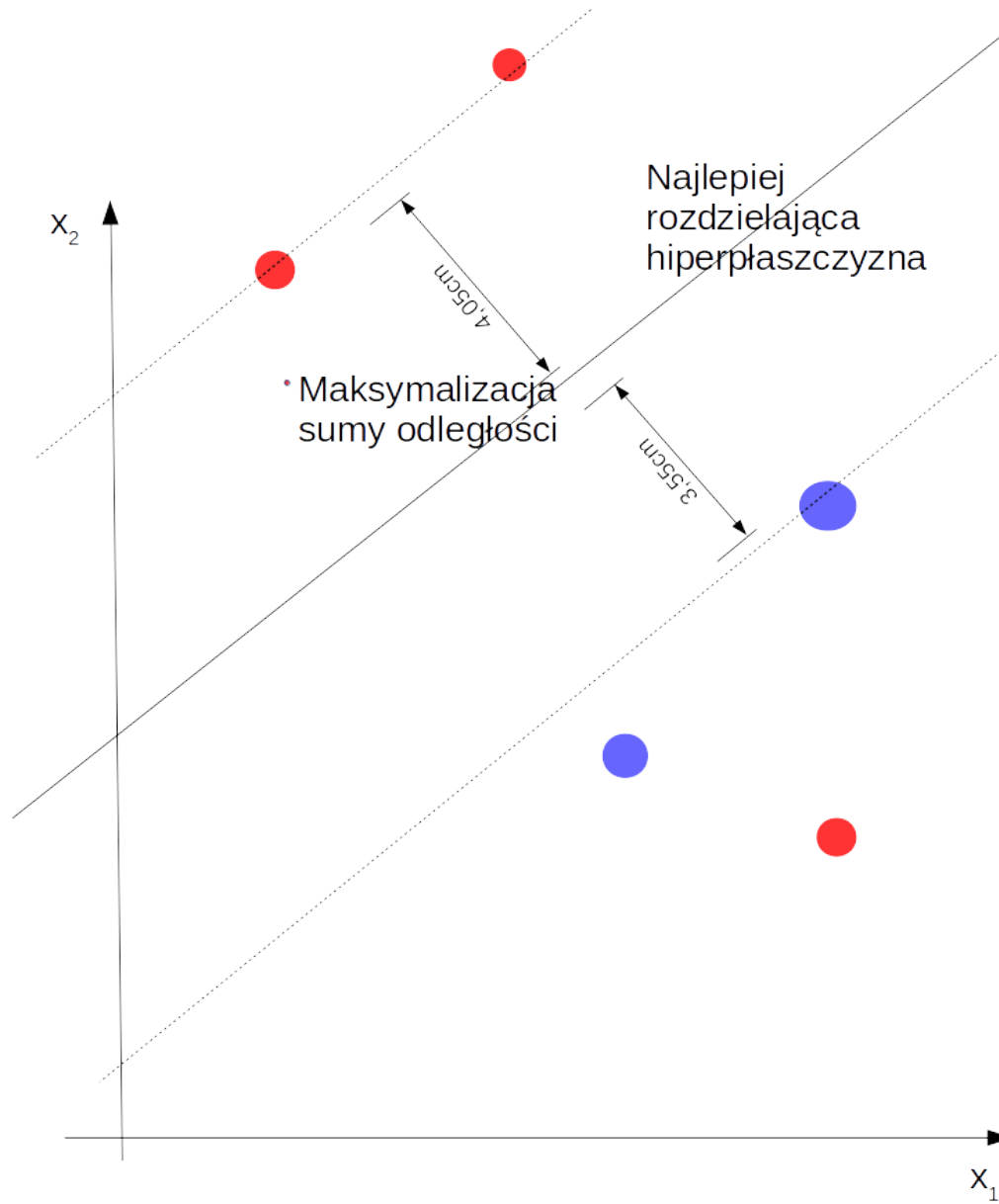
Zarówno w algorytmie indukcji drzewa decyzyjnego jak i w algorytmie szukania nowych cech pojawiła się funkcja `dpoints_strategy.decision(objects)`. W przypadku gdy pojawiają się punkty będące “blisko” siebie mające różne decyzje algorytmowi genetycznemu ciężko będzie “wciśnąć” się pomiędzy te punkty przy losowej optymalizacji. Dlatego w przypadku indukcji drzewa wyliczamy decyzję dla takich punktów w inny, deterministyczny sposób, natomiast przy generowaniu cech takie punkty ignorujemy. Pytanie jakie się nasuwają są dwa:

1. Kiedy uznawać, że punkty są bliskie?
2. W jaki sposób podejmować decyzję dla takich punktów?

W swojej pracy uznaję punkty za bliskie, gdy średnica zbioru jest mniejsza niż pewien procent średnicy całego zbioru (problem pojawia się gdy występują outliery, wtedy można liczyć średnicę bez outlierów). Decyzją zaś jest najczęściej występująca decyzja w danym zbiorze. Kod algorytmu w języku python:

```
|| # max_dist <- srednia calego zbioru obiektow
```

Rysunek 5.3: Podział obiektów przez hiperpłaszczyznę SVM



```

def decision(objects, ratio):
    max_dist_objects = count_max_dist(objects)
    if max_dist_objects / max_dist < ratio:
        return most_common(Counter(decisions(objects)))
    return None

```

Dodatkowo, nawet jeśli punkty nie będą blisko siebie i algorytm genetyczny wyszuka optymalne płaszczyzny może wystąpić ryzyko nadmiernego dopasowania do danych i przeuczenia. Jedną ze momentu stopu użytych w pracy jest odcięcie w momencie, gdy najliczniejsza decyzja stanowi więcej niż pewien ustalony odsetek. W eksperymentach dla generowania nowych cech użyłem właśnie tej strategii odcięcia.

## 5.6. Wyniki eksperymentów

Eksperymenty dla tej części również wykonywałem w laboratorium MIMUW przy użyciu takiego samego zestawu komputerów oraz dla zestawu danych WAVEFORM używanym wcześniej przy dyskretyzacji.

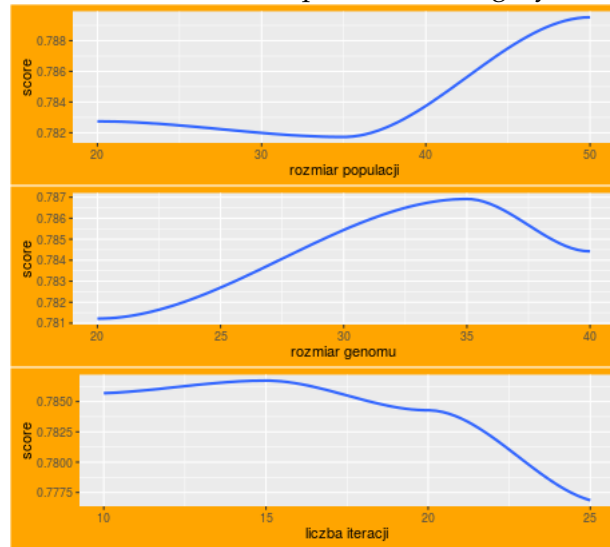
### 5.6.1. Generowanie nowych cech

Przetestowałem jak skuteczność oraz czas trwania algorytmu genetycznego w zależności od parametrów, którymi są:

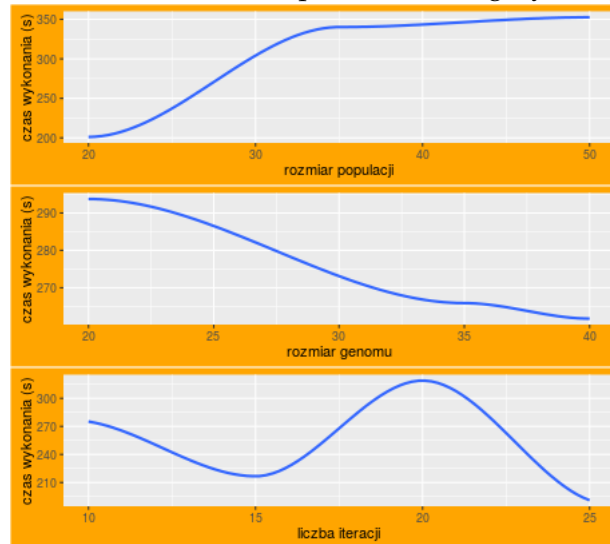
- rozmiar genomu
- rozmiar populacji
- maksymalna liczba iteracji

Jako strategię klasyfikacji bliskich punktów o różnej decyzji wybrałem zatrzymanie przeszukiwania w momencie gdy mamy 0.8 całości obiektów o jednej decyzji. Wyniki można zobaczyć na wykresach:

Rysunek 5.4: Score w zależności od parametrów algorytmu genetycznego



Rysunek 5.5: Czas w zależności od parametrów algorytmu genetycznego

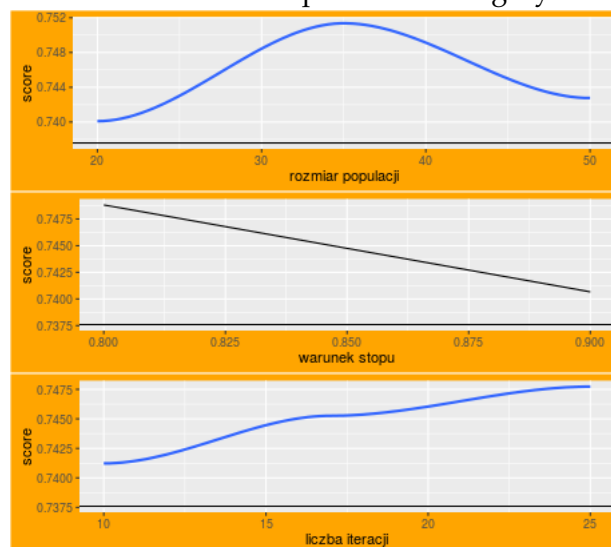


Widzimy, że zwiększając liczbę populacji uzyskujemy lepsze wyniki, jednak odbywa się to kosztem czasu przetwarzania. Jeśli chodzi o maksymalną liczbę iteracji, już przy 15 iteracjach algorytm zbiega do dobrego rozwiązania, zwiększając liczbę iteracji, uciekamy od rozwiązania optymalnego, zwiększając jeszcze czas przetwarzania. Również rozmiar genomu wpływa na czas przetwarzania, im większy tym szybciej uzyskujemy rozwiązanie, a więc lepiej ustalić większy rozmiar genomu. Podsumowując wyniki eksperymentów, najlepsze wyniki uzyskaliśmy dla dużych rozmiarów genomów - 20, małej liczby iteracji - 15 i dużego rozmiaru populacji - 50.

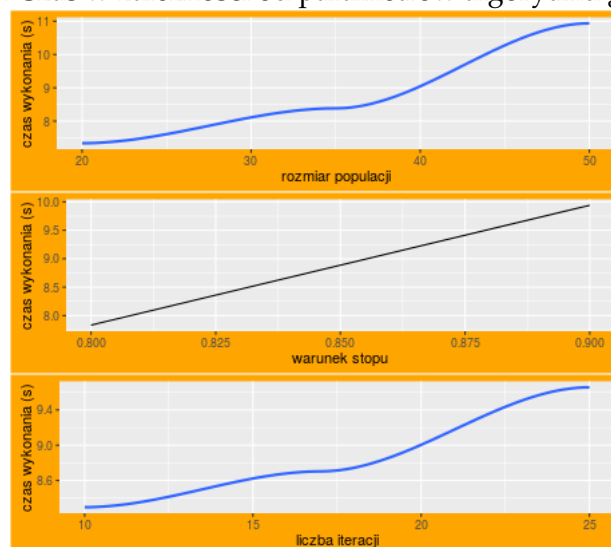
### 5.6.2. Generowanie drzewa decyzyjnego

Podobne eksperymenty przeprowadziłem dla generowania drzewa decyzyjnego przez cięcia hiperpłaszczyznami. Ustaliłem rozmiar genomu na 35 i wprowadziłem parametr mówiący o tym, w którym momencie odcinamy przeszukiwania dla danego węzła: gdy 0.8 obiektów ma taką samą decyzję lub gdy tych obiektów jest 0.9. Brązową kreską został zaznaczony wynik jaki uzyskałem drzewem decyzyjnym z biblioteki sklearn. Otrzymałem następujące wyniki:

Rysunek 5.6: Score w zależności od parametrów algorytmu genetycznego



Rysunek 5.7: Czas w zależności od parametrów algorytmu genetycznego



Widzimy, że zarówno wyniki jak i czas pogarszają się gdy zwiększymy parametr warunku stopu. Przy zbyt dużym parametrze, szukamy płaszczyzn dłużej, co może prowadzić do nadmiernego dopasowania do danych, i w efekcie przeuczenia. W przeciwieństwie do algorytmu generowania nowych cech, tutaj zwiększając liczbę iteracji uzyskujemy lepsze wyniki, zaś optymalnym rozmiarem populacji będzie w tym przypadku 30. W przypadku generowania drzewa mamy prostszą funkcję oceny niż w przypadku generowania nowych cech, dlatego prawdopodobnie potrzebujemy mniejszego rozmiaru populacji.

### 5.6.3. Porównanie wyników

Oto porównanie najlepszych wyników jakie uzyskałem dla zbioru WAVEFORM przy zastosowaniu 5-krotnej krosvalidacji. Wyszukiwanie genetyczne odbywało się na 40 komputerach.

Metoda/parametr	Drzewo decyzyjne	Drzewo decyzyjne SVM	Drzewo decyzyjne genetycznie	Drzewo decyzyjne + nowe cechy
Score (accuracy)	0.7376	0.7482	0.758	0.790
Czas (s)	0.012	0.041	6.04	191.00

Widzimy, że najlepsze wyniki uzyskujemy generując nowe cechy. Drzewa decyzyjne uzyskują niewiele lepsze wyniki niż zwykłe drzewo decyzyjne z biblioteki sklearn. Jednak drzewa wygenerowane metodą SVM lub algorytmem genetycznym ze względu na cięcia hiperpłaszczyznami mają mniejsze głębokości, spójrzmy jak wyglądały średnie głębokości dla drzew wygenerowanych przy krosvalidacji.

Metoda/parametr	Drzewo decyzyjne	Drzewo decyzyjne SVM	Drzewo decyzyjne genetycznie
średnia głębokość drzewa	6.4	3.0	2.4

Drzewa decyzyjne wygenerowane algorytmem genetycznym pozwalają więc osiągnąć najlepszą skuteczność przy jednoczesnej największej prostocie drzewa. Odbywa się to jednak kosztem złożoności czasowej, jednak dzięki użyciu Sparka czas generowania drzewa nie jest wiele większy niż w przypadku zwykłych drzew. Zdecydowanie najlepsze wyniki uzyskujemy poprzez dodanie nowych cech. Czas wyszukiwania jest zdecydowanie najdłuższy, jednak ustalając parametr stopu, możemy zdecydować o kompromisie pomiędzy niedouczeniem/mniejszym czasem wykonania a przeuczeniem.



# Bibliografia

- [1] Hun Son Nguyen, *Approximate Boolean Reasoning: Foundations and Applications in Data Mining*. Institute of Mathematics, Warsaw University, 1998.
- [2] Hun Son Nguyen, *From Optimal Hyperplanes to Optimal Decision Trees*. Institute of Mathematics, Warsaw University, 1998.
- [3] Srilatha Chebrolu, Sriram G Sanjeevi, *Rough set theory for discretization based on boolean reasoning and genetic algorithm*. Department of Computer Science and Engineering, NIT Warangal, India, 2012.
- [4] Spark Documentation, <http://spark.apache.org/docs/latest/programming-guide.html>