Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Uczenie Maszynowe

Inkrementacyjna indukcja reguł
Projekt - dokumentacja

14 lutego 2025

Maciej Lipski, Karol Żelazowski

Spis treści

1. Streszczenie założeń z projektu wstępnego

Projekt dotyczy realizacji algorytmu inkrementacyjnej indukcji reguł, pozwalającego na dynamiczne dostosowywanie zbioru reguł na podstawie danych przychodzących sekwencyjnie. Głównym założeniem jest dostosowanie klasycznych algorytmów wsadowych do inkrementacyjnej aktualizacji reguł, unikając konieczności przetwarzania całego zbioru od nowa.

W ramach projektu zmodyfikowano klasyczny algorytm specjalizacji AQ, przystosowując go do iteracyjnego przyjmowania nowych danych. Algorytm przyjmuje przykłady w porcjach i aktualizuje zbiór reguł, minimalizując konieczność przebudowy. Reguły są generowane dla przyjmowanych przykładów niepokrywanych przez istniejące reguły, z uwzględnieniem zarówno nadchodzących, jak i przetworzonych przykładów. Przewidziano także mechanizm usuwania reguł nadmiarowych lub o wysokim wskaźniku błędu, co minimalizuje ryzyko przeuczenia. W inkrementacyjnej implementacji indukcji reguł ważnym zagadnieniem jest obsługa reguł błędnie klasyfikujących przychodzące przykłady. Przewidziano identyfikację takich i usuwanie ich, gdy cechuje je wysoki wskaźnik błędu.

Zgodnie z założeniami, jakość reguł oceniana jest za pomocą funkcji uwzględniającej liczbę przykładów poprawnie pokrywanych przez regułę, liczbę przykładów negatywnych niepokrywanych oraz liczbę przykładów pokrywanych wyłącznie przez daną regułę.

W zakresie testów działania algorytmu przewidziano testy na dużych i ograniczonych zbiorach, posiadających zarówno atrybuty dyskretne jak i ciągłe. Przy eksperymentach uwzględniono także porównanie przygotowanego algorytmu do innych algorytmów indukcji reguł oraz do implementacji algorytmu AQ dostępnej w Internecie ([5]), a także do wyników innych algorytmów na zbiorach testowych.

2. Pełny opis funkcjonalny

Przygotowany algorytm stanowi implementację modyfikacji algorytmu AQ w języku Python. Umożliwia dynamiczne budowanie, aktualizację i optymalizację zbioru reguł w sposób inkrementacyjny. Przyjmuje przykłady sekwencyjnie lub partiami, generując reguły, które klasyfikują dane wejściowe, przy jednoczesnym minimalizowaniu konfliktów i redundancji. Obsługuje zarówno atrybuty dyskretne jak i ciągłe.

Algorytm przyjmuje dane wejściowe w postaci listy przykładów, gdzie każdy przykład to słownik, który składa się z sekcji *features* opisującej wartości cech oraz etykiety klasy *class*. Przed rozpoczęciem działania algorytm wymaga podania listy atrybutów. Mogą być one liczbowe, określone przez zakres i krok (step - minimalna różnica wartości) lub dyskretne, z ustalonym zbiorem możliwych wartości. Specyfikacja atrybutów w formie słownika umożliwia elastyczną obsługę różnych typów danych i zapewnia zgodność z procesami specjalizacji reguł.

Algorytm rozpoczyna działanie poprzez wsadową inicjalizację (3.2). Polega ona na budowaniu początkowego zbioru reguł na podstawie pierwszej partii danych, tak jak to ma miejsce w klasycznym algorytmie AQ. Dla każdego przykładu sprawdzane jest, czy jest on pokrywany przez istniejące reguły. Jeśli przykład nie jest pokrywany, tworzona jest dla niego nowa reguła. Reguły są optymalizowane i przycinane.

Po wsadowej inicjalizacji możliwe jest inkrementacyjne przyjmowanie przykładów (3.5). Polega ono na dynamicznym dostosowaniu zbioru reguł na podstawie nowych przykładów. Każdy nowy przykład jest analizowany. Dla przykładów niepokrywanych przez istniejące reguły tworzona jest nowa reguła. Jeśli przykład jest błędnie klasyfikowany, wywoływana jest specjalna metoda, która zajmuje się obsługą takiego przypadku (3.6). Reguły generujące znaczące błędy są usuwane, a dla przykładów, które nie są pokrywane ze względu na ich usunięcie, generowane są nowe reguły.

Tworzenie nowych reguł przebiega podobnie jak w klasycznym algorytmie AQ (3.3). Nowe reguły powstają poprzez specjalizację reguł maksymalnie ogólnych. Reguły specjalizuje się iteracyjnie

tak, aby pokrywały dany przykład, jednocześnie minimalizując pokrycie przykładów negatywnych. Ostateczna reguła jest dodawana do zbioru, a redundancje są eliminowane.

Algorytm obsługuje także przycinanie zbioru reguł (3.7). Opiera się ono o redukowanie zbioru reguł, w celu usunięcia redundancji i optymalizacji kompleksów. Usuwane są duplikaty reguł, a liczba reguł jest ograniczana dla każdej klasy do maksymalnej liczby, zależnej od liczby przykładów w tej klasie.

Wytrenowany algorytm obsługuje klasyfikację przykładów na podstawie istniejących reguł (3.9). W procesie klasyfikacji wybierane są reguły pokrywające dany przykład. Przykładowi przypisuje się klasę reguły, która go pokrywa lub jeśli wiele reguł pasuje, wybierana jest klasa reguły o najwyższej jakości (3.8).

3. Opis implementacji algorytmu

W poniższych sekcjach opisano szczegółowo zaimplementowane metody, które składają się na przygotowany algorytm. W wielu sekcjach zamieszczono także kod danej metody lub w wybranych przypadkach ograniczono go do istotnych dla opisu fragmentów. Mniej złożone metody opisano w sekcjach dotyczących funkcji je wykorzystujących.

3.1. Inicjalizacja algorytmu (*init*)

Klasa *RuleInduction*, reprezentująca algorytm, rozpoczyna działanie od inicjalizacji niezbędnych struktur i parametrów algorytmu. Zbiór reguł przechowywany jest w liście *rules* natomiast przetworzone przykłady przechowywane są w liście *processed_examples*. Dodatkowo klasa przechowuje specyfikację atrybutów w postaci słownika *attributes*, który definiuje zakresy dla atrybutów liczbowych (ciągłych) lub możliwe wartości dla atrybutów dyskretnych.

Dla atrybutów liczbowych, które wymagają specjalnej obsługi w procesie specjalizacji, algorytm przechowuje również informacje o kroku kwantyzacji w słowniku *attribute_steps*. Podczas inicjalizacji obiekt reprezentujący algorytm przetwarza zdefiniowane atrybuty wejściowe, rozpoznając, czy dany atrybut jest liczbowy (posiada zakres i krok) czy dyskretny (ustalona lista wartości). Umożliwia mu operowanie zarówno na danych ciągłych, jak i kategorycznych.

```python
class RuleInduction:
    def __init__(self, attributes):
        """
        Inicjalizuje obiekt RuleInduction.
        """
        self.rules = []  # Zbiór reguł
        self.processed_examples = []  # Zbiór przetworzonych przykładów
        self.attributes = {}  # Atrybuty i ich zakres (dla liczbowych)
        self.attribute_steps = {}  # Kroki kwantyzacji dla liczbowych atrybutów
         self.processed = 0  # Licznik przetworzonych przykładów (do kontrolowania postępów)

        # Przetwarzanie wejściowych atrybutów
        for key, value in attributes.items():
            if isinstance(value, dict) and 'range' in value and 'step' in value:
                # Atrybut liczbowy z zakresem i krokiem
                self.attributes[key] = value['range']
                self.attribute_steps[key] = value['step']
            else:
```

```
                # Atrybut dyskretny (lista wartości)
                self.attributes[key] = value
```

3.2. Inicjalizacja wsadowa (*batch_initialization*)

Metoda *batch_initialization* odpowiada za wsadową inicjalizację algorytmu, w którym algorytm przetwarza jednorazowo wszystkie przykłady z podanego zbioru, jak w klasycznym algorytmie AQ. Na początku wszystkie przykłady są dodawane do listy *processed_examples*, która przechowuje przetworzone dane, aby używać je przy ocenie jakości. Następnie algorytm iteracyjnie analizuje każdy przykład i sprawdza, czy jest on pokrywany przez istniejące reguły. Robi to wykorzystując metodę *is_covered_by_rules*, która sprawdza, czy wartość atrybutów dla przykładu znajduje się w zakresie obejmowanym przez obecne w zbiorze reguł reguły.

Dla przykładów, które nie są pokrywane, generowana jest nowa reguła za pomocą metody *create_new_rule* (opisana w sekcji 3.3). Każda nowa reguła jest następnie weryfikowana i dodawana do zbioru reguł za pomocą metody *update_rule_set*, która dodaje reguły do zbioru reguł i dokonuje jego przycinania (metoda *prune_rules* opisana w sekcji 3.7). W trakcie działania metody licznik przetworzonych przykładów jest aktualizowany, w celu drukowania komunikatów o postępach.

```python
def batch_initialization(self, examples):
    """Inicjalizacja wsadowa."""
    self.processed_examples.extend(examples)
    for example in examples:
          if not self.is_covered_by_rules(example):  # Tworzenie nowej reguły dla niepokrytego przykładu
            new_rule = self.create_new_rule(example)
            self.update_rule_set(new_rule)
        self.processed += 1  # Aktualizacja testowego licznika przetworzonych przykładów
        print(f'Przetworzone przykłady: {self.processed}')
```

3.3. Tworzenie nowych reguł (*create_new rule*)

Funkcja *create_new_rule* to kluczowy element algorytmu odpowiedzialny za generowanie nowych reguł na podstawie przykładów treningowych. Rozpoczyna swoje działanie od zainicjalizowania maksymalnie ogólnego kompleksu *G_s*, który obejmuje wszystkie możliwe wartości atrybutów. Dla atrybutów liczbowych są to pełne zakresy ich wartości, a dla dyskretnych wszystkie możliwe opcje.

Następnie algorytm iteracyjnie specjalizuje ten kompleks (opis specjalizacji w sekcji 3.4), aby wykluczyć przykłady negatywne ze zbioru *R0*, które nie należą do klasy danego przykładu pozytywnego. Po każdej iteracji specjalizacji negatywne przykłady, które zostały poprawnie wykluczone, są usuwane z *R0*. Dodatkowo funkcja *select_best_complexes* wybiera jedynie 2 najlepsze kompleksy, na podstawie ich jakości wyznaczanej przez *complex_quality* (według opisu z sekcji 3.8), do następnej tury specjalizacji.

Poza całkowitym wykluczenie pokrywania przykładów negatywnych (sprawdza to funkcja *negative_examples_covered*), iteracje specjalizacji są ograniczone przez dwa warunki zakończenia. Ograniczono je ze względu na brak poprawy jakości kompleksów przez określoną liczbę kolejnych iteracji[1]. Ustalono także ogólną maksymalną liczbę iteracji, aby uniknąć zapętlania algorytmu na specjalizacji wyjątkowych przykładów. Przerwanie ze względu na maksymalną liczbę iteracji ma jednak miejsce bardzo rzadko. W eksperymentach zaobserwowano to jedynie raz. Dodatkowo dopuszczono częściowe pokrywanie przykładów negatywnych przez regułę, aby zapobiec tworzeniu reguł bardzo

[1]Liczbę tą ustalono na 5. Wartość ta może ulec zmianie podczas eksperymentów.

szczegółowych. Funkcję *negative_examples_covered* zaimplementowano tak, aby pozwalała na pokrywanie przez regułę 5% przykładów negatywnych[2].

Po każdej turze specjalizacji specjalizacji, funkcja *select_best_complexes* wybiera 2 najlepsze kompleksy, na podstawie ich jakości wyznaczanej przez *complex_quality* (według opisu z sekcji 3.8). Po zakończeniu specjalizacji, metoda zwraca kompleks o najwyższej jakości, który może być dodany jako nowa reguła.

```python
def create_new_rule(self, example):
    G_s = [{
        'conditions': {
            key: self.attributes[key] if isinstance(self.attributes[key], list) else
                self.attributes[key][0], self.attributes[key][1])
            for key in self.attributes
        },
        'class': example['class']
    }]  # Kompleks maksymalnie ogólny - punkt wyjścia do specjalizacji.
    # Dla atrybutów dyskretnych: wszystkie opcje; dla liczbowych: pełny zakres wartoś

    R0 = [x for x in self.processed_examples if x['class'] != example['class']]

    prev_best_quality = None  # Przechowuje jakość najlepszego kompleksu
    no_change_counter = 0  # Licznik iteracji bez poprawy jakości
    max_no_change_iterations = 5  # Maksymalna liczba iteracji bez poprawy jakości

    max_iterations = 100  # Globalny limit iteracji, potrzebny w awaryjnych sytuacjac
    iteration = 0  # Licznik iteracji

    while self.negative_examples_covered(G_s, R0):
        if iteration >= max_iterations:
            # Przerywanie długich iteracji specjalizacji
            print("Przekroczono maksymalną liczbę iteracji. Przerywanie.")
            break

        G_s = self.specialize(G_s, example, R0)  # Specjalizacja na ziarnie z R0
          G_s = self.select_best_complexes(G_s, n=2)  # Wybranie dwóch najlepszych
kompleksów do dalszej specjalizacji

        # Aktualizacja R0 - usuń przykłady negatywne, które już nie są pokrywane
        R0 = [x for x in R0 if any(self.is_covered_by_rule(complex_, x) for complex_
in G_s)]

       best_quality = self.complex_quality(G_s[0])  # Sprawdzanie jakości najlepszego
kompleksu
        if prev_best_quality is not None and best_quality == prev_best_quality:
            # Monitorowanie, czy specjalizacja przynosi zmianę jakości kompleksu
```

[2]Optymalny procent dopuszczalnych pokrywanych przykładów negatywnych będzie mógł zostać zweryfikowany w trakcje eksperymentów

```
            no_change_counter += 1
            if no_change_counter >= max_no_change_iterations:
                print("Jakość kompleksów nie poprawia się. Przerywanie.")
                break
        else:
            no_change_counter = 0

        prev_best_quality = best_quality
        iteration += 1

    return self.select_best_complexes(G_s, n=1)[0]  # Zwróć jedną najlepsza regułę
```

3.4. Specjalizacja (*specialize*)

Funkcja *specialize* odpowiada za proces specjalizacji kompleksów, w celu wykluczenia przykładów negatywnych, przy jednoczesnym zachowaniu zgodności z przykładem pozytywnym. Specjalizacja polega na iteracyjnym zawężaniu warunków kompleksów, aby przestały pokrywać przykłady negatywne, jednocześnie zachowując możliwość pokrycia przykładu pozytywnego.

Głównym elementem metody jest iteracja przez kompleksy w zbiorze *G_s* i przykłady negatywne. Dla każdego atrybutu kompleksu podejmowana jest próba specjalizacji w celu wykluczenia przykładu negatywnego. Specjalizowane są zarówno atrybuty liczbowe, jak i dyskretne.

W przypadku obsługi atrybutów liczbowych, które są reprezentowane jako przedziały, proces specjalizacji działa w następujący sposób. Jeśli wartość atrybutu w przykładzie negatywnym mieści się w aktualnym przedziale atrybutu kompleksu, sprawdzana jest relacja między wartością pozytywną a negatywną. Jeżeli wartość pozytywna jest większa od wartości negatywnej, przedział jest zawężany od dołu o krok kwantyzacji, podany przy inicjalizacji algorytmu w słowniku atrybutów. Natomiast gdy wartość pozytywna jest mniejsza, przedział zawężany jest od góry o ten sam krok.

Dla atrybutów dyskretnych specjalizacja nie jest możliwa, jeśli wartość przykładu negatywnego jest zgodna z wartością przykładu pozytywnego. W przeciwnym wypadku, jeśli wartość przykładu negatywnego znajduje się w możliwych wartościach kompleksu, atrybut jest specjalizowany poprzez usunięcie tej wartości z warunków kompleksu.

Po zakończeniu iteracji specjalizacji zwracana jest lista nowo powstałych kompleksów. Z pomocą funkcji is_general usuwane są kompleksy, które nie są najbardziej ogólne. Dla atrybutów liczbowych kompleks bardziej szczegółowy to taki, którego zakres wartości przedziału atrybutu mieści się w innych kompleksach. Dla atrybutów dyskretnych kompleks bardziej szczegółowy to taki, którego zbiór wartości jest podzbiorem zbioru wartości kompleksu bardziej ogólnego.

```
def specialize(self, G_s, example, R0):                                    Python
    new_complexes = []
    for complex_ in G_s:  # Iteracja przez kompleksy w G_s
        for negative_example in R0:  # Iteracja przez przykłady negatywne
            specialized_conditions = []
            for attr in complex_['conditions']:
                # Obsługa atrybutów liczbowych - reprezentowanych jako tuple
                if isinstance(complex_['conditions'][attr], tuple):
                      low, high = complex_['conditions'][attr] # Pobranie przedziału
atrybutu
                    if low <= negative_example['features'][attr] <= high:
```

```
                        # Pobranie kroku (kwantu) dla atrybutu
                        step = self.attribute_steps[attr]
                         if example['features'][attr] > negative_example['features']
[attr]:
                            # Zawężenie przedziału od dołu o kwant
                            new_condition = {
                                'conditions': {
                                    **complex_['conditions'],
                                   attr: (negative_example['features'][attr] + step,
high)
                                },
                                'class': complex_['class']
                            }
                        else:
                            # Zawężenie przedziału od góry o kwant
                            new_condition = {
                                'conditions': {
                                    **complex_['conditions'],
                                    attr: (low, negative_example['features'][attr] -
step)
                                },
                                'class': complex_['class']
                            }
                        specialized_conditions.append(new_condition)
                # Obsługa atrybutów dyskretnych
                elif example['features'][attr] == negative_example['features'][attr]:
                     continue  # Jeśli wartość pozytywna i negatywna są takie same,
brak możliwości specjalizacji
                  elif negative_example['features'][attr] in complex_['conditions']
[attr]:
                        new_condition = {
                            'conditions': {
                                **complex_['conditions'],
                                attr: [val for val in complex_['conditions'][attr]
                                       if val != negative_example['features'][attr]]
                            },
                            'class': complex_['class']
                        }
                        specialized_conditions.append(new_condition)
            new_complexes.extend(specialized_conditions)
```

3.5. Iteracyjne przyjmowanie przykładów (*process_examples*)

Metoda *process_examples* odpowiada za inkrementacyjne przetwarzanie nowych przykładów w algorytmie. Na początku nowe przykłady są dodawane do zbioru już przetworzonych przykładów, w celu uwzględnienia ich przy ocenie jakości podczas tworzenia nowych reguł. Następnie, dla każdego nowego przykładu, algorytm sprawdza, czy jest on błędnie klasyfikowany przez istniejące reguły. Jest to konieczne w procesie inkrementacji, ponieważ w przeciwieństwie do wersji wsadowej,

przy tworzeniu nowych reguł nie wykluczono pokrywania wszystkich przykładów negatywnych. W przypadku błędnej klasyfikacji, metoda *handle_misclassification* podejmuje odpowiednie działania, aby zaktualizować zbiór reguł.

Jeśli przykład nie jest pokrywany przez żadną regułę, tworzona jest nowa reguła za pomocą metody *create_new_rule*, a następnie aktualizowany jest zbiór reguł. Po przetworzeniu każdego przykładu aktualizowany jest licznik przetworzonych przykładów, aby umożliwić monitorowanie postępu pracy algorytmu.

```python
def process_examples(self, new_examples):
    self.processed_examples.extend(new_examples)
    for example in new_examples:
       if self.is_misclassified(example):  # Sprawdzenie czy przykład jest niepoprawnie klasyfikowany przez regułę
            # w zbiorze reguł i przetworzenie tego faktu
            self.handle_misclassification(example)
         elif not self.is_covered_by_rules(example):  # Nowa reguła dla przykładu niepokrytego przez żadne
            # reguły
            new_rule = self.create_new_rule(example)
            self.update_rule_set(new_rule)
        self.processed += 1
        print(f'Przetworzone przykłady: {self.processed}')
```

3.6. Obsługa błędnej klasyfikacji (*handle_misclassification*)

O wywołaniu obsługi błędnej klasyfikacji decyduje funkcja *is_misclassified*. Umożliwia ona sprawdzanie, czy dany przykład przy inkrementacyjnym przetwarzaniu został sklasyfikowany błędnie przez istniejące reguły, analizując pokrywające go reguły i porównując przypisaną przez nie klasę z rzeczywistą. Umożliwia to rozpoczęcie procesu obsługi błędnej klasyfikacji.

Za obsługę błędnej klasyfikacji jest odpowiedzialna funkcja *handle_misclassification*. Proces rozpoczyna się od usunięcia duplikatów w zbiorze reguł za pomocą metody *remove_duplicate_rules*, co zapewnia, że reguły są unikalne. Następnie algorytm identyfikuje reguły z wysokim wskaźnikiem błędów, wykorzystując metodę *error_rate*, która oblicza stosunek błędnie klasyfikowanych przykładów do wszystkich przykładów pokrywanych przez regułę. Reguły, których wskaźnik błędów przekracza próg 10% (wartość do potwierdzenia eksperymentalnie), są oznaczane do usunięcia, a następnie usuwane z listy reguł.

Po usunięciu reguł algorytm identyfikuje przykłady, które nie są pokrywane przez żadną z pozostałych reguł, za pomocą metody *is_covered_by_rules*. Dla każdego z takich przykładów tworzona jest nowa reguła za pomocą metody *create_new_rule*, a zbiór reguł jest aktualizowany. Dzięki temu algorytm dostosowuje się do zmian przy obsługiwaniu błędnej klasyfikacji.

```python
def handle_misclassification(self):
    self.rules = self.remove_duplicate_rules(self.rules)  # Usuń duplikaty na początk
    rules_to_remove = []
    # Identyfikacja reguł z wysokim wskaźnikiem błędów
    for rule in self.rules:
        if self.error_rate(rule) > 0.1:  # Usuwanie reguły o błędach powyżej 10%
            rules_to_remove.append(rule)
    # Usunięcie reguł po zakończeniu iteracji
    for rule in rules_to_remove:
```

```
        if rule in self.rules:  # Sprawdzenie, czy reguła wciąż istnieje w liście
            self.rules.remove(rule)
    # Obsługa przykładów, które nie są pokrywane przez żadną regułę
        uncovered_examples = [x for x in self.processed_examples if not self.is_covered_by_rules(x)]
    for ex in uncovered_examples:
        new_rule = self.create_new_rule(ex)
        self.update_rule_set(new_rule)
```

3.7. Przycinanie reguł (*prune_rules*)

Metoda *prune_rules* jest odpowiedzialna za optymalizację zbioru reguł, redukując nadmiarowe lub mniej istotne reguły i jednocześnie zapewniając, że zbiór pozostaje skuteczny w klasyfikacji. Proces ten składa się z dwóch głównych etapów: usuwania duplikatów reguł i ograniczania liczby reguł dla każdej klasy. Wywołuje się w momencie aktualizacji zbioru reguł przez nową regułę. Przygotowana metoda ogranicza ryzyko nadmiernego dopasowania i utrzymuje jak najbardziej wydajny i skuteczny w klasyfikacji nowych przykładów zbiór reguł.

Pierwszym krokiem jest eliminacja duplikatów za pomocą metody *remove_duplicate_rules*, która identyfikuje i usuwa reguły o takich samych warunkach i przypisanej klasie. Sprawia to, że zbiór reguł jest ma najmniejszy możliwy rozmiar, co redukuje redundancję i poprawia wydajność algorytmu.

Następnie reguły są grupowane według klasy, co pozwala na niezależną optymalizację dla każdej grupy. Dla każdej klasy określana jest maksymalna liczba reguł, która może zostać zachowana. Limit ten jest ustalany jako co najmniej 3 reguły dla klasy, ale nie więcej niż 1/4 liczby przykładów przypisanych do danej klasy w przetworzonych danych (wartość do ewentualnej korekcji w eksperymentach). Zastosowany limit zapewnia proporcjonalność między liczbą reguł a liczbą przykładów danej klasy. Jeśli liczba reguł dla danej klasy przekracza wyznaczony limit, algorytm wybiera najlepsze reguły za pomocą metody *select_best_rules*, która ocenia reguły na podstawie ich jakości (zgodnie z opisem w sekcji 3.8). W przeciwnym razie wszystkie reguły dla danej klasy są zachowywane. Po przetworzeniu wszystkich klas zbiór reguł jest aktualizowany, zawierając jedynie te reguły, które przeszły proces przycinania.

```python
def prune_rules(self):
    # Usunięcie duplikatów reguł na podstawie ich warunków i klas
    self.rules = self.remove_duplicate_rules(self.rules)
    # Grupowanie reguł według klasy
    class_rules = {}
    for rule in self.rules:
        class_rules.setdefault(rule['class'], []).append(rule)
    pruned_rules = []
    # Iteracja po grupach reguł według klas
    for cls, rules in class_rules.items():
        # Maksymalna liczba reguł dla klasy: co najmniej 3, maksymalnie 1/4 liczby przykładów tej klasy - do badań
        max_rules = max(3, len([x for x in self.processed_examples if x['class'] == cls]) // 4)

        # Jeśli liczba reguł dla klasy przekracza limit, wybierz najlepsze reguły
        if len(rules) > max_rules:
            pruned_rules.extend(self.select_best_rules(rules, n=max_rules))
```

```
        else:
            pruned_rules.extend(rules)
    self.rules = pruned_rules
```

3.8. Ocena jakości reguł i kompleksów (*quality*, *complex_quality*)

Do oceny jakości reguł i kompleksów przygotowany metody *quality* i *complex_quality*. Są one kluczowe w procesach selekcji reguł w przycinaniu, klasyfikacji oraz w specjalizacji kompleksów, co umożliwia budowanie optymalnego zbioru reguł.

Metoda *quality* oblicza jakość reguły, sumując dwie wartości: liczbę przykładów poprawnie klasyfikowanych przez regułę oraz liczbę przykładów, które nie są przez nią pokrywane, ale należą do innych klas. W ten sposób uwzględniono zarówno zdolność reguły do poprawnej klasyfikacji przykładów, jak i unikania błędnych pokryć. W takiej postaci ocena jakości jest wykorzystywana przy klasyfikacji.

Metoda *complex_quality* rozwija tę koncepcję, dodając dodatkowy element — liczbę przykładów, które są pokrywane wyłącznie przez dany kompleks. Po obliczeniu podstawowej jakości kompleksu przy użyciu metody *quality*, metoda identyfikuje przykłady, które pokrywa wyłącznie dany kompleks. Ta liczba przykładów jest następnie dodawana do podstawowej jakości kompleksu, co pozwala ocenić jak dużą wartość wnosi reguła (kompleks) do zbioru reguł. W tej postaci ocena jakości reguł została zastosowana przy specjalizacji i przycinaniu zbioru reguł.

3.9. Klasyfikacja (*classify*)

Do klasyfikacji przygotowano metodę *classify*, która odpowiada za przypisanie klasy do zadanego przykładu na podstawie istniejącego zbioru reguł. Najpierw identyfikowane są reguły, które pokrywają dany przykład. Jeśli żadna reguła go nie pokrywa, wybierana jest klasa najlepszej reguły spośród wszystkich dostępnych. W przypadku, gdy pokrywające reguły wskazują tylko jedną klasę, jest ona bezpośrednio zwracana. Jeśli reguły sugerują więcej niż jedną klasę, wybierana jest klasa najlepszej reguły spośród pokrywających.

3.10. Podsumowanie opisu implementacji

Przygotowana implementacja inkrementacyjnej indukcji reguł spełnia w większości założenia wstępne, które nie wymagały większej modyfikacji. Algorytm obsługuje uczenie się na bazie przychodzących inkrementacyjnie przykładów oraz klasyfikowanie na podstawie atrybutów ciągłych i dyskretnych.

W trakcie eksperymentów dokonana będzie ewaluacja wartości niektórych zmiennych używanych w algorytmie. Oceniony zostanie dobór maksymalnej liczby reguł w procesie przycinania, dopuszczalny procent przykładów niepoprawnie klasyfikowanych przez reguły w zbiorze reguł, przetworzonych w trakcie procesu uczenia się oraz procent przykładów negatywnych, jaki może pokrywać reguła przed zakończeniem procesu specjalizacji.

4. Testy algorytmu

4.1. Testy na zbiorach

4.1.1. Test na zbiorze Breast Cancer Wisconsin

W pierwszym teście zastosowano zbiór dotyczący raka piersi. Podzieliliśmy go na cztery różne podzbiory:

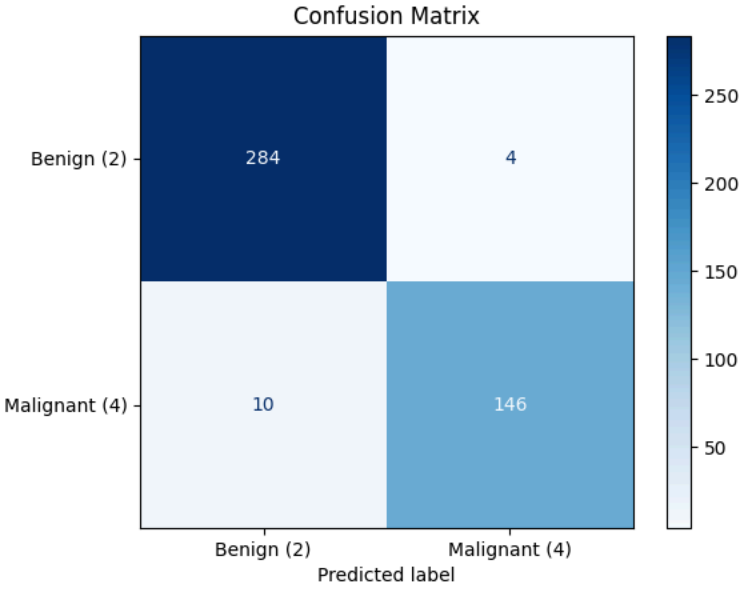
- Zbiór treningowy - zbiór danych do inicjalizacji wsadowej
- Zbiór Inkrementacji 1 - zbiór danych pierwszej inkrementacji algorytmu
- Zbiór Inkrementacji 2 - zbiór danych drugiej inkrementacji algorytmu

- Zbiór testowy - zbiór danych testowych służących do określania dokładności algorytmu po każdym etapie

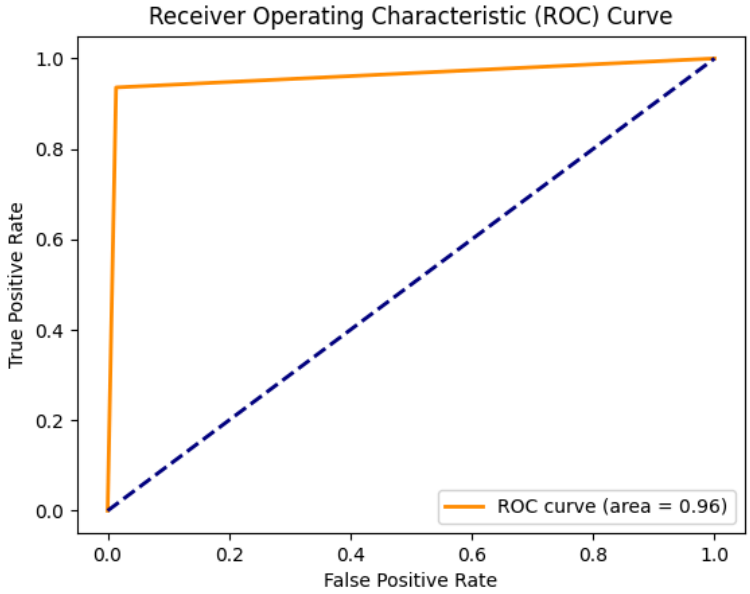
Dane zostały przemieszane w sposób losowy oraz podzielone za pomocą zmiennych `batch_ratio`, `incremental_ratio` w następujący sposób:

- Rozmiar zbióru treningowego = Rozmiar całego zbioru $*$ batch_ratio
- Rozmiar zbioru Inkrementacji 1,2 = Rozmiar całego zbioru $* \frac{\text{incremental\_ratio}}{2}$
- Rozmiar zbioru testowego = Reszta danych

Do podzielenia danych użyliśmy następujących wartości `batch_ratio = 0.05`, `incremental_ratio = 0.30` i przeprowadziliśmy inicjalizację wsadową oraz dwie inkrmentacje za pomocą otrzymanych zbiorów. Wynikiem testu była macierz konfuzji i krzywa roc po każdej z inkrementacji oraz procentowe wyniki dokładności.

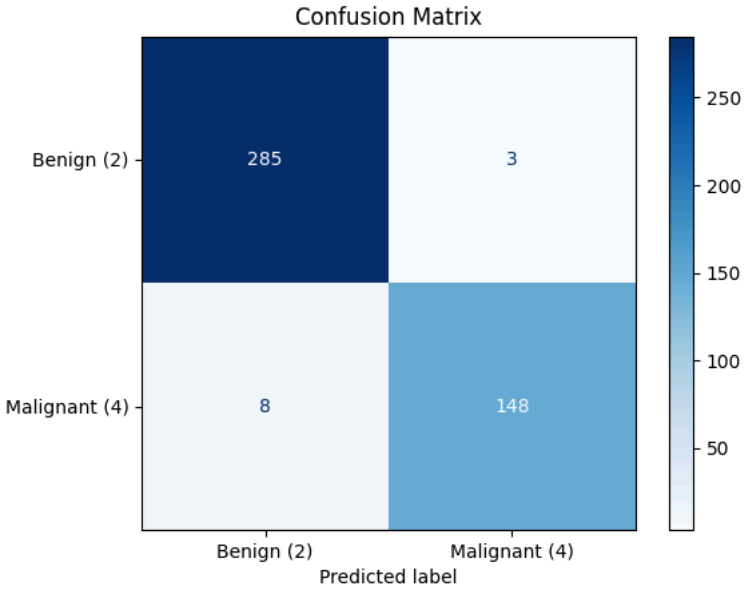


Rys. 2: Macierz konfuzji po pierwszej inkrementacji, `batch_ratio = 0.05, incremental_ratio = 0.30`

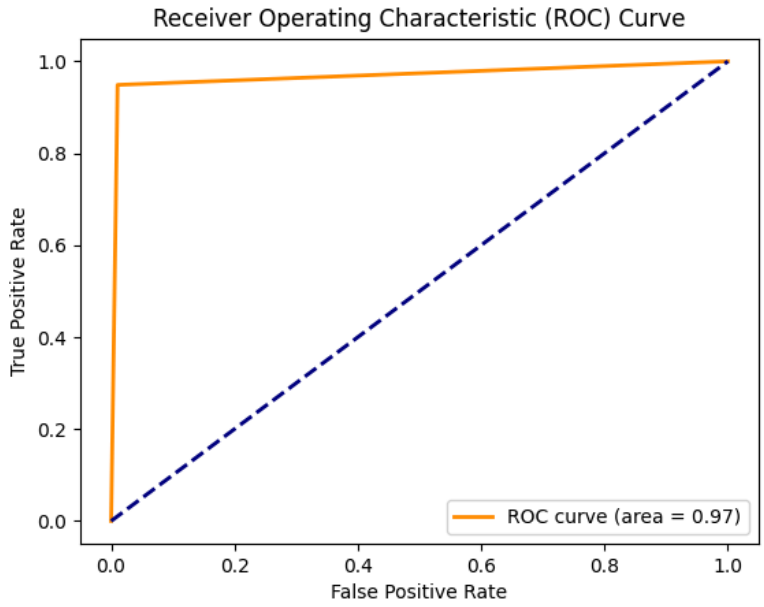


Rys. 3: Krzywa ROC po pierwszej inkrementacji, `batch_ratio = 0.05, incremental_ratio = 0.30`

Na podstawie powstałej macierzy konfuzji oraz krzywej ROC można powiedzieć, że wyniki po pierwszej inkrementacji są bardzo zadowalające.



Rys. 4: Macierz konfuzji po drugiej inkrementacji, `batch_ratio = 0.05, incremental_ratio = 0.30`



Rys. 5: Krzywa ROC po drugiej inkrementacji, `batch_ratio = 0.05, incremental_ratio = 0.30`

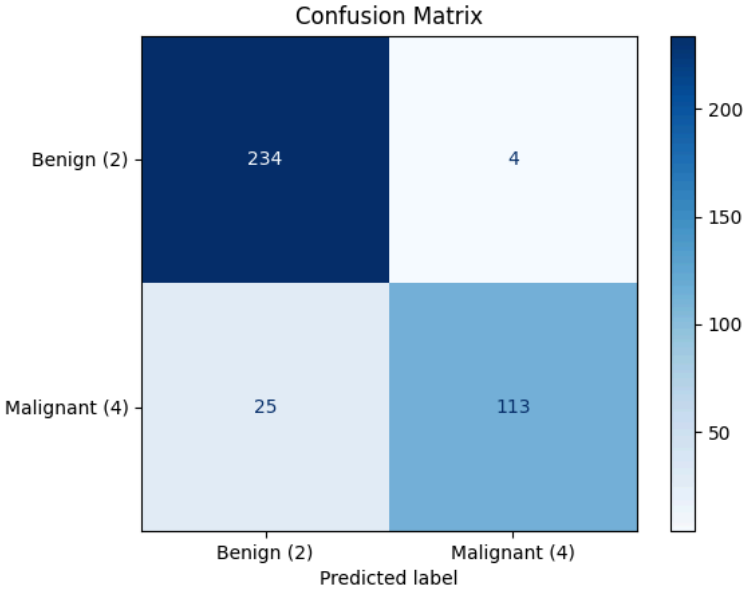
Po drugiej inkrementacji dokładność naszego modelu zwiększyła się niewiele aczkolwiek był do zauważalny wzrost dla krzywej ROC.

```
Wyniki po inicjalizacji wsadowej:
Dokładność: 89.63963963963964%
Błędna klasyfikacja: 10.36036036036036%
Poprawne klasyfikacje: 398
Błędne klasyfikacje: 46
Całkowita liczba przykładów: 444


Wyniki po pierwszej inkrementaci:
Dokładność: 96.84684684684684%
Błędna klasyfikacja: 3.153153153153153%
Poprawne klasyfikacje: 430
Błędne klasyfikacje: 14
Całkowita liczba przykładów: 444


Wyniki po drugiej inkrementaci:
Dokładność: 97.52252252252252%
Błędna klasyfikacja: 2.4774774774774775%
Poprawne klasyfikacje: 433
Błędne klasyfikacje: 11
Całkowita liczba przykładów: 444
```

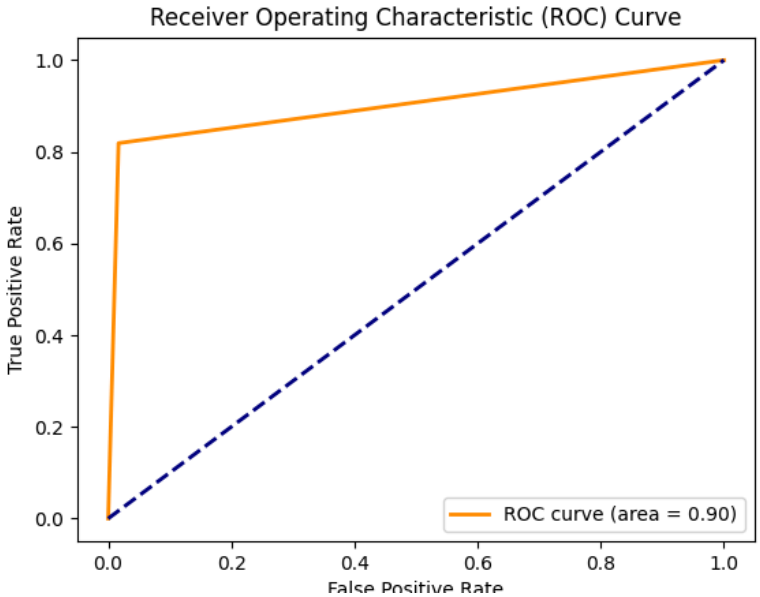
Rys. 6: Wyniki dla każdego etapu, `batch_ratio = 0.05, incremental_ratio = 0.30`

Jak możemy zauważyć wynik po inicjalizacji wsadowej był już bliski 90% po kolejnych dwóch inkrementacjach poprawił się zauważalnie i na koniec miał już dokładność na poziomie 97.5% co uznajemy za znakomity wynik

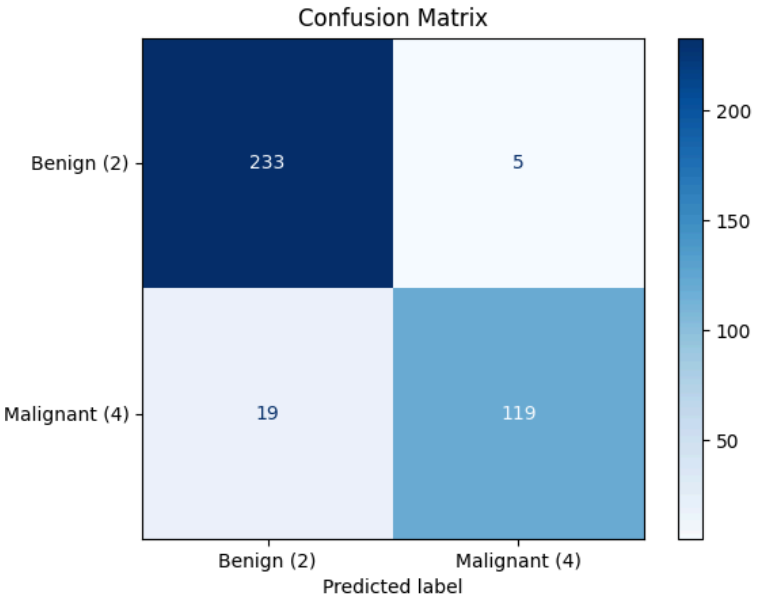
Następnie postanowiliśmy zwiększyć rozmiar zbioru testowego i inkrementalnego kosztem rozmiaru zbioru testowego. Wyniki dla wartości `batch_ratio = 0.1, incremental_ratio = 0.4` są następujące:



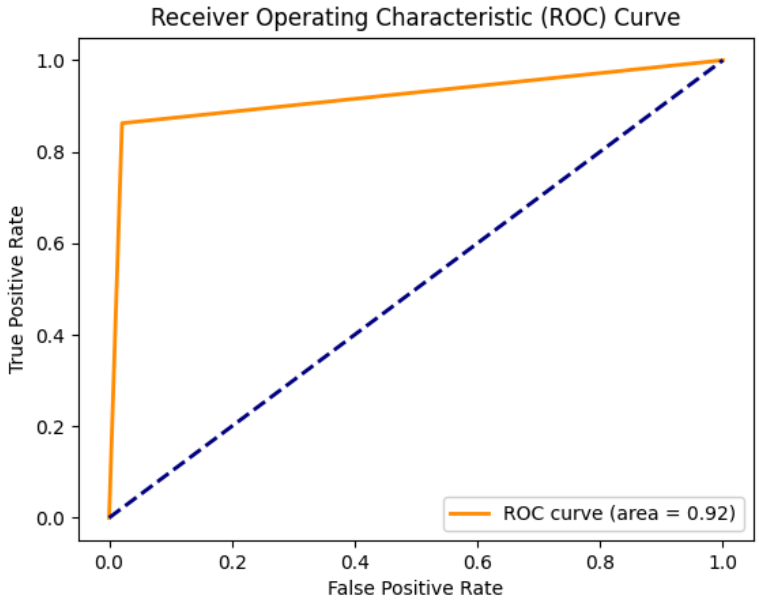
Rys. 7: Macierz konfuzji po pierwszej inkrementacji `batch_ratio = 0.1, incremental_ratio = 0.4`



Rys. 8: Krzywa ROC po pierwszej inkrementacji, `batch_ratio = 0.1, incremental_ratio = 0.4`



Rys. 9: Macierz konfuzji po drugiej inkrementacji `batch_ratio = 0.1, incremental_ratio = 0.4`



Rys. 10: Krzywa ROC po drugiej inkrementacji `batch_ratio = 0.1, incremental_ratio = 0.4`

```
Wyniki po inicjalizacji wsadowej:
Dokładność: 90.69148936170212%
Błędna klasyfikacja: 9.308510638297872%
Poprawne klasyfikacje: 341
Błędne klasyfikacje: 35
Całkowita liczba przykładów: 376


Wyniki po pierwszej inkrementaci:
Dokładność: 92.2872340425532%
Błędna klasyfikacja: 7.712765957446808%
Poprawne klasyfikacje: 347
Błędne klasyfikacje: 29
Całkowita liczba przykładów: 376


Wyniki po drugiej inkrementaci:
Dokładność: 93.61702127659575%
Błędna klasyfikacja: 6.382978723404255%
Poprawne klasyfikacje: 352
Błędne klasyfikacje: 24
Całkowita liczba przykładów: 376
```

Rys. 11: Wyniki dla każdego z poszczególnych etapów `batch_ratio = 0.1, incremental_ratio = 0.4`

Jak możemy zauważyć przez to, że wzrósł zbiór treningowy dokładność po inicjalizacji wsadowej jest powyżej 90%. Natomiast ciekawą rzeczą jest brak tak dużego wzrostu dokładności po pierwszej i drugiej inkrementacji. Podejrzewamy, że może mieć to coś wspólnego z ilością reguł wygenerowanych przez inicjalizacje wsadową.

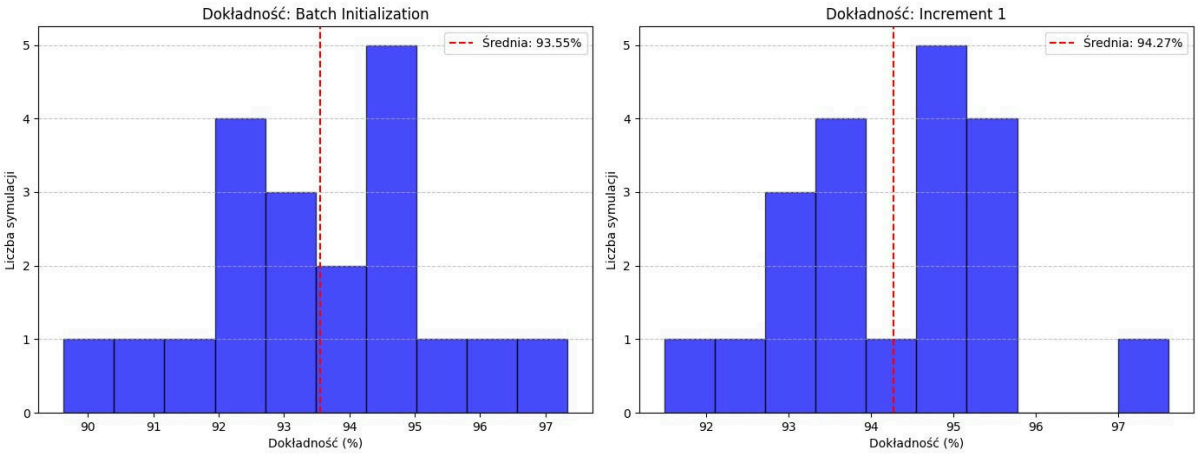
Ze względu na złożoność obliczeniową naszego algorytmu i rozmiar tego zbioru danych zdecydowaliśmy się na przeprowadzenie symulacji tylko jednej dwudziestokrotnej symulacji zawierającą tylko jedną iterację. W innym przypadku praca algorytmu jest znacznie wydłużona. Przez to, że wykorzystujemy tutaj tylko jedną inkrementacje podzieliliśmy zbiór tylko na trzy podzbiory:

- Zbiór trenujący - zbiór używany do inicjalizacji wsadowej
- Zbiór inkrementujący - zbiór używany do inkrementacji
- Zbiór testowy - zbiór używany do testów

W każdej symulacji dane są najpierw losowo mieszane i dzielone w następujący sposób za pomocą zmiennej `train_ratio=0.15`:

- Rozmiar zbióru treningowego = Rozmiar całego zbioru ∗ train_ratio
- Rozmiar zbioru Inkrementacji 1,2 = Rozmiar całego zbioru ∗ train_ratio ∗ 3
- Rozmiar zbioru testowego = Reszta danych

Wyniki po przeprowadzeniu dwudziestu symulacji:



Rys. 12: Średnia dokładność algorytmu z jedną iteracją

Średnia wartość dokładności po jednej iteracji wynosi około 94.27%

Rys. 13: Wyniki innych algorytmów podanych przez stronie [1]

Przez to, że znalezione przez nas wcześniej implementacje algorytmów zwykłej indukcji reguł były wadliwe nie mogliśmy porównać jakie one osiągają wyniki na takim samym zbiorze danych. Wspomniana wcześniej implementacja AQ dostępna na Githubie, nie udawała się uruchomić. Brakowało do niej także dokumentacji, wyjaśniającej jej wykorzystanie. Implementacja CN2 z biblioteki Orange także odmawiała współpracy, ze względu na wysoką złożoność wykorzystania algorytmu oraz skromną dokumentację implementacji CN2. Z tego względu nasz algorytm porównywaliśmy do wyników innych algorytmów, dostępnych w dokumentacjach zbiorów danych.

4.1.2. Test na zbiorze „Irys”

W tym teście podzieliliśmy zbiór danych Irys na pięć różnych podzbiorów:

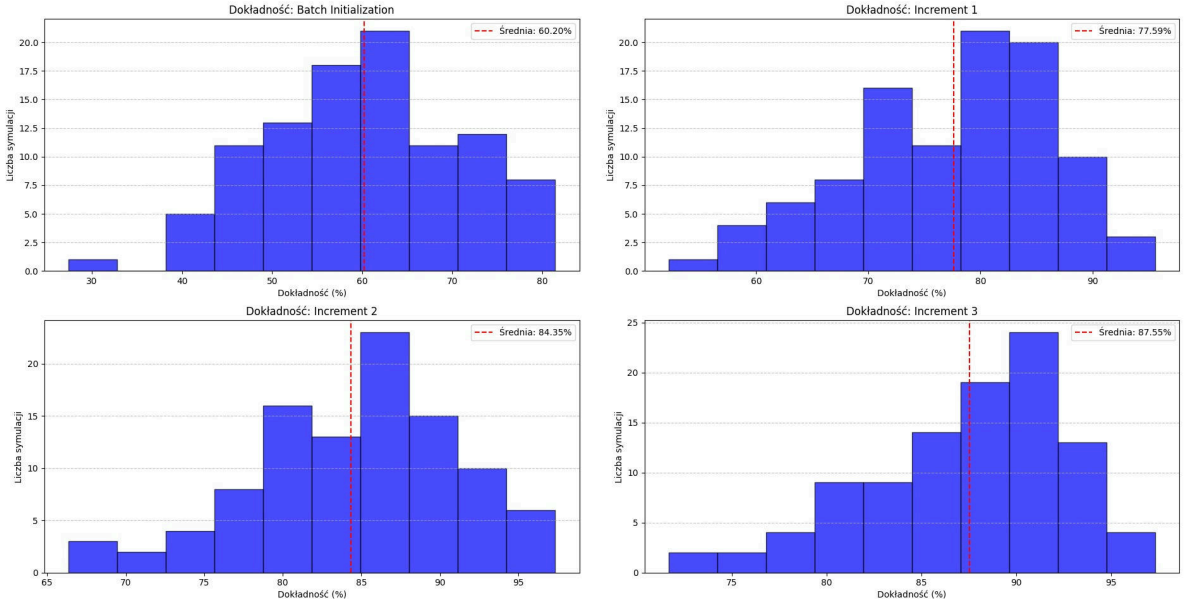
- Zbiór treningowy - zbiór danych do inicjalizacji wsadowej
- Zbiór Inkrementacji 1 - zbiór danych pierwszej inkrementacji algorytmu
- Zbiór Inkrementacji 2 - zbiór danych drugiej inkrementacji algorytmu

- Zbiór Inkrementacji 3 - zbiór danych trzeciej inkrementacji algorytmu
- Zbiór testowy - zbiór danych testowych służących do określania dokładności algorytmu po każdym etapie

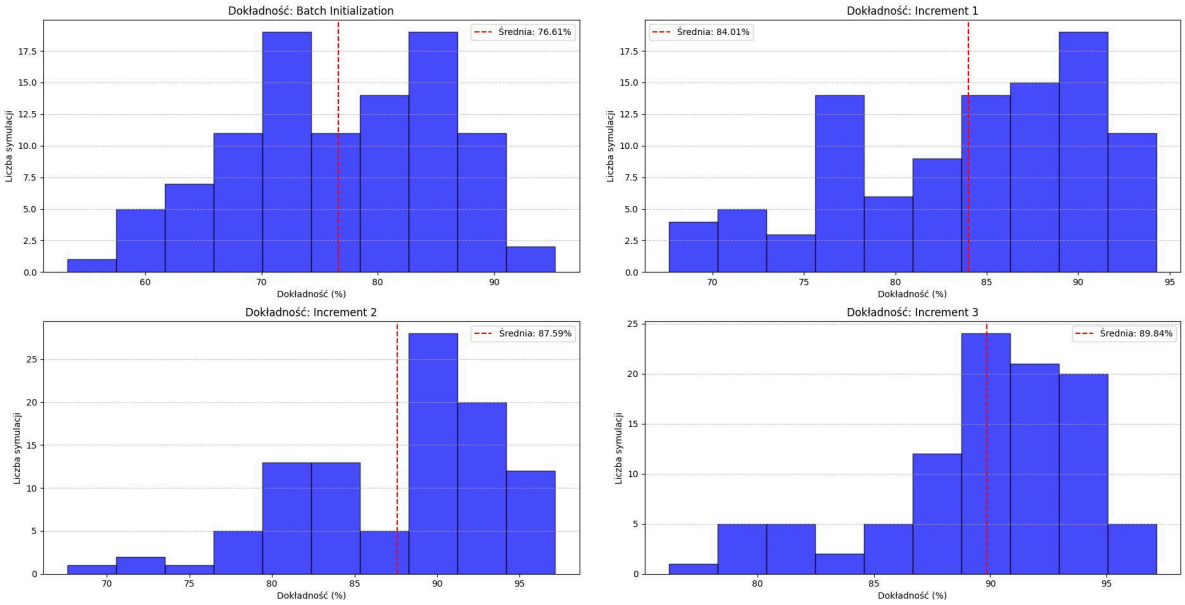
Dane zostały losowo przemieszane oraz zostały podzielone w następujący sposób za pomocą dwóch zmiennych `batch_ratio, incremental_ratio`, które określały rozmiar poszczególnych zbiorów:

- Rozmiar zbióru treningowego = Rozmiar całego zbioru $*$ $\frac{\text{batch\_ratio}}{3}$
- Rozmiar zbioru Inkrementacji 1,2,3 = Rozmiar całego zbioru $*$ $\frac{\text{incremental\_ratio}}{3}$
- Rozmiar zbioru testowego = Reszta danych

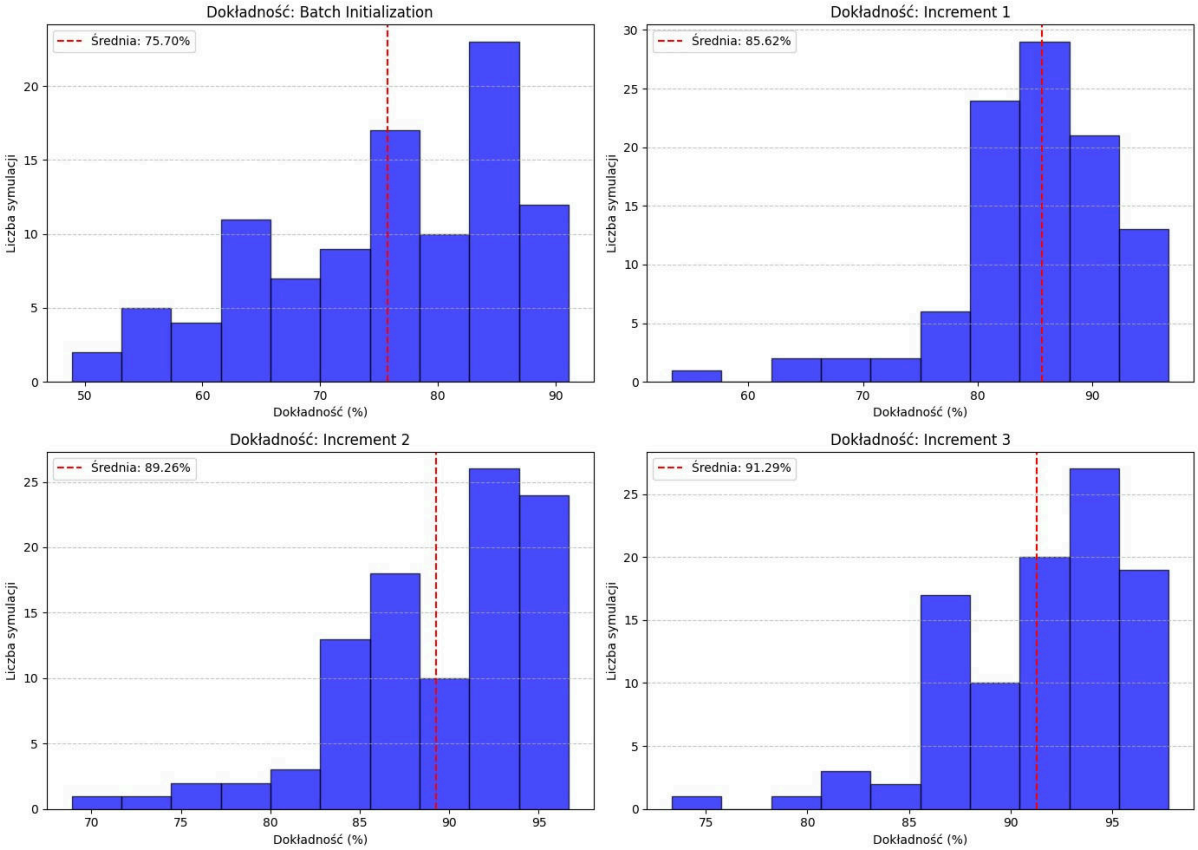
Następnie przeprowadziliśmy kilka stukrotnych symulacji uczenia się naszego algorytmu z różnymi wartościami zmiennych `batch_ratio, incremental_ratio`.



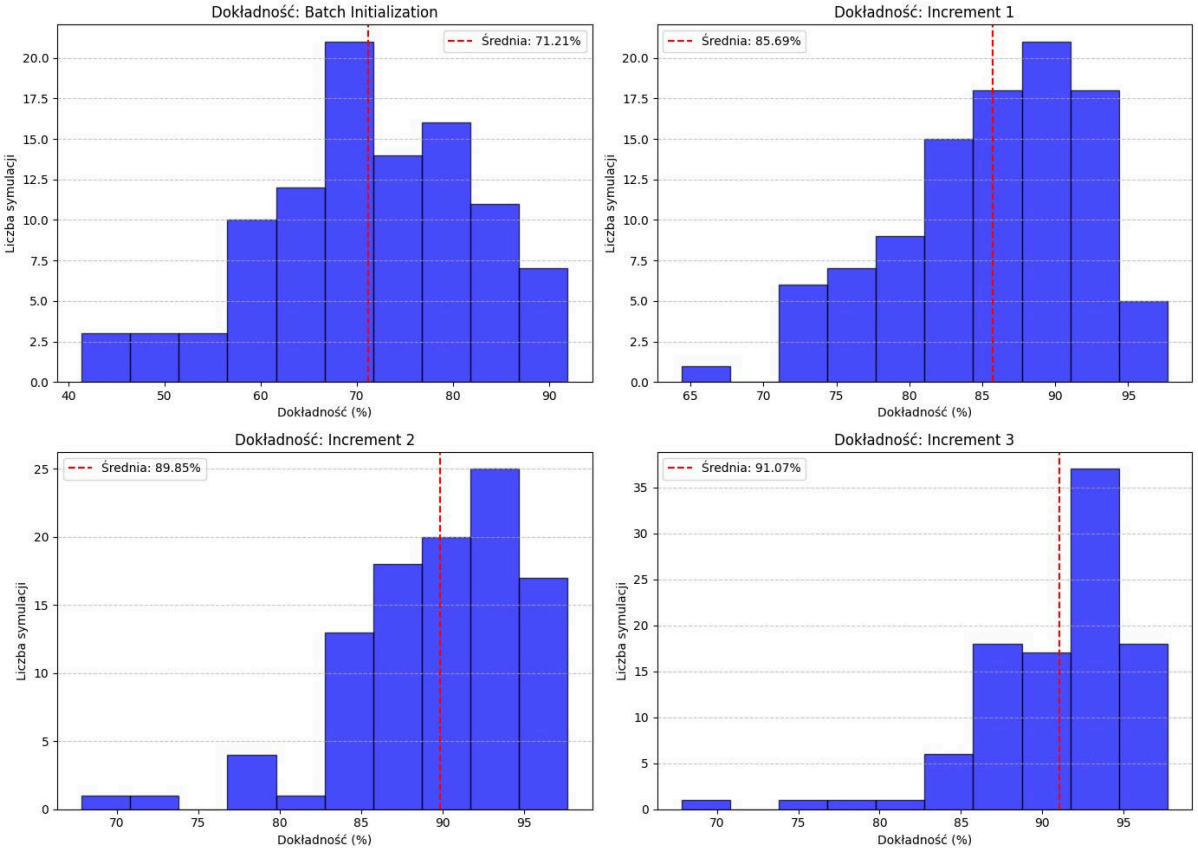
Rys. 14: `batch_ratio = 0.05, incremental_ratio = 0.2`



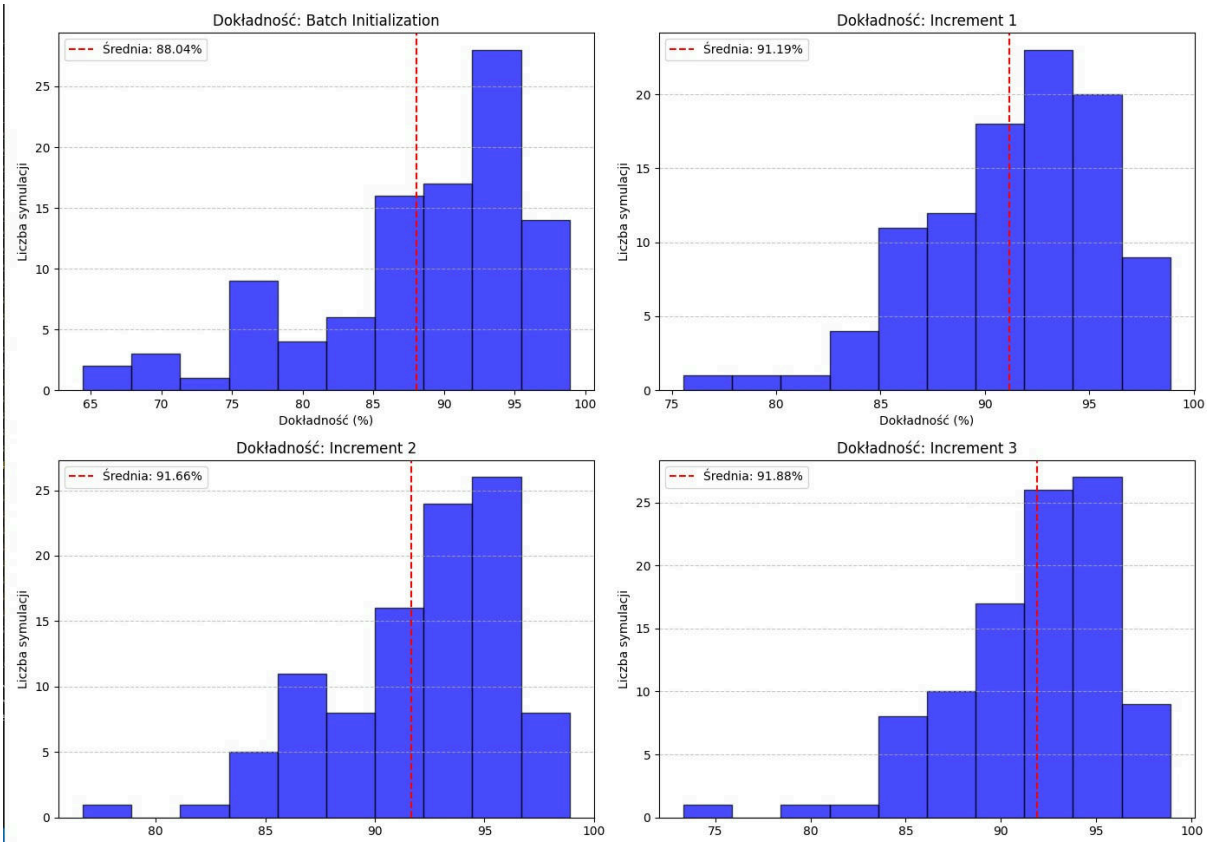
Rys. 15: `batch_ratio = 0.10, incremental_ratio = 0.2`



Rys. 16: `batch_ratio = 0.10, incremental_ratio = 0.3`



Rys. 17: `batch_ratio = 0.08, incremental_ratio = 0.35`



Rys. 18: `batch_ratio = 0.3, incremental_ratio = 0.10`

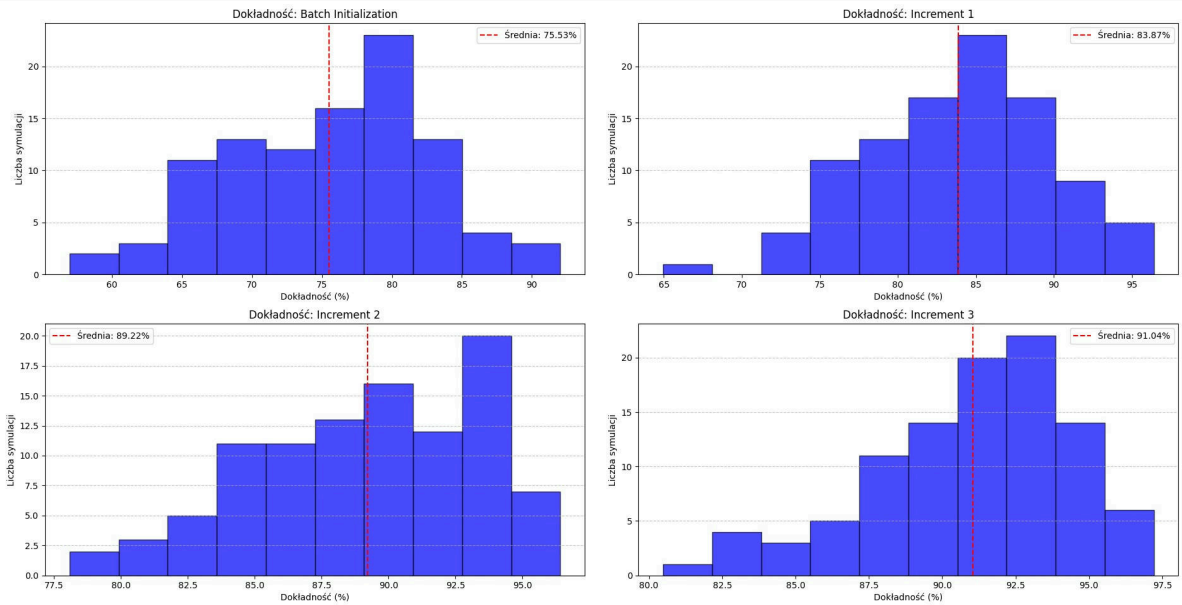
W niemal każdym przypadku możemy przypadku po końcowej iteracji otrzymujemy średnią dokładność na poziomie około 90%.

Kolejną rzeczą, którą można zauważyć to jest, że prawie w każdym przypadku histogram dla inicjalizacji wsadowej ma więcej przypadków bardzo słabej dokładności w porównaniu do kolejnych iteracji. Można to wytłumaczyć tym, że przed inicjalizacją wsadową nie wysępują jeszcze żadne reguły klasyfikacji oraz każdym przypadku poza jednym zbiór treningowy, na którym odbywa się inicjalizacji wsadowa jest mniejszy od zbiorów kolejnych inkrementacji reguł algorytmu. Natomiast w przypadku, w którym zbiór treningowy jest większy od zbiorów inkrementacyjnych możemy już zauważyć mniejszą rozbieżność wyników dokładności oraz większą mninimalną dokładność algorytmu.

4.1.3. Test na zbiorze Palmer Penguins

Za pomocą zbioru Palmer Penguins przetestowaliśmy, jak algorytm radzi sobie z trochę bardziej skomplikowanym zbiorem danych, który zawiera więcej atrybutów niż zbiór Irys. W teście podzieliliśmy zbiór Palmer Penguins na pięć różnych podzbiorów. Dane zostały losowo przemieszane oraz zostały podzielone w następujący sposób za pomocą dwóch zmiennych `batch_ratio`, `incremental_ratio`, które określały rozmiar poszczególnych zbiorów:

- Rozmiar zbióru treningowego = Rozmiar całego zbioru $*$ $\frac{\text{batch\_ratio}}{3}$
- Rozmiar zbioru Inkrementacji 1,2,3 = Rozmiar całego zbioru $*$ $\frac{\text{incremental\_ratio}}{3}$
- Rozmiar zbioru testowego = Reszta danych



Rys. 19: `batch_ratio = 0.05, incremental_ratio = 0.20`

Jak widzimy po ostatniej iteracji algorytm osiągnął średnią dokładność równą 91.04%, co oznacza, że poradził sobie lepiej niż w przypadku zbioru Irysów przy takich samych ustawieniach `batch_ratio, incremental_ratio`. Może to jednak wynikać z tego, że zbiór z pingwinami ma dwa razy więcej przykładów, dzięki czemu algorytm mógł wytworzyć lepsze reguły do klasyfikacji pingwinów. Tak samo jak w przypadku Irysów wyniki po inicjalizacji wsadowej są bardziej rozległe w porównaniu do wyników po kolejnych inkrementacjach. Największy przeskok w dokładności miał miejsca pomiędzy inicjalizacją wsadową a wynikami po pierwszej inkrementacji, wynosił on 8.34 punktów procentowych. Pomiędzy pierwszą, a drugą inkrementacją skok był równie zauważalny aczkolwiek mniejszy, wynosił 5.35 punktu procentowego. Przeskok pomiędzy drugą, a trzecią inkrementacją skok był już zauważalnie mniejszy - 1.82 punktu procentowego.

4.1.4. Test na zbiorze jakości wina

Test na zbiorze wina zaczęliśmy od tego że przetestowaliśmy inicjalizację wsadową na zbiorze składający się z dziesięciu procent całego rozmiaru danych. Przez rozmiar danych i ilość klas algorytm okazał się bardzo wolny i niedokładny. Sama inicjalizacja wsadowa wygenerowała 27 reguł. Dla porównania dla zbioru związanego z rakiem piersi inicjalizacja wsadowa zazwyczaj generowała 3 reguły.

```
Wyniki po wsadowym uczeniu:
correct_rate: 2.31344976185076
error_rate: 97.68655023814924
correct: 102
incorrect: 4307
total: 4409
```

Rys. 20: Wynik testu po inicjalizacji wsadowej

4.2. Opis zbiorów danych

4.2.1. Zbiór Breast Cancer Wisconsin (Original) [1]

Zebrany przez Dr Wolberg'a zbiór danych dotyczących klasyfikacji raka piersi jako złośliwego bądź łagodnego na podstawie następujących atrybutów:

- Clump Thickness - Grubość skupisk komórek. Ocena wielkości i grubości skupisk komórkowych. Wartości dyskretne w zakresie od 1 do 10.

- Uniformity of Cell Size - Jednorodność rozmiaru komórek. Ocena, czy komórki mają jednolitą wielkość. Wartości dyskretne w zakresie od 1 do 10.
- Uniformity of Cell Shape - Jednorodność kształtu komórek. Analiza podobieństwa kształtów komórek. Wartości dyskretne w zakresie od 1 do 10.
- Marginal Adhesion - Przyleganie komórek na brzegach. Sprawdzanie, jak mocno komórki przylegają do siebie. Wartości dyskretne w zakresie od 1 do 10.
- Single Epithelial Cell Size - Rozmiar pojedynczych komórek nabłonkowych. Pomiar wielkości indywidualnych komórek nabłonkowych. Wartości dyskretne w zakresie od 1 do 10.
- Bare Nuclei - Nagie jądra komórkowe. Ocena obecności jąder niezawierających cytoplazmy. Wartości dyskretne w zakresie od 1 do 10.
- Bland Chromatin - Chromatyna o łagodnym wyglądzie. Sprawdzanie struktury chromatyny. Wartości dyskretne w zakresie od 1 do 10.
- Normal Nucleoli - Obecność jąderka. Ocena obecności i wielkości jąderka w komórkach. Wartości dyskretne w zakresie od 1 do 10.
- Mitoses - Podziały komórkowe. Liczba podziałów komórkowych. Wartości dyskretne w zakresie od 1 do 10.

Zawiera 699 przykładów.

4.2.2. Zbiór danych „Irys" [2]

Zbiór danych klasyfikujący Irysy na trzy gatunki: „Iris Setosa", „Iris Versicolour", „Iris Virginica". Każdy z kwiatków opisany jest za pomocą następujących 4 cech.

- Sepal length - Długość działek kielicha kwiatu irysa (zielonej struktury przypominającej liść, która otacza pąk kwiatowy). Ciągła wartość podana w centymetrach. W zakresie od 4 do 8 z przeskokiem co 0.1
- Sepal width - Szerokość działek kielicha kwiatu irysa. Ciągła wartość podana w centymetrach. W zakresie od 2 do 4.5 z przeskokiem co 0.1
- Petal length - Długość płatków kwiatu irysa (kolorowa struktura kwiatu). Ciągła wartość podana w centymetrach. W zakresie od 1 do 7 z przeskokiem co 0.1
- Petal width - Szerokość płatków kwiatu irysa. Ciągła wartość podana w centymetrach. W zakresie od 0.1 do 2.5 z przeskokiem co 0.1

Zawiera 150 przykładów.

4.2.3. Zbiór Palmer Penguins [3]

Zbiór danych klasyfikujący 3 gatunki pingwinów żyjących na wyspach Archipelagu Palmera na Antarktyce. Klasyfikowane gatunki pingwinów to: Adélie, Gentoo, Chinstrap. Każdy z pingwinów opisany jest następującymi cechami:

- Wyspa na której żyje: Torgersen, Biscoe, Dream
- Długość dziobu wyrażona w milimetrach. W zakresie od 32.1 do 59.6 z przeskokiem o 0.1
- Głębokość dzioba pingwina wyrażona w milimetrach. W zakresie od 13.1 do 21.5 z przeskokiem o 0.1
- Długość płetwy wyrażona w milimetrach. W zakresie od 172 do 231 z przeskokiem o 1
- Masa ciała wyrażona w gramach. W zakresie od 2700 do 6300 z przeskokiem o 25
- Płeć: męska, żeńska

Zawiera 344 przykładów.

4.2.4. Zbiór jakości wina [4]

Zbiór klasyfikujący jakość białego wina od 1 do 10 w zależności od cech chemicznych danego wariantu wina.

- Fixed Acidity (Stała kwasowość): To miara kwasowości wina, związana z kwasami organicznymi (takimi jak kwas winowy, cytrynowy, jabłkowy), które są obecne w winie. W zakresie od 3.8 do 14.2 z przeskokiem o 0.1
- Volatile Acidity (Latanie kwasowości): To kwasowość, która jest związana z lotnymi kwasami, głównie kwasem octowym (octem). W zakresie od 0.1 do 1.1 z przeskokiem o 0.1
- Citric Acid (Kwas cytrynowy): Kwas cytrynowy występuje naturalnie w winogronach i ma wpływ na kwasowość oraz smak wina, dodając mu świeżości i lekkości. W zakresie od 0.0 do 1.7 z przeskokiem o 0.1
- Residual Sugar (Cukier resztkowy): To ilość cukru, która pozostała w winie po fermentacji. W zakresie od 0.6 do 65.8 z przeskokiem o 0.1
- Chlorides (Chlorki): Chlorki to obecność soli w winie, które mogą pochodzić z wody używanej podczas produkcji lub z winogron. W zakresie od 0.01 do 0.35 z przeskokiem o 0.1
- Free Sulfur Dioxide (Wolny dwutlenek siarki): Dwutlenek siarki jest często stosowany w winie jako konserwant, aby zapobiec utlenianiu i rozwojowi niepożądanych bakterii i drożdży. W zakresie od 2 do 289 z przeskokiem o 1
- Total Sulfur Dioxide (Całkowity dwutlenek siarki): Całkowita ilość dwutlenku siarki, w tym zarówno wolny dwutlenek siarki, jak i związany z innymi substancjami w winie. W zakresie od 9 do 440 z przeskokiem o 1
- Density (Gęstość): Gęstość wina zależy od zawartości alkoholu i cukru. W zakresie od 0.987 do 1.039 z przeskokiem o 0.001
- pH : pH wina mierzy jego kwasowość. W zakresie od 0.2 do 1.1 z przeskokiem o 0.01
- Sulphates (Siarkany): Siarkany to związki siarki, które występują w winie i również działają jako środki konserwujące. W zakresie od 8 do 14.2 z przeskokiem o 0.01
- Alcohol (Alkohol): Zawartośc procentowa alkoholu w winie. W zakresie od 8 do 14.2 z przeskokiem o 0.01

Zawiera 4898 przykładów.

4.2.5. Zbiór danych dotyczących gry w golfa

Zbiór danych dotyczących gry w golfa zaproponowany w wstępnej dokumentacji okazał się nie użyteczny ze względu na jego bardzo mały rozmiar, dlatego postanowiliśmy go nie używać w testowaniu.

4.3. Raport z przeprowadzonych testów i wnioski

Według naszych obserwacji wnioskujemy, że nasz algorytm najlepiej nadaje się do klasyfikacji danych, które zakładają małą liczbę klas, najlepiej klasy binarne lub dane trzy-klasowe. Widzieliśmy to dobrze na przykładzie zbioru danych dotyczących raka piersi, który miał klasy binarne i jego dokładność na podstawie tych danych osiągała wartości koło 94%, natomiast na danych dotyczących jakości wina gdzie było aż dziesięć klas nasz algorytm kompletnie sobie nie radził.

Na dokładność naszego algorytmu nie wpływa ilość atrybutów. Natomiast przez złożoność obliczeniową związaną z indukcją reguł, czym większa liczba atrybutów tym czas obliczeń rośnie w sposób, że od pewnej ilości nasz algorytm staje się mało użyteczny ze względu na czas jaki zajmuje mu wyindukowanie odpowiednich reguł.

Podsumowując algorytm wydaje się być użyteczny dla zbiorów do 1000 przykładów, zawierających niewielką ilość klas (max 4). W takich zbiorach jest on w stanie zawsze osiągnąć średnią dokładność na poziomie 85% przy 20% zbiorze trenującym. Inkrementacyjność przyjmowania nowych danych wydaje się działać poprawnie. Przy ich przyjmowaniu obserwujemy przebudowę zbioru reguł i poprawę wyników klasyfikacji.

5. Opis wykorzystanych narzędzi

W naszym projekcie używaliśmy następujących bibliotek:

- Matplotlib używaliśmy do rysowania histogramów
- Scikit-learn była używana do generowania macierzy konfuzji oraz do krzywej ROC
- Csv - używana była do wczytywania danych z pliku
- Random - była używana do losowego mieszania zbiorów danych

6. Podsumowanie i wnioski

Reasumując projekt udało nam się uzyskać algorytm inkrementacyjnej indukcji reguł, który bardzo dobrze sobie radzi z generowaniem modeli z danych, które zawierają mała liczbę klas. Niestety przez złożoność obliczeniową indukcji reguł ma swoje ograniczenia, takie jak czas indukowania reguł lub niską dokładność dla zestawów danych, które zawierają dużą liczbę klas.

Z projektu wyciągnęliśmy dużo wniosków dotyczących praktycznej implementacji algorytmów maszynowego uczenia się. Mogliśmy głębiej poznać istotę algorytmów polegających na indukowaniu reguł. Ważnym aspektem projektu było również zapoznanie się z metodami określania algorytmu takie jak macierz konfuzji lub badanie dokładności za pomocą F1-score.

Bibliografia

[1] Wolberg WIlliam, „Breast Cancer Wisconsin (Original)”. [Online]. Dostępne na: https://doi.org/10.24432/C5HP4Z

[2] Fisher R, „Iris [Dataset]”. [Online]. Dostępne na: https://doi.org/10.24432/C56C76.

[3] Allison Horst, „Palmer Penguins”. [Online]. Dostępne na: https://archive.ics.uci.edu/dataset/690/palmer+penguins-3

[4] Cortez, „Wine Quality”. [Online]. Dostępne na: https://doi.org/10.24432/C56S3T

[5] Patrick Canny, „Implementation of the AQ (Max Star) Data Mining Algorithm”. [Online]. Dostępne na: https://github.com/patrickcanny/AQ

[6] Paweł Cichosz, *Systemy uczące się*. 2000.

[7] Paweł Cichosz, „Uczenie Maszynowe - Prezentacje wykładowe”. 2024.

[8] Demsar J, „Orange: Data Mining Toolbox in Python”. [Online]. Dostępne na: https://orangedatamining.com/

[9] Aditya Rahman, „Golf weather dataset”. [Online]. Dostępne na: https://gist.github.com/kudaliar032/b8cf65d84b73903257ed603f6c1a2508