POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych

Instytut Systemów Elektronicznych

Piotr Tąkiel

nr albumu: 212512

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy

Praca wykonana pod kierunkiem

dr inż. Roberta Nowaka

Warszawa, 19 października 2014

Streszczenie

Tytuł pracy

Streszczenie

Imię i nazwisko: Piotr Tąkiel

Specjalność: Inżynieria Systemów Informacyjnych

Data urodzenia: 12 stycznia 1990

Data rozpoczęcia studiów: 1 października 2009

ŻYCIORYS

Życiorys

Spis treści

1. Wstęp . . . 5

1.1 O pracy . . . 5

1.2 Cel i zakres pracy . . . 5

1.2.1 Cel pracy . . . 5

1.2.2 Zakres pracy . . . 5

2. Wprowadzenie do problemu klasyfikacji . . . 6

3. Założenia i wymagania systemu . . . 7

3.1 Wymagania funkcjonalne . . . 7

3.1.1 Przetwarzanie wstępne i klasyfikacja . . . 7

3.1.2 Przeszukiwanie . . . 8

3.1.3 Opis systemu komputerowego . . . 11

3.2 Wymagania niefunkcjonalne . . . 12

3.3 Wymagania ograniczeń . . . 13

3.4 Podsumowanie wymagań . . . 13

4. Projekt systemu komputerowego . . . 15

4.1 Projekt biblioteki obliczeniowej . . . 16

4.1.1 Struktura biblioteki . . . 16

4.1.2 Dostęp do danych . . . 16

4.1.3 Deskryptory operacji . . . 17

4.1.4 Przeszukiwanie przestrzeni kombinacji . . . 20

4.2 Projekt aplikacji . . . 25

4.3 Projekt testów . . . 25

4.3.1 Testy biblioteki obliczeniowej . . . 25

4.3.2 Testy aplikacji . . . 25

5. Elementy implementacji systemu . . . 28

5.1 Implementacja narzędzia . . . 28

5.2 Implementacja interfejsu webowego . . . 28

6. Testy systemu . . . 29

6.1 Testy narzędzia 29
6.2 Testy interfejsu webowego 29

7. Wyniki badań 30

8. Wnioski 31

Bibliografia 32

1. Wstęp

1.1 O pracy

1.2 Cel i zakres pracy

1.2.1 Cel pracy

1.2.2 Zakres pracy

2. Wprowadzenie do problemu klasyfikacji

3. Założenia i wymagania systemu

Zdefiniowanie wymagań ma bardzo duże znaczenie podczas realizacji projektu informatycznego. Właściwe sformułowanie wymagań uszczegóławia wizję systemu oraz jest źródłem wiedzy o pożądanych możliwościach i cechach rozwiązania. Dodatkowo, prawidłowa identyfikacja wymagań nakreśla cel projektu i, co za tym idzie, pozwala uniknąć potrzeby wprowadzania kosztownych poprawek.

Wymagania dzielimy na funkcjonalne, jakościowe (niefunkcjonalne) oraz wymagania ograniczeń. Do pierwszej grupy zaliczamy wymagania opisujące możliwości, jakie projektowany system daje użytkownikom. Przykładem wymagania funkcjonalnego może być klasyfikowanie danych przy użyciu algorytmu drzew decyzyjnych. Wymagania jakościowe skupiają się na ilościowych regułach oceniania systemu, np. czy baza danych potrafi odzyskać utracone dane w mniej niż jedną sekundę. Ostatnia kategoria wymagań precyzuje granice rozwiązania.

3.1 Wymagania funkcjonalne

Zgodnie z tym, co przedstawiłem w rozdziale poświęconym celowi pracy, tworzona przeze mnie aplikacja służy ułatwieniu zadań związanych z klasyfikacją danych. Aplikacja ma za zadania wykonywać operacje, które dla użytkownika zajęły by bardzo dużo czasu. Do czynności takich można zaliczyć wstępne przetwarzanie danych oraz ich późniejszą klasyfikację, jak również wyszukiwanie kombinacji metod oraz parametrów, których użycie prowadzi do najlepszej jakości klasyfikacji. Takie sformułowanie wymagań prowadzi do wyłonienia dwóch podstawowych funkcjonalności projektowanego rozwiązania.

3.1.1 Przetwarzanie wstępne i klasyfikacja

Pierwszą z głównych funkcjonalności jest przetwarzanie wstępne oraz klasyfikacja danych zgodnie z nastawami użytkownika. W dalszej części pracy dla obu tych czynności stosować będę jeden termin – *przetwarzanie*. Odbiorca aplikacji powinien mieć pełną kontrolę nad tym w jaki sposób dane zostaną zmodyfikowane (o ile w ogóle) oraz jak ma przebiegać klasyfikacja. Powołując się na to, co opisałem we rozdziale o klasyfikacji, użytkownik powinien być w stanie:

1. Wybrać metodę radzenia sobie z brakującymi danymi w wierszach
2. Wskazać kolumny, które powinny zostać usunięte ze zbioru danych
3. Wskazać kolumny z danymi, które powinny zostać poddane normalizacji
4. Określić metodę kwantyzacji danych w kolumnach wraz z parametrami
5. Określić metodę klasyfikacji danych wraz z parametrami

Spośród wyżej wymienionych operacji tylko pierwsza i ostatnia powinny być zawsze określone. Wynika to z faktu, że wiele algorytmów klasyfikacji zakłada obecność wszystkich danych i nie uwzględnia sytuacji, w których pewnych wartości brakuje. Inaczej jest w przypadku pozostałych czynności. Usuwanie, normalizacja oraz kwantyzacja kolumn mogą mieć zasadniczy wpływ na przebieg całego algorytmu, ale nie są konieczne.

Należy zwrócić uwagę na wymaganie dotyczące kwantyzacji wartości w kolumnach. Aplikacja powinna pozwalać na wykorzystanie wielu różnych algorytmów łączenia jednocześnie. Dla każdego wybranego algorytmu kwantyzacji użytkownik powinien móc wskazać grupę kolumn poddanych łączeniu oraz parametry algorytmu.

Rezultatem działania aplikacji powinna być wartość walidacji krzyżowej na przetworzonych danych z użyciem określonego algorytmu.

3.1.2 Przeszukiwanie

Drugą podstawową funkcjonalnością jest wyszukiwanie najlepszej metody przetwarzania określonych danych. Ogromna rozmaitość dostępnych ustawień całego algorytmu sprawia, że zazwyczaj udaje nam się przejrzeć jedynie niewielki fragment wszystkich możliwości. Z tego powodu dla ludzi zadanie to wiąże się często z dobieraniem kombinacji metod oraz ich parametrów „na chybił trafił”. Czynność, która dla wielu ludzi okazuje się czasochłonna i nużąca, może być w znacznie szybszy sposób zrealizowana przez komputer.

Przeszukiwanie polegać będzie zatem na dobieraniu parametrów całego algorytmu maksymalizujących wartość funkcji celu - walidacji krzyżowej. Aby uniknąć przeuczenia i nadmiernego dopasowania ustawień do wprowadzonych danych, przeszukiwanie wykorzystywać będzie jedynie część danych. Wielkość tej części użytkownik będzie mógł ustalić samodzielnie w postaci liczby procentowej. Domyślną wartością tej części uczącej będzie 90%. Pozostałe dane zostaną użyte do ostatecznego określenia jakości parametrów algorytmu - odbędzie się to ponownie przy użyciu walidacji krzyżowej. Jak widać zatem, dane zostaną podzielone na zbiór uczący dla poszukiwań oraz na zbiór testowy. Zbiór uczący będzie dzielony na podzbiory uczące i testowe podczas określania jakości parametrów algorytmu w każdej iteracji.

Całkowita liczba wszystkich możliwych kombinacji metod przetwarzania wstępnego i klasyfikacji zależy wykładniczo od liczby atrybutów (kolumn) danych. Przykładowo, liczba możliwych kombinacji atrybutów, które aplikacja podda normalizacji, wynosi 2^n, gdzie n oznacza liczbę kolumn. Aby uniknąć długiego czasu przeszukiwania związanego z eksplozją wykładniczą, użytkownik powinien być w stanie określić podzbiór wszystkich kombinacji, które program ma przeanalizować.

Dobierając metodę określania podprzestrzeni poszukiwań, należy osiągnąć kompromis pomiędzy precyzyjnością metody oraz stopniem jej skomplikowania. Dokładna metoda mogłaby wymagać od użytkownika bezpośredniego wprowadzania opisów kombinacji, które aplikacja miałaby obsłużyć. Jednakże przy gigantycznej liczbie wszystkich możliwych wariantów, rozwiązanie to nie znalazłoby zastosowania. Skrajną alternatywą mogłoby być pytanie użytkownika, czy przeszukiwanie powinno uwzględniać określone operacje, np. usuwanie kolumn lub ich kwantyzację. Odpowiedzi „tak lub nie" byłyby wygodne dla odbiorcy, ale odbierałyby jakikolwiek wpływ na proces wyszukiwania.

Zaproponowane przeze mnie rozwiązanie stanowi złoty środek pomiędzy dwoma skrajnymi metodami. Polega ono na podziale przestrzeni poszukiwań na podprzestrzenie i na precyzowaniu każdej z osobna.

Na samym początku użytkownik powinien wybrać metody radzenia sobie z brakującymi danymi, na przykład: $\{\text{usuń wiersze}, \text{wartość średnia}\}$.

W następnej kolejności użytkownik powinien określić zbiór atrybutów R_{attr}, spośród których aplikacja może wybrać te, które zostaną usunięte. Dodatkowo użytkownik powinien określić zbiór liczb $\mathbf{R} = \{r_1, r_2, \ldots, r_n\}$, taki, że każda liczba r_i oznacza liczbę atrybutów, które algorytm będzie mógł usunąć jednocześnie ze zbioru R_{attr}.

Analogicznie powinno przebiegać określanie szczegółów normalizacji atrybutów. Użytkownik powinien wybrać zbiór atrybutów N_{attr} oraz wprowadzić zbiór liczb $\mathbf{N}$, którego każdy element oznaczać będzie liczbę atrybutów normalizowanych jednocześnie.

Opis podprzestrzeni przeszukiwań związanej z kwantyzacją okazuje się bardziej skomplikowany. Podobnie jak poprzednio, odbiorca powinien wskazać zbiór atrybutów poddanych kwantyzacji Q_{attr}. Następnie użytkownik powinien określić zbiór algorytmów kwantyzacji $\mathbf{Q}$, które mogą być użyte. W kolejnym kroku należy wprowadzić zbiór liczb Q_{pass}. Każdy element zbioru oznaczać będzie liczbę jednoczesnych przebiegów kwantyzacji. W dalszej kolejności odbiorca określi liczbę n_{max} oznaczającą maksymalną liczbę atrybutów, które mogą być łączone przez algorytmy ze zbioru $\mathbf{Q}$. Na samym końcu osoba będzie musiała opisać podzbiór wszystkich kombinacji parametrów, jakie mogą przyjąć wybrane wcześniej algorytmy kwantyzacji.

Aby uprościć to zadanie, użytkownik wprowadzi tylko jedną liczbę g, oznaczającą *granularność* podprzestrzeni parametrów.

Wiele algorytmów często przyjmuje parametry początkowe należące do określonych dziedzin. Przykładami takich dziedzin mogą być: wartości „tak" i „nie", zbiór liczb rzeczywistych od 0.01 do 100. W ogólnym przypadku nie jest możliwe przeglądanie wszystkich kombinacji parametrów dla dowolnego algorytmu – jest ich zbyt dużo lub ich wartości są nieprzeliczalne. Z tego powodu, w mojej pracy przeszukiwanie przestrzeni ograniczam do jej wybranych elementów. Naturalnym podejściem do tego problemu jest wybór takich wartości z każdej dziedziny, które byłby dla niej reprezentatywne.

Granularność liniową g_l dla zbioru $\{a, \ldots, b\}$ definiuję jako liczność zbioru $\mathbf{L} = \{l_1, l_2, \ldots, l_n\}$ takiego, że:

$$l_1 = a$$
$$l_n = b$$
$$l_{i+1} - l_i = c$$

gdzie c jest stałą, a $i \in \{1, \ldots, n-1\}$. Intuicyjnie granularność liniowa to liczba od 1 większa od liczby podziałów zbioru na przedziały o jednakowej długości.

Granularność wykładniczą g_w dla zbioru $\{a, \ldots, b\}$ definiuję jako liczność zbioru $\mathbf{W} = \{w_1, w_2, \ldots, w_n\}$ takiego, że:

$$w_1 = a$$
$$w_n = b$$
$$w_{i+1} = w_i * q$$

gdzie q jest stałą, a $i \in \{1, \ldots, n-1\}$. Posługując się intuicją, granularność wykładniczą można rozumieć jako liczbę o 1 większą od liczby podziałów zbioru na przedziały należące do różnych rzędów wielkości. Zbiory **L** i **W** nazywam *zbiorami granularnymi*.

Granularnością nazywam granularność dziedziny każdego parametru algorytmu. Przeszukiwanie przestrzeni parametrów algorytmu sprowadzam zatem do przeglądania tylko tych wartości, na jakie wskazuje granularność. Konkretniej oznacza to przeglądanie elementów iloczynu kartezjańskiego zbiorów granularnych każdego parametru algorytmu. Metoda ta wymaga wcześniejszego określenia dziedzin parametrów, ale nie stanowi to problemu, ponieważ zazwyczaj są one znane. Posługiwanie się granularnością pozwala zatem na ilościowe określenie dokładności przeszukiwań bez względu na to, z ilu parametrów dany algorytm korzysta.

Po zakończeniu ustalania szczegółów kwantyzacji użytkownikowi pozostanie ustawienie procesu klasyfikowania danych. Odbiorca aplikacji będzie musiał wskazać algorytmy, których działanie ma zostać sprawdzone, jak również granularność dla tych algorytmów.

Podsumowując wymagania dotyczące przeszukiwania przestrzeni kombinacji, użytkownik będzie mógł:

1. Wskazać metody radzenia sobie z brakującymi danymi.
2. Wytypować możliwe kombinacje atrybutów przeznaczonych do usunięcia
3. Ustalić możliwe kombinacje atrybutów przeznaczonych do normalizacji
4. Określić kombinacje algorytmów łączenia atrybutów oraz ich parametry
5. Wskazać zbiór algorytmów klasyfikacji oraz ich parametry

3.1.3 Opis systemu komputerowego

Biorąc pod uwagę różnorodność popularnie używanych systemów operacyjnych, projektowany system nie powinien wymuszać na użytkowniku żadnej konkretnej platformy. Projektowane rozwiązanie powinno działać równie dobrze zarówno na systemach z rodziny MS Windows, jak i Linux (na przykład). Jednocześnie aplikacja powinna umożliwiać przetwarzanie wstępne, klasyfikowanie danych oraz przeszukiwanie w prosty i intuicyjny sposób. W tym celu system powinien oferować przejrzysty i nieskomplikowany graficzny interfejs użytkownika. Interfejs musi pozwalać na wprowadzanie danych do przetwarzania oraz na łatwe ustawianie parametrów opisanych w poprzednich częściach rozdziału.

Projektowane narzędzie powinno radzić sobie z odczytem danych zgromadzonych w popularnych formatach plików, np. MS Excel lub CSV (*Comma Separated Values*). Z racji tego, że wprowadzane dane poddawane są algorytmom klasyfikacji, plik powinien posiadać wyszczególnioną kolumnę zawierającą informacje o kategorii obserwacji. Nagłówek tej kolumny powinien nosić nazwę *Index*.

Zadaniem systemu jest ułatwienie odnajdywania ustawień algorytmu klasyfikacji. Typowa sesja pracy z aplikacją polegać powinna na wyszukiwaniu najlepszych ustawień „na chybił trafił” lub automatycznie (przeszukiwanie). W obu wariantach pożyteczne byłoby przechowywanie informacji o wynikach wcześniejszych operacji. Mając łatwy dostęp do wcześniejszych rezultatów, użytkownik mógłby w prostszy sposób wnioskowań na temat skuteczności użytych metod. Biorąc to pod uwagę, aplikacja powinna zapisywać informacje o wynikach przeprowadzonych operacji. W przypadku przetwarzania, system zapamiętywałby wprowadzone parametry oraz wynik walidacji krzyżowej. W przypadku procesu przeszukiwania,

aplikacja zapisywałaby dwie rzeczy. Po pierwsze, opis podprzestrzeni przeszukiwań. Po drugie, najlepszy znaleziony wynik oraz opis metody, która do niego prowadzi. Aplikacja powinna umożliwiać również usuwanie wybranych zapisów.

Biorą pod uwagę możliwie długi czas przeszukiwania, system komputerowy musi pozwalać na wykonywanie tej operacji w tle. Po uruchomieniu przeszukiwania użytkownik powinien być w stanie wykonywać inne operacje w systemie. Stąd też kolejne wymaganie – system powinien umożliwiać przeprowadzanie wielu przeszukiwań jednocześnie. Oprócz tego, system powinien pozwalać odbiorcy na podglądanie aktualnego postępu operacji. W dowolnym momencie użytkownik powinien móc dowiedzieć się, jaka część wszystkich wybranych kombinacji została już sprawdzona.

Dodatkowo system nie powinien wymagać dostępu do Internetu. Aplikacja powinna być w stanie wykonać wszystkie konieczne obliczenia na pojedynczym komputerze, nie korzystając z sieci.

Poza wyżej wymienioną funkcjonalnością system komputerowy powinien oferować nieskomplikowaną metodę instalacji.

3.2 Wymagania niefunkcjonalne

System komputerowy powinien umożliwiać wykonywanie co najmniej 20 operacji przeszukiwań jednocześnie. Wymaganie to pokrywa większość scenariuszy użycia operacji przeszukiwania. Nie spodziewamy się bowiem, aby użytkownik uruchamiał ponad kilkadziesiąt operacji w tym samym czasie.

Aplikacja powinna pozwalać na zapisanie co najmniej 100000 wyników klasyfikacji oraz 100000 wyników przeszukiwań. Takie wymaganie gwarantuje, że odbiorca, będący wielkim entuzjastą uczenia maszynowego, mógłby dokonywać 50 badań każdego typu codziennie przez ponad pięć lat[1].

Ze względu na to, że posiadane przeze mnie dane zostały dostarczone w postaci pliku MS Excel, aplikacja powinna potrafić odczytywać pliki tego typu. Odczyt danych z innych rodzajów plików nie jest konieczny.

Ostatnim wymaganiem jest określenie liczby algorytmów klasyfikacji i łączenia, jakie aplikacja będzie oferowała. Aby umożliwić użytkownikowi szeroki wybór, aplikacja pozwalać

[1] Pięćdziesiąt operacji przez trzysta sześćdziesiąt pięć dni w roku przez pięć lat daje $50 * 365 * 5 = 91250$ przeprowadzonych operacji, czyli nieco mniej niż 100000.

będzie na skorzystanie z co najmniej pięciu algorytmów klasyfikacji oraz tej samej liczby metod klasteryzacji.

3.3 Wymagania ograniczeń

Projektując oprogramowanie, należy uwzględnić obecność wielu czynników ograniczających. Do czynników ograniczających można zaliczyć ograniczenia finansowe, związane z określonym budżetem przeznaczonym na realizację projektu lub jego określonej fazy. Innym rodzajem ograniczenia jest specyfika środowiska uruchomieniowego. Aplikacje wykonujące przetwarzanie rozproszone mogą wymagać określonej architektury sieci, podczas gdy niektóre systemy wbudowane potrafią działać poprawnie tylko na dedykowanych platformach sprzętowych. Ograniczenia nałożone na projekt mają często bezpośredni wpływ na architekturę końcowego rozwiązania i dlatego są równie istotne jak wymagania funkcjonalne i jakościowe.

Podstawowym ograniczeniem w mojej pracy środowisko uruchomieniowe – jest nim pojedynczy komputer PC. Poprawne działanie aplikacji nie wymaga pracy wielu komputerów, wystarczyć powinna jedna maszyna. Wymaganie to gwarantuje, że odbiorca będzie mógł korzystać z systemu np. na laptopie, będąc w podróży.

3.4 Podsumowanie wymagań

Tabela 3.1 stanowi zestawienie wymagań projektowych opisanych w tym rozdziale.

Rodzaj wymagania	Opis wymagania
Funkcjonalne	Przetwarzanie wstępne i klasyfikacja zgodnie z ustawieniami użytkownika
Funkcjonalne	Przeszukiwanie przestrzeni kombinacji metod przetwarzania zgodnie z ustawieniami użytkownika
Funkcjonalne	Praca pod systemami operacyjnymi z rodziny MS Windows i Linux
Funkcjonalne	Prosty i intuicyjny interfejs graficzny
Funkcjonalne	Odczyt danych posiadających wyszczególnioną kolumnę kategorii
Funkcjonalne	Zapisywanie rezultatów przetwarzania: ustawień algorytmu i wyniku walidacji krzyżowej
Funkcjonalne	Zapisywanie rezultatów przeszukiwania: ustawień przeszukiwania oraz nastaw najlepszej znalezionej metody oraz wyniku walidacji krzyżowej dla tej metody
Funkcjonalne	Uruchamianie wielu operacji przeszukiwania w tle
Funkcjonalne	Aktualizowanie informacji o postępie procesu przeszukiwania
Funkcjonalne	Nieskomplikowana metoda instalacji
Funkcjonalne	Aplikacja nie wymaga dostępu do Internetu
Niefunkcjonalne	Wykonywanie 20 operacji przeszukiwania jednocześnie
Niefunkcjonalne	Zapisywanie co najmniej 100000 wyników klasyfikacji oraz 100000 wyników przeszukiwań
Niefunkcjonalne	Obsługa formatu MS Excel
Niefunkcjonalne	Wybór spośród co najmniej 5 algorytmów klasyfikacji
Niefunkcjonalne	Wybór spośród co najmniej 5 algorytmów łączenia atrybutów
Ograniczeń	Aplikacja przeznaczona na pojedynczy komputer PC

Tablica 3.1. Wymagania projektowe

4. Projekt systemu komputerowego

W tym rozdziale skupię się nad projektem systemu komputerowego ułatwiającego zadania klasyfikacji danych. Projektując aplikację, postanowiłem podzielić ją oddzielne, współpracujące ze sobą, elementy. Zaproponowany przeze mnie system komputerowy składa się z **Opis modułów**. **Dłuższy opis modułów**.

Moduły współpracują ze sobą przy użyciu zdefiniowanych przeze mnie interfejsów. **Opis interfejsów**.

Dzięki rozdzieleniu całości na powiązane, lecz odrębne, moduły, prace nad wybranym fragmentem systemu można prowadzić w oderwaniu od reszty. Dodatkowo, luźno powiązana architektura umożliwia testowanie każdego podsystemu oddzielnie. Upraszcza to znacząco testowanie całej aplikacji.

Rysunek 4.1 przybliży czytelnikowi szczegóły podziału, jak również elementy wchodzące w skład każdego modułu systemu.

Rysunek 4.1. Diagram architektury systemu komputerowego

4.1 Projekt biblioteki obliczeniowej

4.1.1 Struktura biblioteki

Biblioteka obliczeniowa składać się będzie z trzech części. Pierwszą z nich będzie zbiór klas odpowiedzialnych za odczyt i dostęp do danych. Klasy te pozwalać będą również na manipulowanie danymi (np. na normalizację atrybutów) oraz na ich klasyfikację. Drugą częścią biblioteki będą klasy związane z przetwarzaniem wstępnym i klasyfikacją danych w sposób zdefiniowany przez użytkownika. Stanowić one będą opis operacji, jakie powinny być wykonane przez aplikację. Ostatnim fragmentem biblioteki obliczeniowej będą klasy związane z przeszukiwaniem przestrzeni metod.

4.1.2 Dostęp do danych

Klasa *Sample* stanowić będzie reprezentację próbki danych zawartych w pliku dostarczonym przez użytkownika. Klasa ta zawierać będzie pola odpowiadające cechom próbki (liczba wierszy, liczba i nazwy atrybutów) jak również udostępniać będzie metody manipulujące danymi. Instancje obiektów klasy *Sample* przechowywać będą wyłącznie atrybuty o ustalonych wartościach numerycznych. Opis pól i metod prezentuje listing 4.1.

Interfejs *DataFile* zapewnia dostęp do danych, które nadają się do wstępnego przetwarzania i klasyfikacji. Interfejs ten pozwala w ujednolicony sposób traktować dane pochodzące z różnych formatów plików (np. MS Excel, CSV). Metody interfejsu pokazuje listing 4.2.

Interfejs *DataFile* jest implementowany przez klasę *XlsFile*. Obiekty tej klasy umożliwiają dostęp do danych zapisanych w formatach z rodziny MS Excel.

Interfejs *AbstractClusterer* definiuje dostęp do algorytmów związanych z łączeniem atrybutów (kwantyzacją). Jedyną metodą interfejsu jest metoda *transform*. Parametrem metody jest lista wektorów z których każdy zawiera wartości wybranego atrybutu danych. Metoda zwraca pojedynczy wektor powstały poprzez złączenie wektorów wejściowych odpowiednią metodą (tabela 4.3).

Interfejs *AbstractClusterer* implementują klasy *EqualDistributionClusterer*, *KMeansClusterer* i *KMeansPlusPlusClusterer*. Stanowią one implementacje algorytmów, odpowiednio: łączenia o równej liczności grup, K-Średnich i K-Średnich plus plus **Odnośnik do rozdziału o klasyfikacji**.

Zaprojektowane klasy przedstawia diagram 4.2.

Nazwa	Opis
Pole *attributes*	Macierz zawierająca wszystkie wartości atrybutów w pliku.
Pole *categories*	Wektor zawierający wartości kategorii poszczególnych wierszy.
Pole *columns*	Lista nazw atrybutów.
Pole *ncols*	Liczba atrybutów.
Pole *nrows*	Liczba wierszy.
Metoda *from_file*	Tworzy instancję obiektu klasy *Sample*, ładując jej zawartość z instancji klasy implementującej interfejs *DataFile* oraz reperując brakujące dane przy użyciu wybranego algorytmu. Metoda ignoruje kolumny z danymi o wartościach nienumerycznych.
Metoda *remove_column*	Usuwa z próbki atrybut o wybranej nazwie.
Metoda *normalize_column*	Normalizuje atrybut o wskazanej nazwie.
Metoda *merge_columns*	Łączy atrybuty o wybranych nazwach korzystając z instancji obiektu implementującego interfejs *AbstractClusterer*.

Tablica 4.1. Opis pól i metod klasy *Sample*

Nazwa	Opis
Metoda *columns*	Zwraca listę nazw kolumn w pliku.
Metoda *rows*	Zwraca listę wierszy w pliku.

Tablica 4.2. Metody interfejsu *DataFile*.

4.1.3 Deskryptory operacji

Projektując bibliotekę postanowiłem oddzielić kod związany z wyborem parametrów przetwarzania od kodu wykonującego to przetwarzanie. W tym celu zaprojektowałem specjalne klasy, tzw. *deskryptory*. Klasy te czerpią ze wzorca projektowego *Polecenie* [1], stanowiąc opisy operacji, które mają zostać wykonane. Każdy z deskryptorów dotyczy szczególnej części przetwarzania i posiada metody wykonujące daną operację oraz metody sprawdzające poprawność opisu operacji.

Użycie deskryptorów posiada wiele zalet. Po pierwsze, logika związana z definiowaniem kolejnych kroków algorytmu jest odseparowana do właściwego algorytmu. Efektem tego jest

Nazwa	Opis
Metoda *transform*	Zwraca atrybut powstały poprzez złączenie wybranych atrybutów.

Tablica 4.3. Metody interfejsu *AbstractClusterer*.

Rysunek 4.2. Diagram klas związanych z dostępem do danych

prostszy i czytelniejszy kod. Po drugie, każdy deskryptor posiada metody umożliwiające sprawdzenie, czy wprowadzony opis operacji jest poprawny. Pomaga to w zarządzaniu kodem związanym z nieprawidłowymi ustawieniami wprowadzonymi przez użytkownika. Kolejną zaletą jest ograniczenie duplikacji kodu. Zarówno funkcjonalność związana z przetwarzaniem danych jak i ta polegająca na przeszukiwaniu mogą wykorzystywać klasy deskryptorów. Dodatkowym atutem podziału kodu na deskryptory jest uproszczenie testów jednostkowych. Testy projektowane będą pod określony deskryptor, dzięki czemu ich kod będzie czytelniejszy i prostszy w utrzymaniu.

W mojej pracy planuję zaimplementować trzy rodzaje deskryptorów. Pierwszym z nich będzie klasa *PreprocessingDescriptor* związana ze wstępnym przetwarzaniem danych. Instancje obiektu klasy *PreprocessingDescriptor* posiadać będą pola zawierające informacje o metodzie radzenia sobie z brakującymi danymi, atrybutach poddawanych normalizacji itd. Klasa *PreprocessingDescriptor* posiadać będzie również metodę wytwórczą zwracającą instancję obiektu klasy *Sample*. Struktura klasy pokazana jest w tabeli 4.4.

Następnym deskryptorem będzie klasa *QuantizationDescriptor*. Przechowywać będzie ona informacje o kwantyzacji atrybutów danych z wykorzystaniem pojedynczego, określonego algorytmu. Poza metodą sprawdzającą, klasa *QuantizationDescriptor* oferować będzie również metodę wykonującą operację łączenia na wskazanej próbce danych. Pola i metody klasy przedstawia tabela 4.5.

Nazwa	Opis
Konstruktor	Przyjmuje jako parametry: metodę naprawiania brakujących danych, atrybuty do usunięcia, atrybuty do normalizacji, kolekcję obiektów klasy *QuantisationDescriptor*.
Metoda *create_sample*	Zwraca obiekt typu *Sample* zgodny z parametrami przekazanymi do konstruktora. Jako argument pobiera obiekt implementujący interfejs *DataFile*.
Metoda *validate*	Sprawdza poprawność parametrów przekazanych do konstruktora.

Tablica 4.4. Pola i metody klasy *PreprocessingDescriptor*.

Nazwa	Opis
Konstruktor	Przyjmuje parametry: nazwy atrybutów przeznaczonych do złączenia, nazwę algorytmu kwantyzacji oraz parametry algorytmu.
Metoda *execute*	Jako argument pobiera obiekt klasy *Sample*. Wykonuje operację kwantyzacji na wskazanych atrybutach danych zawartych wewnątrz argumentu.
Metoda *validate*	Sprawdza poprawność parametrów przekazanych do konstruktora.

Tablica 4.5. Pola i metody klasy *PreprocessingDescriptor*.

Ostatnim deskryptorem jest klasa, związana z klasyfikacją danych, o nazwie *ClassificationDescriptor*. Klasa ta będzie miała za zadanie przechowywać informacje o algorytmie klasyfikacji użytym podczas przetwarzania. Podobnie jak klasa *PreprocessingDescriptor*, również ta klasa posiadać będzie metodę wytwórczą. Różnica będzie polegać na tym, że zwracać ona będzie obiekt klasyfikatora implementujący interfejs *Classifier*.

Interfejs *Classifier* udostępniać będzie dwie metody: *fit* oraz *predict*. Pierwsza z metod przyjmować będzie dwa parametry: macierz atrybutów oraz wektor z wartościami kategorii odpowiadającym poszczególnym wierszom macierzy. Wywołanie metody spowoduje nauczenie klasyfikatora zgodne z wybranym algorytmem uczenia. Metoda *predict* przyjmować będzie pojedynczy wiersz danych i zwracać kategorię, do której dane zostały zaklasyfikowane. Interfejs prezentuje rysunek 4.3.

Diagram sekwencji 4.4 ilustruje przykładowe wykorzystanie deskryptorów w programie.

Rysunek 4.3. Diagram interfejsu *IClassifier*.

Nazwa	Opis
Konstruktor	Przyjmuje parametry: nazwę algorytmu klasyfikacji, listę parametrów algorytmu.
Metoda *create_classifier*	Zwraca obiekt klasyfikatora.
Metoda *validate*	Sprawdza poprawność parametrów przekazanych do konstruktora obiektu.

Tablica 4.6. Pola i metody klasy *ClassificationDescriptor*.

4.1.4 Przeszukiwanie przestrzeni kombinacji

Drugą z kluczowych funkcjonalności systemu będzie przeszukiwanie przestrzeni wariantów przetwarzania danych. Zgodnie z założeniami opisanymi w rozdziale poświęconym wymaganiom systemu, aplikacja umożliwiać będzie sprecyzowanie podprzestrzeni poszukiwań a następnie przystąpienie do wyszukiwania najlepszego rozwiązania. Aplikacja sprawdzać będzie poszczególne kombinacje operacji związanych z przetwarzaniem wstępnym oraz klasyfikacją. Przeszukiwanie sprawdzi wszystkie warianty, które będą odpowiadać opisowi wprowadzonemu przez użytkownika. W zależności od opisu, czas wykonywania operacji przewiduję na od kilku sekund do kilkunastu godzin. Z tego względu postarałem się zaprojektować przeszukiwanie w sposób umożliwiający łatwe sprawdzenie jego poprawności. Dodatkowo, przygotowany przeze mnie projekt rozdziela przeszukiwanie na luźno powiązane fragmenty, uproszczając tym samym bardzo skomplikowaną implementację.

Przeszukiwanie, najprościej mówiąc, polegać będzie na przeglądaniu wszystkich kombinacji metod przeszukiwania, w celu znalezienia tej, która dawać będzie najwyższy wynik

Rysunek 4.4. Wykorzystanie klasyfikatorów *PreprocessingDescriptor*, *QuantizationDescriptor* oraz *ClassificationDescriptor*.

Rysunek 4.5. Poglądowy algorytm przeszukiwania

walidacji krzyżowej. Pomysł ten postanowiłem sprowadzić do sekwencji zagnieżdżonych pętli 4.5. Każda pętla odpowiadać będzie osobnej operacji związanej z przetwarzaniem danych. Zadaniem wszystkich pętli będzie przyrostowe tworzenie opisu czynności do wykonania. Polegać to będzie na uzupełnianiu pól deskryptorów informacjami związanymi z etapami przetwarzania odpowiadającym danym pętlom. Po uzupełnieniu odpowiedniego pola deskryptora, każda pętla przekaże go do pętli wewnątrz niej. Dzięki temu, podczas każdej iteracji pętli najgłębiej zagnieżdżonej otrzymamy deskryptory opisujące konkretną kombinację elementów przetwarzania. Informacja zawarta w deskryptorach będzie wystarczająca aby wykonać pojedyncze przetwarzanie i otrzymać wynik walidacji krzyżowej.

Wątek (lub proces) generujący opisy operacji będzie mógł rozdzielać je mi

Rysunek 4.6. Interfejs *AbstractSearchSpace* oraz klasy implementujące go.

Należy wspomnieć, że kolejność zagnieżdżania pętli odgrywa tu rolę. Dzieje się tak, ponieważ niektóre operacje mogą mieć wpływ na decyzję o wykonaniu innych operacji. Przykładowo, po usunięciu atrybutu o nazwie *wiek* nie jest możliwe normalizowanie atrybutu o tej samej nazwie - z tego powodu pętla odpowiadająca normalizacji musi być umieszczona wewnątrz pętli odpowiadającej usuwaniu.

W celu uproszczenia kodu związanego z obsługą każdej z pętli, iterowanie po każdym poziomie zagnieżdżenia realizowane będzie przy pomocy obiektów specjalnych klas. Każda z klas będzie wykorzystywać wzorzec projektowy *Iterator* [1]. W mojej pracy sprowadzi się to do tego, że każda z klas implementować będzie interfejs *AbstractSearchSpace*. Diagram klas 4.6 ilustruje zależności między klasami.

Interfejs *AbstractSearchSpace* udostępniać będzie tylko jedną metodę o nazwie *generate*. Argumentem metody będzie deskryptor - w założeniu pochodzący z pętli z wyższego poziomu zagnieżdżenia. Metoda zwracać będzie ten sam deskryptor lecz uzupełniony o informacje związane z danym etapem przetwarzania.

Pierwszą z klas implementujących interfejs *AbstractSearchSpace* jest klasa *FixSpace*. Obiekt tej klasy ma za zadanie uzupełnić deskryptor o informację związaną z metodą pozbywania się brakujących danych. Argumentem konstruktora klasy będzie lista metod, z których algorytm przeszukiwania może skorzystać aby pozbyć się brakujących danych. Dla każdej metody z listy obiekt klasy wygeneruje pojedynczy deskryptor korzystający z danej metody.

Następną klasą implementującą interfejs będzie klasa o nazwie *RemoveSpace*. Konstruktor klasy będzie przyjmować dwa argumenty: listę nazw atrybutów przeznaczonych do usunięcia **L** oraz listę krotności. Lista krotności zawierać będzie liczby odpowiadające liczbom atrybutów usuwanych jednocześnie. Dla każdej liczby krotności n algorytm wygeneruje wszystkie

podzbiory zbioru **L** o liczności n. Przykładowo, dla listy atrybutów $\{a, b, c, d\}$ oraz listy krotności $\{1, 2\}$ algorytm wytworzy $\binom{4}{1} + \binom{4}{2} = 10$ kombinacji.

W podobny sposób zachowywać się będą obiekty klasy *NormalizeSpace*. Zarówno jak w przypadku klasy *RemoveSpace* konstruktor oczekiwać będzie listy nazw atrybutów oraz listy krotności. Różnica polegać będzie na tym, czego otrzymane kombinacje będą dotyczyć. Obiekt klasy *NormalizeSpace* będzie bowiem uzupełniał dany deskryptor informacją dotyczącą normalizacji kolumn.

Obie klasy – *RemoveSpace* i *NormalizeSpace* – wykorzystywać będą algorytm generujący kolejne podzbiory o określonej liczności. W celu uniknięcia duplikacji kodu, algorytm odpowiedzialny za generowanie podzbiorów zostanie zaimplementowany w oddzielnej klasie o nazwie *SubsetGenerator*. Klasa ta, podobnie jak klasy implementujące interfejs *AbstractSearchSpace*, oparta będzie na wzorcu projektowym iteratora.

Kolejną klasą implementującą interfejs *AbstractSearchSpace* będzie klasa o nazwie *QuantifySpace*. Zadaniem obiektu tej klasy jest generowanie deskryptorów posiadających informację o kwantyzacjach, które mają zostać wykonane.

Konstruktor klasy przyjmuje zestaw argumentów odpowiadający wymaganiom funkcjonalnym przeszukiwania opisanym w rozdziale 3.1.2. Pierwszym argumentem jest lista nazw atrybutów, które mogą wziąć udział w procesie kwantyzacji. Drugim argumentem jest lista obiektów implementujących interfejs *AbstractClusterer* reprezentujących poszczególne algorytmy kwantyzacji. Następny argument to lista liczb odpowiadających liczbom kwantyzacji, które mogą zachodzić w tym samym czasie. Argument *max_cols* określa maksymalną liczbę atrybutów, które każdy z algorytmów kwantyzacji może łączyć. Ostatni argument określa granularność przeszukiwania.

Ostatnią klasą odpowiedzialną za szczególny etap przeszukiwania będzie klasa *ClassificationSpace*. Konstruktor klasy przyjmować będzie jedynie dwa parametry – listę obiektów implementujących interfejs *Classifier* oraz granularność operacji. Obiekt klasy *ClassificationSpace* wygeneruje wszystkie kombinacje przekazanych algorytmów klasyfikacji oraz ich parametrów, które określać będzie wartość granularności.

Iterowanie z wykorzystaniem powyższych klas zaimplementowane zostanie w klasie *SearchSpace*. Klasa ta, będąca kolejnym przykładem iteratora, pozwoli na łatwe generowanie kolejnych deskryptorów. Konstruktor klasy przyjmować będzie obiekty klas *FixSpace*, *RemoveSpace*, *NormalizeSpace*, *QuantifySpace* oraz *ClassificationSpace*. Przy każdej iteracji obiekt klasy *SearchSpace* zwróci parę deskryptorów – obiekty klas *PreprocessingDescriptor* oraz *ClassificationDescriptor*.

Rysunek 4.7. Diagram klasy *SearchAlgorithm*.

Wykorzystanie klas, jakimi są deskryptory, umożliwi rozdzielenie kodu odpowiedzialnego za wytwarzanie deskryptorów od kodu przetwarzającego dane zgodnie z opisem zawartym w deskryptorach. Rozdzielenie tych obowiązków posiada bardzo dużą zaletę. Na komputerach wyposażonych w przynajmniej dwa rdzenie procesora aplikacja będzie mogła w prosty sposób przetwarzać dane równolegle. Możliwość ta wynika z faktu, że wygenerowanie pojedynczego deskryptora zajmować będzie znacznie mniej czasu, niż przetworzenie danych związanych z nim. Wątek (lub proces) będzie „rozdawał" opisy operacji innym wątkom (lub procesom) i co jakiś czas odbierał rezultaty obliczeń. Rozwiązanie to pozwoli na skrócenie czasu przeszukiwania.

Przeprowadzenie przeszukiwania z wykorzystaniem wyżej opisanych klas zostanie umieszczone wewnątrz klasy *SearchAlgorithm*. Diagram klasy prezentuje rysunek 4.7. Konstruktor klasy pobierać będzie trzy parametry. Pierwszym z nich będzie ścieżka do pliku z danymi. Drugim parametrem będzie obiekt klasy *SearchSpace* reprezentujący przestrzeń kombinacji do przeszukania. Ostatnim parametrem będzie liczba wątków wykonania użyta podczas przeszukiwania.

Przeszukiwanie rozpocznie się po wywołaniu metody *start* obiektu klasy *SearchAlgorithm*. Po wywołaniu tej metody zostanie utworzony osobny wątek w którym rozpocznie się wyszukiwanie. Metoda *progress* pozwoli na podejrzenie postępu przeszukiwania. Zwróci ona liczbę z przedziału $[0, 1]$ oznaczającą postęp przeszukiwań. O tym, czy przeszukiwanie zostało zakończone informować będzie wartość zwrócona przez metodę *running*. Po zakończeniu przeszukiwania metoda *result* zwróci strukturę zawierającą parę deskryptorów dającą najlepszy wynik jak również sam wynik. Rozpoczęte przeszukiwanie będzie można zatrzymać metodą *stop*.

4.2 Projekt aplikacji

4.3 Projekt testów

4.3.1 Testy biblioteki obliczeniowej

Biblioteka obliczeniowa stanowi fundament całej aplikacji. Wyniki działania biblioteki decydować będą o rezultatach badań, jakie przeprowadzać będzie użytkownik. Z tego powodu niezmiernie istotne jest upewnienie się, że biblioteka obliczeniowa działa poprawnie.

W celu sprawdzenia tego istotnego warunku, poprawność działania klas w bibliotece będzie sprawdzana przy pomocy testów jednostkowych. Dzięki modularnej strukturze biblioteki opisanej w punkcie 4.1.1, możliwe jest oddzielne testowanie poszczególnych klas biblioteki. Weryfikacja poprawności działania poszczególnych fragmentów biblioteki umożliwi wyeliminowanie większości możliwych błędów.

Tabela 4.7 prezentuje opis testów jednostkowych badających poprawność działania części biblioteki odpowiedzialnej za odczyt i przetwarzanie wstępne danych. Tabela 4.8 pokazuje testy klas-deskryptorów. Testy jednostkowe klas odpowiadających za przeszukiwanie zostały umieszczone w tabeli 4.9.

4.3.2 Testy aplikacji

Nazwa klasy	Opis testów
XlsFile	Weryfikacja stan obiektu po odczytaniu różnych poprawnych danych poprawnych, weryfikacja rzucania wyjątków podczas odczytu danych niepoprawnych.
KMeansClusterer	Sprawdzenie, czy wynik klasteryzacji podlega założeniom algorytmu K-Średnich, weryfikacja rzucania wyjątków w przypadku otrzymania nieprawidłowych parametrów.
KMeansPlusPlus-Clusterer	Sprawdzenie, czy wynik klasteryzacji podlega założeniom algorytmu K-Średnich++, weryfikacja rzucania wyjątków w przypadku otrzymania nieprawidłowych parametrów.
EqualDistribution-Clusterer	Sprawdzenie, czy wynik klasteryzacji podlega założeniom algorytmu łączenia o równej liczności grup, weryfikacja rzucania wyjątków w przypadku otrzymania nieprawidłowych parametrów.
Sample	Badanie stanu obiektu po odczycie poprawnych danych, badanie rzucania wyjątków podczas odczytu danych niepoprawnych. Badanie poprawności działania metod modyfikujących atrybuty danych, weryfikacja rzucania wyjątków podczas wołania metod z nieprawidłowymi argumentami.

Tablica 4.7. Testy klas biorących udział w odczycie i przetwarzaniu wstępnym danych.

Nazwa klasy	Opis testów
Quantification-Descriptor	Sprawdzenie czy opis w deskryptorze odpowiada operacjom kwantyzacji przeprowadzanym przez deskryptor, badanie rzucania wyjątków dla nieprawidłowych parametrów deskryptora.
Preprocessing-Descriptor	Weryfikowanie czy opis w deskryptorze odpowiada przetwarzaniu wstępnemu przeprowadzonemu przez deskryptor, badanie rzucania wyjątków dla nieprawidłowych parametrów deskryptora.
Classification-Descriptor	Badanie czy opis w deskryptorze odpowiada operacji klasyfikacji przeprowadzonej przez deskryptor, badanie rzucania wyjątków dla nieprawidłowych parametrów deskryptora.

Tablica 4.8. Testy jednostkowe deskryptorów

Nazwa klasy	Opis testów
FixSpace	Testowanie poprawności kombinacji wygenerowanych przez obiekt klasy, badanie rzucania wyjątków dla nieprawidłowych parametrów konstruktora.
RemoveSpace	Testowanie poprawności kombinacji wygenerowanych przez obiekt klasy, badanie rzucania wyjątków dla nieprawidłowych parametrów konstruktora.
NormalizeSpace	Testowanie poprawności kombinacji wygenerowanych przez obiekt klasy, badanie czy kombinacje uwzględniają ew. usunięcie niektórych atrybutów. Weryfikowanie rzucania wyjątków dla nieprawidłowych parametrów konstruktora.
QuantifySpace	Testowanie poprawności kombinacji wygenerowanych przez obiekt klasy, badanie czy kombinacje uwzględniają ew. usunięcie niektórych atrybutów. Weryfikowanie rzucania wyjątków dla nieprawidłowych parametrów konstruktora.
ClassificationSpace	Testowanie poprawności kombinacji wygenerowanych przez obiekt klasy, badanie rzucania wyjątków dla nieprawidłowych parametrów konstruktora.
SearchSpace	Badanie poprawności deskryptorów wygenerowanych przez obiekt klasy przy różnych ustawieniach klas odpowiadających za poszczególne etapy przeszukiwania.
SearchAlgorithm	Sprawdzenie poprawności stanu obiektu przed, w trakcie i po zakończeniu przeszukiwania. Testowanie informacji przedstawiającej postęp przeszukiwania oraz końcowy wynik.

Tablica 4.9. Testy klas związanych z przeszukiwaniem

5. Elementy implementacji systemu

5.1 Implementacja narzędzia

5.2 Implementacja interfejsu webowego

6. Testy systemu

6.1 Testy narzędzia

6.2 Testy interfejsu webowego

7. Wyniki badań

8. Wnioski

Bibliografia

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*