

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	3
1.2	Struktura pracy	3
2	Podstawy teoretyczne	4
2.1	Uczenie maszynowe	4
2.1.1	Systemy klasyfikujące	5
	Formalizacja problemu klasyfikacji	5
	Metody oceny skuteczności klasyfikacji	5
2.1.2	Miary skuteczności klasyfikacji	6
2.2	SVM — Maszyny wektorów wspierających	7
2.2.1	Klasyfikatory liniowe	7
2.2.2	Maszyny wektorów wspierających	10
2.2.3	Funkcje jądrowe	12
	Algorytmy wykorzystujące funkcje jądrowe	13
2.2.4	Klasyfikacja więcej niż dwóch klas	14
	Jeden przeciw wszystkim	14
	Klasyfikacja parami	14
2.3	Obliczenia ewolucyjne	14
	Selekcja	15
	Krzyżowanie	17
	Mutacja	17
	Genotyp	17
2.3.1	Programowanie genetyczne	17
	Inicjalizacja populacji	18
	Krzyżowanie	19
	Mutacja	19
	Selekcja	20
2.4	Optymalizacja parametrów SVM — przegląd literatury	21
2.4.1	Miary przystosowania (fitness)	21
2.4.2	Optymalizacja parametrów	22
2.4.3	Ewolucja kerneli	22
	Genetic Kernel SVM	22
	KTree	23
	Evolutionary Kernel Machine	24
	Kernel GP	24

3	Algorytm Kernel GP+	26
3.1	Opis algorytmu	26
3.1.1	Inicjalizacja populacji	27
	Generowanie funkcji	27
3.1.2	Ewaluacja kerneli	28
3.1.3	Selekcja i zapobieganie przerostowi	29
3.1.4	Krzyżowanie i mutacja	29
3.1.5	Walidacja rozwiązania	30
3.2	Implementacja	30
4	Wyniki działania algorytmu na popularnych zbiorach danych	33
4.1	Metodologia pomiarów	33
4.1.1	Parametry procesu ewolucyjnego	34
4.2	Opis zbiorów danych	34
4.3	Fitness	35
4.4	Wyniki klasyfikacji zbioru testującego	40
4.4.1	Trafność	40
4.4.2	Prawdopodobieństwo	44
4.5	Czas wykonania	47
4.6	Wnioski	49
5	Podsumowanie	50
5.1	Napotkane problemy	50
5.2	Kierunek dalszych prac	51
	Literatura	52
	Zasoby internetowe	54

Rozdział 1

Wprowadzenie

Jako cechę wyróżniającą nasz gatunek, *Homo sapiens sapiens*, podaje się zwykle umiejętność inteligentnego myślenia oraz wytwarzania i używania narzędzi. Korzystając zarówno z inteligencji jak i wytworzonych wcześniej narzędzi, człowiek przez tysiąclecia rozwoju kultury i cywilizacji tworzył coraz to nowsze i doskonalsze narzędzia, które ułatwiały mu wykonywanie trudnych i monotonicznych prac, a czasami wykonywały te prace za niego. Ten proces udoskonalania wytworów człowieka, zbliżony jest do procesu ewolucji naturalnej, której podlega człowiek, dlatego rozwój człowieka jako gatunku opisuje się często jako złożenie ewolucji naturalnej i kulturowej. Narzędzia, które sprawdzają się lepiej od już istniejących zastępują je, rozwiązania krzyżują się, te powstałe na potrzeby jednej dziedziny są stosowane w innych dziedzinach. Narzędzia, których używa człowiek, mogą być postrzegane jako jego „rozszerzenie” czy „przedłużenie” (ang. *extension*) [CC98]. Na przykład tak prosty przyrząd jak łopata może być uznany za „przedłużenie” rąk i dłoni. Inne narzędzia mogą stanowić rozszerzenie ludzkiego umysłu: notatnik stanowi rozszerzenie ludzkiej pamięci pozwalając odciążyć ją od konieczności przechowywania pewnych informacji. O ile rozwój narzędzi stanowiących rozszerzenie ludzkich zdolności fizycznych ma miejsce od początków dziejów, o tyle rozszerzenia umysłu przez tysiąclecia zmieniały się tylko kilka razy. Za największy wynalazek zwiększający nasze możliwości intelektualne należy uznać rozwój języka, pisma i druku. Poza nim tworzone przez tysiąclecia narzędzia nie przynosiły rewolucyjnych zmian. Dopiero wynalazki powstałe w XIX wieku, takie jak telegraf, telefon, radio zapoczątkowały rewolucję w sposobie w jaki człowiek zdobywał i przekazywał informacje. Rewolucja ta osiągnęła swój pełen rozkwit wraz z pojawieniem się i rozwojem komputerów. Są one rozszerzeniem umysłu, które wspomaga człowieka w najbardziej złożonych i skomplikowanych operacjach mentalnych. O związku łączącym komputer i umysł świadczy nawet etymologia słowa 'komputer' — w języku angielskim początkowo oznaczało ono 'osobę, która liczy' (ang. *one who computes*), wykonuje obliczenia matematyczne. W języku polskim z kolei istniało określenie 'mózg elektronowy' oznaczające komputer.

Komputer stanowi rozszerzenie ludzkiego umysłu między innymi poprzez automatyzację procesu przetwarzania informacji — zwalnia człowieka z konieczności wykonywania monotonicznych, powtarzających się czynności umysłowych. Dzięki możliwości operowania na większej liczbie elementów naraz oraz szybszym tempie wykonywania prostych operacji komputer umożliwia również wykonywanie zadań, których człowiek bez jego pomocy w ogóle nie byłby w stanie wykonać. Użycie komputera w celu rozwiązania jakiegoś problemu wymaga jednak od człowieka jego zdefiniowania w taki sposób, żeby komputer mógł je w jednoznaczny sposób przełożyć na ciąg prostych operacji matematycznych wykonywanych na danych. Taka definicja sposobu działania komputera, które prowadzi do rozwiązania przez niego pewnego zadania to program komputerowy. W przypadku współczesnych komputerów każdy program musi ostatecznie sprowadzać się do ciągu instrukcji, które mają być po kolei

wykonywane. Najbardziej bezpośrednim sposobem zapisu programu jest bezpośrednie podanie ciągu takich instrukcji, na przykład w języku assemblera, który odnosi się do elementarnych operacji wykonywanych przez komputer. Używanie takiej formy definicji zadania jest jednak bardzo niewygodne dla człowieka. Dlatego istnieją języki programowania o różnym stopniu abstrakcji — od języków niskiego poziomu, zawierających proste operacje, związane ściśle ze sposobem pracy komputera, po języki wysokiego poziomu, abstrahujące od procesów przebiegających w komputerze i operujące na wyrażeniach odnoszących się do w sposób bardziej bezpośredni do rozwiązywanego problemu.

Mimo używania języków wysokiego poziomu, człowiek wciąż musi wykonać sporo pracy, żeby przełożyć swoje potrzeby na język zrozumiały dla komputera. W pewnym sensie komputer przypomina niedoświadczonego, niezbyt inteligentnego pracownika, którego przełożony musi krok po kroku instruować, nieraz spędzając przy tym więcej czasu niż zajęłoby mu wykonania danego zadania samodzielnie. Na szczęście zarówno w przypadku komputera jak i niedoświadczonego pracownika trud włożony w napisanie programu/wy tłumaczenie pracownikowi sposobu wykonywania zadania opłaca się, jeśli tylko program/zadanie musi być wykonywane wiele razy. Problem pojawia się wtedy, gdy zmieniają pewne warunki zadania, na przykład dane, na których zdanie jest wykonywane — wymagają one elastyczności. Elastyczność programu komputerowego może być w pewnym stopniu zapewniona przez parametry, którymi można regulować jego pracę — ich odpowiednie ustawienie wymaga jednak inteligencji osoby uruchamiającej program. W przypadku pracownika, jeśli nie wykaze się on wystarczającą elastycznością i nie poradzi sobie z problemem po jego nieznacznym zmodyfikowaniu przełożony może stwierdzić, że nie powinien zajmować się powierzonymi mu problemami, ponieważ przy każdej modyfikacji zadania potrzebna jest ingerencja osób bardziej od niego doświadczonych. Taka elastyczność w rozwiązywaniu problemów, wykonywaniu zadań w zmieniających się warunkach jest często uważana za jeden z przejawów inteligencji. Programy, które wykazują się elastycznością na zmieniające się dane, potrafią same dostosowywać swoje parametry, również zwane są inteligentnymi. Dziedzina informatyki zajmująca się między innymi takimi programami to *sztuczna inteligencja* (ang. *Artificial Intelligence (AI)*).

Jednym z działów sztucznej inteligencji są obliczenia ewolucyjne. Sposób w jaki algorytmy należące do tej grupy rozwiązują problemy jest zainspirowany biologiczną ewolucją i zasadą selekcji naturalnej. Użycie obliczeń ewolucyjnych pozwala na automatyzację doboru parametrów programu a nawet na automatyczne generowanie jego fragmentów przez komputer. Choć z punktu widzenia informatyki algorytmy ewolucyjne mogą być postrzegane po prostu jako pewien rodzaj metaheurystyk, czyli uniwersalnych algorytmów optymalizacyjnych, to z perspektywy filozoficznej ich znaczenie może być o wiele bardziej donośne. O to bowiem narzędzia wytworzone przez człowieka (algorytmy ewolucyjne), są w stanie niemal samodzielnie wytwarzać inne narzędzia, służące realizacji celów zadanych przez człowieka. Podobna sytuacja ma miejsce w przypadku automatyzacji produkcji przemysłowej, gdzie narzędzia składające się na taśmę produkcyjną produkują inne narzędzia, ale w ten sposób nie powstaje żaden wytwór, którego człowiek wcześniej nie wymyślił - automatyzacja zastępuje tu tylko pracę fizyczną a nie umysłową człowieka. Za to algorytmy ewolucyjne są teoretycznie w stanie tworzyć w sposób, który można by określić jako kreatywny. W wizjach niektórych futurologów zdolność programów do samo modyfikacji oraz tworzenia innych programów może doprowadzić do powstania maszyn przewyższających inteligencją człowieka oraz skokowego wzrostu tempa rozwoju technologicznego.

Umiejętność uczenia się jest podawana jako jeden z wyznaczników inteligencji. Systemy uczące się, to systemy (np. programy komputerowe), które potrafią doskonalić sposób swojego działania wraz z nabywanym doświadczeniem. Dziedzina sztucznej inteligencji zajmująca się ich konstruowaniem to *uczenie maszynowe* (ang. *machine learning*). Systemy uczące pozwalają na automatyczne analizowanie danych, pozyskiwanie informacji, które nie są dane bezpośrednio. Mogą służyć do prognozowania

przebiegu procesów, klasyfikacji obiektów - wykonują zadania, z którymi wcześniej poradzić mógł sobie tylko człowiek używając w tym celu swojej inteligencji.

Niniejsza praca dotyczy zagadnień należących do dwóch wspomnianych powyżej dziedzin sztucznej inteligencji: *Uczenia maszynowego* oraz *Obliczeń ewolucyjnych*. Pierwsza z nich jest w pracy reprezentowana przez *maszyny wektorów wspierających* (ang. *Support Vector Machines* (SVM)), będące rodzajem systemu klasyfikującego. Drugą reprezentuje programowanie genetyczne.

1.1 Cel i zakres pracy

Celem niniejszej pracy jest stworzenie oraz analiza wyników działania algorytmu programowania genetycznego optymalizującego działanie klasyfikatora SVM.

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 oraz przedstawiono sposób jego implementacji. Rozdział czwarty przedstawia wyniki działania algorytmu na standardowych zbiorach danych używanych do testowania algorytmów maszynowego uczenia. Rozdział piąty 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]. Podstawową cechą takich systemów jest to, że potrafią one zmieniać sposób swojego działania w miarę jak napływają do nich kolejne dane. Zmiana działania systemu może mieć różną skalę — od zmiany pojedynczych parametrów programu, przez zapamiętywanie danych wejściowych po całkowitą 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. Jak ujmuje to Tom Mitchell [Mit97] (s.2):

System uczy się z doświadczenia E ze względu na pewną klasę zadań T i ocenę wykonania P jeśli ocena wykonania zadań należących do klasy T rośnie wraz z doświadczeniem E .

Systemy uczące się mają wiele zastosowań, między innymi [KS03]

- 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

2.1.1 Systemy klasyfikujące

Jednym z typów systemów uczących się są *systemy klasyfikujące* (inaczej *klasyfikatory*). Operują one na zbiorach *przykładów* opisanych za pomocą pewnego zbioru *atrybutów*. *Przykłady* (zwane też *obserwacjami*) reprezentują pewne obiekty, które różnią się od siebie wartościami atrybutów. Każdy przykład jest całkowicie scharakteryzowany przez swoje wartości atrybutów, co oznacza, że dwa przykłady o identycznych wartościach atrybutów są z punktu widzenia systemu klasyfikującego nieodróżnialne. Przykładem zbioru obserwacji może być np. zbiór pacjentów, zaś zbiorem atrybutów zbiór cech takich jak wiek, płeć, wzrost, wyniki testów laboratoryjnych. Wśród zbioru atrybutów wyróżnia się jeden specjalny atrybut zwany atrybutem decyzyjnym (w odróżnieniu od pozostałych — atrybutów warunkowych) zwany też *klasą* lub *etykietą* obiektu. Zazwyczaj wartość tego atrybutu nie jest znana bezpośrednio i niesie ze sobą pewne istotne informacje, których bezpośrednie zdobycie może być niemożliwe lub nieopłacalne. W przytoczonym przypadku pacjentów takim atrybutem może być na przykład diagnoza choroby lub prognoza jej rozwoju. Uczenie systemu klasyfikującego polega na dostarczeniu do systemu zbioru przykładów z przypisanymi etykietami. Zbiór taki nazywamy *zbiorem trenującym / uczącym*. Na podstawie przykładów ze zbioru uczącego system wytwarza wewnętrzną reprezentację, która następnie umożliwia przypisanie nieznanym klasyfikatorowi etykiet/klas nowym przykładom, które nie występowały w zbiorze uczącym.

Formalizacja problemu klasyfikacji

W celu uściślenia dalszych rozważań konieczne jest wprowadzenie notacji formalnej opisującej problem klasyfikacji [KS03]. Zbiór wszystkich możliwych obiektów x_i , których dotyczy dany problem klasyfikacji, nazywany jest dziedziną i jest oznaczany przez U . Atrybut $a_j(x_i) : U \rightarrow V_{aj}$ to dowolna funkcja określona na dziedzinie U z przeciwdziedziną A_j . Zbiór wszystkich atrybutów oznaczamy przez $A = \{a_1, a_2, \dots, a_n\}$.

Każdy przykład $x \in X$ można opisać jako wektor w n -wymiarowej przestrzeni atrybutów Ω , czyli $\langle a_1(x), a_2(x), \dots, a_n(x) \rangle$.

Problem klasyfikacji polega na znalezieniu odwzorowania, które każdemu $x_i \in D$ (gdzie D to zbiór danych wejściowych) przypisuje jego klasę y_i . W przypadku klasyfikacji binarnej $y_i \in \{-1, 1\}$.

Metody oceny skuteczności klasyfikacji

W celu oceny skuteczności systemu należy za jego pomocą dokonać klasyfikacji przypadków, które nie były użyte podczas jego uczenia i których etykiety są znane (choć nie dostępne klasyfikatorowi). Wyniki klasyfikacji porównuje się z właściwymi etykietami i w ten sposób szacuje skuteczność klasyfikacji. W tym celu można wydzielić ze zbioru przykładów specjalny podzbiór, zwany *zbiorem testującym*, który jest używany do testowania a w fazie uczenia klasyfikator nie ma do niego dostępu. Czasami wydziela się też *zbiór walidujący*, który jest używany w trakcie uczenia w celu optymalizacji parametrów algorytmu. Stałego podziału zbioru przykładów na zbiór trenujący, testujący i walidujący można dokonać tylko wtedy, gdy zbiory te są wystarczająco liczne. W przeciwnym przypadku może okazać się, że nie są one wystarczająco reprezentatywne i na przykład rozkład przykładów z poszczególnych klasy jest mocno skrzywiony w którymś ze zbiorów. Aby tego uniknąć można posłużyć się metodą *k-krotnej walidacji krzyżowej*. Polega ona na podzieleniu zbioru na k podzbiorów i następnie powtarzanych k -razy fazach uczenia i testowania klasyfikatora, przy czym za każdym razem k -ty podzbiór służy jako zbiór testujący/walidujący a pozostałe podzbiory jak zbiór uczący. Skuteczności klasyfikacji oblicza się wtedy jako średnią sprawność osiąganą we wszystkich k testach.

2.1.2 Miary skuteczności klasyfikacji

Do oceny jakości klasyfikacji można używać różnych miar. W przypadku klasyfikacji binarnej (czyli kiedy rozróżniamy tylko dwie klasy przykładów) większość z nich można wyrazić za pomocą stosunku kilku z czterech wartości wyrażających liczbę przypadków klasyfikowanych w określony sposób. Wartości te są odnoszą się zawsze do jednej z klas, która jest w pewien sposób wyróżniona. Na przykład w przypadku diagnozy medycznej zazwyczaj taką klasą jest grupa osób chorych na jakąś chorobę. Przypadki zaklasyfikowane jako należące do tej klasy określane są jako zaklasyfikowane *pozytywnie* (ang. *positive*) (+) natomiast przypadki zaklasyfikowane jako do niej nienależące jako zaklasyfikowane *negatywnie* (ang. *negative*) (−). Słowa ang. 'True' oraz ang. 'False' odnoszą się odpowiednio do przypadków zaklasyfikowanych prawidłowo i nieprawidłowo:

- **True Positive (TP)** — liczba przypadków **poprawnie** zaklasyfikowanych jako **należące** do wyróżnionej klasy,
- **True Negative (TN)** — liczba przypadków **poprawnie** zaklasyfikowanych jako **nienależące** do wyróżnionej klasy,
- **False Positive (FP)** — liczba przypadków **niepoprawnie** zaklasyfikowanych jako **należące** do wyróżnionej klasy (inaczej błąd pierwszego rodzaju)
- **False Negative (FN)** — liczba przypadków **niepoprawnie** zaklasyfikowanych jako **nienależące** do wyróżnionej klasy (inaczej błąd drugiego rodzaju)

TABLICA 2.1: Miary jakości klasyfikacji

		Rzeczywista klasa		
		+	−	
Przewidziana klasa	+	TP	FP	Wartość predykcyjna dodatnia (Precyzja) $\frac{\sum TP}{\sum TP + \sum FP}$
	−	FN	TN	Wartość predykcyjna ujemna $\frac{\sum TN}{\sum TN + \sum FN}$
		Czułość $\frac{\sum TP}{\sum TP + \sum FN}$	Swoistość $\frac{\sum TN}{\sum TN + \sum FP}$	

Poniżej zostały opisane miary, o których będzie mowa w dalszej części pracy. Wszystkie one zawierają się w przedziale $\langle 0, 1 \rangle$.

- **Precyzja/Wartość predykcyjna dodatnia** (ang. *precision/Positive predictive value*) — określa jaka część przypadków zaklasyfikowanych jako należące do wyróżnionej klasy rzeczywiście do niej należy. Dana jest wzorem:

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

- **Wartość predykcyjna ujemna** (ang. *Negative Predictive Value*) — określa jaka część przypadków zaklasyfikowanych jako nienależące do wyróżnionej klasy rzeczywiście do niej nie należy. Dana jest wzorem:

$$precision = \frac{TN}{TN + FN}$$

- **Kompletność/czułość** (ang. *recall/sensitivity*) — określa jaka część przypadków należących do wyróżnionej klasy została prawidłowo zaklasyfikowana jako należące do niej. Dana jest wzorem:

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

- *Swoistość* (ang. *specificity*) — określa jaka część przypadków nienależących do wyróżnionej klasy została prawidłowo zaklasyfikowana jako nienależąca do niej. Dana jest wzorem:

$$specificity = \frac{TN}{TN + FP}$$

- *definicja Trafność (lub dokładność)* (ang. *Accuracy*) — stosunek liczby przypadków ze zbioru walidującego, które zostały zaklasyfikowane poprawnie do liczby wszystkich przypadków w zbiorze testującym. Może być wyrażona jako:

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

- *Miara F_1* (ang. *F_1 measure*) — miara uwzględniająca zarówno precyzję (ang. *precision*) jak i kompletność (ang. *recall*). Miara ta nie uwzględnia wartości TN . Jej wartość jest dana wzorem:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

- *MCC ang. Matthews correlation coefficient* — miara, która w przeciwieństwie do miary F_1 bierze pod uwagę wszystkie cztery wartości (TP , TN , FP i FN). Dana wzorem:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

- *Średnie prawdopodobieństwo wyboru właściwej klasy* — niektóre klasyfikatory zamiast przypisywać każdemu z przykładów jedną z klas potrafią zwrócić dla każdego przykładu rozkład przynależności do wszystkich rozważanych klas. Jakość klasyfikacji można wtedy obliczyć jako uśrednioną po wszystkich przykładach wartość prawdopodobieństwa przypisanego klasie, do której przykład należy. Wartość taka może wahać się od wartości 0 (kiedy dla każdego przykładu do jego właściwej klasy zostało przypisane prawdopodobieństwo 0) do wartości 1 (kiedy dla każdego przykładu do jego właściwej klasy zostało przypisane prawdopodobieństwo 1).

W przypadku problemów, w których wyróżnia się $k > 2$ klas miara korzystająca z wartości TP, TN, FP i FN jest obliczana jako średnia wartość tej miary dla k problemów binarnych polegających na zaklasyfikowaniu przykładów jako należących lub nienależących do wybranej klasy.

2.2 SVM — Maszyny wektorów wspierających

Maszyna wektorów wspierających (SVM, ang. *Support Vector Machine*) to rodzaj klasyfikatora binarnego. Stanowi on rozszerzenie *klasyfikatora liniowego*, lecz w przeciwieństwie do niego jest w stanie poprawnie klasyfikować dane *nieseparowalne liniowo*. Jest to możliwe dzięki dokonywanej przez SVM transformacji danych do wyższych wymiarów za pomocą *funkcji jądrowych*.

2.2.1 Klasyfikatory liniowe

Jednym z najprostszych klasyfikatorów jest *klasyfikator liniowy*. Rozwiązuje on problem klasyfikacji binarnej poprzez znalezienie w przestrzeni atrybutów Ω hiperpłaszczyzny, która dzieli ją na dwie części odpowiadające dwóm klasom decyzyjnym: $\{-1, 1\}$.

Definicja 2.2.1 *Hiperpłaszczyzna w przestrzeni Ω to zbiór:*

$$\{x \in \Omega \mid \langle w, x \rangle + b = 0\}, w \in \Omega, b \in \mathbb{R} \quad (2.1)$$

$\langle x, y \rangle$ oznacza iloczyn skalarny wektorów x i y :

$$\langle x, y \rangle = \sum_{i=1}^N x[i]y[i]$$

gdzie $x[i]$ to i -ta wartość wektora x .

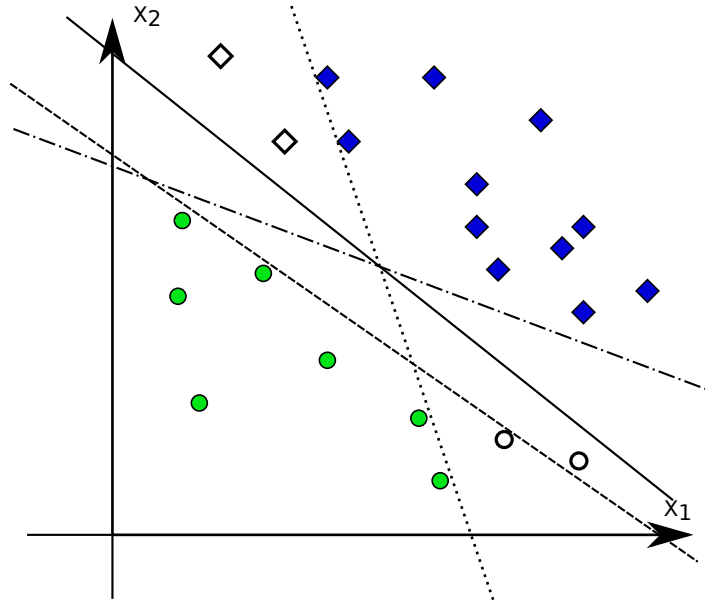
Wektor w we wzorze 2.1 to wektor wag, normalny do hiperpłaszczyzny, $\|w\|$ to norma euklidesowa tego wektora, czyli jego długość, a $b/\|w\|$ to odległość płaszczyzny od początku układu współrzędnych. Oba te parametry można razem dowolnie przeskalowywać, to jest pomnożyć w i b przez tę samą stałą zachowując tę samą hiperpłaszczyznę. Dlatego wprowadza się ograniczenie, po którego zastosowaniu otrzymujemy tak zwaną postać kanoniczną hiperpłaszczyzny:

Definicja 2.2.2 Dla danego zbioru obserwacji $x_1, x_2, \dots, x_m \in \Omega$ wektor w i parametr b wyznaczają **kanoniczną postać hiperpłaszczyzny** jeśli:

$$\min_{i=1..m} |\langle w, x_i \rangle + b| = 1 \quad (2.2)$$

Hiperpłaszczyzna dana wzorem 2.1 definiuje funkcję decyzyjną, która każdemu przypadkowi z Ω przypisuje klasę decyzyjną:

$$\begin{aligned} f_{w,b} : \Omega &\rightarrow \{\pm 1\} \\ x &\mapsto f_{w,b}(x) = \text{sgn}(\langle w, x \rangle + b) \end{aligned} \quad (2.3)$$



RYSunek 2.1: Hiperpłaszczyzny separujące dwa zbiory punktów w przestrzeni dwuwymiarowej. Każda z nich poprawnie separuje punkty ze zbioru uczącego — zielone koła i niebieskie kwadraty. Przykłady ze zbioru testowego (puste kwadraty i kółka) są poprawnie separowane jedynie przez dwie proste (prosta narysowana linią ciągłą i prosta narysowana kropkami i kreskami).

Wynikiem uczenia klasyfikatora liniowego jest znalezienie hiperpłaszczyzny i odpowiadającej jej funkcji decyzyjnej, która przykładom ze zbioru uczącego $(x_i, y_i) \in \Omega$ przypisuje prawidłowe etykiety, czyli dla każdego (jeśli zbiór jest liniowo separowalny), lub dla jak największej liczby przypadków x_i zachodzi $f_{w,b}(x_i) = y_i$. Zazwyczaj kilka hiperpłaszczyzn równie dobrze rozdziela przypadki ze zbioru uczącego, mogą się jednak one różnić zdolnością do klasyfikacji zbioru testowego, co pokazano na

rysunku 2.2.1. Optymalna hiperpłaszczyzna separująca to taka, która charakteryzuje się największym marginesem, czyli odległością hiperpłaszczyzny do najbliższych położonych obserwacji [SS02].

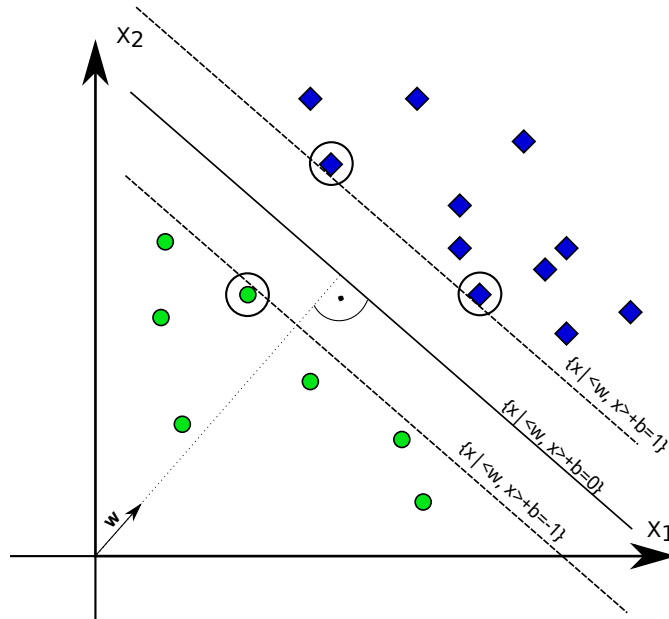
Definicja 2.2.3 Dla hiperpłaszczyzny danej wzorem $x \in \Omega | \langle w, x \rangle + b = 0$ oraz zbioru obserwacji $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ marginesem tego zbioru od hiperpłaszczyzny nazywamy minimalną odległość hiperpłaszczyzny od punktów z tego zbioru:

$$\rho_{w,b} := \min_{i=1..m} y_i (\langle w, x_i \rangle + b) / \|w\| \quad (2.4)$$

Żeby maksymalizować margines powinniśmy minimalizować $\|w\|$, zachowując warunek 2.2. Problem znalezienia optymalnej hiperpłaszczyzny separującej zbiór przykładów $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ można zatem zapisać jako problem optymalizacyjny:

$$\begin{aligned} \min_{w \in \Omega, b \in \mathbb{R}} \quad & \tau(w) = \frac{1}{2} \|w\|^2 \\ \text{p.o.} \quad & y_i (\langle w, x_i \rangle + b) \geq 1 \quad \text{dla } i = 1..m \end{aligned} \quad (2.5)$$

Ograniczenia w powyższym problemie zapewniają, że wartość funkcji decyzyjnej $f_{w,b}(x_i)$ będzie równa y_i , czyli, że hiperpłaszczyzna poprawnie odseparuje przykłady z dwóch grup. Osiągnięcie celu optymalizacji zapewnia znalezienie hiperpłaszczyzny o maksymalnym marginesie.



RYСУNEK 2.2: Hiperpłaszczyzna separująca dwa zbiory punktów w przestrzeni dwuwymiarowej wraz z marginesami. Przykłady w kółku to wektory podpierające.

Powyższy problem programowania matematycznego jest podany w tak zwanej *formie prymalnej*. W praktyce rozwiązuje się wersję *dualną* problemu, która ma przyjmować następującą postać:

Definicja 2.2.4 Wersja dualna problemu znalezienia optymalnej hiperpłaszczyzny:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ \text{p.o.} \quad & \alpha_i \geq 0, \text{ dla } i = 1..m \\ & \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned} \quad (2.6)$$

Dla formy dualnej problemu optymalizacji funkcja decyzyjna przyjmuje postać:

$$f(x) = \operatorname{sgn} \left(\sum_{i=1}^m y_i a_i \langle x, x_i \rangle + b \right) \quad (2.7)$$

We wzorze 2.5 dla większości przykładów wartość współczynników α_i wyniesie 0. Przykłady te nie mają wpływu na wynik optymalizacji i otrzymaną hiperpłaszczyznę. Pozostałe przykłady, dla których $\alpha_i > 0$ noszą nazwę *wektorów wspierających*. Leżą one dokładnie na marginesie (na hiperpłaszczyźnie wyznaczonej przez margines, czyli w przypadku kanonicznej postaci hiperpłaszczyzny separującej w odległości $1/\|w\|$ od niej). Wektory wspierające są jedynymi przykładami ze zbioru trenującego, które należy zapamiętać w celu wyznaczenia hiperpłaszczyzny separującej. Ilość wektorów wspierających wyznacza złożoność hipotezy, którą stanowi otrzymana funkcja decyzyjna i pozwala oszacować górną granicę oczekiwanego prawdopodobieństwa błędu klasyfikacji. Górna granica prawdopodobieństwa błędu klasyfikacji klasyfikatora uczonego na zbiorze o wielkości $m-1$ równa się liczbie wektorów otrzymanych przy uczeniu na zbiorze o wielkości m podzielonej przez m .

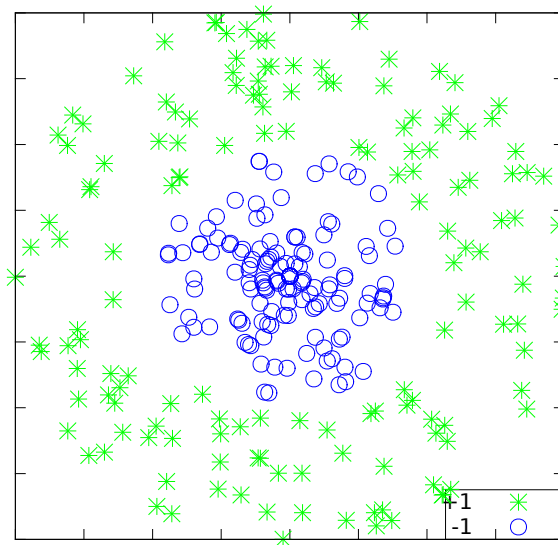
W przypadku, w którym dane nie są separowalne liniowo ze względu na szum, tzn. nie istnieje taka hiperpłaszczyzna, która dla danego zbioru przykładów spełnia ograniczenia ze wzoru 2.5, do problemu optymalizacyjnego wprowadza się tak zwane *zmienne osłabiające* (ang. *slack variables*) $\zeta_i, i = 1..m$, oraz stałą $C > 0$, która jest parametrem algorytmu. Wzór 2.5 przyjmuje wówczas postać:

$$\begin{aligned} \min_{w \in \Omega, b \in \mathbb{R}} \quad & \tau(w) = \frac{1}{2} \|w\|^2 + \frac{C}{m} \sum_{i=1}^m \zeta_i \\ \text{p.o.} \quad & y_i (\langle w, x \rangle + b) \geq 1 - \zeta_i \quad \text{dla } i = 1..m \\ & \zeta_i \geq 0 \quad \text{dla } i = 1..m \end{aligned} \quad (2.8)$$

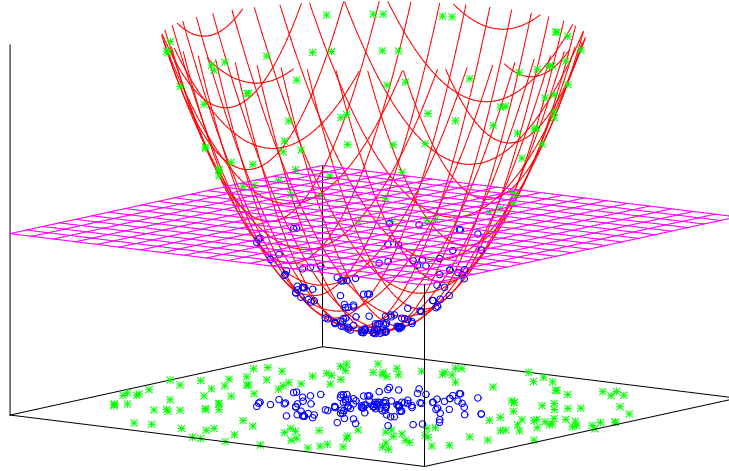
Natomiast wzór 2.6:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ \text{p.o.} \quad & 0 \leq \alpha_i \leq \frac{C}{m}, \text{ dla } i = 1..m \\ & \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned} \quad (2.9)$$

2.2.2 Maszyny wektorów wspierających



RYСУNEK 2.3: Zbiór danych opisanych w przestrzeni 2D. Klas (niebieskie kółka - klasa +1, zielone krzyżyki - klasa -1) nie da się odseparować za pomocą hiperpłaszczyzny



RYСУNEK 2.4: Zbiór z rysunku 2.2.2 po dokonaniu transformacji do przestrzeni 3-wymiarowej. Na wysokości 0 pokazano dane przed transformacją. W nowej przestrzeni klasy mogą być rozdzielone hiperpłaszczyzną.

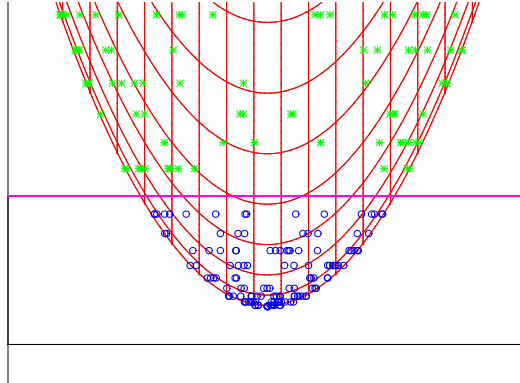
Opisane powyżej klasyfikatory są w stanie klasyfikować tylko dane, które są separowalne liniowo. Jeśli dane cechuje pewna inna niż liniowa zależność między wartościami atrybutów decyzyjnych i warunkowych (na przykład taka jak przedstawiona na rysunku 2.2.2), to klasyfikator liniowy nie poradzi sobie z ich klasyfikacją. Aby rozwiązać ten problem można dokonać mapowania danych wejściowych do przestrzeni o większej liczbie wymiarów, w której te dane mogą stać się liniowo separowalne. Na rysunku 2.2.2 pokazano dane z rysunku 2.2.2 po transformacji do przestrzeni trójwymiarowej za pomocą funkcji wielomianowej. Jak widać w nowej przestrzeni dane są separowalne przez hiperpłaszczyznę, co widać dokładnie na rysunku 2.2.2.

Mapowanie z oryginalnej przestrzeni atrybutów do nowej przestrzeni jest dokonywane przez funkcję $\Phi: \Omega^m \mapsto \Omega^{m+k}$, która dane n -wymiarowe zamienia w dane $n+k$ -wymiarowe. Na przykład mapowanie wektora $x = [x_1, x_2]$ z przestrzeni 2-wymiarowej do przestrzeni 3-wymiarowej może wyglądać następująco: $\Phi(x) = \Phi([x_1, x_2]) = [x_1^2, x_2^2, x_1 x_2]$. Aby zastosować mapowanie w klasyfikatorze wystarczy zamienić wszystkie wystąpienia wektorów wynikiem ich mapowań, czyli na przykład cel optymalizacji ze wzoru 2.6 przyjąłby postać:

$$\max_{\alpha \in R^m} W(\alpha) = \sum_{i=1}^m \alpha_i - \sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \langle \Phi(x_i), \Phi(x_j) \rangle \quad (2.10)$$

natomiast funkcja decyzyjna ze wzoru 2.7:

$$f(x) = \text{sgn} \left(\sum_{i=1}^m y_i a_i \langle \Phi(x), \Phi(x_i) \rangle + b \right) \quad (2.11)$$



RYSUNEK 2.5: Zbiór z rysunku 2.2.2 po dokonaniu transformacji do przestrzeni 3-wymiarowej, rzut wzdłuż osi X

Powyższe podejście charakteryzuje się dużą złożonością obliczeniową. Zarówno obliczanie mapowania dla każdego przykładów jak i obliczenie iloczynu skalarnego przykładów w nowej przestrzeni, która może mieć bardzo dużo wymiarów, stanowi spory narzut obliczeniowy. Dokonywanie wszystkich mapowań a następnie obliczanie ich iloczynu nie jest jednak konieczne, można obie te operacje wykonać w jednym kroku, stosując tak zwany ang. *Kernel Trick* [BGV92]. Polega on na zastosowaniu podstawienia $\langle \Phi(x), \Phi(x') \rangle \mapsto k(x, x')$ gdzie $k(x, x')$ to *funkcja jądrowa* (ang. *kernel*). Po zastosowaniu *kernel trick* otrzymujemy następujący problem optymalizacyjny:

$$\begin{aligned} \max_{\alpha \in R^m} \quad & W(\alpha) = \sum_{i=1}^m \alpha_i - \sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ \text{p.o.} \quad & \alpha_i \geq 0, \text{ dla } i = 1..m \\ & \sum_{i=1}^m \alpha_i y_i = 0 \end{aligned} \quad (2.12)$$

oraz następującą funkcję decyzyjną:

$$f(x) = \text{sgn} \left(\sum_{i=1}^m y_i \alpha_i k(x, x_i) + b \right) \quad (2.13)$$

2.2.3 Funkcje jądrowe

Funkcję jądrowe pozwalają obliczyć iloczyn skalarny wyników mapowania ($\Phi()$) dwóch wektorów (x, x') do wysoko wymiarowej przestrzeni atrybutów (Ω^{m+k}), bez bezpośredniego dokonywania mapowania:

$$\Phi : \Omega^m \mapsto \Omega^{m+k}$$

$$\langle \Phi(x), \Phi(x') \rangle \mapsto k(x, x')$$

Najczęściej używane funkcje jądrowe to:

- Wielomianowa: $k(x, x') = \langle x, y \rangle^d$

- Gausowska (RBF): $k(x, x') = e^{-\sigma * ||x-y||^2}$
- Sigmoidalna: $k(x, x') = \tanh(\gamma \langle x, y \rangle + \tau)$

Jako funkcję jądrową w SVM można użyć jednak dowolnej symetrycznej funkcji, która jest dodatnio określona (ang. *positive definite*), czyli dla której macierz:

$$K_{ij} := k(x_i, x_j)$$

spełnia warunek:

$$\sum_{i,j} c_i c_j K_{ij} \geq 0 \quad (2.14)$$

dla wszystkich $c_i \in \mathbb{R}$ co jest równoważne temu, że jej wszystkie wartości własne są dodatnie [SS02].

Spełnienie powyższego warunku gwarantuje, że rozwiązywany problem programowania matematycznego będzie wypukły, czyli będzie miał tylko jedno minimum lokalne, będące zarazem minimum globalnym. Dla funkcje jądrowych, które nie są dodatnio określone, rozwiązywany w SVM problem wciąż może być wypukły, pod warunkiem, że są one warunkowo dodatnio określone (ang. *conditionally positive definite*). Jest to możliwe dzięki ograniczeniu 2.12, które wyklucza pewne wartości współczynników α . Wyniki eksperymentalne pokazują, że w praktyce funkcje, które są jedynie warunkowo dodatnio określone, mogą dawać równie dobre wyniki co funkcje dodatnio określone [?]. Funkcja jest warunkowo dodatnio określona, jeśli spełnia warunek ze wzoru 2.14 przy czym $\sum_{i,j} c_i = 0$.

Funkcje jądrowe cechuje własność domknięcia ze względu na pewne operacje. Oznacza to, że funkcje powstałe poprzez połączenie poprawnych funkcji jądrowych za pomocą tych operatorów również są poprawnymi funkcjami jądrowymi [STC04]. Jeżeli $k_1(x, z)$ i $k_2(x, z)$ są kernelami, to również poniższe funkcje są kernelami:

$$\begin{aligned} k(x, x') &= k_1(x, z) + k_2(x, x') \\ k(x, x') &= k_1(x, z) \times k_2(x, x') \\ k(x, x') &= a \times k_1(x, x') \\ k(x, x') &= e^{k_2(x, x')} \\ k(x, x') &= p(k_1(x, x')) \end{aligned} \quad (2.15)$$

gdzie $a \in \mathbb{R}^+$, a $p()$ to wielomian o dodatnich współczynnikach.

Algorytmy wykorzystujące funkcje jądrowe

SVM nie jest jedynym algorytmem wykorzystującym funkcję jądrową do dokonania mapowania danych do wysoko wymiarowych przestrzeni. Ich użycie jest zasadne wszędzie tam, gdzie zamiast działać wprost na wejściowych wektorach danych można operować na ich iloczynach skalarnych [SS02]. W związku z tym funkcje jądrowe znalazły zastosowania w wielu algorytmach, zwanych zbiorczo jak metody jądrowe (ang. *kernel methods*):

- *Kernel-NN* [YJZ02] — modyfikacja k -NN korzystająca z funkcji jądrowych do obliczania odległości między przykładami
- definicja PCA [SS02] — modyfikacji algorytmu analizy głównych składowych *Principal Component Analysis* (PCA), który może służyć m.in. do konstrukcji cech opisujących przykłady, klasyfikowane potem za pomocą jakiegoś algorytmu klasyfikacji.
- *Support Vector Regression* (SVR) — regresja za pomocą SVM
- *Kernel Fisher Discriminant* (KFD) [SS02] — zastosowanie funkcji jądrowych w liniowym dyskryminatorze Fishera, który może służyć do redukcji wymiarów opisujących przykłady i/lub do klasyfikacji.

2.2.4 Klasyfikacja więcej niż dwóch klas

Klasyfikator SVM jest ze swej natury klasyfikatorem binarnym, to znaczy potrafi separować jedynie dwie klasy przykładów. W praktyce jednak wiele problemów klasyfikacji dotyczy zbiorów, w których wyróżniono więcej niż dwie klasy. Aby zastosować SVM do rozwiązania takich problemów stosuje się zazwyczaj jedną z dwóch technik: *jeden przeciw wszystkim* (ang. *One versus the Rest*) oraz *klasyfikacja parami* (ang. *pairwise classification*).

Jeden przeciw wszystkim

Metoda *jeden przeciw wszystkim* polega na nauczaniu M klasyfikatorów binarnych, gdzie M oznacza liczbę klas. Każdy z nich separuje jedną klasę od pozostałych $M - 1$ klas, traktując te ostatnie jak jedną klasę. Klasyfikacja polega na zastosowaniu wszystkich M klasyfikatorów na testowanym przykładzie i wybraniu tego, który maksymalizuje wartość ze wzoru 2.13, przed zastosowaniem funkcji sgn , czyli:

$$\arg \max_{j=1..M} g^j(x)$$

gdzie
$$g^j(x) = \sum_{i=1}^m y_i a_i^j k(x, x_i) + b^j$$

Jako wady tej metody wymienia się to, że pod-problemy przez nią rozwiązywane są niesymetryczne (jeśli w wyjściowym problemie dystrybucja przykładów do klas jest równomierna, to w każdym pod-problemie klasa agregująca "pozostałe" klasy jest $M - 1$ razy większa). Z problemem asymetryczności lepiej radzi sobie druga z wymienionych metod.

Klasyfikacja parami

Metoda *klasyfikacji parami* polega na stworzeniu wszystkich możliwych kombinacji klasyfikatorów binarnych: dla każdej z M klas tworzy się $M - 1$ klasyfikatorów oddzielających ją od poszczególnych klas, co daje łącznie $(M - 1) * M/2$ klasyfikatorów. Każdy z nich uczony jest tylko na przykładach należących do 2 wybranych klas. Klasyfikacja przykładu odbywa się poprzez głosowanie: każdy z $(M - 1) * M/2$ klasyfikatorów oddaje głos na klasę, do której przypisał dany przykład. Przykładowi przypisuje się klasę, która zdobyła największą liczbę głosów. Choć w tej metodzie liczba klasyfikatorów, które trzeba nauczyć a potem użyć do klasyfikacji jest znacznie większa niż w metodzie *jeden przeciw wszystkim*, to jednak uczenie odbywa się na mniejszej liczbie przykładów a klasyfikacja z użyciem mniejszej liczby wektorów wspierających, co może przyspieszyć sprawia, że etapy te przebiegają szybciej niż w metodzie *jeden przeciw wszystkim*. Ponieważ jednak liczba klasyfikatorów rośnie potęgowo względem liczby klas M , to dla dużych M *klasyfikacja parami* może okazać się wolniejsza niż *jeden przeciw wszystkim*.

2.3 Obliczenia ewolucyjne

Obliczenia ewolucyjne (ang. *Evolutionary computation (EC)*) to grupa inspirowanych biologicznie technik obliczeniowych. Algorytmy należące do tej grupy to *Algorytmy Ewolucyjne* (ang. *Evolutionary Algorithms (EA)*). Algorytmy te stanowią rodzaj metaheurystyk, czyli uniwersalnych algorytmów służących rozwiązywaniu różnych problemów optymalizacji, często wykorzystujących w trakcie działania bardziej specyficzne algorytmy. Algorytmy ewolucyjne wyróżnia wśród innych metaheurystyk między innymi przynależność do grupy algorytmów populacyjnych, które charakteryzują się równoległym przeszukiwaniem wielu rozwiązań, między którymi zachodzą interakcje wpływające na ocenę poszczególnych rozwiązań jak i kierunek ich zmian. Idea obliczeń ewolucyjnych czerpie swoje inspiracje z biologicznych teorii ewolucji, z nich również zapożycza terminologię. Poniżej przedstawiono znaczenie terminów używanych w kontekście obliczeń ewolucyjnych [Luk09]:

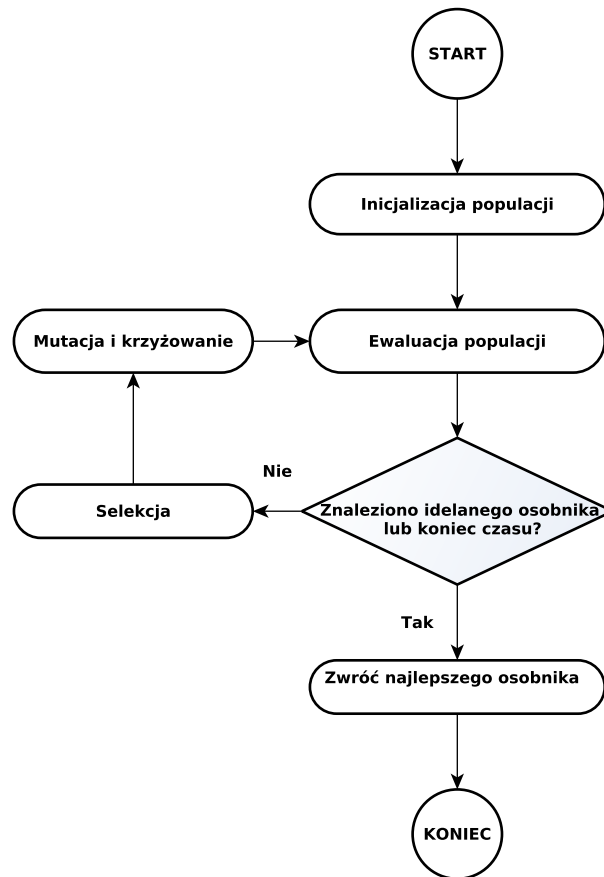
- *osobnik* (ang. *individual*) — jedno z potencjalnych rozwiązań problemu
- *genotyp* (ang. *genotype*) — dane opisujące osobnika, podlegające procesom mutacji i krzyżowania
- *relacja rodzic-potomek* (ang. *parent-parent*) — potomek to osobnik powstały poprzez mutację rodzica lub w wyniku krzyżowania dwóch rodziców
- *mutacja* (ang. *mutation*) — zmiana genotypu pojedynczego osobnika w wyniku której powstaje osobnik o nowym genotypie
- *krzyżowanie* (ang. *crossover*) — stworzenie jednego lub więcej nowych osobników, których genom jest kombinacją genomów rodziców
- *populacja* (ang. *population*) — zbiór osobników item *pokolenie* (ang. *generation*) — kompletny cykl operacji wykonanych na pokoleniu, również pokolenie będące jego wynikiem
- *przystosowanie / wartość funkcji przystosowania* (ang. *fitness*) — wartość określająca jakość osobnika ze względu na cel optymalizacji przyjęty w rozwiązywanym problemie
- *selekcja* (ang. *selection*) — wybór podzbioru osobników z populacji na podstawie ich przystosowania
- *powielanie* — utworzenie nowego osobnika poprzez skopiowanie istniejącego osobnika bez zmian
- *reprodukcja* (ang. *breeding*) — proces tworzenia nowego pokolenia (nowej generacji) osobników z osobników już istniejących poprzez procesy mutacji, krzyżowania i selekcji.

Przebieg typowego algorytmu ewolucyjnego przedstawiono na rysunku 2.6, poniżej przedstawiono jego opis:

1. Utwórz początkową populację osobników korzystając z losowego mechanizmu lub użyj istniejącej populacji
2. Oblicz wartość *funkcji dopasowania* każdego osobnika z populacji (dokonaj ewaluacji populacji).
3. Jeśli znaleziono idealnego osobnika (wartość fitness wyniosła 1) lub skończył się czas, zwróć najlepszego znalezione dotąd osobnika
4. Dokonaj selekcji osobników
5. Utwórz nową populację poprzez mutację, krzyżowanie lub powielanie wybranych w poprzednim kroku osobników
6. Wróć do punktu 2

Selekcja

Selekcja polega na wybraniu podzbioru P osobników z populacji. Jej celem jest promowanie dobrych rozwiązań, które powstały w wyniku generowania nowych osobników, mutacji oraz krzyżowania. Dzięki dokonywaniu selekcji geny osobników lepiej przystosowanych do rozwiązania postawionego przed algorytmem problemu są "przekazywane dalej", biorąc udział w tworzeniu nowych osobników. "Dobroć rozwiązań jest oceniana poprzez funkcję przystosowania, która przyjmuje wartości z przedziału $\langle 0, 1 \rangle$, gdzie 0 oznacza najgorsze rozwiązanie a 1 rozwiązanie idealne. Najprostszym, naiwnym", sposobem przeprowadzenia selekcji jest wybór z niej określonej liczby osobników o najwyższej wartości fitness. Rozwiązanie to jednak nie jest optymalne — czasami pożądane fragmenty genotypu są



RYSUNEK 2.6: Diagram przepływu algorytmu ewolucyjnego GP.

"ukryte" w osobnikach, które nie osiągają najwyższych wartości fitness, dlatego warto dopuścić również niektóre ze słabszych osobników do etapu krzyżowania. Głównym parametrem wpływającym na proces selekcji jest *napór selekcyjny*. Im większy napór selekcyjny tym trudniej osobnikom przejść selekcję co prowadzi do ujednolicania się osobników w populacji, ponieważ do następnego pokolenia przechodzi tylko wąska grupa najlepszych osobników, zmniejszając tym samym pulę genów pokolenia. Zmniejszanie się różnorodności jest niepożądanym procesem, ponieważ prowadzi do stagnacji procesu ewolucyjnego — kiedy wszystkie osobniki mają ten sam genotyp jedyne zmiany w populacji możliwe są poprzez mutację. Stagnacja procesu ewolucyjnego oznacza, że populacja z pokolenia na pokolenie się nie zmienia, ponieważ przez etap selekcji przechodzą zawsze te same, podobne do siebie osobniki. Jest to równoważne z utknięciem w optimum lokalnym (które może, choć nie musi być optimum globalnym rozwiązywanego problemu). Dlatego, żeby nie dopuścić do zbyt szybkiego ujednolicenia populacji stosuje się różne, bardziej wyrafinowane niż opisany powyżej naiwny, algorytmy selekcji:

- *ruletka* — metoda ruletki, zwana też wyborem losowym powtórzeniami ang. *Fitness-Proportionate Selection* polega na losowaniu osobników z prawdopodobieństwem wylosowania proporcjonalnym do ich wartości fitness. Metaforycznie można ją przedstawić w ten sposób, że każdemu osobnikowi przypisuje się pole na kole od ruletki o wielkości proporcjonalnej do jego wartości fitness i kręcąc ruletką losuje ustaloną liczbę osobników. Ta metoda selekcji dopuszcza wybranie jednego osobnika kilka razy a także umożliwia przypuszczenie osobników o małych wartościach fitness, co jest korzystne dla różnorodności populacji. Wadą tej metody jest ukryte w niej założe-

nie o ilorazowym charakterze skali wartości fitness — osobnik A o k -krotnie większym fitness od osobnika B będzie średnio wybierany k razy częściej niż osobnik B, choć w ogólności nie musi być tak, że k -krotnie większy fitness oznacza k -krotnie lepszego lub pożądanego z punktu widzenia puli genów populacji osobnika. Z tym problemem radzi sobie następna opisana metoda.

- *turniej* — metoda turniejowa (ang. *tournament selection*) jest najbardziej popularną i jednocześnie najprostszą z metod selekcji [Luk09]. Polega na losowaniu ze zwracaniem t osobników i wybraniu tego, który ma największą wartość fitness. Parametr t to tak zwany *rozmiar turnieju*, zwiększając go zwiększamy presję selekcyjną. Dla $t = 1$ selekcja sprowadza się do losowego wyboru osobników, dla $t = P$ otrzymamy populację składającą się z P kopii osobnika o najwyższym fitness.

Krzyżowanie

Osobniki wybrane w procesie selekcji z ustalonym jako parametr procesu ewolucyjnego prawdopodobieństwem ulegają krzyżowaniu, czyli wymieszaniu ich genotypów, w czego wyniku powstaje nowy osobnik o przypuszczalnie nieobecnym wcześniej genotypie.

Mutacja

Mutacja polega na losowej, zazwyczaj nieznacznej, zmianie genotypu osobnika. Wpływa korzystnie na różnorodność populacji wprowadzając fragmenty genomu, które mogły wcześniej nie być obecne w puli genów populacji. Mutacja podobnie jak krzyżowanie jest stosowana z pewnym prawdopodobieństwem. Im to prawdopodobieństwo większe a zmiany dokonywane przez mutację bardziej rozległe, tym bardziej eksploratywny staje się algorytm ewolucyjny — trudniej mu "utknąć" w lokalnym optimum.

Oprócz krzyżowania i mutacji możliwe jest również kopiowanie osobników do nowej populacji w niezmięnionej formie. Mechanizm elit czyni tak z określoną liczbą osobników o najwyższym fitness.

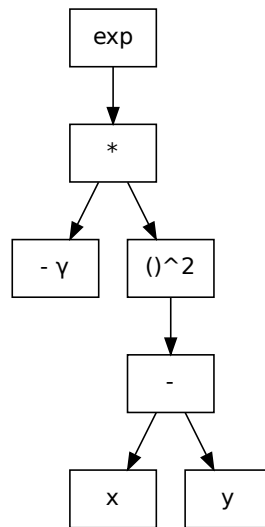
Genotyp

Poszczególne algorytmy ewolucyjne różnią się od siebie przede wszystkim ze względu na sposób reprezentacji osobników i powiązane z nim metody mutacji i krzyżowania. Selekcja osobników także może przebiegać na różne sposoby. Opisanie wszystkich możliwych parametrów i metod używanych w algorytmach ewolucyjnych nie jest celem tej pracy, ich opis można znaleźć na przykład w [Luk09]. Poniżej opisano jeden z rodzajów algorytmów ewolucyjnych jaką jest programowanie genetyczne, ze względu na jej wykorzystanie w niniejszej pracy.

2.3.1 Programowanie genetyczne

Programowanie genetyczne (GP, ang. *Genetic Programming*) to rodzaj algorytmów ewolucyjnych, które wyróżniają się specyficzną reprezentacją — ewolucji podlegają wyrażenia reprezentowane za pomocą drzew, na przykład wyrażenia matematyczne, logiczne lub kod programu komputerowego. Najczęściej w definicji programowania genetycznego podaje się właśnie ten ostatni rodzaj osobników [Luk09] [PLM08], lecz ewolucja innych wyrażeń reprezentowanym za pomocą drzew przebiega w algorytmie programowania genetycznego w identyczny sposób.

Drzewo generowane przez algorytm GP (np. takie jak pokazane na rysunku 2.7) składa się z *terminali* (liście) oraz *funkcji* (węzły wewnętrzne oraz korzeń). Terminale to wartości wejściowe generowanego wyrażenia — stałe i zmienne — na rysunku są to zmienne x i y oraz stała λ . Funkcje to operacje wykonywane wartościach pochodzących z terminali lub innych funkcji, w przypadku drzewa z ry-



RYСУNEK 2.7: Przykładowe drzewo, które może być wynikiem działania algorytmu GP. Drzewo przedstawia wyrażenie $e^{-\gamma * (x-y)^2}$

sunku 2.7 są to: funkcja wykładnicza $exp()$ oraz funkcja potęgowa (korzeń). Każdy węzeł wewnętrzny posiada łuk(i) wejściowe, którymi funkcja "pobiera" wartości swoich argumentów oraz łuk wyjściowy, którym przekazuje wartość do kolejnych funkcji (lub w przypadku korzenia zwraca wartość obliczoną przez całe wyrażenie).

Inicjalizacja populacji

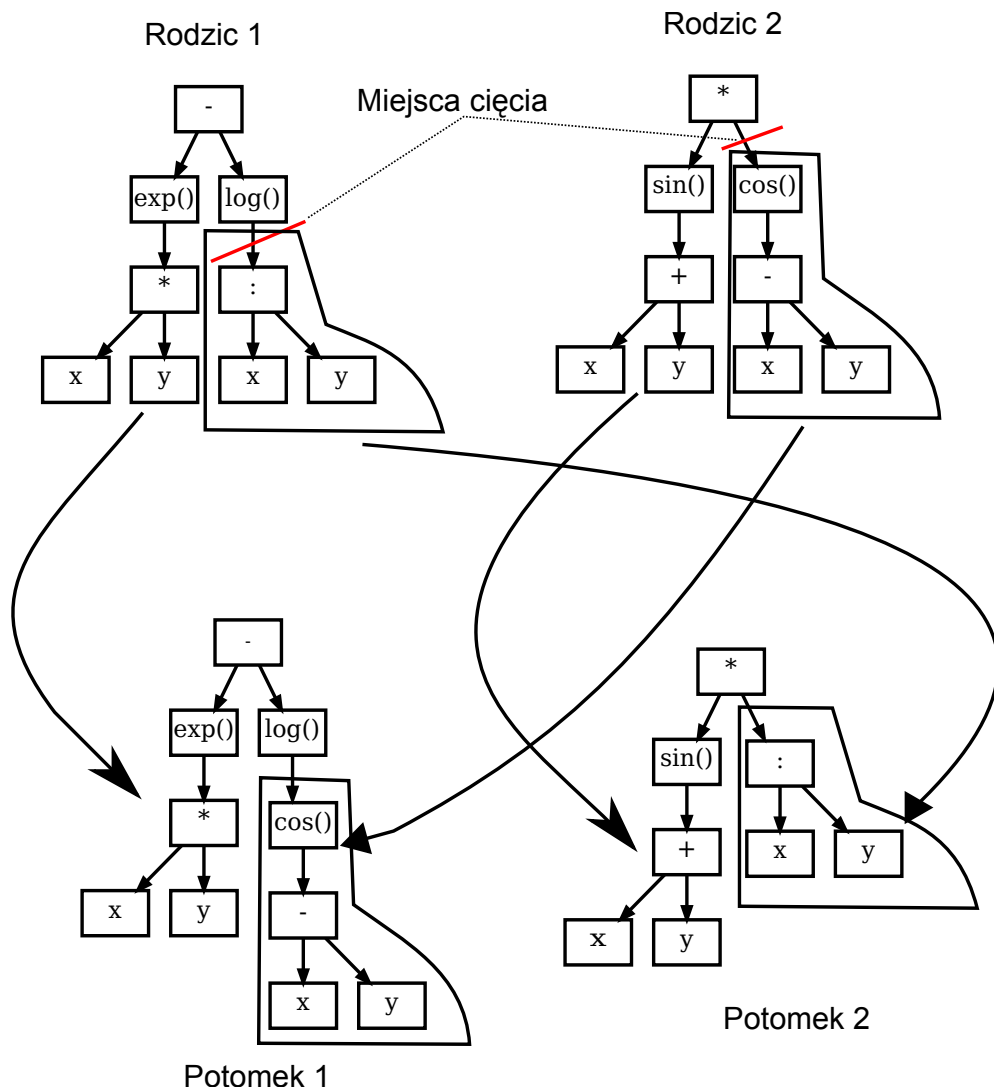
Wybranie sposobu inicjalizacji populacji sprowadza się do określenia metody losowego generowania osobników. Drzewa są budowane przez łączenie ze sobą funkcji wybieranych ze zbioru funkcji oraz terminali. Każda funkcja ma określoną liczbę argumentów oraz opcjonalnie typ danych wejściowych i wyjściowych. W najprostszym przypadku wszystkie węzły drzewa operują na tych samych typach danych, wtedy nie narzuca się ograniczeń na to które węzły mogą się ze sobą łączyć. Jeśli jednak zbiór funkcji i terminali jest heterogeniczny ze względu na wartości typów danych, konieczne jest wprowadzenie ograniczeń. Odmiana GP, w której każda funkcja ma zdefiniowane typy danych, na których operuje nazywany jest *Strongly Typed GP* STGP. Kolejnym ograniczeniem najczęściej narzucanym na generowane drzewa jest ich głębokość, czyli odległość od najdalszego liścia do korzenia. Ze względu na to ograniczenie wyróżnia się 3 podstawowe metody generowania osobników:

- *full* — generuje drzewa o maksymalnej dozwolonej głębokości, w których każdy liść jest oddalony od drzewa o maksymalną odległość. Genrowanie polega na budowaniu drzewa zaczynając od korzenia poprzez losowanie kolejnych funkcji ze zbioru funkcji i dołączenie ich do funkcji już istniejących w drzewie, tak długo aż wszystkie gałęzie osiągną oczekiwaną głębokość. Na końcu dla każdej gałęzi losuje się kończący ją terminal.
- *grow* — w tej metodzie węzły budujące drzewo są losowane z obu zbiorów: funkcji i terminali. Dzięki temu, w przypadku wylosowania terminala dana gałąź kończy swój wzrost i otrzymujemy z gałęziami o potencjalnie zróżnicowanej długości.

- *ramped half-and-half* — w metodzie tej połowę populacji inicjalizuje się za pomocą metody *grow* a drugą połowę za pomocą algorytmu *full*, dodatkowo zmieniając maksymalną możliwą głębokość. Metoda ta pozwala otrzymać populację najbardziej urozmaiconą pod względem kształtu i wielkości drzew.

Krzyżowanie

Najpowszechniejsza metoda krzyżowania w programowaniu genetycznym to *krzyżowanie poddrzew* ang. subtree crossover. Polega ona na zamianie poddrzew dwóch rodziców w czego wyniku powstaje dwóch potomków. Poddziewa otrzymuje się poprzez wybór punktu cięcia, w którym poddrzewo jest odcinane od rodzica. Przykładowy proces krzyżowania zilustrowano na rysunku 2.3.1.

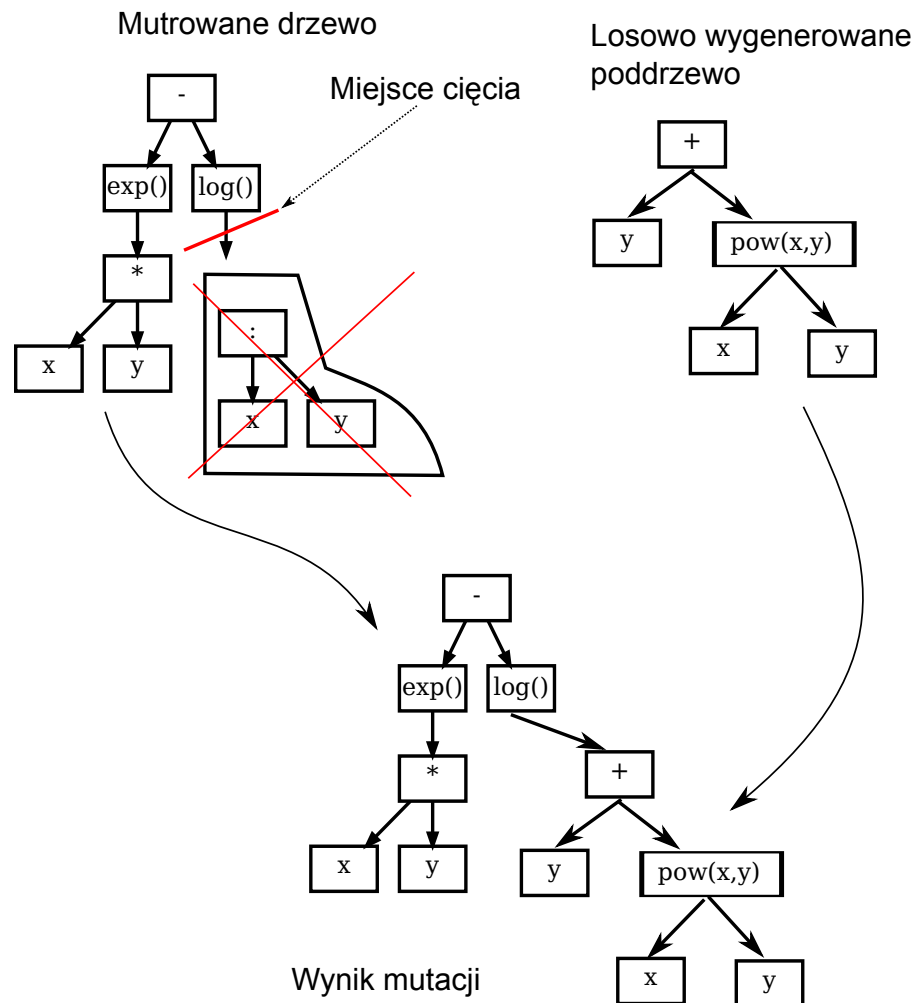


RYСУNEK 2.8: Krzyżowanie drzew. U góry pokazano drzewa rodziców, punkty cięcia zaznaczono czerwonymi kreskami. Na dole widać dwóch potomków będących wynikiem krzyżowania.

Mutacja

Mutacja drzewa polega na zamianie jego losowo wybranego fragmentu (poddziewa) przez losowo wygenerowane drzewo odpowiedniej długości. Pod tym względem mutacja przypomina krzyżowanie,

z tą różnicą, że odcięty fragment mutowanego drzewa jest zastępowany przez losowo wygenerowane drzewo a nie poddrzewo pochodzące od drugiego rodzica. Przykładową mutację drzewa pokazano na rysunku 2.3.1.



RYСУNEK 2.9: Przykładowa mutacja drzewa. U góry po lewej pokazano drzewo poddawane mutacji, u góry po prawej losowo wygenerowane poddrzewo, na dole wyniki mutacji. Odcinana część mutowanego drzewa (na rysunku przekreślona) zostaje odrzucona (nie jest już nigdzie wykorzystywana).

Selekcja

Selekcja w GP może być dokonywana jedną z metod używanych w innych algorytmach ewolucyjnych, jednak istnieją też metody specyficzne dla algorytmów GP. Jednym z problemów programowania genetycznego jest zjawisko *przerostu* (ang. *bloat*). Polega ono na tym, 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 może być regulowana przez kilka mechanizmów. Najprostszy z nich to proste ograniczenie na maksymalną głębokość drzewa — podczas inicjalizacji lub mutacji wzrost drzewa (poddrzewa) jest zatrzymywany kiedy osiągnie ono maksymalną głębokość. Algorytm może również odrzucać zbyt duże drzewa powstałe podczas krzyżowania. Grupa bardziej wyrafinowanych mechanizmów o angielskiej nazwie *parsimony pressure*, promuje mniejsze drzewa podczas selekcji. Do grupy tej należą algorytmy

[Sea10]:

- *Selekcja turniejowa leksykograficzna* — przebiega tak samo jak selekcja turniejowa, ale w przypadku, gdy porównywane są dwa osobniki o równej wartości fitness, wygrywa ten, który jest mniejszy
- *Selekcja turniejowa leksykograficzna z koszykami* (ang. *bucket lexicographic tournament selection*) — osobniki w populacji sortowane są według przystosowania, następnie grupuje się 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.
- *Selekcja turniejowa proporcjonalna* (ang. *proportional tournament selection*) — przebiega tak samo jak selekcja turniejowa, z tym, że z pewnym prawdopodobieństwem osobniki są porównywane według przystosowania albo według wielkości
- *Podwójna selekcja turniejowa* (ang. *double tournament selection*) — selekcja odbywa się w dwóch rundach — najpierw w N turniejach wybiera się N osobników, według ich wielkości (przystosowania), następnie tych N osobników bierze udział w turnieju, w którym porównywani są według przystosowania (wielkości)
- *ang. Tarpeian statistics* — nie jest to metoda selekcji, lecz sposób przypisania wartości przystosowania. Pewnej losowo wybranej części osobników o rozmiarze większym niż średni rozmiar w populacji przypisuje się bardzo niską miarę fitness i w danej rundzie nie są one ewaluowane.

2.4 Optymalizacja parametrów SVM — przegląd literatury

Na wyniki osiągane przez algorytm SVM duży wpływ ma dobór stosowanej funkcji jądrowej wraz z jej parametrami (σ dla jądra RBF, d dla jądra wielomianowego, oraz γ i τ dla jądra sigmoidalnego) oraz parametrów samego algorytmu (na przykład parametru C ze wzoru 2.9) [HCL03].

Problem wyboru funkcji jądrowej i parametrów stosowanych przez klasyfikator SVM można rozwiązać na kilka sposobów. Najczęściej stosuje się po prostu jedną z popularnych funkcji — RBF, sigmoidalną lub wielomianową, kierując się doświadczeniem, "dobrymi praktykami", lub ręcznie sprawdzając wyniki osiągane z użyciem tych funkcji i różnych wartości ich parametrów. Najczęściej stosowaną funkcją jądrową wydaje się być RBF [HCL03] [HM05].

Możliwa jest również automatyzacja poszukiwania optymalnych parametrów. Proces ten nosi nazwę *optymalizacji hiperparametrów* (ang. *Hyperparameter optimization*). Jest to określenie zbioru metod służących optymalizacji parametrów jakiegoś algorytmu.

2.4.1 Miary przystosowania (fitness)

W przypadku wszystkich metod optymalizacji parametrów SVM konieczne jest wybranie jakiejś funkcji celu, która będzie optymalizowana. Funkcja ta informuje o tym, czy zmiany parametrów idą w dobrą stronę i pozwala oceniać i porównywać różne ich kombinacje. Oprócz miar uniwersalnych dla wszystkich problemów klasyfikacji (patrz rozdział 2.1.2), możliwe jest stosowanie miar specyficznych dla algorytmu SVM, takich jak: [HM05] [RS04]

- liczba wektorów wspierających (minimalizacja)
- promień najmniejszej sfery R , o środku w początku układu współrzędnych, obejmującej wszystkie przykłady uczące (minimalizacja): $R = \max_{1 \leq i \leq m} (K(x_i, x_i))$

- suma współczynników α : $\sum_{i=1}^N \alpha_i$ Dąży się do minimalizacji tej miary, co prowadzi do maksymalizacji marginesu hiperpłaszczyzny [HM05].
- suma *zmiennych osłabiających* (ang. *slack variables*) ζ : $\sum_{i=1}^N \zeta_i$, której minimalizacja prowadzi do mniejszej liczby niepoprawnie zaklasyfikowanych przykładów ze zbioru uczącego

oraz miar będących kombinacjami powyższych, takich jak: [RS04]

- $R^2 \sum_{i=1}^m \alpha_i + \sum_{i=1}^m \zeta_i$
- $(R^2 + 1/C)(\|\omega\|^2 + 2 + 2C \sum_{i=1}^m \zeta_i)$
- $(R^2 + 1/C) \sum_{i=1}^m \alpha_i$

Miary te pomagają zapobiec w przeuczeniu algorytmu, to znaczy zapewnić, że skuteczność klasyfikacji zbioru testującego będzie podobna do tej osiągananej na zbiorze uczącym, ponieważ nie mierzą skuteczności wprost.

2.4.2 Optymalizacja parametrów

W najprostszym przypadku do dokonania hiperoptymalizacji można posłużyć się jedną z prostych metaheurystyk takich jak *Random Search* [E] lub *Grid Search* [HCL03]. *Grid Search* polega na zachłanym przeszukiwaniu przestrzeni parametrów ograniczonej przez podane zakresy dozwolonych wartości każdego parametru. Przeszukiwanie takie odbywa się z określoną rozdzielczością, czyli dla każdego parametru jest określona wielkość kroku δ , o którą zmieniany jest parametr. Ponieważ przeszukiwanie takie jest nieefektywne, zaproponowano również użycie jego ulepszonej wersji [Sta03], polegające na stopniowym zwiększaniu rozdzielczości przeszukiwania i jednoczesnym zawężaniu przeszukiwanej przestrzeni do tego podobszaru, w którym dla aktualnej rozdzielczości otrzymano najlepsze rezultaty.

Do optymalizacji parametrów SVM, używa się również algorytmów ewolucyjnych. Na przykład używano *Strategii ewolucyjnych* (ang. *Evolutionary Strategy*) w celu doboru optymalnych wartości C i σ dla jądra RBF [RS04].

2.4.3 Ewolucja kerneli

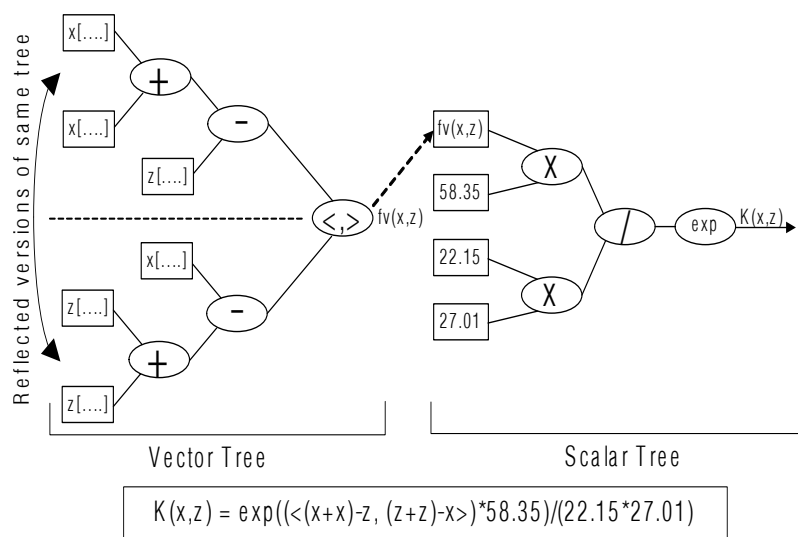
Niemniej ważny od doboru parametrów jest dobór funkcji jądrowej użytej przez SVM do dokonania mapowania danych do wysoko wymiarowej przestrzeni. Dlatego próbuje się wykorzystywać różnego rodzaju algorytmy ewolucyjne w celu znalezienia funkcji, która daje najlepsze wyniki klasyfikacji.

Genetic Kernel SVM

W roku 2005 Howley i Madden zaproponowali użycie programowania genetycznego do projektowania funkcji jądrowych na potrzeby SVM [HM05]. Swoją metodę nazwali ang. *Genetic Kernel Support Vector Machine* (GK SVM). Algorytm GP w ich podejściu buduje funkcje ze zbioru operacji dodawania, odejmowania i mnożenia, w dwóch wersjach: skalarnej i wektorowej. Terminale to wektory opisujące przykłady uczące. Tak stworzone funkcje są służą potem do stworzenia funkcji jądrowej według wzoru: $K(x, z) = \langle treeEval(x, z), treeEval(x, z) \rangle$, gdzie *treeEval* to wyrażenie zwrócone przez algorytm GP. Takie podejście zapewnia symetryczność funkcji jądrowej, choć nie zapewnia, że będzie ona dodatnio określona (nie musi zatem być poprawną funkcją jądrową w świetle teorii Mercera), a zatem nie zapewnia, że zostanie znalezione globalne optimum problemu optymalizacji rozwiązywanego wewnątrz SVM. Mimo to, takie funkcje mogą dawać dobre wyniki klasyfikacji SVM [BHB02]. Jako miarę przystosowania (fitness) Howley i Madden użyli błędu klasyfikacji i dodatkowo funkcji $fitness = \sum_{i=1}^m \alpha_i * R^2$

jako tak zwanego ang. *tiebreaker*, czyli funkcji służącej do uporządkowania osobników osiągających taki sam fitness. Autorzy przeprowadzili eksperymenty na sześciu zbiorach danych i porównali trafność klasyfikacji z użyciem funkcji wielomianowej, sigmoidalnej, RBF oraz stworzonych przez algorytm GP. Dla większości zbiorów wyewoluowane funkcje dawały wyniki lepsze lub porównywalne z najlepszymi wynikami otrzymanymi z pomocą jednej ze standardowych funkcji jądrowych z różnymi wartościami parametrów.

KTree



RYСУNEK 2.10: Przykładowe drzewo wygenerowane przez algorytm KTree (rysunek pochodzi z pracy [HM06]). Po lewej stronie widać część wektorową, po prawej część skalarną. Funkcja reprezentowana przez to drzewo to $e^{((x+x)-z, (z+z)-x) * 58.35 / (22.15 * 27.01)}$

Rok później ci sami autorzy zaprezentowali ulepszoną wersję swojego podejścia, tym razem zwaną *KTree* [HM06]. W podejściu tym drzewo reprezentujące generowany kernel składa się z dwóch części: wektorowej i skalarnej. Część wektorowa znajduje się na dole drzewa, zawiera jego liście, będące wektorami cech oraz funkcje operujące na wektorach, takie jak dodawanie i odejmowanie wektorów. Podobnie jak w poprzedniej wersji algorytmu buduje się drzewo symetryczne do wygenerowanego i oblicza ich iloczyn skalarny. Jego wynik, wraz z terminalami zawierającymi losowe stałe, stanowią liście drugiej części drzewa, która zawiera operacje wektorowe. Dzięki zastosowaniu takiej reprezentacji możliwe jest modelowanie za jej pomocą standardowych funkcji jądrowych, takich jak RBF. Przykład takiego drzewa pokazano na rysunku 2.4.3. W pracy przebadano 3 różne funkcje fitness: dwie oparte na błędzie klasyfikacji zbioru uczącego, jako *tiebreak* używające kolejno wielkości drzewa oraz sumy współczynników α : $\sum_{i=1..m} \alpha_i$ oraz trzecią: opartą na 3-krotnej cross walidacji na zbiorze uczącym z wielkością drzewa jako *tiebreaker*-em. Dodatkowo autorzy testowali użycie "filtra Mercera" (ang. *Mercer filter*). Jeśli dla podzbioru danych uczących macierz $K = (k(x, z)_{i,j=1}^N)$ ma ujemne wartości własne, to filtr przypisuje funkcji jądrowej k najgorszą możliwość fitness tym samym eliminując funkcje nie spełniające warunków narzuconych przez teorię Mercera. W przeprowadzonych przez autorów eksperymentach metoda KTree osiągała na większości z 9 przetestowanych zbiorów danych wyniki lepsze lub porównywalne do SVM ze standardowymi kernelami. Najlepszą funkcją fitness okazała się potrójna walidacja krzyżowa. Użycie filtra Mercera okazało się nie wpływać korzystnie na fitness ani osiąganą trafność klasyfikacji, jednak autorzy ostrzegają przed użyciem algorytmu bez filtra Mercera ze względu na groźbę przeuczenia.

Evolutionary Kernel Machine

Oryginalne podejście do problemu przedstawia metoda definicja Evolutionary Kernel Machine (*EKM*), przedstawiona w 2006 roku w [GSST06]. Funkcje jądrowe generowane są za pomocą GP, przy czym zbiór terminali stanowią funkcje operujące na wektorach a zwracające wartości skalarne (np. iloczyn skalarny, norma euklidesowa maksimum/minimum z wartości), natomiast wewnętrzne węzły pochodzą ze zbioru funkcji operujących na wartościach skalnych i zwracających wartości skalarne. Co nietypowe w tym podejściu, to funkcja fitness, która pochodzi z miary oceny marginesu zaproponowanej w [GBNT04] a używanej oryginalnie do oceny marginesu w algorytmie 1-NN:

$$F(K) = \frac{1}{m} \sum_{i=1}^m \delta_K(x'_1, y'_i) - l$$

, gdzie l to wielkość zbioru prototypów, a m to wielkość zbioru walidującego.

Wyrażenie $\delta_K(x, y)$ występujące w powyższym wzorze można przedstawić jako $\delta_K(x, y) = n(x, y) - p(x, y)$. Aby obliczyć $n(x, y)$ oraz $p(x, y)$ należy posortować przykłady ze zbioru prototypów $\{(x_1, y_1), \dots, (x_m, y_m)\}$ względem podobieństwa do przykładu (x, y) . $p(x, y)$ oznacza najniższą na posortowanej liście przykładów pozycję przykładu należącego do klasy y , czyli tej samej klasy co przykład (x, y) , $n(x, y)$ oznacza zaś najniższą pozycję przykładu należącego do klasy innej niż y . Miara podobieństwa, według której sortowane są przykłady pochodzi z użycia wygenerowanej funkcji jądrowej i wzorowana jest na algorytmie *Kernel-NN* [YJZ02]:

$$d_K(x, x')^2 = K(x, x) + K(x', x') - 2K(x, x')$$

Podsumowując: autorzy aby ocenić funkcje jądrowe na użytek algorytmu SVM, używają ich do skonstruowania klasyfikatora *Kernel-NN* i oceniają margines w tym algorytmie przy pomocy zbioru walidującego. Dodatkowo, oprócz ewolucji kerneli, autorzy stosują kooperatywną koewolucję zbioru prototypów ze zbiorem kerneli oraz koewolucję typu żywiciel-pasożyt zbioru walidującego ze zbiorem kerneli. Wyniki eksperymentów zaprezentowane w artykule pokazują porównanie błędów klasyfikacji otrzymanych za pomocą algorytmów $k-NN$, tradycyjnego SVM z jądrem RBF i metody *EKM*. W przypadku 2 z 6 przebadanych zbiorów danych algorytm *EKM* okazał się istotnie statystycznie lepszy od pozostałych dwóch algorytmów.

Kernel GP

W 2007 roku ukazał się artykuł autorstwa Seana Luke-a (autora m.in. wykorzystanej w tej pracy biblioteki ECJ [Sea10]) i Keith Sullivan opisujący algorytm definicja Kernel GP (*KGP*) [SL07]. Algorytm ten, podobnie jak opisane powyżej algorytmy, wykorzystuje programowanie genetyczne do wygenerowania funkcji jądrowych, jednak jako jedyny gwarantuje, że powstałe w ten sposób kernele będą spełniać warunki poprawności definiowane przez teorię Merera. Jest to możliwe dzięki własności zamknięcia zbioru kerneli ze względu na określone operacje. Własność ta umożliwia konstruowanie poprawnych funkcji jądrowych poprzez łączenie funkcji już istniejących (o których wiadomo, że są poprawne) za pomocą określonych operacji. W algorytmie *Kernel GP* zbiór terminali to zbiór trzech standardowych funkcji jądrowych: wielomianowej, sigmoidalnej i gaussowskiej. Zbiór wewnętrznych węzłów drzewa stanowią operacje dodawania, mnożenia przez stałą, mnożenia oraz funkcja wykładnicza. Ponieważ funkcje jądrowe jako argumenty przyjmują wektory a zwracają wartości skalarne, operacje łączące kernele zaś przyjmują i zwracają wartości skalarne, autorzy użyli (opisanego w części 2.3.1) silnie typowanego programowania genetycznego, żeby zapewnić zgodność typów danych przekazywanych w drzewie. Jako funkcji fitness autorzy używają wyników 10-krotnej walidacji krzyżowej na zbiorze trenującym. W eksperymentach autorzy porównali trafność klasyfikacji wydzielonego zbioru testującego przez algorytm *Kernel GP* oraz *Grid Search* (opis w części 2.4.2) użyty do SVM z jądrem

gaussowskim. Dla połowy z przebadanych wzorów lepsze wyniki osiągnął *Kernel GP*, dla drugiej *Grid Search*.

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.

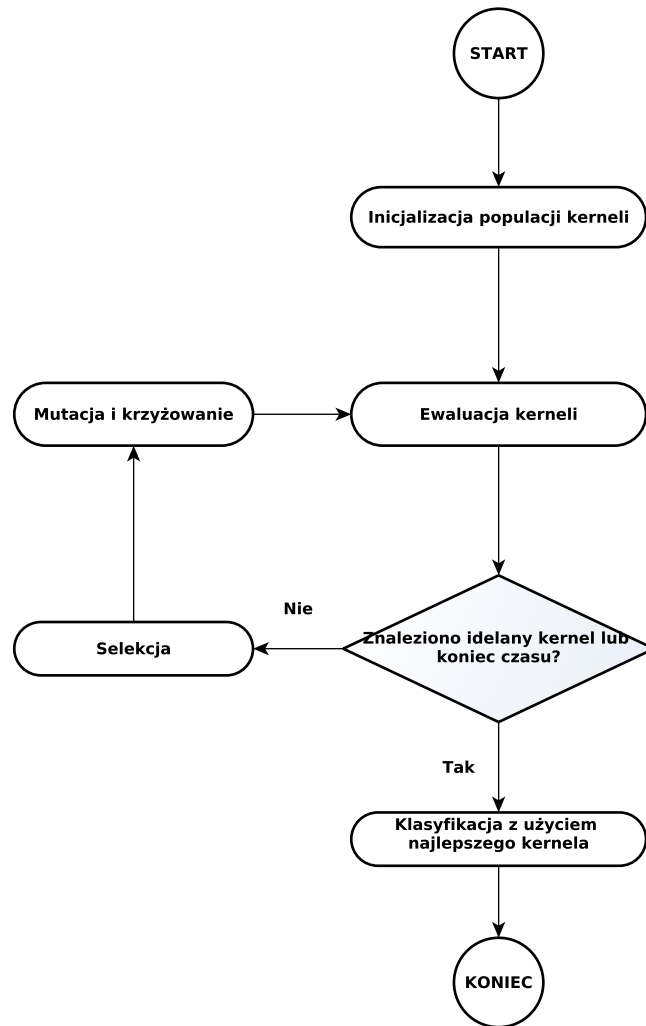
Celem algorytmu *Kernel GP+* jest odnalezienie optymalnej dla danego problemu funkcji jądrowej wraz z jej parametrami. Dzięki opisanej w poprzednim rozdziale własności domknięcia zbioru kerneli ze względu na pewne operacje arytmetyczne możliwe jest tworzenie nieograniczonej ilości dowolnie złożonych funkcji na podstawie kilku podstawowych kerneli. Opisany algorytm przeszukuje przestrzeń takich funkcji za pomocą *programowania genetycznego*. Szukana jest taka funkcja, przy której użyciu klasyfikator SVM osiągnie największą wartość wybranej miary jakości klasyfikacji (miary zostały opisane w części 2.1.2).

Algorytm *Kernel GP+* jest wzorowany na algorytmie *Kernel GP* [SL07]. Główna różnica, to poszerzony zbiór funkcji jądrowych, z których algorytm "buduje" bardziej złożone funkcje, użycie stałych *ERC* (opis: 3.1.1) oraz użycie innych miar przystosowania niż trafność klasyfikacji.

Przebieg algorytmu jest typowy dla algorytmów genetycznych:

1. Utwórz początkową populację kerneli
2. Oblicz wartość *funkcji dopasowania* każdego z kerneli: trafność klasyfikacji SVM z użyciem tego kernela
3. Jeśli znaleziono idealny kernel (wartość funkcji fitness wyniosła 1) 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.

RYSUNEK 3.1: Diagram przepływu algorytmu *Kernel GP+*.

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 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

względem na operacje wykonywane przez te funkcje funkcja powstała przez połączenie dwóch kerneli funkcją łączącą jest również poprawnym kernelem [STC04].

- Dodawanie: $k(x, y) = k_1(x, y) + k_2(x, y)$
- Mnożenie: $k(x, y) = k_1(x, y) * k_2(x, y)$
- Mnożenie przez stałą: $k(x, y) = a * k_1(x, y)$
- Funkcja wykładnicza: $k(x, y) = e^{k_1(x, y)}$

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: $k(x, y) = \langle x, y \rangle$
 - Wielomianowa: $k(x, y) = \langle x, y \rangle^d$
 - Gausowska *RBF): $e^{-\sigma * \|x-y\|^2}$
 - Sigmoidalna: $k(x, y) = \tanh(\gamma \langle x, y \rangle + \tau)$
 - Logarytmiczna: $k(x, y) = -\log(\|x - y\|^d + 1)$
 - Potęgowa: $k(x, y) = (\alpha x^T z + c)^d$
 - Cauchego: $k(x, y) = \frac{1}{1 + \frac{\|x-y\|^2}{\sigma}}$
 - Wykładnicza: $k(x, y) = \exp\left(-\frac{\|x-y\|}{2\sigma^2}\right)$

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

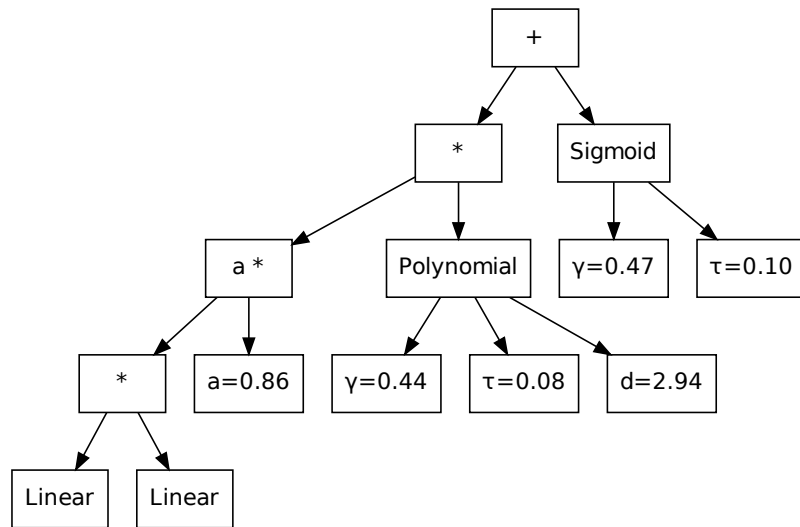
- Stałe *ERC* (ang. *Ephemeral Random Constant*): liczby rzeczywiste lub całkowite, które służą jako parametry innych funkcji. Są one liśćmi w drzewie, nie przyjmują żadnych argumentów. Mogą losowo zmieniać swoją wartość podczas mutacji, w ramach określonego przedziału.
 - γ : liczba rzeczywista z zakresu $\langle 0.1, 2.0 \rangle$
 - 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.??.

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

3.1.2 Ewaluacja kerneli

Każda wygenerowana przez algorytm GP funkcja zostaje poddana ocenie, w wyniku której zostaje jej przypisana wartość funkcji przystosowania (ang. *fitness*). W tym celu funkcja ta jest wykorzystywana przez algorytm SVM jako funkcja jądrowa a jakość wyników klasyfikacji stanowi ocenę funkcji jądrowej. Ewaluacja funkcji jądrowej może odbywać się na jeden z dwóch sposobów. Jeśli ze zbioru uczącego wydzielono zbiór walidujący, to sprawdzany kernel jest używany do klasyfikacji danych ze zbioru walidującego. Ocena jakości klasyfikacji zostaje przeliczona na wartość *funkcji przystosowania* ewaluowanej funkcji jądrowej. Jeśli ze zbioru uczącego nie wydzielono zbioru walidującego, to



RYSUNEK 3.2: Przykładowe drzewo generowane przez algorytm.

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.

Do oceny jakości wyników klasyfikacji używana jest jedna z miar opisanych w części 2.1.2. Ponieważ wszystkie te miary należą do zakresu $\langle 0, 1 \rangle$, to mogą być bezpośrednio użyte jako wartość fitness ewaluowanego kernela.

3.1.3 Selekcja i zapobieganie przerostowi

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 podczas inicjalizacji populacji czy podczas krzyżowania i mutacji. Drugi mechanizm, o angielskiej nazwie ang. *parsimony pressure*, promuje mniejsze drzewa podczas selekcji. W tym celu stosowany jest algorytm selekcji turniejowej leksykograficznej z koszykami (ang. *Bucket Lexicographic Tournament Selection*), opisany w części 2.3.1.

3.1.4 Krzyżowanie i mutacja

Krzyżowanie polega na odcięciu dwóch losowych poddrzew z dwóch różnych osobników i zamianę 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, które nie były używane podczas uczenia klasyfikatora SVM ani podczas ewaluacji kerneli. Najpierw algorytm SVM jest uczony na połączonych zbiorach trenującym i testującym, przy pomocy tej funkcji jądrowej. Następnie dokonywana jest klasyfikacja zbioru walidującego. Otrzymane w wyniku tej klasyfikacji miary jakości klasyfikacji (opisane w części 2.1.2 są miarą oceny całego algorytmu).

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 mechanizmów *obliczeń ewolucyjnych* w tym *programowania genetycznego*. *LibSVM* to klasyfikator SVM napisany oryginalnie w języku C, przepisany w języku Java.

W stworzonym programie można wydzielić następujące części:

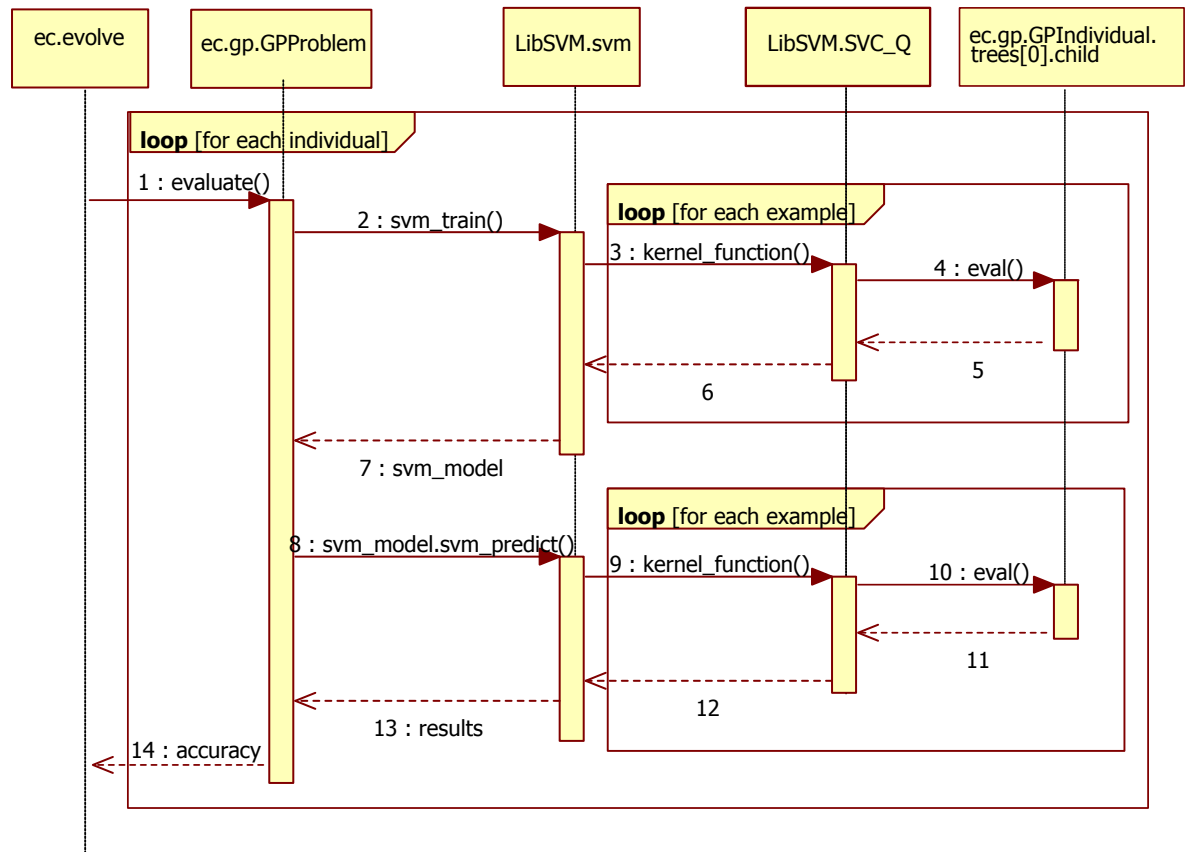
- pakiet *libsvm*, zawierający przerobiony kod biblioteki *LibSVM* wraz z kilkoma dodanymi klasami potomnymi. Ze względu na to, że implementacja *LibSVM* w języku Java jest po prostu przepisany kod oryginalnej wersji napisanej w C i większość użytych w niej metod jest statyczna i prywatna, trudno jest wprowadzać modyfikacje w bibliotece poprzez pisanie klas potomnych dziedziczących po już istniejących w bibliotece. Dlatego zdecydowano się na bezpośrednią modyfikację istniejącego kodu źródłowego biblioteki.
- pakiet *textc.app.kernel_gp*, zawierający implementacje problemu, funkcji węzłów i terminali oraz klas zbierających statystyki przebiegu algorytmu, dziedziczące z klas należących do biblioteki *ECJ*
- bibliotekę *ECJ*
- skrypty napisane w języku Python służące do przeprowadzania eksperymentów

Uruchomienie algorytmu następuje poprzez uruchomienie klasy *ec.Evolve* należącej do biblioteki *ECJ*, wraz z podaniem jako jeden z parametrów ścieżki do pliku konfiguracyjnego. Plik konfiguracyjny oraz parametry wywołania programu zawierają informacje takie jak:

- wielkość populacji
- liczba pokoleń przez które będzie wykonany algorytm
- prawdopodobieństwa mutacji, krzyżowania
- rodzaj i parametry procesu selekcji, np. wielkość turnieju
- ograniczenia na wielkość generowanych drzew
- typy danych używane w drzewie
- zbiory ograniczeń narzucone na liczbę i typ wartości przyjmowanych i zwracanych przez węzły drzewa

- zbiór funkcji i terminali, każdej funkcji i terminalowi przypisane są: wspomniane w poprzednim punkcie ograniczenia, klasa Javy, która zawiera jej implementację
- ścieżka do klasy java implementującej problem rozwiązywany przez algorytm GP

Klasa wymieniona w ostatnim z powyższych punktów jest rozszerzeniem klasy `GPPProblem` z biblioteki *ECJ*. Zawiera ona między innymi definicję funkcji *evaluate*, która jest wykonywana dla każdego wygenerowanego przez GP rozwiązania i ma za zadanie przypisać mu wartość fitness. W funkcji tej zostaje wywołana funkcja będąca rozszerzeniem funkcji *svm_train()* z biblioteki *LibSVM*. Jako jeden z parametrów jej wywołania przekazywany jest obiekt reprezentujący wygenerowane przez *ECJ* drzewo. Podczas wywołania funkcji *svm_train()*, w metodzie *kernel_function()* obliczana jest wartość funkcji jądrowej. Standardowo, w bibliotece *LibSVM* jest to jedna z trzech standardowych funkcji w zależności od parametru wywołania. Wprowadzona modyfikacja polega na wywołaniu w ciele funkcji *kernel_function()* metody *eval()* przekazanego wcześniej do *svm_train()* obiektu drzewa, która to metoda zwraca wartość obliczoną przez drzewo dla podanych wartości wejściowych. Diagram sekwencji UML przedstawiający przepływ kontroli pomiędzy wymienionymi powyżej funkcjami ukazano na rysunku 3.3.



RYСУNEK 3.3: Diagram UML sekwencji działania algorytmu Kernel GP

W celu przeprowadzenia eksperymentów obliczeniowych z użyciem różnych kombinacji parametrów został napisany skrypt w języku Python. Jego zadaniem jest iteracyjne ustawianie kolejnych wartości parametrów, wywołanie właściwego algorytmu z użyciem tych parametrów oraz zapis zwróconych przez algorytm wyników do pliku. Wymienione funkcje zostały zaimplementowane w plikach:

- `experiment.py` — główny skrypt służący do przeprowadzania eksperymentu

- `pyDataSet.pt` — skrypt służący do przetwarzania danych wejściowych w celu wydzielenia podzbiorów testowego, uczącego i walidującego, dzielenia zbioru na potrzeby walidacji krzyżowej, konwersji danych z plików `*.arff` pakietu *WEKA* na pliki kompatybilne z *LibSVM*.
- `group.py` – skrypt dzielący pliki z wynikami klasyfikacji i statystykami zwracane przez `experiment.py` według wartości w zadanych kolumnach, w celu ułatwienia ich obsługi przez skrypty `gnuplot`, służące do rysowania wykresów

Wszystkie pliki programów, dane wejściowe a także pliki z wynikami i statystykami są zawarte na dołączonej do niniejszej pracy płycie CD. Aby uruchomić eksperyment obliczeniowy należy w konsoli wydać polecenie `python experiment.py` z odpowiednimi argumentami. Opis wszystkich parametrów skryptu można otrzymać poprzez uruchomienie skryptu z przełącznikiem `-h`. Wynikiem działania skryptu są 3 rodzaje plików:

- pliki o rozszerzeniu `*.dat` — zawierają wyniki działania algorytmu: wartości wszystkich miar klasyfikacji (trafność, *f1*, *MCC*, prawdopodobieństwo), czasy wykonania, oraz wartości uśrednione dla wszystkich podziałów zbioru danych. Wyniki najlepszego do danego momentu osobnika są zapisywane co pokolenie. W każdym wierszu podano również parametry, z którymi był wywołany proces ewolucyjny. Dane w pliku mają formę tabelaryczną, kolumny są oddzielone spacjami. Jest to format kompatybilny z programem `gnuplot`.
- pliki o rozszerzeniu `*.stat` — zawierają zapis wszystkich ewaluowanych przez *ECJ* w przebiegu algorytmu funkcji jądrowych wraz z osiągniętą przez nie wartością funkcji przystosowania i czasem ich obliczenia.
- pliki o rozszerzeniu `*.ecjstat` — zawierają statystyki generowane przez program *ECJ*, zapisywane co pokolenie, zawierające takie dane jak średnia wielkość generowanych drzew, średnia wartość funkcji przystosowania, najlepszy osobnik znaleziony w danym pokoleniu a także, dodane w toku implementacji algorytmu *Kernel GP+*, wyniki klasyfikacji zbioru testującego z użyciem najlepszego osobnika znalezione w danym pokoleniu, z użyciem różnych miar jakości.

Rozdział 4

Wyniki działania algorytmu na popularnych zbiorach danych

4.1 Metodologia pomiarów

Oszacowanie trafności klasyfikacji osiąganej przez skonstruowany system było dokonywane na dwa sposoby. Pierwszy z nich (ang. *random subsampling*) polegał na podziale zbioru danych na zbiór uczący i *testujący*, a w przypadku algorytmu *Kernel-GP+* również wydzielaniu ze zbioru uczącego podzbioru *walidującego*, 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.

Drugi sposób polegał na zastosowaniu k-krotnej walidacji krzyżowej. Walidacja krzyżowa może być stosowana zarówno na etapie oceny osobników przez algorytm programowania genetycznego (wtedy niepotrzebny staje się wydzielony zbiór walidujący), jak i na etapie testowania ostatecznego rozwiązania znalezione przez ten algorytm. Metoda k-krotnej walidacji krzyżowej dawała lepsze wyniki, lecz poprzez konieczność wykonania walidacji rozwiązania k razy wydłużała czas wykonania algorytmu. Jeśli nie zaznaczono inaczej, przedstawione poniżej wyniki działania *Kernel GP+* dotyczą pierwszej metody testowania i walidacji rozwiązań.

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

Aby ocenić skuteczność algorytmu genetycznego w poszukiwaniu optymalnych funkcji jądrowych oraz oszacować optymalną wielkość populacji, liczbę ewaluowanych pokoleń oraz najlepszą funkcję przystosowania przeprowadzono szereg eksperymentów obliczeniowych, w których uruchamiano algorytm dla coraz to większych wartości tych parametrów. Dla każdego przebiegu algorytmu obliczano i zapisywano kilka miar trafności klasyfikacji zbioru *walidującego* (miary te zostały opisane w części 2.1.2).

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 pokoleń, przez które poszukiwano rozwiązania. W szczególności

TABLICA 4.1: Parametry procesu ewolucyjnego

	mimumum	maksimum
Wielkość populacji	1	100
Liczba pokoleń	1	50
Głębokość generowanych drzew	3	8
Prawdopodobieństwo mutacji	-	0.55
Prawdopodobieństwo krzyżowania	-	0.4
Prawdopodobieństwo klonowania	-	0.05
Prawdopodobieństwo mutacji stałych	-	0.9
Wielkość turnieju	•	7
Ilość koszyków	•	10

ciekawym przypadkiem jest ten, gdy liczba populacji 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 osiąganą w trakcie jednego pokolenia i coraz większej ich liczby można ocenić czy proces ewolucyjny przebiega poprawnie.

4.1.1 Parametry procesu ewolucyjnego

Tabela 4.1 przedstawia użyte w eksperymentach wartości parametrów procesu ewolucyjnego.

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 [BL13] [D]. Zbiory zostały opisane w tabelce 4.2. Użyte nazwy zbiorów są zgodne z tymi ze strony *libsvm* [C].

TABLICA 4.2: Zbiory danych użyte do testowania systemu. W przypadku eksperymentów przeprowadzanych z zastosowaniem walidacji krzyżowej zbiór walidujący nie był wyodrębniany.

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	409	274	33.00%
Heart (statlog)	2	13	270	180	90	33.00%
Ionosphere	2	34	351	234	117	33.00%
Liver disorders	2	6	345	230	115	33.00%
Pima (diabets)	2	8	768	512	256	33.00%

TABLICA 4.3: Zbiory danych użyte do testowania systemu.

Nazwa zbioru	Liczba atrybutów ciągłych	Liczba atrybutów nominalnych	Liczba klas	Proporcje klas
DNA	0	180	3	464/485/1051
Vowel	10	0	11	Każda klasa 48 razy
Breast	10	0	2	239/444
Heart	7	6	2	150/120
Ionosphere	34	0	2	225/126
Liver	6	0	2	200/145
Pima	8	0	2	500/268

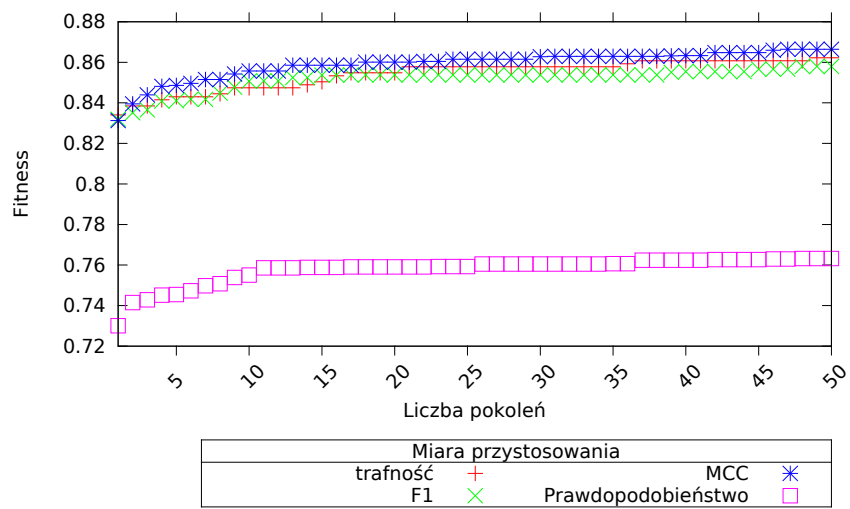
4.3 Fitness

Aby ocenić dobór parametrów procesu ewolucyjnego zapisywano wartości fitness osiągane przez najlepszego osobnika w każdym pokoleniu. Wartości te zostały przedstawione na wykresach przedstawiających wartość fitness najlepszego osobnika w funkcji czasu trwania algorytmu (ilości dotychczas wygenerowanych populacji), czyli tak zwanych ang. *fitness graphs*. Na wykresach 4.1 – 4.5 widać jak zmienia się wartość funkcji fitness wraz z kolejnymi pokoleniami dla różnych miar jakości klasyfikacji użytych jako funkcja fitness (miary te zostały opisane w części 2.1.2 a ich użycie w części 3.1.2).

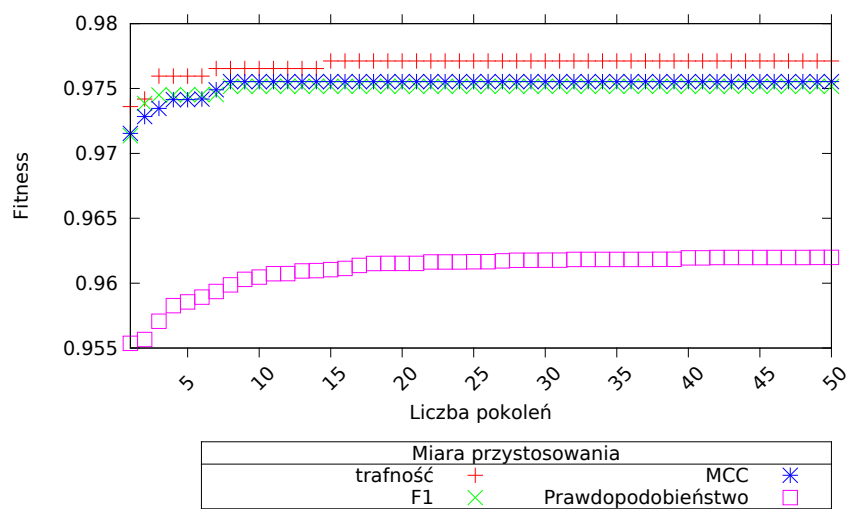
Dla wszystkich przetestowanych zbiorów widać, że fitness najlepszego osobnika w kolejnych pokoleniach rośnie, a przynajmniej nie maleje. To, że przystosowanie nigdy nie maleje jest wynikiem zastosowania mechanizmu *elit* (opisanego w części 2.3). Niestety, w przypadku zbiorów *breast* (rys. 4.2 i *liver disorders* (rys. 4.4 fitness dość szybko (ok. 15-tego pokolenia) przestaje rosnać. Jest to moment, w którym można by zakończyć algorytm ewolucyjny, choć a-priori ciężko określić kiedy należy to zrobić - na przykład w przypadku zbioru *ionosphere* (rys. 4.3 po chwilowym wypłaszczeniu funkcji przystosowania (pokolenia 10-20) zaczyna ona znowu rosnać.

Wybór miary, która służy za funkcję przystosowania zależy przede wszystkim od dziedziny klasyfikacji, kosztów pomyłek. W przypadku większości zbiorów fitness rośnie podobnie niezależnie od wyboru miary użytej do jego obliczenia. Wyjątkiem jest zbiór *liver disorders*, w przypadku którego średnie prawdopodobieństwo przypisane właściwej klasie rośnie cały czas, w przeciwieństwie do trafności klasyfikacji, która w okolicach 15 pokolenia ulega stagnacji.

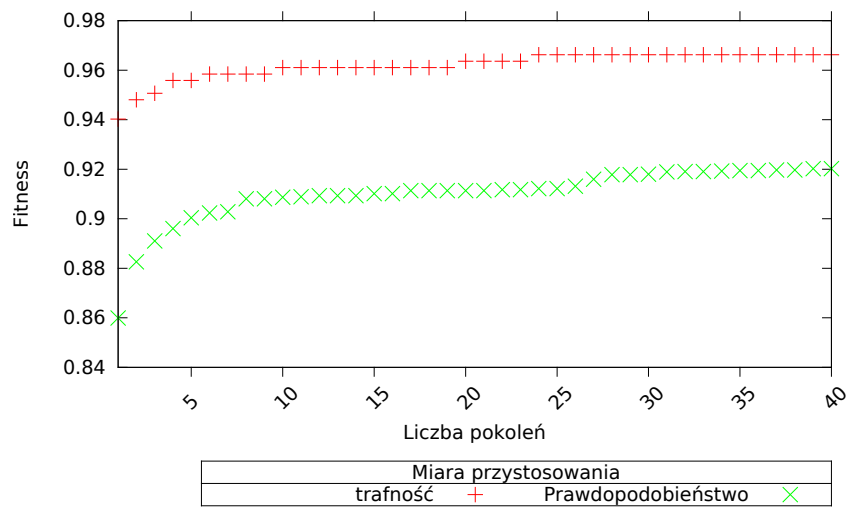
Wykresy 4.6 – 4.8 podobnie jak poprzednie pokazują wzrost przystosowania w funkcji liczby pokoleń, jednak w ich przypadku są to wyniki pochodzące z 10-krotnej walidacji krzyżowej. W celu analizy zjawiska przeuczenia pokazano na nich zarówno trafność klasyfikacji zbioru trenującego (walidacja) jak i testowego (testowanie). W przypadku zbioru *heart* przeuczenie nie występuje - mimo spadku w okolicach 10–15 pokolenia trafność na zbiorze testującym rośnie wraz z liczbą pokoleń. W przypadku dwóch pozostałych zbiorów widać, że algorytm genetyczny dostosowuje funkcję jądrową do zbioru trenującego, co nie ma wpływu na trafność osiąganą na zbiorze testowym - ta zachowuje się w sposób losowy.



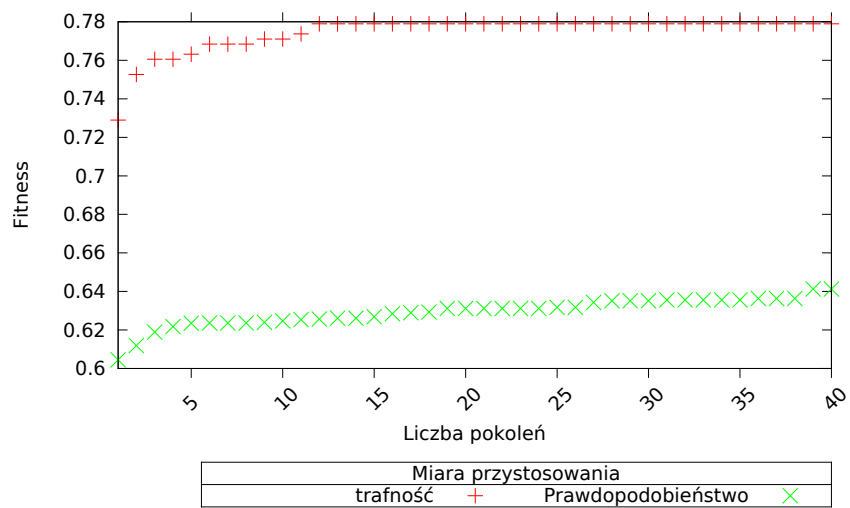
RYSUNEK 4.1: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *heart*.



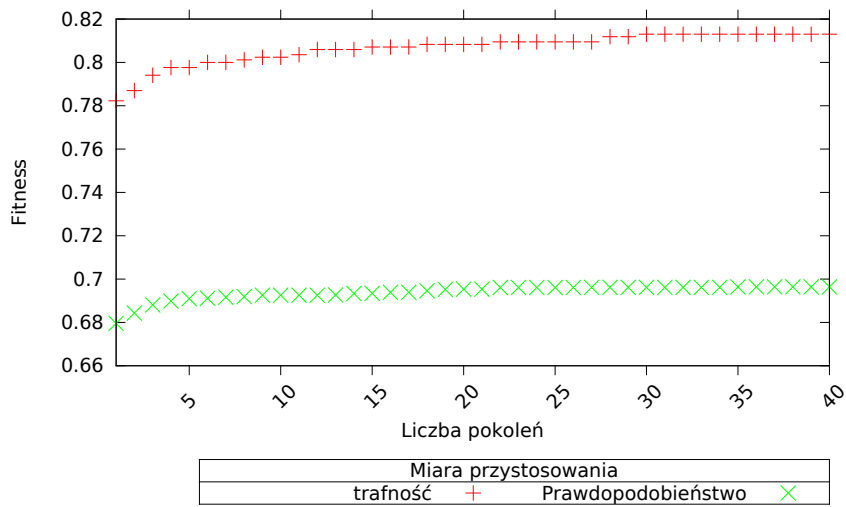
RYSUNEK 4.2: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *breast*.



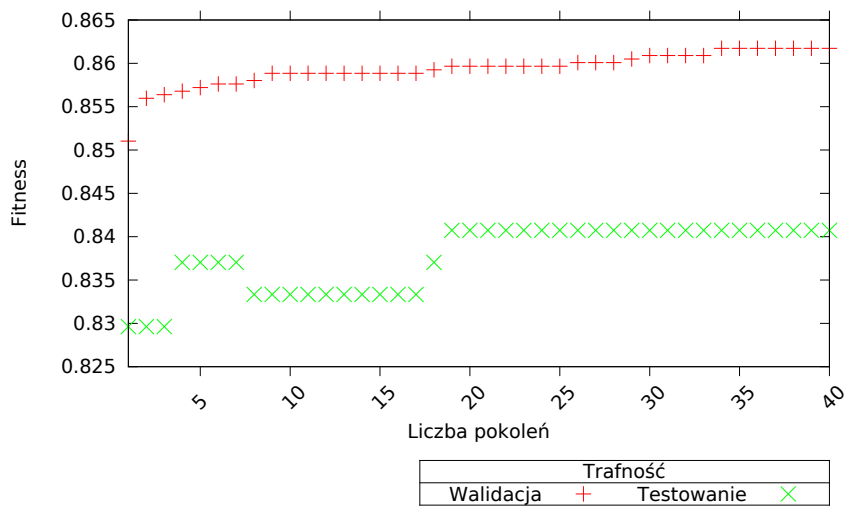
RYSUNEK 4.3: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *ionosphere*.



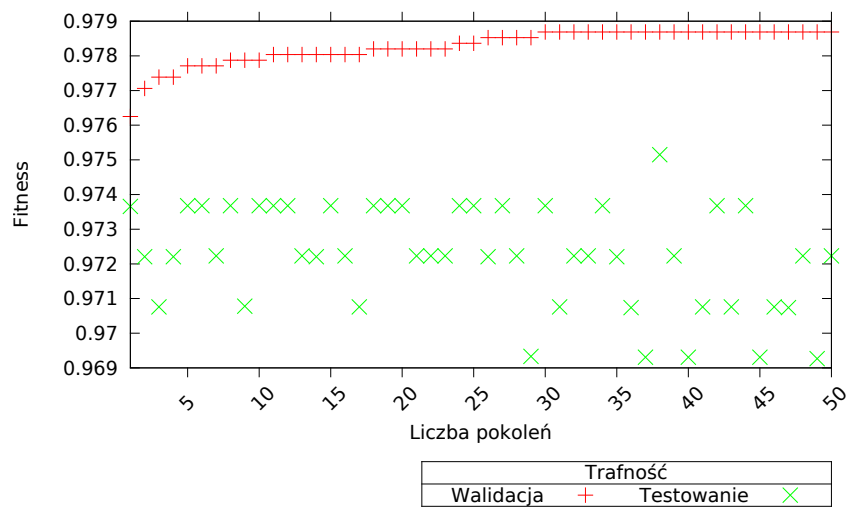
RYSUNEK 4.4: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *liver disorders*.



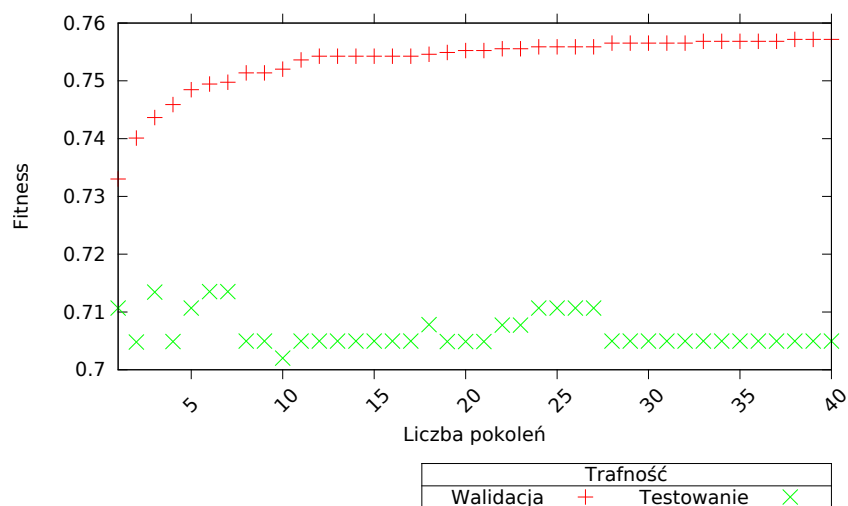
RYСУNEK 4.5: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *pima-diabetes*.



RYСУNEK 4.6: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *heart*, obliczana za pomocą 10-krotnej walidacji krzyżowej, oraz trafność osiągnięta przez najlepsze znalezione rozwiązanie, również obliczana za pomocą 10-krotnej walidacji krzyżowej.



RYSUNEK 4.7: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *breast*, obliczana za pomocą 10-krotnej walidacji krzyżowej, oraz trafność osiągnięta przez najlepsze znalezione rozwiązanie, również obliczana za pomocą 10-krotnej walidacji krzyżowej.



RYSUNEK 4.8: Najlepsza wartość funkcji przystosowania dla kolejnych pokoleń dla zbioru *liver disorders*, obliczana za pomocą 10-krotnej walidacji krzyżowej, oraz trafność osiągnięta przez najlepsze znalezione rozwiązanie, również obliczana za pomocą 10-krotnej walidacji krzyżowej.

4.4 Wyniki klasyfikacji zbioru testującego

Wyniki klasyfikacji zostały ocenione za pomocą miar opisanych w części 2.1.2. Dla każdej z tych miar przedstawiono jej wartości dla przebiegów algorytmu, w których jako funkcja fitness była wybrana właśnie ta miara. Dodatkowo porównano otrzymane wartości z wynikami uzyskanymi w wyniku klasyfikacji za pomocą trzech standardowych funkcji jądrowych (*Wielomianowej*, *Sigmoidalnej* i *RBF*).

Trafność klasyfikacji osiąganą przez algorytmy pokazano w tabelach 4.4 i 4.5. Przedstawiono w niej dodatkowo wyniki algorytmów opisanych w części 2.4.3, pochodzące z prac, w których te algorytmy zostały zaprezentowane.

4.4.1 Trafność

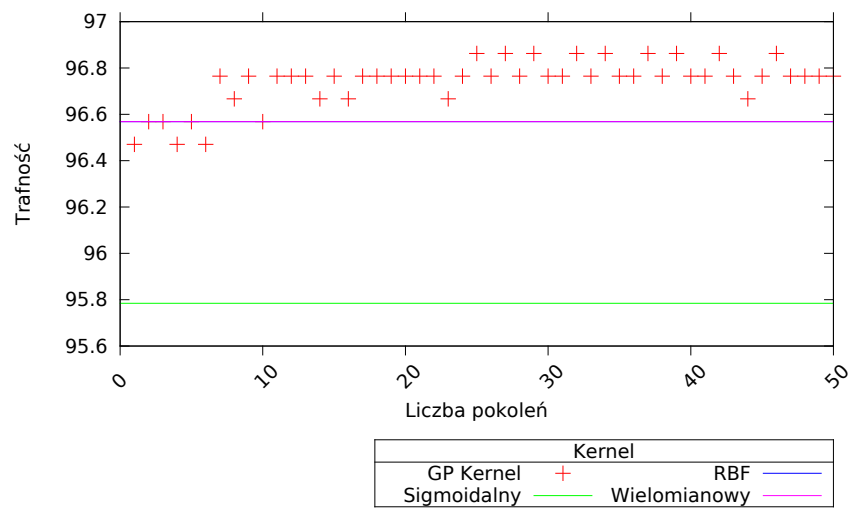
Jak widać na rysunkach 4.9-4.13 osiągnięta jakość klasyfikacji jest mocno zależna od konkretnego zbioru danych. Dla zbiorów *ionosphere* i *liver disorders* i *pima diabetes* algorytm *Kernel GP+* osiąga lepsze wyniki niż SVM ze standardowymi funkcjami jądrowymi. dla zbiorów *breast*, *heart* i *ionosphere* widoczna jest poprawa wyników wraz z kolejnymi pokoleniami algorytmu ewolucyjnego. W przypadku zbioru *liver disorders* już około piątego pokolenia algorytm osiąga *plateau* i dalej trafność oscyluje wokół poziomu 70%. Może to oznaczać, że rozmiar populacji jest na tyle duży, że już w początkowej populacji znajduje się rozwiązanie zbliżone do optymalnego.

TABLICA 4.4: Zestawienie wyników klasyfikacji. Tabela przedstawia trafność klasyfikacji osiąganą przez LibSVM dla różnych funkcji jądrowych przy domyślnych ustawieniach oraz wyniki uzyskane za pomocą przeszukiwania Grid Search dostępnego wraz z biblioteką LibSVM.

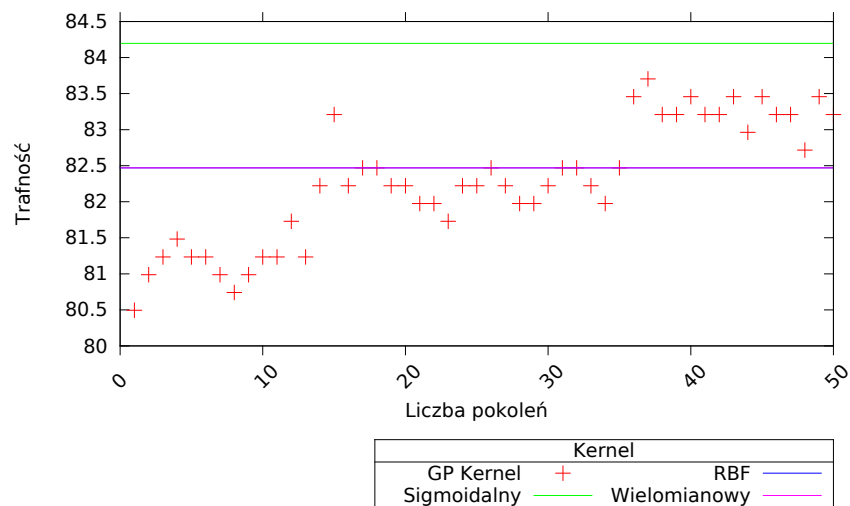
	Algorytm			
Zbiór	LibSVM Polly	LibSVM RBF	LibSVM Sigmoid	Grid LibSVM
Breast	0,970	0,966	0,961	0,972
Heart	0,827	0,822	0,829	0,844
Ionosphere	0,875	0,935	0,870	0,952
Liver	0,612	0,597	0,603	0,739
Pima	0,746	0,755	0,732	0,777

TABLICA 4.5: Zestawienie wyników klasyfikacji. Tabela przedstawia trafność klasyfikacji osiąganą przez poszczególne algorytmy dla przetestowanych zbiorów danych. Wyniki algorytmów K-Tree, Kernel-GP i EKM pochodzą z prac omówionych w rozdziale 2.4.3.

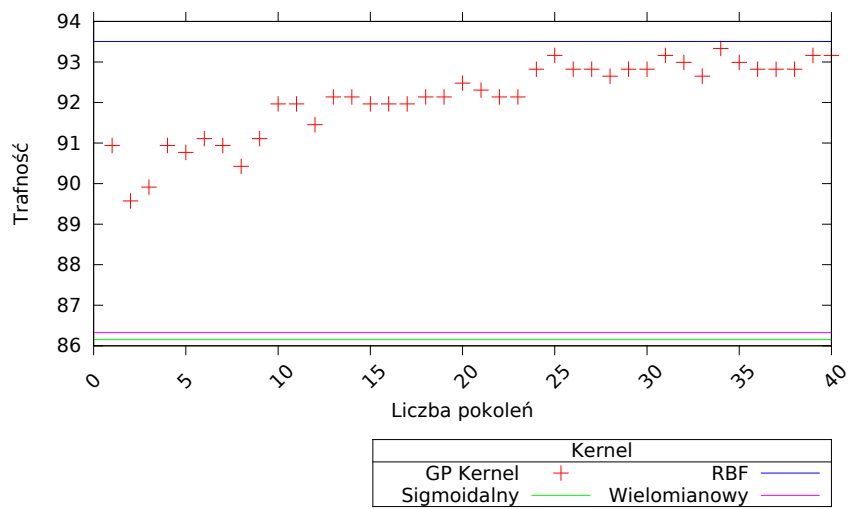
	Algorytm					
Zbiór danych	Grid LibSVM	K-Tree	Kernel-GP	EKM	Kernel-GP+	Kernel-GP+ 10-fold CV
Breast	0,972	•	0,955	0,970	0,957	0,972
Heart	0,844	0,822	0,776	•	0,829	0,841
Ionosphere	0,952	0,943	0,940	0,905	0,932	•
Liver	0,739	0,723	•	0,691	0,705	0,705
Pima	0,777	0,775	•	0,748	0,773	•



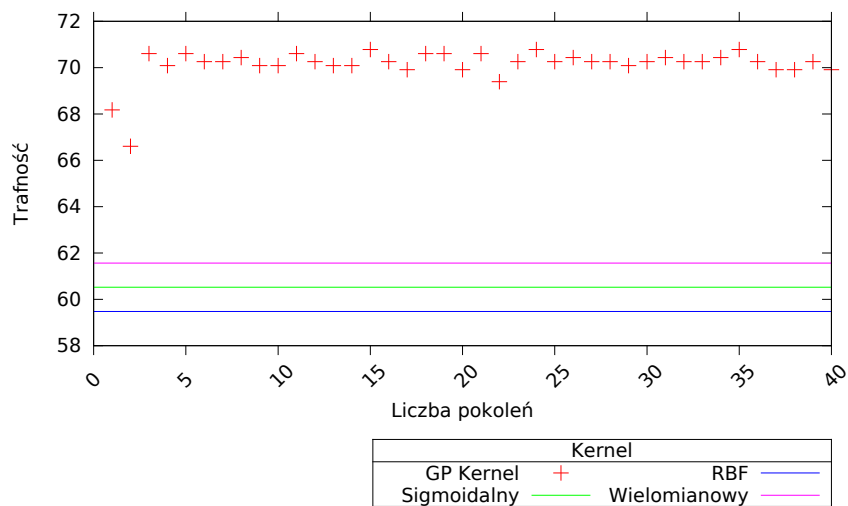
RYСУNEK 4.9: Trafność (ang. *accuracy*) klasyfikacji dla zbioru *breast* w funkcji czasu wykonania (ilości pokoleń).



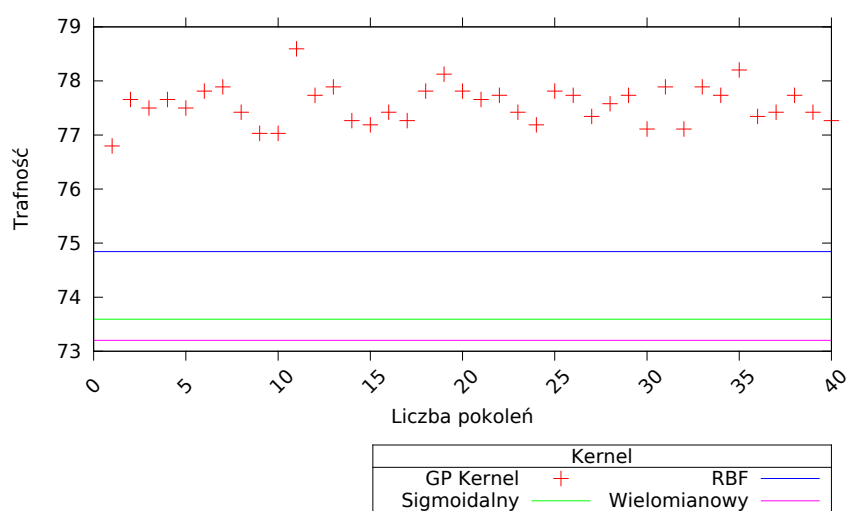
RYСУNEK 4.10: Trafność (ang. *accuracy*) klasyfikacji dla zbioru *heart* w funkcji czasu wykonania (ilości pokoleń).



RYСУNEK 4.11: Trafność (ang. *accuracy*) klasyfikacji dla zbioru *ionosphere* w funkcji czasu wykonania (ilości pokoleń).



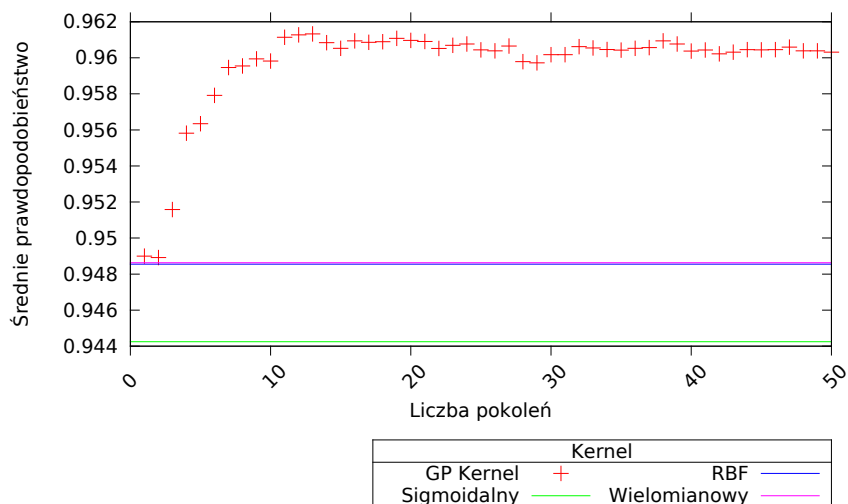
RYСУNEK 4.12: Trafność (ang. *accuracy*) klasyfikacji dla zbioru *live disorders* w funkcji czasu wykonania (ilości pokoleń).



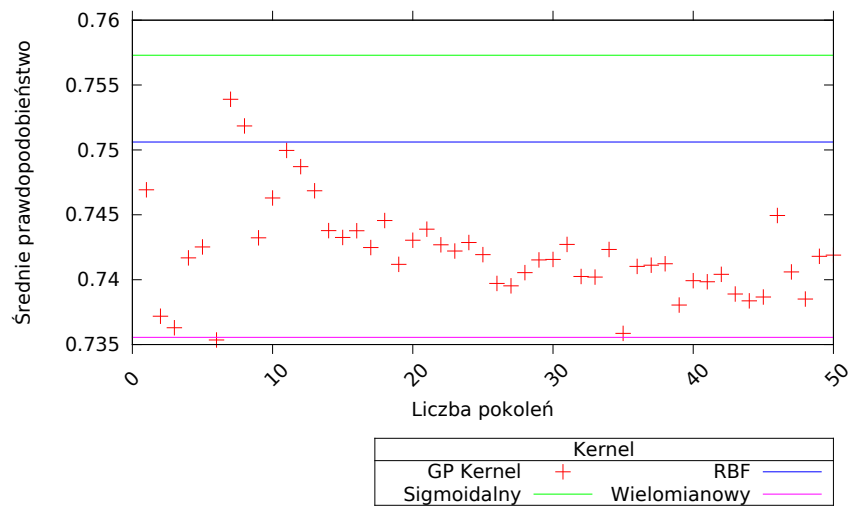
RYSUNEK 4.13: Trafność (ang. *accuracy*) klasyfikacji dla zbioru *pima diabetes* w funkcji czasu wykonania (ilości pokoleń).

4.4.2 Prawdopodobieństwo

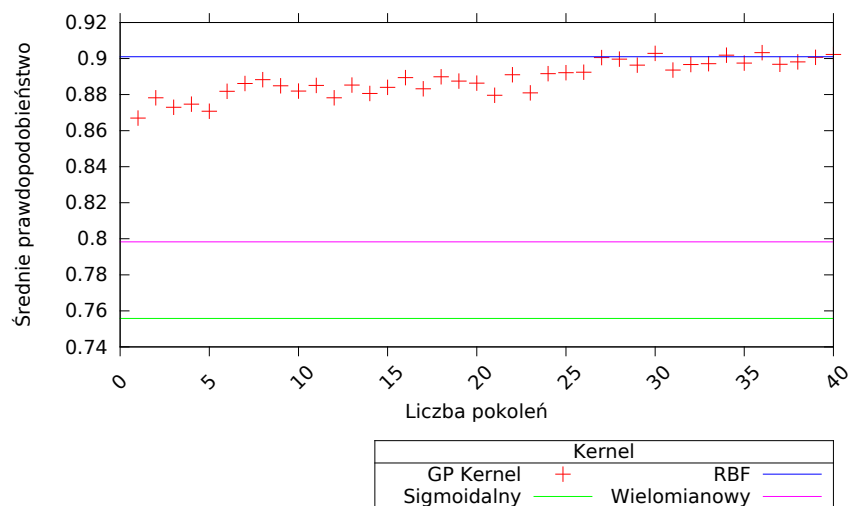
Wykresy 4.14 – 4.18 ukazują zmieniającą się wraz z kolejnymi pokoleniami wartość miary średniego prawdopodobieństwa przypisywanego przez klasyfikator klasie, do której rzeczywiście należy dany przypadek, opisanej w części 2.1.2. Dla zbiorów *breast* i *liver* obserwujemy wzrost wartości tej miary i osiąganie wyników lepszych niż standardowy SVM. W przypadku zbioru *heart* występuje zjawisko przeuczenia, które uniemożliwia algorytmowi osiągnięcie wyników na poziomie tych osiągniętych przez SVM z kernelem sigmoidalnym.



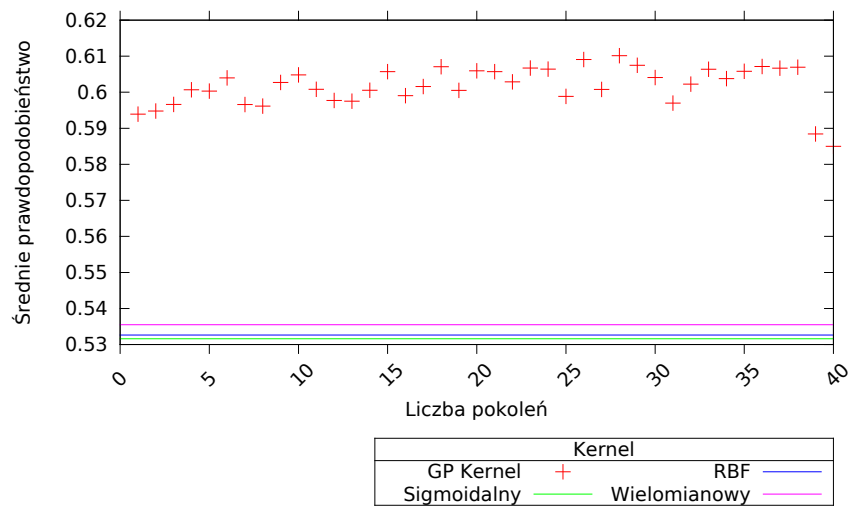
RYSunEK 4.14: Średnia wartość prawdopodobieństwa przypisywanego przez SVM właściwej dla klasyfikowanego przykładu klasie w funkcji czasu wykonania (ilości pokoleń). Zbiór *breast*



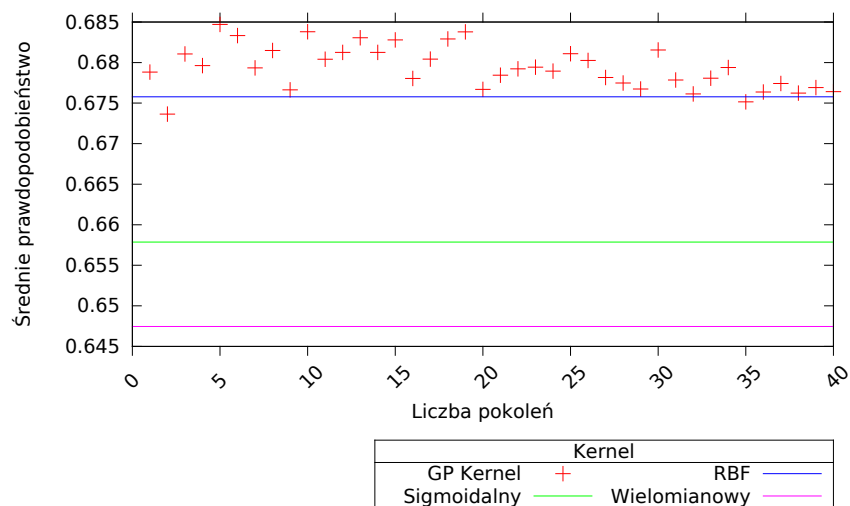
RYСУNEK 4.15: Średnia wartość prawdopodobieństwa przypisywanego przez SVM właściwej dla klasyfikowanego przykładu klasie w funkcji czasu wykonania (ilości pokoleń). Zbiór *heart*.



RYСУNEK 4.16: Średnia wartość prawdopodobieństwa przypisywanego przez SVM właściwej dla klasyfikowanego przykładu klasie w funkcji czasu wykonania (ilości pokoleń). Zbiór *ionosphere*.



RYSUNEK 4.17: Średnia wartość prawdopodobieństwa przypisywanego przez SVM właściwej dla klasyfikowanego przykładu w funkcji czasu wykonania (ilości pokoleń). Zbiór *liver-disorders*.



RYSUNEK 4.18: Średnia wartość prawdopodobieństwa przypisywanego przez SVM właściwej dla klasyfikowanego przykładu w funkcji czasu wykonania (ilości pokoleń). Zbiór *pima-diabetes*.

4.5 Czas wykonania

Główną wadą proponowanego podejścia są spore czasy wykonania algorytmu. Algorytm programowania genetycznego uruchamia klasyfikator SVM, najpierw w trybie uczenia, potem klasyfikacji, za każdym razem kiedy dokonuje ewaluacji (oceny) osobnika, czyli $G \times P$ razy, gdzie P to wielkość populacji a G liczba pokoleń przez które działa algorytm. Dla parametrów używanych najczęściej podczas przeprowadzonych eksperymentów, czyli $P = 100$ i $G = 50$ daje to 5000 wywołań *LibSVM*. Do tego należy jeszcze doliczyć narzut związany z operacjami selekcji, mutacji i krzyżowania dokonywanymi przez *ECJ*. Dodatkowo można się spodziewać, że obliczenie wartości zwracanej przez funkcję jądrową będzie, dla tej samej funkcji, bardziej czasochłonne kiedy jest ona obliczana przez *ECJ* jako wartość drzewa, niż w przypadku, kiedy jest ona liczona jako wartość wyrażenia Javy zapisanego w *LibSVM* - ze względu na narzut związany z przechodzeniem przez gałęzie drzewa. Co więcej funkcje jądrowe generowane przez GP z założenia są kombinacją kilku standardowych funkcji jądrowych, a więc wymagają większej liczby obliczeń.

W celu przeanalizowania źródła czasochłonności obliczeń Kernel-GP podczas przeprowadzania eksperymentów zapisywano czas ich wykonania. Wyniki przedstawiają tabele 4.6 oraz 4.7 a także wykresy 4.20 i 4.19.

Patrząc na wykresy 4.20 i 4.19 widać, że narzut związany z mechanizmami algorytmu GP nie jest znaczący. Problem stanowi spora zmienność czasów wykonania algorytmu Kernel-GP - odchylenie standardowe czasu wykonania czasami jest większe niż średnia. Jest to przypuszczalnie spowodowane użyciem funkcji jądrowych, które nie są dodatnio określone (na przykład funkcja sigmoidalna) i dla których rozwiązywany przez SVM problem optymalizacji może nie być wypukły. Czas wykonania nie rośnie liniowo wraz z liczbą dokonywanych ocen generowanych funkcji - dla 10 pokoleń wielkości 10 osobników, a więc 100 ewaluacji, czas wykonania jest co najwyżej 10 razy większy niż przy jednym osobniku i jednym pokoleniu, czyli 1 ewaluacji. Jest to spowodowane pewnym stałym kosztem obliczeniowym związanym z inicjalizacją procesu ewolucyjnego, która wykonuje się tylko raz oraz z tym, że osobniki identyczne do już ocenianych nie są oceniane po raz drugi. Im proces ewolucyjny trwa dłużej, tym bardziej homogeniczna staje się populacja a co za tym idzie więcej występuje w niej identycznych osobników.

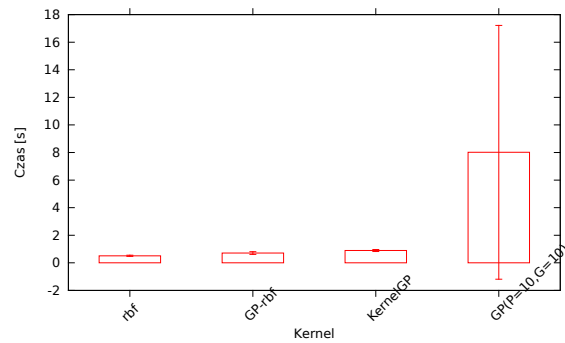
Porównanie czasów przeszukiwania metodą *Grid search*, dostępną jako skrypt dostarczany z bibliotekę *LibSVM*, z czasami wykonania algorytmu *Kernel GP+* jest zasadne, ponieważ oba algorytmy osiągają zbliżoną jakość klasyfikacji (patrz tabela 4.5). Niestety, pod względem szybkości działania *Kernel-GP* jest nieporównywalnie gorszy od *Grid Search*. Osiągnięcie wyników ukazanych w tabeli 4.5 zajmuje temu pierwszemu od stu, do nawet ponad tysiąca sekund (z użyciem pięcio-krotnej walidacji krzyżowej), podczas gdy *Grid search* znajduje swoje rozwiązanie po kilku-kilkunastu sekundach. Przyczyną tego stanu rzeczy jest fakt, że przestrzeń, którą przeszukuje szybszy algorytm jest znacznie mniejsza: manipulacji ulegają jedynie dwa parametry: C i γ .

TABLICA 4.6: Czasy uczenia na zbiorze uczącym i klasyfikacji zbioru testującego przez klasyfikator *LibSVM*.

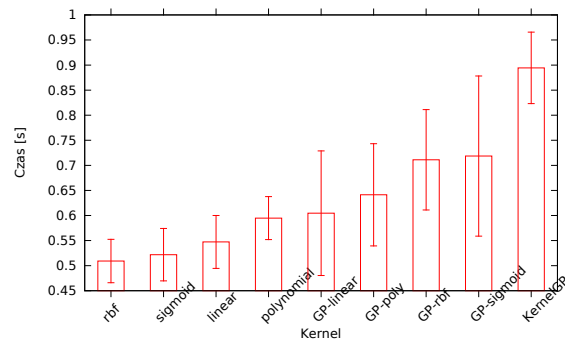
dataset	LibSVM kernels			
	LibSVM linear	LibSVM polynomial	LibSVM rbf	LibSVM Sigmoid
breast	0.55 ± 0.05	0.59 ± 0.04	0.51 ± 0.04	0.52 ± 0.05
heart	0.53 ± 0.03	0.55 ± 0.05	0.48 ± 0.03	0.49 ± 0.05
ionosphere	0.78 ± 0.02	0.82 ± 0.03	0.76 ± 0.06	0.84 ± 0.07
liver disorders	0.54 ± 0.07	0.54 ± 0.04	0.54 ± 0.04	0.54 ± 0.03
pima diabetes	0.81 ± 0.08	0.77 ± 0.03	0.81 ± 0.07	0.81 ± 0.04

TABLICA 4.7: Czasy wykonania algorytmu kernel-GP. Kolumny GP Linear/Poly/Sigmoid/RBF zawierają czas wykonania algorytmu, w którym podczas oceny drzewa wygenerowanego przez GP tak naprawdę wywołano jedną z funkcji jądrowych zdefiniowanych w LibSVM (wartość drzewa nie była więc obliczona). Dwie ostatnie kolumny zawierają czas wykonania Kernel-GP dla różnych wartości parametrów wielkości populacji i liczby pokoleń.

	GP-generated Kernels					
dataset	GP Linear	GP Poly	GP RBF	GP Sigmoid	GP(P=1, G=1)	GP(P=10, G=10)
breast	0.60 ± 0.12	0.64 ± 0.10	0.71 ± 0.10	0.72 ± 0.16	0.89 ± 0.07	8.02 ± 9.2
heart	0.54 ± 0.12	0.67 ± 0.15	0.62 ± 0.12	0.64 ± 0.07	4.05 ± 3.64	18.14 ± 20.49
ionosphere	0.71 ± 0.12	0.73 ± 0.18	0.81 ± 0.16	0.76 ± 0.13	1.41 ± 0.7	5.88 ± 2.7
Liver-disorders	0.63 ± 0.1	0.67 ± 0.13	0.68 ± 0.12	0.67 ± 0.12	0.75 ± 0.16	5.66 ± 3.87
Pima-diabetes	0.85 ± 0.12	0.93 ± 0.12	0.90 ± 0.09	1.0 ± 0.13	6.26 ± 11.48	6.30 ± 1.16



RYSUNEK 4.19: Czasy uczenia i klasyfikacji zbioru breast dla różnych funkcji jądrowych i algorytmów.



RYSUNEK 4.20: Czasy uczenia i klasyfikacji zbioru breast dla różnych funkcji jądrowych i algorytmów.

4.6 Wnioski

Przedstawione w rozdziale wyniki pokazują, że zastosowany algorytm programowania genetycznego działa - można zaobserwować wzrost funkcji fitness wraz kolejnymi pokoleniami. W większości przebadanych przypadków algorytm *Kernel GP+* okazał się skuteczny: pozwalał na znalezienie takiej funkcji jądrowej i jej parametrów, które dawały lepsze wyniki klasyfikacji niż klasyfikator *SVM* ze standardowymi funkcjami jądrowymi i parametrami. Z drugiej strony, osiągnęte przez ten algorytm wyniki klasyfikacji są nie lepsze niż te otrzymane z pomocą przeszukiwania *grid search*, przy czym to drugie jest znacznie mniej czasochłonne. Analiza funkcji jądrowych, które w procesie ewolucyjnym osiągały najwyższe wartości fitness wykazuje, że zazwyczaj były to standardowe funkcje jądrowe. Stąd nasuwa się wniosek, że standardowe funkcje jądrowe używane najczęściej w metodach jądrowych są wystarczające i należy skupić się na optymalizacji ich parametrów, oraz parametrów samego klasyfikatora *SVM* takich jak parametr *C*. Warto dodać, że parametr *C* nie podlegał manipulacji w algorytmie *Kernel GP+*.

Choć różnice metodologiczne w przeprowadzaniu eksperymentów nie pozwalają na istotne statystycznie porównanie, to można zaryzykować stwierdzenie, że w porównaniu z podobnymi podejściami generującymi funkcje jądrowe za pomocą programowania genetycznego algorytm *Kernel GP+* wykazuje się lepszą lub porównywalną trafnością klasyfikacji. Szczególnie dobrze wypada porównanie go z oryginalnym algorytmem *Kernel GP*.

Rozdział 5

Podsumowanie

W ramach niniejszej pracy stworzono system, który za pomocą programowania genetycznego generuje funkcje jądrowe na potrzeby klasyfikatora SVM. System spełnia swe zadanie, czyli znajduje funkcję jądrową, która poprzez użycie jej przez algorytm SVM w tzw. *kernel trick*, przyczynia się do dokonania możliwie dobrej, ze względu na pewną miarę jakości, klasyfikacji zbioru danych. Algorytm osiąga wyniki porównywalne lub lepsze niż podobne podejścia używające programowania genetycznego.

W trakcie tworzenia pracy przetestowano kilka pomysłów, które nie zostały wykorzystane w ostatecznej wersji algorytmu. Jednym z nich było użycie węzła „selekcji”, który dla każdej standardowej funkcji jądrowej użytej w drzewie wybiera podzbiór atrybutów, na którym ona operuje. Operator ten został zaprojektowany jako wektor binarny o długości L , gdzie L to ilość atrybutów opisujących przypadki w zbiorze danych. Jeśli bit odpowiadający danemu atrybutowi przyjmował wartość 0, to dana funkcja nie korzystała z tego atrybutu. Analiza wyników otrzymanych z użyciem operatora selekcji wykazała, że okazał się on negatywnie wpływać na wyniki klasyfikacji i w toku ewolucji liczebność osobników zawierających ten operator malała z pokolenia na pokolenie.

5.1 Napotkane problemy

Najpoważniejszym problemem napotkanym w trakcie prac nad algorytmem są bardzo długie czasy wykonania, które zmniejszają zakres praktycznych zastosowań algorytmu a także znacząco utrudniają prace nad algorytmem oraz jego testowanie. Próbą rozwiązania tego problemu może być zrównoleglenie procesu ewolucyjnego. Mechanizmy równoległej oceny osobników są zawarte w bibliotece *ECJ*, ale ich wykorzystanie ma sens tylko w przypadku wykonywania na komputerze posiadającym kilka rdzeni. Przeprowadzone eksperymenty były wykonywane z użyciem pojedynczych wątków w celu zapewnienia powtarzalności wyników, której nie dało się osiągnąć przy użyciu kilku wątków. W zastosowaniach praktycznych, gdzie nie zachodzi potrzeba porównywania parametrów procesu ewolucyjnego i ich wpływu na wyniki klasyfikacji (ponieważ te parametry są już wcześniej znane, a celem jest jedynie znalezienie optymalnej funkcji jądrowej), użycie kilku równoległych wątków do oceny osobników nie wiąże się z żadnymi problemami i może przyspieszyć działanie algorytmu. Ponadto, w zastosowaniach praktycznych stworzonego systemu długi czas poszukiwania przez algorytm programowania genetycznego funkcji jądrowej może nie stanowić przeszkody - zazwyczaj bardziej istotny jest czas klasyfikacji nowych obiektów, a ten, gdy funkcja jądrowa jest już znaleziona, nie różni się od czasu klasyfikacji przez bibliotekę *LibSVM*.

Bardzo dobre wyniki osiąga prosty i szybki algorytm *grid search* dostarczony z biblioteką *LibSVM*, który jako funkcji jądrowej używa funkcji RBF i manipuluje jedynie dwoma parametrami: stałą γ oraz

parametrem C klasyfikatora SVM. Stworzony algorytm *Kernel GP+* nie manipuluje parametrem C , jego wartość została dobrana eksperymentalnie i użyta dla wszystkich zbiorów i generowanych funkcji. Manipulacja przez proces ewolucyjny tym parametrem mogłaby więc przynieść pozytywne efekty. Stwarza to jednak problemy natury implementacyjnej - parametr C nie jest częścią funkcji jądrowej i w momencie, kiedy funkcja jądrowa jest ewaluowana powinien już być on znany.

Porównanie wyników osiąganych przez *Kernel GP+* i *grid search* oraz analiza funkcji reprezentowanych przez osobniki „wygrywające” ewolucję kerneli sugeruje, że funkcje jądrowe konstruowane ze zbioru kilku standardowych funkcji jądrowych z użyciem operacji, ze względu na które zbiór kerneli jest zamknięty, nie pozwalają dokonać odmiennych jakościowo mapowań danych do wysoko wymiarowych przestrzeni, w których dane te byłyby łatwiej separowalne, niż przy użyciu pojedynczych standardowych funkcji.

5.2 Kierunek dalszych prac

Dalsze prace nad udoskonaleniem algorytmu mogą obejmować wspomniane powyżej próby zrównoleglenia algorytmu oraz manipulacji parametrem C . Niektóre prace, przytoczone w 2.4.3, sugerują możliwość polepszenia wyników poprzez użycie funkcji przystosowania opartej na bardziej „wewnętrznych” miarach oceny algorytmu, takich jak opisane w części 2.4.1. Warte rozpatrzenia jest również poszerzenie przeszukiwanej przestrzeni funkcji jądrowych, tak, żeby wykroczyć poza te będące jedynie złożeniem kilku funkcji „standardowych”.

Literatura

- [BGV92] Bernhard E. Boser, Isabelle M. Guyon, Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. *Proceedings of the fifth annual workshop on Computational learning theory, COLT '92*, strony 144–152, New York, NY, USA, 1992. ACM.
- [BHB02] Claus Bahlmann, Bernard Haasdonk, Hans Burkhardt. On-line handwriting recognition with support vector machines a kernel approach. *Proceedings of the Eighth International Workshop on Frontiers in Handwriting Recognition (IWFHR'02)*, IWFHR '02, strony 49–, Washington, DC, USA, 2002. IEEE Computer Society.
- [BL13] K. Bache, M. Lichman. UCI machine learning repository, 2013.
- [CC98] Andy Clark, David Chalmers. The extended mind. *Analysis*, 58(1):7–19, 1998.
- [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.
- [GBNT04] Ran Gilad-Bachrach, Amir Navot, Naftali Tishby. Margin based feature selection - theory and algorithms. *Proceedings of the twenty-first international conference on Machine learning, ICML '04*, strony 43–, New York, NY, USA, 2004. ACM.
- [GSST06] Christian Gagné, Marc Schoenauer, Michèle Sebag, Marco Tomassini. Genetic programming for kernel-based learning with co-evolving subsets selection. *Proceedings of the 9th international conference on Parallel Problem Solving from Nature, PPSN'06*, strona 1008–1017, Berlin, Heidelberg, 2006. Springer-Verlag.
- [HCL03] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin. *A practical guide to support vector classification*. 2003.
- [HM05] Tom Howley, Michael G. Madden. The genetic kernel support vector machine: Description and evaluation. *Artificial Intelligence Review*, 24(3-4):379–395, 2005.
- [HM06] Tom Howley, Michael G. Madden. An evolutionary approach to automatic kernel construction. Stefanos Kollias, Andreas Stafylopatis, Włodzisław Duch, Erkki Oja, redaktorzy, *Artificial Neural Networks – ICANN 2006*, wolumen 4132 serii *Lecture Notes in Computer Science*, strony 417–426. Springer Berlin Heidelberg, 2006.
- [KS03] K. Krawiec, J. Stefanowski. *Uczenie maszynowe i sieci neuronowe*. Wydaw. Politechniki Poznańskiej, 2003.
- [Luk09] Sean Luke. *Essentials of Metaheuristics*. Sean Luke, 2009. available at <http://cs.gmu.edu/~sean/book/metaheuristics/>.
- [Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, wydanie 1, 1997.
- [PLM08] Riccardo Poli, William B. Langdon, Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.
- [RS04] Thomas Philip Runarsson, Sven Sigurdsson. Asynchronous parallel evolutionary model selection for support vector machines. *Neural Information Processing—Letters and Reviews*, 3(3):59–68, 2004.

- [Sea10] Luke Sean. *The ECJ Owner's Manual*. Yale Univ Pr, Pa/xdziernik 2010.
- [SL07] Keith M. Sullivan, Sean Luke. Evolving kernels for support vector machine classification. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, strona 1702–1707, New York, NY, USA, 2007. ACM.
- [SS02] B. Schölkopf, A.J. Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Adaptive computation and machine learning. MIT Press, 2002.
- [Sta03] Carl Staelin. Parameter selection for support vector machines. *Hewlett-Packard Company, Tech. Rep. HPL-2002-354R1*, 2003.
- [STC04] John Shawe-Taylor, Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.
- [YJZ02] Kai Yu, Liang Ji, Xuegong Zhang. Kernel nearest-neighbor algorithm. *Neural Process. Lett.*, 15(2):147–156, Kwiecie/n 2002.

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/>

[E] Hyperopt - Distributed Asynchronous Hyperparameter Optimization in Python

<http://jaberg.github.io/hyperopt/>

[F] <http://crsouza.blogspot.com> - lista funkcji jądrowych <http://crsouza.blogspot.com/2010/03/kernel-functions-for-ma.html>



© 2013 Tomasz Ziętkiewicz

Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu L^AT_EX.

Wykresy wygenerowane za pomocą programu *gnuplot*

Bib_LT_EX:

```
@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",
}
```