

Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

**OPTYMALIZACJA KLASYFIKATORA SVM ZA POMOCĄ PROGRAMOWANIA
GENETYCZNEGO**

Tomasz Ziętkiewicz

Promotor
dr hab. Krzysztof Krawiec

Poznań, 2013

Spis treści

1	Wprowadzenie	1
1.1	Cel i zakres pracy	1
1.2	Struktura pracy	1
2	Podstawy teoretyczne	2
2.1	Uczenie maszynowe	2
2.1.1	SVM	3
2.2	Obliczenia ewolucyjne	3
2.2.1	Programowanie genetyczne	3
2.3	Ewolucja kerneli	3
3	Algorytm Kernel GP	4
3.1	Opis algorytmu	4
3.1.1	Inicjalizacja populacji	4
	Generowanie funkcji	5
3.1.2	Ewaluacja kerneli	7
3.1.3	Selekcja	7
3.1.4	Krzyżowanie i mutacja	7
3.1.5	Walidacja rozwiązania	7
3.2	Implementacja	7
3.3	Złożoność obliczeniowa	8
4	Wyniki działania algorytmu na popularnych zbiorach danych	9
4.1	Metodologia pomiarów	9
4.2	Opis zbiorów danych	10
4.3	Trafność klasyfikacji	10
	Monotoniczność funkcji trafności	10
4.3.1	Porównanie z tradycyjnym algorytmem SVM	18
4.4	Czas wykonania	21
4.5	Użycie pamięci	21
4.6	Podsumowanie wyników	21
5	Case study - klasyfikacja danych ADHD 200	22
5.1	Opis zbioru danych	22
5.1.1	Surowe dane	22
5.1.2	Preprocessing	23
5.2	Konstrukcja i selekcja cech	23
5.3	Wyniki klasyfikacji	23

5.3.1	Kernel GP	23
5.3.2	Porównanie z innymi algorytmami	23
6	Podsumowanie	24
	Literatura	25
	Zasoby internetowe	26

Rozdział 1

Wprowadzenie

1.1 Cel i zakres pracy

Niniejsza praca ma dwa podstawowe cele:

- Stworzenie algorytmu programowania genetycznego optymalizującego parametry klasyfikatora SVM
- Zastosowanie stworzonego algorytmu do klasyfikacji danych ze zbioru ADHD-200

Realizacja drugiego z powyższych celów służyć ma przede wszystkim sprawdzeniu efektywności stworzonego algorytmu, ale jest też wyzwaniem samym w sobie. Zbiór danych ADHD-200 nie poddaje się łatwo klasyfikacji za pomocą metod uczenia maszynowego, dlatego każda poprawa wyników klasyfikacji względem wyników dotychczas osiągniętych będzie sporym sukcesem.

1.2 Struktura pracy

Struktura pracy jest następująca: rozdział drugi przedstawia ważniejsze zagadnienia teoretyczne związane z pracą oraz zawiera przegląd literatury. W rozdziale trzecim opisano zaimplementowany algorytm Kernel GP oraz przedstawiono sposób jego implementacji. Rozdział czwarty przedstawia wyniki działania algorytmu na standardowych zbiorach danych używanych do testowania algorytmów maszynowego uczenia. W rozdziale piątym prezentowane są wyniki działania algorytmu na zbiorze ADHD-200. Rozdział szósty zawiera podsumowanie.

Rozdział 2

Podstawy teoretyczne

2.1 Uczenie maszynowe

Uczenie maszynowe (ang. *Machine Learning*) to dziedzina informatyki zajmująca się konstruowaniem *systemów uczących się* [KS03]. Zdefiniowanie *systemu uczącego* nie jest zadaniem trywialnym, jednak wydaje się, że podstawową cechą takich systemów jest to, że potrafią one dostosowywać sposób swojego działania do danych wejściowych, na których operują. Zmiana działania systemu może mieć różną skalę - od zmiany pojedynczych parametrów programu, przez zapamiętywanie informacji po zmianę wykonywanego algorytmu. Niezależnie od skali każda taka zmiana powinna mieć wpływ na jego przyszłe działanie i powinna mieć na celu uzyskanie jak najwyższej *oceny* pracy systemu. Funkcja oceny uczącego się systemu jest od niego niezależna, ale musi mieć on dostęp do jej wartości. Dostosowanie sposobu działania, nie zaś samego działania, w ten sposób, żeby zostało ono jak najwyżej ocenione wydaje się być cechą charakterystyczną systemów uczących się. Wszak przecież nawet prosty kalkulator dostosowuje swoje działanie (wyświetlany wynik) do danych wejściowych, jednak sposób działania pozostaje niezmienny - po wprowadzeniu tych samych danych wejściowych zawsze otrzymamy ten sam wynik. Gdyby w arytmetyce nastąpił nagle przełom, kalkulator nie potrafiłby dostosować sposobu swojego działania do nowych reguł arytmetyki, przez co zostałby albo zmodyfikowany przez człowieka, albo po prostu wyrzucony. Porównanie systemów uczących się do kalkulatora nie jest bynajmniej tak abstrakcyjne jak mogłoby się wydawać - ostatecznie najbardziej wyrafinowany system uczący też wykonuje pewną deterministyczną funkcję na danych wejściowych, jednak proces obliczania wyniku może być rozłożony w czasie - jako dane wejściowe można traktować zarówno dane użyte do uczenia systemu, jego oceny jak i dane aktualnie do niego wprowadzane.

Systemy uczące się mają wiele zastosowań, między innymi:

- Rozpoznawanie mowy ludzkiej
- Rozpoznawanie tekstu pisanego (OCR, ang. *Optical Character Recognition*)
- Diagnostyka medyczna
- Klasyfikacja tekstów, np. na potrzeby filtrowania niechcianych wiadomości
- Automatyczna identyfikacja zagrożeń na podstawie obrazu z kamer przemysłowych
- Kierowanie autonomicznymi pojazdami
- Prognozowanie pogody
- Prognozowanie zmian kursów akcji na giełdzie

- Wykrywanie podejrzanych transakcji finansowych
- Biometria - identyfikacja ludzi na podstawie cech takich jak głos, wygląd twarzy, odciski palców, sposób chodzenia
- Wspomaganie podejmowania decyzji

Jednym z rodzajów systemów uczących są systemy klasyfikujące....

- *zbiór uczący*
- *zbiór testujący*
- *zbiór walidujący*
- *walidacja krzyżowa*
- *trafność*

2.1.1 SVM

Maszyna wektorów wspierających (SVM, ang. Support Vector Machine)

- *funkcja jądrowa*
- *wektor wspierający*
- *hiperpłaszczyzna*
-
-

2.2 Obliczenia ewolucyjne

- *populacja*
- *osobnik*
- *mutacja*
- *krzyżowanie*
- *selekcja*
- *funkcja przystosowania (ang. fitness)*

2.2.1 Programowanie genetyczne

Programowanie genetyczne (GP, ang. Genetic Programming) to

Funkcje, które generuje algorytm programowania genetycznego są w nim reprezentowane w postaci drzew. Węzłami takiego drzewa są elementarne funkcje zadeklarowane w kodzie programu. Każda z takich funkcji ma przypisane pewne ograniczenia co do ilości i typu argumentów, które przyjmuje oraz co do typu wartości, który zwraca. Drzewo jako całość również ma zadeklarowany typ zwracanej wartości.

2.3 Ewolucja kerneli

Rozdział 3

Algorytm Kernel GP

3.1 Opis algorytmu

Jedną z trudności, która wiąże się z używaniem klasyfikatora SVM jest dobór odpowiedniej do zbioru danych *funkcji jądrowej*. Wymaga to doświadczenia lub przebiega na zasadzie prób i błędów. Ponadto zbiór powszechnie używanych funkcji jest ubogi - zazwyczaj ogranicza się do trzech podstawowych funkcji. Oprócz wyboru funkcji konieczne jest również ustawienie odpowiednich wartości ich parametrów.

Celem algorytmu Kernel GP jest odnalezienie optymalnej dla danego problemu funkcji jądrowej wraz z jej parametrami. Dzięki opisanej w poprzednim rozdziale własności domknięcia zbioru kerneli ze względu na pewne operacje arytmetyczne możliwe jest tworzenie nieograniczonej ilości dowolnie złożonych funkcji na podstawie kilku podstawowych kerneli. Opisywany algorytm przeszukuje przestrzeń takich funkcji za pomocą *programowania genetycznego*. Szukana jest taka funkcja, przy której użyciu klasyfikator SVM osiągnie największą *dokładność (accuracy)* klasyfikacji.

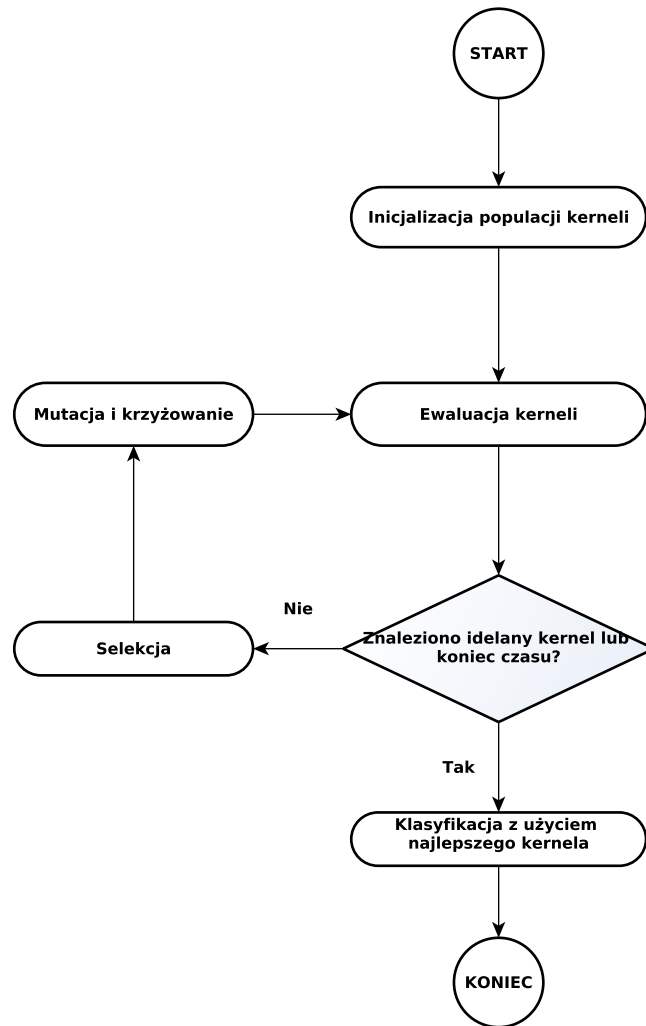
Przebieg algorytmu jest typowy dla algorytmów genetycznych:

1. Utwórz początkową populację kerneli
2. Oblicz wartość *funkcji dopasowania* każdego z kerneli: dokładność klasyfikacji SVM z użyciem tego kernela
3. Jeśli znaleziono idealny kernel (dokładność klasyfikacji 100) lub skończył się czas, użyj tego kernela do klasyfikacji zbioru walidującego, zwróć wyniki klasyfikacji i zakończ algorytm.
4. Dokonaj selekcji najlepszych funkcji z populacji
5. Utwórz nową populację poprzez mutację i krzyżowanie wybranych w poprzednim kroku funkcji
6. Wróć do punktu 2

Algorytm pokazano również na diagramie przepływu na rycinie 3.1. Poszczególne kroki algorytmu zostaną opisane poniżej.

3.1.1 Inicjalizacja populacji

Podczas inicjalizacji początkowo pusta populacja jest wypełniana przez generowane w sposób losowy drzewa reprezentujące funkcje. Generowane drzewa muszą być poprawne, czyli spełniać narzucone ograniczenia na głębokość drzewa, liczbę węzłów, typ wartości zwracanych przez drzewo. Wielkość populacji jest jednym z parametrów algorytmu. Zbyt mała populacja powoduje losowe zawężenie



RYSUNEK 3.1: Diagram przepływu algorytmu Kernel GP.

przeszukiwanej przestrzeni i zmniejsza prawdopodobieństwo znalezienia optymalnej funkcji. Z drugiej strony zbyt duża wielkość populacji upodabnia algorytm genetyczny do pełnego przeszukiwania, co oczywiście zwiększa szanse znalezienia optymalnego kernela, ale wydłuża czas działania algorytmu.

Generowanie funkcji

Generowanie drzew reprezentujących funkcje jądrowe polega na łączeniu ze sobą funkcji elementarnych zgodnie z przypisanymi im ograniczeniami. Funkcje elementarne wraz z ograniczeniami zdefiniowane w algorytmie:

- Funkcje łączące - jako argument przyjmują wynik dwóch lub jednej funkcji jądrowej i ewentualnie stałą ERC. Zwracają wartość rzeczywistą. Dzięki właściwości domknięcia zbioru kerneli ze względu na operacje wykonywane przez te funkcje funkcja powstała przez połączenie dwóch kerneli funkcją łączącą jest również poprawnym kernelem [STC04].

- Dodawanie: $k(x, z) = k_1(x, z) + k_2(x, z)$
- Mnożenie: $k(x, z) = k_1(x, z) * k_2(x, z)$
- Mnożenie przez stałą: $k(x, z) = a * k_1(x, z)$

- Funkcja wykładnicza: $k(x, z) = e^{k_1(x, z)}$

Gdzie a to stała rzeczywista generowana jako stała ERC.

- Podstawowe funkcje jądrowe - jako argument przyjmują odpowiednią do funkcji liczbę stałych ERC. Zwracają wartość rzeczywistą.

- Liniowa: $k(x, z) = \langle x, z \rangle$
- Wielomianowa: $k(x, z) = \langle x, z \rangle^d$
- Gausowska: $e^{-\gamma * \|x - z\|^2}$
- Sigmoidalna: $k(x, z) = \text{tgh}(\gamma \langle x, z \rangle + \tau)$
- Logarytmiczna: $k(x, z) = -\log(\|x - y\|^d + 1)$

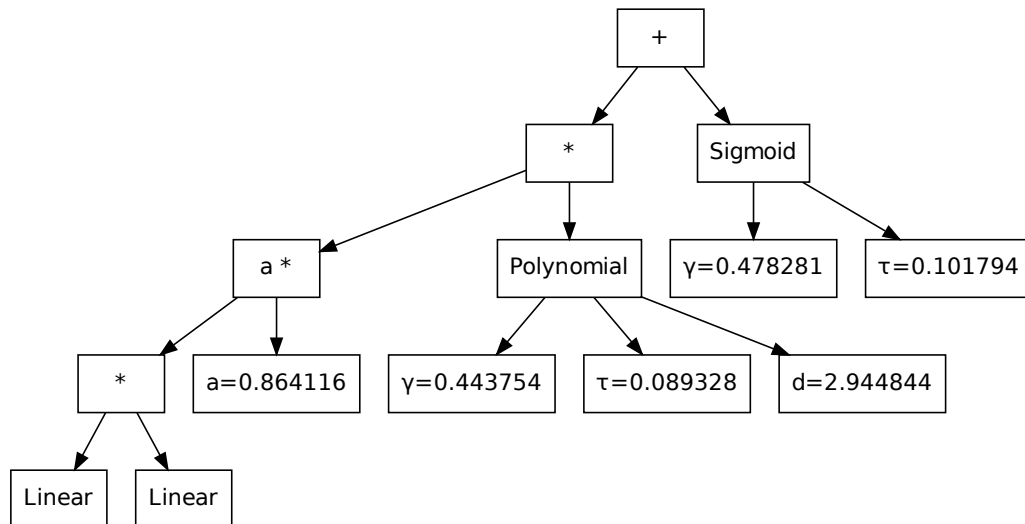
Gdzie γ , τ oraz d to wartości stałe generowane jako stałe ERC. a $\langle x, y \rangle$ to iloczyn skalarny wektorów x i y .

- Stałe ERC (ang. *Ephemeral Random Constant*) liczby rzeczywiste lub całkowite, które służą jako parametry innych funkcji. Są one liśćmi w drzewie, nie przyjmują żadnych argumentów. Mogą losowo zmieniać swoją wartość podczas mutacji.

- γ : liczba rzeczywista z zakresu $\langle 0.1, 2.0 \rangle$
- τ : liczba rzeczywista z zakresu $\langle 0.1, 1.0 \rangle$
- d : liczba całkowita z zakresu $\langle 1.0, 10.0 \rangle$
- a : liczba rzeczywista z zakresu $\langle -10.0, 10.0 \rangle$

Przykładowe drzewo wygenerowane przez algorytm pokazana na ryc.3.2.

Wektory cech będące najważniejszymi argumentami funkcji jądrowych nie są wyodrębnione jako osobne funkcje budujące drzewo.



RYСУNEK 3.2: Przykładowe drzewo generowane przez algorytm.

3.1.2 Ewaluacja kerneli

Ewaluacja funkcji jądrowej może odbywać się na jeden z dwóch sposobów. Jeśli w zbiorze danych oprócz zbioru uczącego wydzielono zbiory testowy i walidujący, to sprawdzany kernel jest używany do klasyfikacji danych ze zbioru testującego. Trafność klasyfikacji zostaje przeliczona na wartość *funkcji przystosowania* ewaluowanej funkcji jądrowej. Jeśli w zbiorze danych wydzielono tylko dwa podzbiory: uczący i walidujący, to zdolność klasyfikacji przez kernel jest oceniana za pomocą *walidacji krzyżowej* (ang. *cross-validation*). Walidacja krzyżowa pozwala użyć więcej danych podczas fazy uczenia, jednak wiąże się ze znacznym wzrostem złożoności obliczeniowej - zamiast jednej klasyfikacji musimy przeprowadzić k procesów uczenia i k klasyfikacji.

3.1.3 Selekcja

Jednym z problemów programowania genetycznego jest to, że drzewa powstałe w wyniku procesu ewolucyjnego mogą być bardzo duże, co nie jest pożądaną cechą - większe drzewo dłużej oblicza zwracaną wartość, zajmuje więcej miejsc w pamięci. Dlatego wielkość drzew należy ograniczać, jeśli wzrost drzewa nie prowadzi do zwiększenia wartości funkcji dopasowania. Wielkość generowanych drzew jest regulowana przez dwa mechanizmy. Pierwszy to proste ograniczenie na maksymalną głębokość drzewa. Wartość tę ustawiono na 6 - drzewa o większej głębokości nie zostaną w ogóle wygenerowane przez podczas inicjalizacji populacji czy podczas krzyżowania i mutacji. Drugi mechanizm, o angielskiej nazwie *parsimony pressure*, promuje mniejsze drzewa podczas selekcji. W tym celu stosowany jest algorytm selekcji turniejowej leksykograficznej z koszykami (ang. Bucket Lexicographic Tournament Selection). Algorytm ten sortuje populację według przystosowania osobników, następnie grupuje je w N "koszyki". Następnie selekcja przebiega według zasad selekcji turniejowej, z tym, że porównuje się nie przystosowanie osobników, ale koszyk, do którego są przypisane. W przypadku gdy w turnieju porównywane są dwa osobniki z tego samego koszyka wygrywa ten, który jest mniejszy.

3.1.4 Krzyżowanie i mutacja

Krzyżowanie polega na odcięciu dwóch losowych poddrzew z dwóch różnych osobników i zamianie ich miejscami. Wygenerowane w ten sposób drzewo musi spełniać narzucone na drzewo ograniczenia dotyczące typów i wielkości. Mutacja drzew polega na zamianie losowo wybranego poddrzewa przez losowo wygenerowane drzewo. Dodatkowo mutowane są również węzły ERC. Ich mutacja polega na dodaniu losowej wartości o rozkładzie normalnym do wartości przechowywanej w węźle. Wartość ta może być ujemna lub dodatnia.

3.1.5 Walidacja rozwiązania

Walidacja polega na użyciu najlepszego znalezionej kernela do klasyfikacji przykładów ze zbioru walidującego, który nie był używany podczas uczenia klasyfikatora SVM ani podczas ewaluacji kerneli. Najpierw algorytm SVM jest uczony na połączonych zbiorach trenującym i walidującym, przy pomocy tej funkcji jądrowej. Następnie dokonywana jest klasyfikacja zbioru walidującego. Otrzymana w wyniku tej klasyfikacji trafność jest miarą oceny całego algorytmu Kernel GP.

3.2 Implementacja

Algorytm został napisany w języku Java z użyciem bibliotek *ECJ* (*Evolutionary Computing in Java*) [Sea10] oraz *LibSVM* [CL11]. Pierwsza z nich dostarcza mechanizmy *obliczeń ewolucyjnych* w tym *programowania genetycznego*. *LibSVM* to klasyfikator SVM napisany oryginalnie w języku C z dostępną

implementacją w Javie. Mechanizmy ECJ stanowią trzon algorytmu zapewniając tworzenie populacji funkcji, ich selekcję, mutację oraz krzyżowanie. LibSVM został użyty na etapie ewaluacji wygenerowanych przez ECJ funkcji.

3.3 Złożoność obliczeniowa

Rozdział 4

Wyniki działania algorytmu na popularnych zbiorach danych

4.1 Metodologia pomiarów

Żeby oszacować trafność klasyfikacji osiąganą przez skonstruowany system konieczne było podzielenie zbioru danych na zbiór uczący i *walidujący*, a w przypadku algorytmu Kernel-GP również wydzielenie ze zbioru uczącego podzbioru *testującego*, używanego do obliczania miary przystosowania (fitness) podczas przebiegu algorytmu genetycznego. Ponieważ sposób podziału zbioru danych ma wpływ na osiąganą trafność klasyfikacji, dokonywano 5 takich podziałów a następnie wyciągano średnią oraz odchylenie standardowe z wyników otrzymanych dla tych podziałów. Ta procedura dotyczyła zarówno testowania algorytmu *Kernel-GP* jak i porównawczych testów klasyfikatora SVM z biblioteki *LibSVM*. Dla obu algorytmów stosowano te same podziały danych, przy czym w przypadku klasyfikatora *LibSVM* ze zbioru uczącego nie wydzielano zbioru testującego.

Jeśli nie zaznaczono inaczej, w poniższym omówieniu wyników słowa "dokładność" lub "trafność" klasyfikacji odnoszą się do trafności klasyfikacji danych ze zbioru *walidującego*.

Algorytm genetyczny jest w swej naturze stochastyczny, korzysta więc z funkcji generujących liczby pseudolosowe. Aby zapewnić powtarzalność wyników i umożliwić ich porównanie ziarno generatora liczb pseudolosowych ustawiono na stałą wartość.

Aby ocenić skuteczność algorytmu genetycznego w poszukiwaniu optymalnych funkcji jądrowych oraz oszacować optymalną wielkość populacji oraz czas trwania (liczbę ewaluowanych generacji) algorytmu przeprowadzono szereg eksperymentów obliczeniowych, w których uruchamiano algorytm dla coraz to większych wartości tych parametrów. Dla każdego przebiegu algorytmu zapisywano trafność klasyfikacji zbioru *walidującego*.

Analizując tak zebrane dane można przeanalizować na ile poszukiwanie funkcji jądrowej przez algorytm genetyczny było podobne do losowego przeszukiwania a na ile było ono zbieżne. W pierwszym przypadku na wyniki osiągane przez algorytm powinna mieć wpływ przede wszystkim wielkość populacji, w drugim również liczba generacji przez które poszukiwano rozwiązania. W szczególności ciekawym przypadkiem jest ten, gdy liczba generacji wynosi 1, czyli cały algorytm ogranicza się do wygenerowania populacji losowych osobników i wybrania jednego z nich - w tym przypadku algorytm genetyczny sprowadza się do losowego poszukiwania rozwiązania. Porównując różnicę w trafności osiąganej w trakcie jednej generacji i coraz większej ich liczby można ocenić czy proces ewolucyjny przebiega poprawnie.

4.2 Opis zbiorów danych

Do oceny pracy algorytmu użyto standardowych zbiorów danych służących do testowania systemów maszynowego uczenia się, dostępnych na stronie biblioteki *LIBSVM* [CL11] oraz w repozytorium UCI [D]. Zbiory zostały opisane w tabelce 4.1. Użyte nazwy zbiorów są zgodne z tymi ze strony libsvm [C].

TABLICA 4.1: Zbiory danych użyte do testowania systemu.

Nazwa zbioru	Liczba klas	Liczba atrybutów	Wielkość zbioru	Wielkość zbioru uczącego	Wielkość zbioru testującego	Wielkość zbioru walidującego
Iris	3	4	150	68	33	49
Letter	26	16	15000	9000	4400	6600
DNA	3	180	2000	1435	700	1051
Vowel	11	10	528	447	217	326
Breast cancer	2	10	683	343	170	170
heart	2	13	270	136	67	67

TABLICA 4.2: Zbiory danych użyte do testowania systemu.

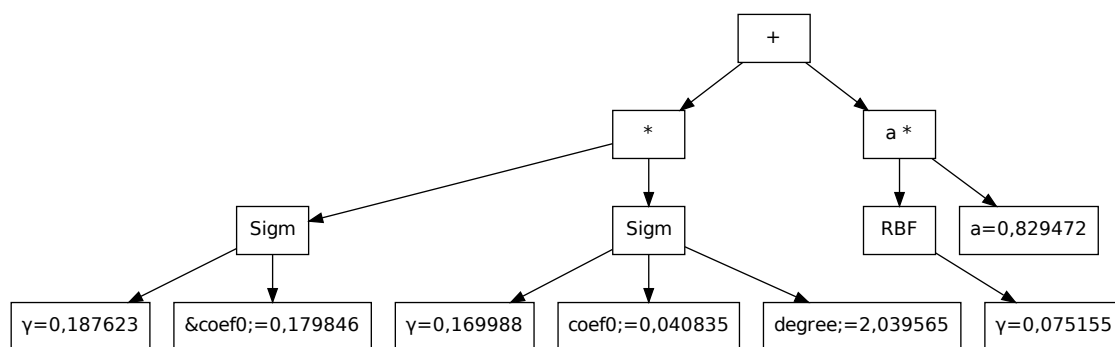
Nazwa zbioru	Liczba atrybutów ciągłych	Liczba atrybutów nominalnych	Liczba klas	Proporcje klas
Breast Cancer	10	0	2	239/444
Heart	7	6	2	150/120
DNA	0	180	3	464/485/1051
Vowel	10	0	11	Każda klasa 48 razy

4.3 Trafność klasyfikacji

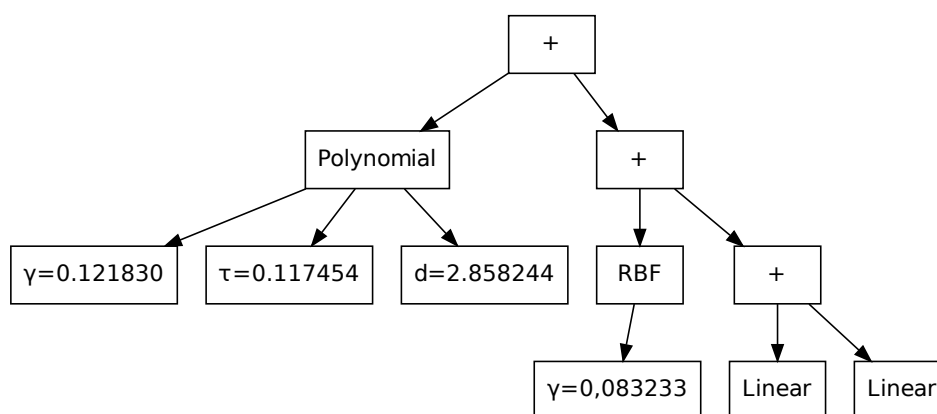
Jak widać na rysunkach 4.6-4.15 sprawność algorytmu zależy mocno od konkretnego zbioru danych. Na przykład zbiór IRIS jest na tyle łatwy w klasyfikacji, że ciężko jest poprzez dobranie optymalnej funkcji jądrowej polepszyć znacznie wyniki klasyfikacji. Na osiąganą trafność klasyfikacji wpływ ma też podział zbioru danych na testujący, uczący i walidujący - widać to po odchyleniu standardowym widocznym na wykresach.

Monotoniczność funkcji trafności

Miejscami funkcja trafności nie jest monotoniczna, a ściślej niemalejąca, względem liczby generacji oraz wielkości populacji (co widać np. na wykresach 4.7 i 4.12). Wydawałoby się, że tak być nie powinno (algorytm genetyczny zwraca najlepszego osobnika z całego swojego przebiegu, więc wszystkie osobniki, które pojawiły się podczas przebiegu z 5 generacjami pojawią się podczas przebiegu z 7 generacjami, więc trafność dla przebiegu z 7 generacjami powinna być co najmniej tak dobra jak dla przebiegu z 5 generacjami). Jednak może się tak zdarzyć ze względu na to, że trafność pokazana na wykresach to trafność klasyfikacji zbioru walidującego, natomiast trafność użyta przez algorytm genetyczny jako miara dostosowania (ang. *fitness*) to trafność klasyfikacji zbioru testującego. Widać to na wykresie 4.8, który przedstawia wartość przystosowania dla tych samych danych, dla których na wykresie 4.7 jest pokazana trafność klasyfikacji na zbiorze walidującym - tutaj funkcja wykazuje mniej braku monotoniczności.



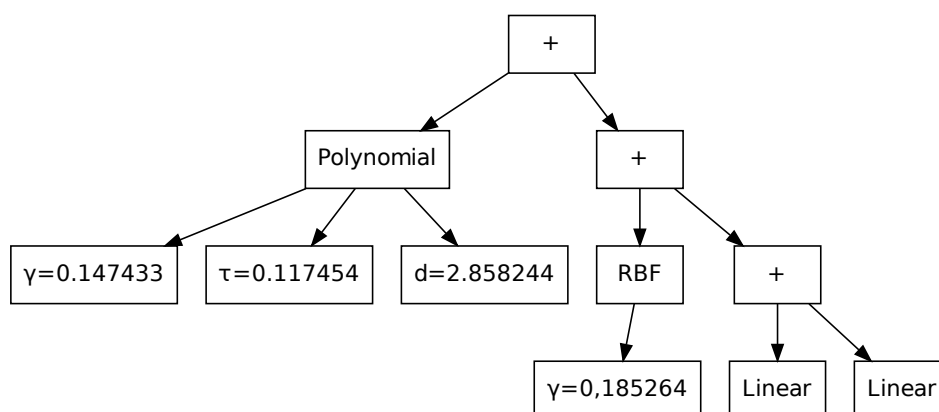
RYSUNEK 4.1: Funkcja z pierwszej generacji, która w przebiegu z wielkością populacji 4 osiągnęła fitness 0.4242424. Przodek funkcji z rys.4.2



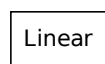
RYSUNEK 4.2: Funkcja z trzeciej generacji, która w przebiegu z wielkością populacji 4 osiągnęła fitness 0.78030306. Potomek funkcji z rys.4.1, przodek zwycięskiej funkcji (rys.4.3) z przebiegu z populacją o wielkości 4.

Zatem przynajmniej część braku monotoniczności funkcji trafności na zbiorze walidującym wynika z przeuczenia algorytmu - znaleziona przez algorytm genetyczny funkcja jądrowa lepiej sprawdza się przy klasyfikacji zbioru testującego niż walidującego. Nie jest to jednak jedyna przyczyna braku monotoniczności - widać to na wykresie 4.13 przedstawiającym wartość funkcji przystosowania dla zbioru *vowel* - jej przebieg jest bardzo podobny do przebiegu ukazanej na rys. 4.12 funkcji trafności klasyfikacji zbioru walidującego na tym samym zbiorze. Co więc jest przyczyną braku monotoniczności? Warto zauważyć, że funkcja jest niemalejąca ze względu na ilość generacji oraz że dla jednej generacji funkcja jest monotoniczna. Sugeruje to, że "winnym" może być selekcja - w praktyce nie zachodzi ono w przypadku gdy algorytm genetyczny działa przez jedną generację.

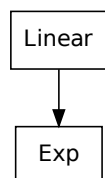
Gdy przyjrzeć się dokładnie przebiegowi ewolucji widać, że rzeczywiście tak jest. Dodatkowy osobnik (rys. 4.1), który odróżnia w generacji pierwszej populacje o wielkości 3 i 4 jest przodkiem innego osobnika (rys. 4.2), który w trzeciej generacji osiąga fitness większy niż osobnik (rys. 4.4), który w przypadku populacji wielkości 3 był przodkiem osobnika (rys. 4.5), który to okazał się najlepszym podczas przebiegu całego algorytmu. W rezultacie tego "geny" potencjalnego zwycięzcy nie przetrwały w przebiegu algorytmu z populacją liczącą 4 osobników. Jak widać osobnik najlepszy we wszystkich genera-



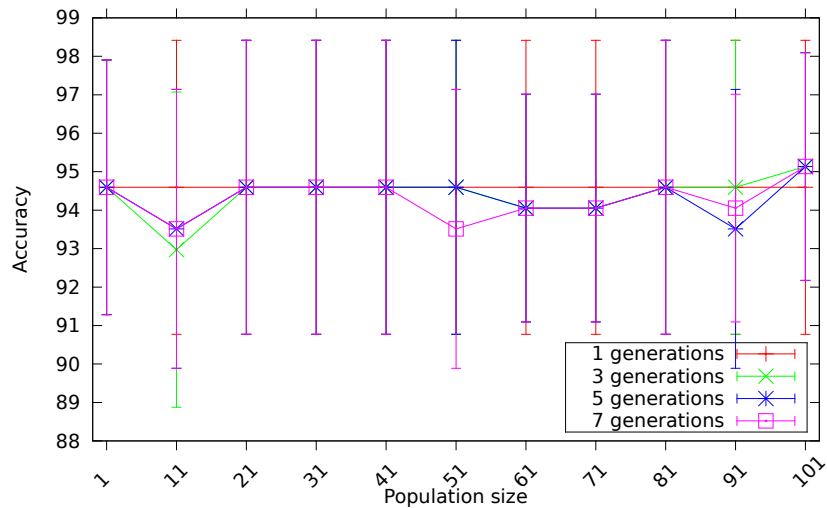
RYSUNEK 4.3: Funkcja z ostatniej generacji w przebiegu z wielkością populacji 4 osiągnęła fitness 0.833333. Przodek zwycięskiej funkcji z rys.4.5



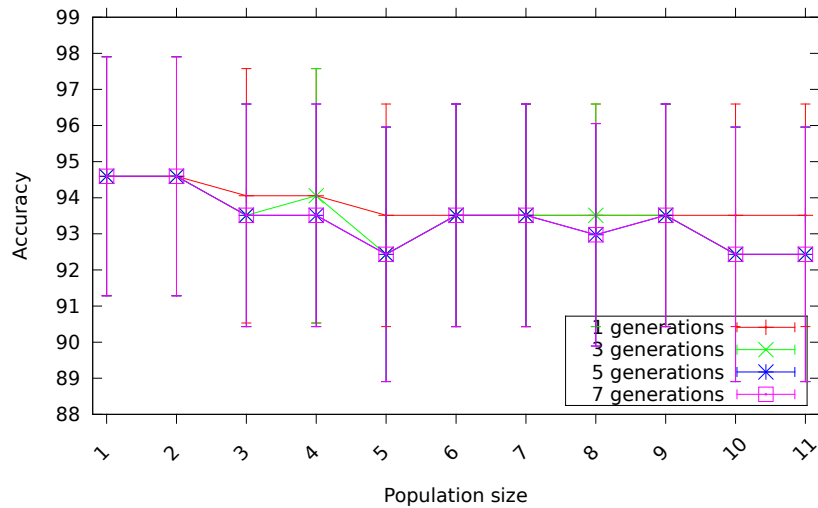
RYSUNEK 4.4: Funkcja z trzeciej generacji, która w przebiegu z wielkością populacji 3 i 4 osiągnęła fitness 0.6515151. Przodek zwycięskiej funkcji z rys.4.5



RYSUNEK 4.5: Zwycięska funkcja w przebiegu z populacją wielkości 3, potomek funkcji z rys.4.4. Osiągnęła fitness 0.9015151.

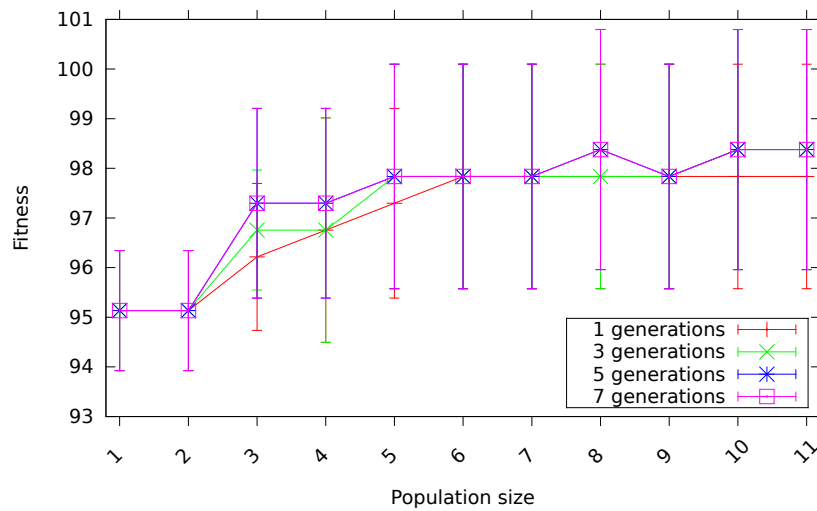


RYSUNEK 4.6: Dokładność klasyfikacji dla zbioru *iris* w funkcji rozmiaru populacji dla różnych ilości generacji.

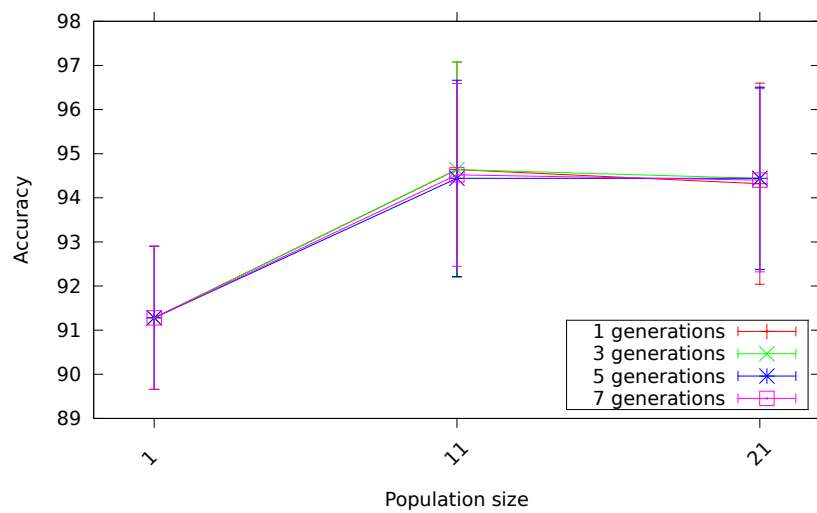


RYSUNEK 4.7: Dokładność klasyfikacji dla zbioru *iris* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.

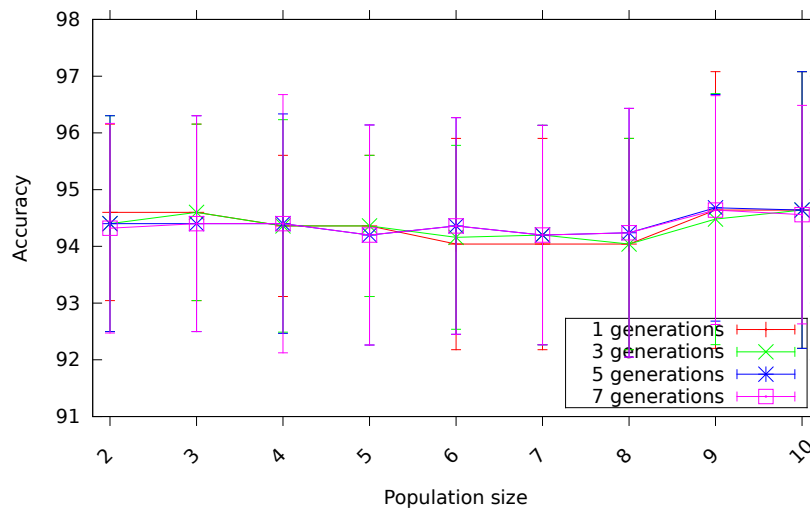
ciach nie musi być wcale potomkiem osobników najlepszych w poszczególnych generacjach - czasem połączenie dwóch osobników przeciętnych może dać osobnika bardzo dobrego.



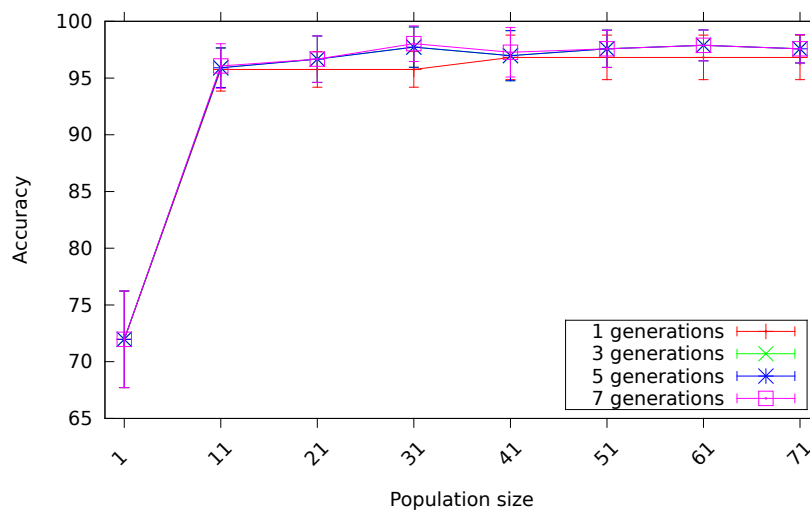
RYСУNEK 4.8: Najlepsza wartość funkcji przystosowania (ang. *fitness*) *iris* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.



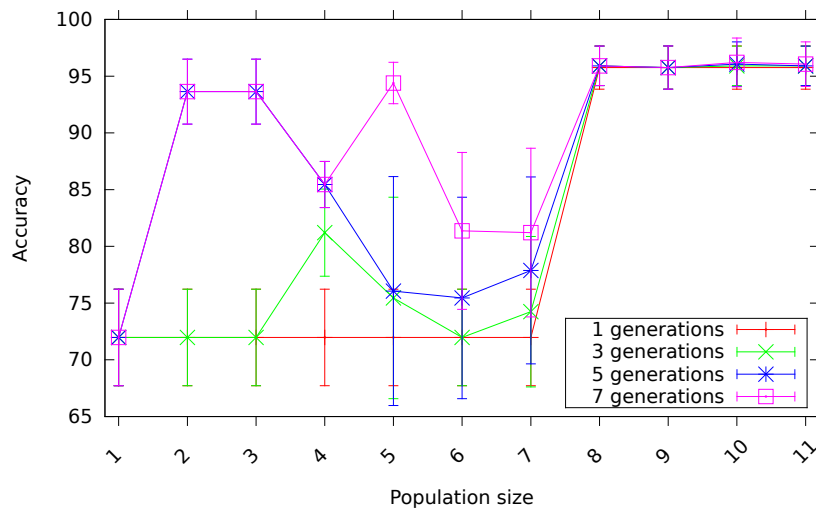
RYСУNEK 4.9: Dokładność klasyfikacji dla zbioru *DNA* w funkcji rozmiaru populacji dla różnych ilości generacji.



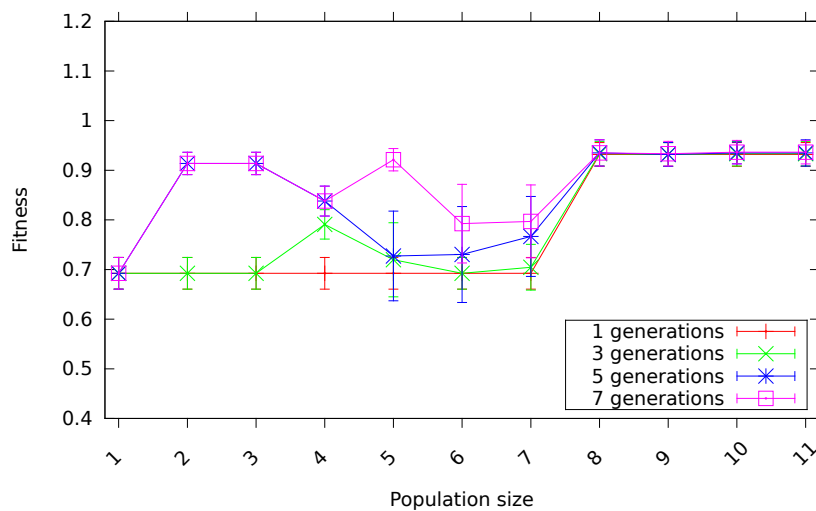
RYSunEK 4.10: Dokładność klasyfikacji dla zbioru *DNA* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.



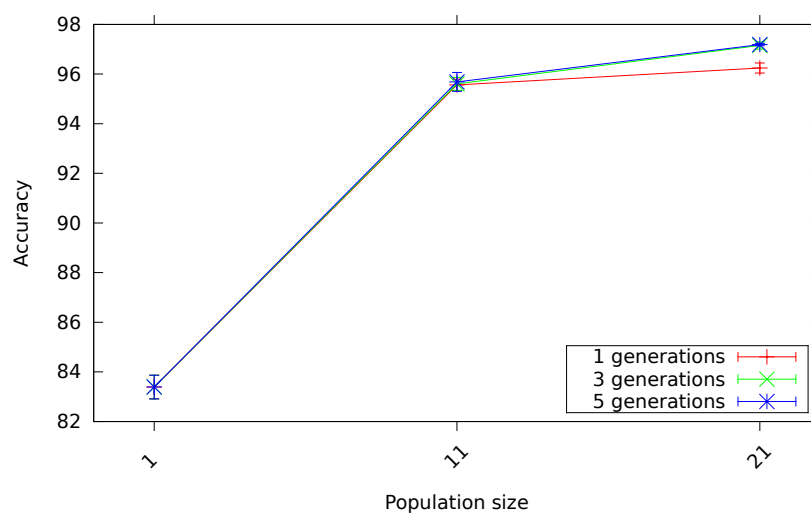
RYSunEK 4.11: Dokładność klasyfikacji dla zbioru *vowel* w funkcji rozmiaru populacji dla różnych ilości generacji.



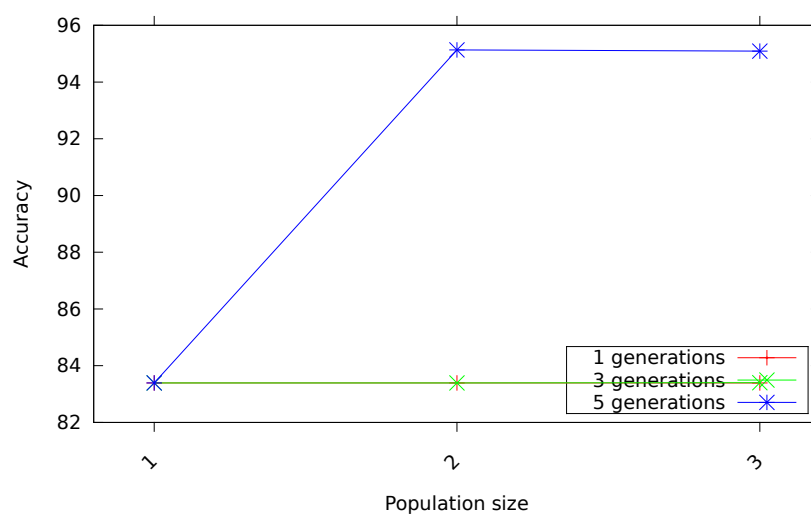
RYСУNEK 4.12: Dokładność klasyfikacji dla zbioru *vowel* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.



RYСУNEK 4.13: Dokładność klasyfikacji dla zbioru *vowel* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.



RYСУNEK 4.14: Dokładność klasyfikacji dla zbioru *letter* w funkcji rozmiaru populacji dla różnych ilości generacji.

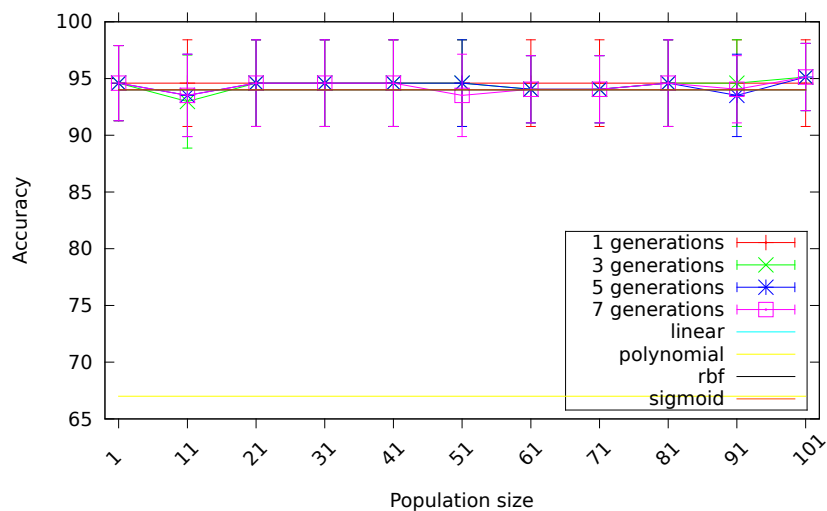


RYСУNEK 4.15: Dokładność klasyfikacji dla zbioru *letter* w funkcji rozmiaru populacji dla różnych ilości generacji, dla małych populacji.

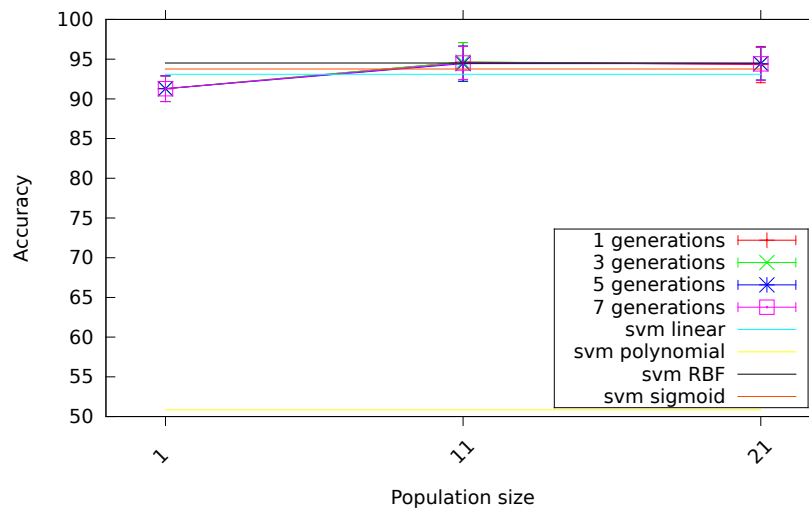
4.3.1 Porównanie z tradycyjnym algorytmem SVM

Na wykresach 4.16-4.19 przedstawiono porównanie trafności klasyfikacji zbioru walidującego przez algorytm SVM z użyciem czterech podstawowych funkcji jądrowych (liniowej, wielomianowej, RBF i sigmoidalnej) i przez stworzony algorytm Kernel-GP. W przypadku dwóch zbiorów: *vowel* i *letter* udało się uzyskać polepszenie trafności klasyfikacji względem standardowych funkcji jądrowych. Są to zbiory, dla których standardowy SVM osiąga słabe wyniki - ok. 70% – 80%.

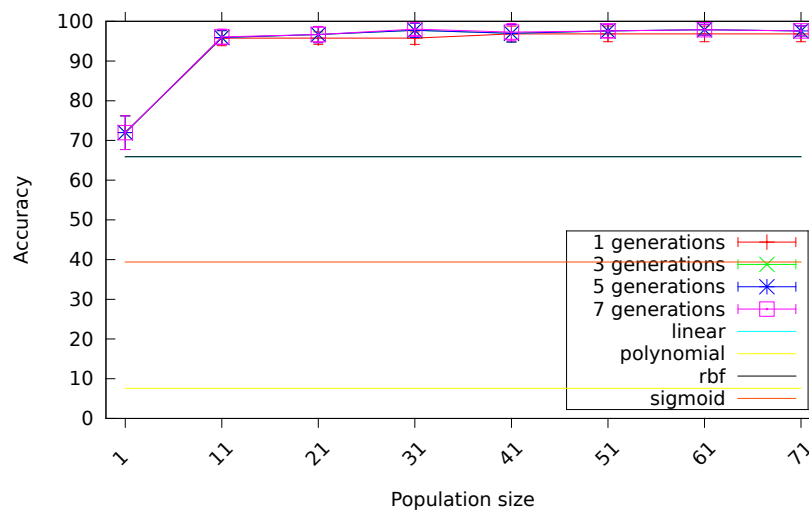
W przypadku dwóch pozostałych zbiorów niezależnie od użytej funkcji jądrowej osiągana trafność klasyfikacji jest bardzo wysoka - ok. 95% - sugeruje to, że zbiory te są dość łatwo separowalne z wyjątkiem 5% przypadków.



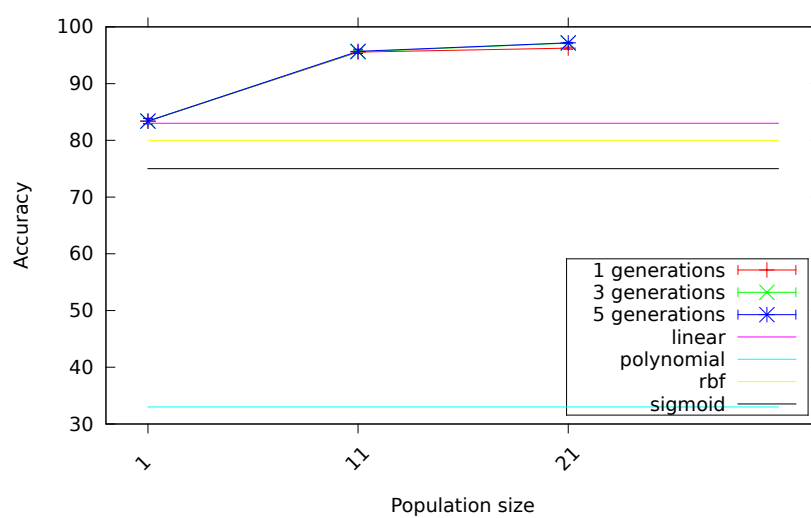
RYSUNEK 4.16: Porównanie dokładności klasyfikacji dla zbioru *iris* przez algorytm SVM z różnymi funkcjami jądrowymi i algorytm Kernel-GP



RYСУNEK 4.17: Porównanie dokładności klasyfikacji dla zbioru *dna* przez algorytm SVM z różnymi funkcjami jądrowymi i algorytm Kernel-GP.



RYСУNEK 4.18: Porównanie dokładności klasyfikacji dla zbioru *vowel* przez algorytm SVM z różnymi funkcjami jądrowymi i algorytm Kernel-GP.



RYSUNEK 4.19: Porównanie dokładności klasyfikacji dla zbioru *letter* przez algorytm SVM z różnymi funkcjami jądrowymi i algorytm Kernel-GP

4.4 Czas wykonania

4.5 Użycie pamięci

4.6 Posdumowanie wyników

Rozdział 5

Case study - klasyfikacja danych ADHD 200

ADHD 200 był międzynarodowym konkursem, który zakończył się we wrześniu 2011 roku. Dzięki współpracy ośmiu szpitali i ośrodków naukowych z całego świata w ramach konkursu udostępniono zbiór zawierający dane medyczne 776 dzieci, z czego 285 z ADHD. Zadaniem uczestników konkursu było skonstruowanie klasyfikatora diagnozującego ADHD na podstawie tych danych. Zbiór testowy zawierał dane xxx dzieci, których danych nie było w zbiorze uczącym, bez podanej diagnozy. Celem skonstruowanego klasyfikatora było przypisanie diagnozy do przykładów ze zbioru testującego.

Wyniki konkursu pokazały, że zbiór danych był trudny w klasyfikacji. Największa osiągnięta trafność klasyfikacji wyniosła 60.51% (szczegółowe wyniki dostępne na stronie konkursu: [B]).

5.1 Opis zbioru danych

5.1.1 Surowe dane

Dane dostarczone przez organizatorów konkursu składają się z:

- Danych klinicznych:
 - Płeć
 - Wiek
 - Współczynnik IQ
 - Prawo/lewo ręczność
- Danych obrazowych:
 - Strukturalnych - dane pochodzące z obrazowania rezonansu magnetycznego (*MRI*, ang. *Magnetic Resonance Imaging*). Są to trójwymiarowe obrazy o rozdzielczości ok. 256x254x160 punktów, obrazujące strukturę mózgu osoby badanej
 - Funkcjonalnych - dane pochodzące z obrazowania funkcjonalnego rezonansu magnetycznego (*FMRI*, ang. *functional Magnetic Resonance Imaging*) będące sekwencją ok 120 trójwymiarowych obrazów o rozdzielczości ok 250x250x250 punktów, obrazującą aktywność mózgu osoby badanej rejestrowaną przez ok 6 minut.

5.1.2 Preprocessing

5.2 Konstrukcja i selekcja cech

5.3 Wyniki klasyfikacji

5.3.1 Kernel GP

5.3.2 Porównanie z innymi algorytmami

SVM

Inne klasyfikatory

Rozdział 6

Podsumowanie

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla purus purus, fermentum in condimentum nec, sodales nec enim. Fusce auctor auctor porta. Proin tempus lacinia tortor, eget aliquam ante condimentum id. Morbi viverra congue posuere. Nunc non odio eros, sollicitudin pulvinar metus. Sed eget ligula ligula, a congue orci. Proin laoreet aliquet vulputate. Vivamus ut enim sed diam pretium fringilla. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Etiam convallis mi nec dolor pretium sed aliquam orci sagittis. Maecenas aliquam dictum neque vel mollis. Morbi vel vehicula mauris.

Literatura

- [CL11] Chih-Chung Chang, Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, Maj 2011.
- [KS03] K. Krawiec, J. Stefanowski. *Uczenie maszynowe i sieci neuronowe*. Wydaw. Politechniki Poznańskiej, 2003.
- [Sea10] Luke Sean. *The ECJ Owner's Manual*. Yale Univ Pr, Pa/xdziernik 2010.
- [STC04] John Shawe-Taylor, Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

Zasoby internetowe

[A] ECJ

<http://cs.gmu.edu/~eclab/projects/ecj/>

[B] ADHD 200

http://fcon_1000.projects.nitrc.org/indi/adhd200/

[C] Libsvm datasets

<http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>

[D] UCI Machine Learning Repository

<http://archive.ics.uci.edu/ml/datasets/>



© 2013 Tomasz Ziętkiewicz

Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu L^AT_EX.

Bib_TE_X:

```
@mastersthesis{ mnowak-masterthesis,  
  author = "Tomasz Ziętkiewicz",  
  title = "{Optymalizacja klasyfikatora SVM za pomocą programowania genetycznego}",  
  school = "Poznan University of Technology",  
  address = "Pozna{\n}, Poland",  
  year = "2013",  
}
```