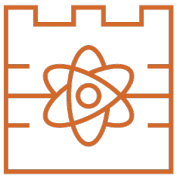
POLITECHNIKA KRAKOWSKA
im. Tadeusza Kościuszki
Wydział Inżynierii Materiałowej i Fizyki
Katedra Fizyki



Kierunek studiów: Fizyka Techniczna
Specjalność: Modelowanie Komputerowe

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

Marek Matejko

135388

Identyfikacja pulsarów przy użyciu metod uczenia maszynowego

Pulsars identification using machine learning methods

Promotor pracy dyplomowej:

dr Radosław Kycia

Recenzent pracy dyplomowej:

prof. dr hab. Włodzimierz Wójcik

Kraków, rok akad. 2022/23

Spis treści

Abstrakt

Celem pracy było szukanie optymalnego algorytmu uczenia nadzorowanego, który jest w stanie sklasyfikować prawdziwego pulsara wśród zbioru danych HTRU2. Do wyznaczenia najlepszego modelu wykorzystano, środowisko programistyczne Jupyter Notebook, język programowania Python oraz jego biblioteki, Scikit-learn, Pandas, Matplotlib, Seaborn. Opisana została metryka służąca do ewaluacji naszych modeli. Przeprowadzona została walidacja krzyżowa, która miała na celu uzyskać wyniki algorytmów. Najlepsze wyniki zostały uzyskane przy metodzie agregacji pasków startowych.

Abstract

The goal of the thesis was to search for the optimal supervised learning algorithm that is able to classify a real pulsar among the HTRU2 data set. Jupyter Notebook environment and Python programming language including its libraries Scikti-learm, Pandas, Mathplotlib, Seaborn were used to determine the best model. The metric used to evaluate our models has been described. Cross-validation was carried out to obtain the results of the algorithms. The best results were obtained with the bootstrap aggregation.

Wstęp

0.1 Cel pracy

Celem pracy jest przegląd metod uczenia maszynowego w celu znalezienia optymalnego algorytmu, który w danych HTRU2 potrafi znaleźć prawdziwego pulsara wśród sygnałów potencjalnych kandydatów.

0.2 Zakres pracy

Zakres pracy obejmuje część teoretyczną, w której przedstawione zostaną pulsary, uczenie maszynowe oraz algorytmy drzewa decyzyjnego, agregacji pasków startowych, lasu losowego, SVM oraz K-najbliższych sąsiadów wraz z metodami ich walidacji. Przestawione zostanie środowisko programistyczne Jupyter Notebook oraz język Python wraz z jego bibliotekami Scikit-learn, Pandas, Matplotlib, Seaborn. Opisany zostanie zbiór danych HTRU2 oraz metryka potrzebna do zmierzenia jakości naszych modeli.

W części praktycznej zostanie przedstawiony sposób wyboru najlepszego algorytmu znajdującego prawdziwe pulsary oraz wyniki i podsumowanie pracy.

0.3 Metodyka pracy

Analiza zbioru danych HTRU2 oraz tworzenie modeli algorytmów drzewa decyzyjnego, agregacji pasków startowych, lasu losowego, SVM oraz K-najbliższych sąsiadów. Wyniki pracy zostaną przygotowane przy pomocy środowiska programistycznego Jupyter Notebook, a także języka Python oraz jego bibliotek.

Część I

Część Teoretyczna

Rozdział 1

Pulsary

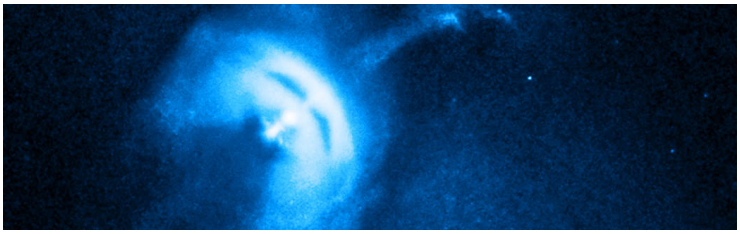
1.1 Definicja pulsarów

Pulsary są to szybko rotujące gwiazdy neutronowe, wysyłające wiązkę promieniowania elektromagnetycznego w regularnych odstępach czasu. Wiązka taka nazywana jest dżetem i jeśli jest zwrócona w stronę Ziemi, to jesteśmy w stanie zauważyć gwiazdę neutronową. Pulsar powstaję w zjawisku zwanym wybuchem supernowej. Eksplozja ta kończy żywot masywnej gwiazdy i istnieją dwa takie przypadki.

Pierwszy przypadek następuje, gdy w masywnej gwieździe przestają zachodzić procesy termojądrowe wywołane brakiem paliwa do prowadzenia syntezy jądrowej. Ciśnienie zewnętrze, które rozpychało gwiazdę, pod wpływem tych procesów spada, oznacza to, że jądro gwiazdy będzie się kurczyć w wyniku sił grawitacji. Gdy jądro gwiazdy osiągnie punkt krytyczny, czyli moment, w którym nie jest w stanie się bardziej skurczyć, nastąpi wybuch supernowej.

Drugi przypadek odnosi się do białego karła, czyli zapadniętej pozostałości po gwieździe, która miała masę równą lub mniejszą od około 8 mas słońca. Obiekt ten może pobierać materie z sąsiadującej gwiazdy do czasu uzyskania maksymalnej masy, jaką może posiadać. Po przekroczeniu tego punktu powstanie eksplozja supernowej.

Podczas wybuchu supernowej jądro gwiazdy ulega kompresji. Zewnętrzne warstwy gwiazdy zapadają się, po czym są rozrzucane w przestrzeń kosmiczną w potężniej eksplozji, po której zostaje mgławica planetarna. Skompresowane jądro, które pozostaje po takim wybuchu, nazywamy gwiazdą neutronową. Więcej informacji możemy znaleźć w książce [1].



Rysunek 1.1: Dżet o długości 0,7 roku świetlnego z pulsara Vela. Źródło [2].

1.2 Właściwości pulsarów

Średnica pulsara zazwyczaj jest niewielka i wynosi około 20 km, jednak jego masa może wynosić nawet 1,5 masy słońca. Dzieje się tak, ponieważ materia w gwieździe neutronowej jest gęsto upakowana przez wpływ grawitacji gwiazdy. Pulsary bardzo szybko rotują, ponieważ nagła zmiana promienia gwiazdy przy jednoczesnym zachowaniu masy spowoduje, że prędkość obrotu wzrośnie z powodu działania zachowania momentu pędu. Najszybciej rotująca gwiazda

obraca się raz na 1,4 milisekundy, czyli 716 razy na sekundę. Natomiast najwolniejszy pulsar, który znajduje się w gwiazdozbiorze Kasjopei, rotuje raz na 23,5 sekundy [3].

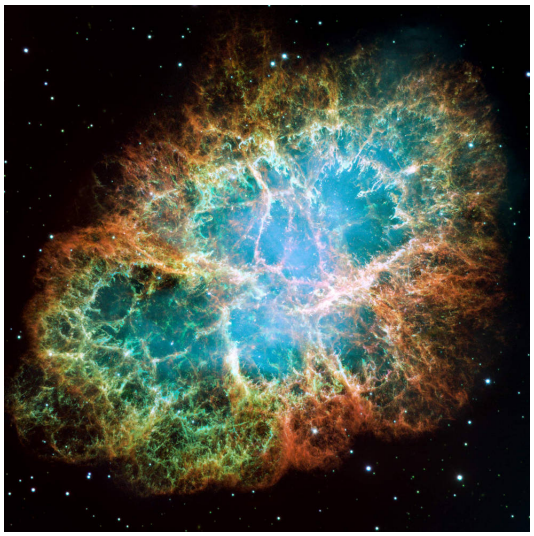
Pulsary opisane w naszych danych mają dwa parametry i są to:

- **DM/SNR, czyli Dispersion Measure/Signal-to-Noise Ratio** [4] oznacza Miarę Dyspersji/Stosunek Sygnału do Szumu. Dyspersja, w tym przypadku, odnosi się do fal radiowych emitowanych przez pulsary docierających do Ziemi, po przebyciu dużych odległości w przestrzeni wypełnionej swobodnymi elektronami. Fale o niższej częstotliwości docierają do teleskopu później, niż te o wyższej częstotliwości. Ta różnica czasowa jest związana z gęstością elektronów znajdujących się na drodze fali [5].
- **Integrated profile** możemy przetłumaczyć jako **uśredniony profil pulsara** [5]. Każdy pulsar wytwarza unikalny wzór emisji impulsów. Jeżeli zaobserwujemy dużą ilość takich pulsów, możemy wtedy stworzyć uśredniony profil pulsara. Jednak emisja impulsów pulsara zmienia się nieznacznie w każdym okresie obrotu z racji tego, potrzebujemy uśrednić wiele tysięcy obrotu by, profil był stabilniejszy.

1.3 Historia pulsarów

Pulsary są stosunkowo niedawno odkrytymi obiektami. W 1967 roku Jocelyn Bell zaobserwowała, za pomocą skonstruowanego przez siebie radioteleskopu, że źródło radiosygnałów oznaczone symbolem CP 1919 wysyła regularne impulsy radiowe. Przypuszczano, że te impulsy są sztuczne z powodu ich regularności, jednak kolejne znalezione takie obiekty w 1968 roku oznaczały, że wcale tak nie jest. Ostatecznie źródło radiosygnałów oznaczone symbolem CP 1919 pochodziło od pulsara, który został nazwany PSR B1919+21 i stał się pierwszym znalezionym pulsarem.

Jednak pomimo tego, że pulsary zostały odkryte w XX wieku ludzkość, mogła zauważyć powstanie gwiazdy neutronowej w świetle widzialnym, a konkretnie wybuch supernowej SN 1054 w 1054 roku. Wybuch ten został wspomniany w ówczesnej astronomii chińskiej, a także przez świat islamski. Mgławica, która powstała po tym wybuchu nosi dzisiaj nazwę mgławicy kraba i w jej centrum znajduje się Pulsar Kraba PSR B0531+21, który został odkryty w 1969 roku.



Rysunek 1.2: Mgławica kraba. Źródło [6].

Rozdział 2

Uczenie maszynowe

Uczenie maszynowe jest metodą analizy danych i jednym z działów sztucznej inteligencji. Jedną z najważniejszych cech uczenia maszynowego jest eksploracja danych, która polega na uzyskaniu wartościowych informacji z dużej ilości danych. Jeżeli nasz dobrze dobrany algorytm korzysta z większego zbioru danych to prognozy przewidywane przez niego będą lepsze. Obecnie uczenie maszynowe jest używane w bardzo wielu przypadkach, takich jak przetwarzanie danych swojego klienta w celu lepszej rekomendacji reklam. Więcej informacji możemy się dowiedzieć w książce [7].

2.1 Rodzaje uczenia maszynowego

Najczęściej wymieniane są trzy rodzaje uczenia maszynowego - uczenie nadzorowane, nienadzorowane oraz przez wzmacnianie:

- **Uczenie nienadzorowane** [7] - ten rodzaj uczenia maszynowego zostaje wykorzystany przez algorytmy, które analizują dane bez etykiety, czyli bez klasy. Algorytmy z tego rodzaju próbują znaleźć w zbiorze danych wzorce lub zależności. Przykładem uczenia nienadzorowanego może być segmentacja klientów, czyli podzieleniu klientów na mniejsze grupy, według wyznaczonych przez nas kryteriów, takich jak wiek lub płeć.
- **Uczenie przez wzmacnianie** [7] - jest wykorzystywane przez programy lub maszyny, które starają się znaleźć najlepsze rozwiązanie lub najlepszą akcję, jaką powinny obrać w danej sytuacji. W algorytmach uczenia przez wzmacnianie jest wykorzystywany agent sztucznej inteligencji, który może być kodem lub mechanizmem, a dąży on do ustalonego przez nas celu. Takim celem może być wygranie partii szachów z przeciwnikiem.
- **Uczenie nadzorowane** [7] – w przeciwieństwie do uczenia nienadzorowanego wykorzystuje zbiór danych z etykietą klas. Taki zbiór jest wykorzystywany przez nasz algorytm, który uczy się klasyfikacji danych lub przewidywania wyników. Algorytmy uczenia nadzorowanego mogą zostać wykorzystywane w klasyfikowaniu spamu na poczcie elektronicznej.

2.2 Opis algorytmów

W uczeniu nadzorowanym występuje wiele algorytmów, które mogą zostać wykorzystane w celu klasyfikacji prawdziwych pulsarów. W tej pracy zastosujemy pięć algorytmów, takich jak drzewo decyzyjne, agregacja pasków startowych, las losowy, maszyna wektorów nośnych, K-najbliższych sąsiadów.

2.2.1 Drzewo decyzyjne

Model drzewa decyzyjnego [8] działa na zasadzie stworzenia prostych reguł decyzyjnych, przy pomocy których algorytm będzie w stanie wyznaczyć przyszłe obserwacje. Przykłady takich reguł znajdują się na Rys. 2.3. Jesteśmy w stanie pokazać proces tworzenia reguł poprzez wykorzystanie stworzonego przeze mnie zbioru danych z Rys. 2.1. O zbiorze danych możemy się dowiedzieć w sekcji 3.2.2.

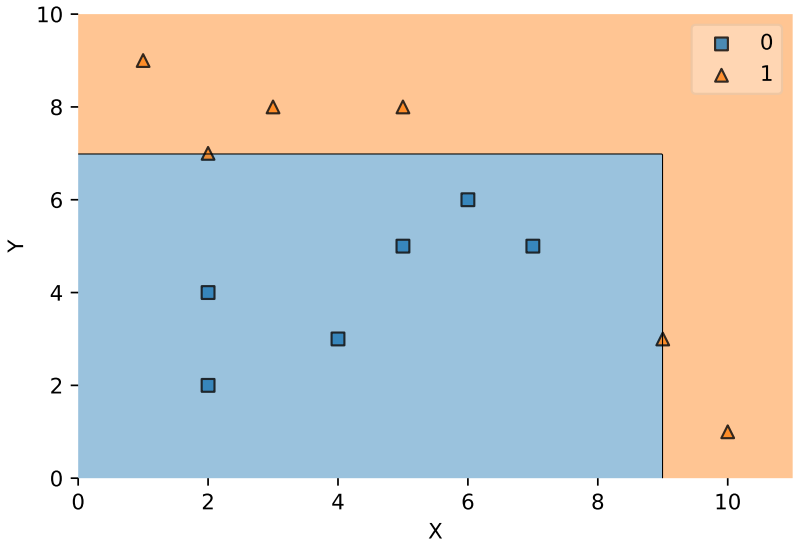
	X	Y	Klasa
0	2	4	0
1	2	2	0
2	7	5	0
3	6	6	0
4	4	3	0
5	5	5	0
6	9	3	1
7	10	1	1
8	2	7	1
9	5	8	1
10	1	9	1
11	3	8	1

Rysunek 2.1: Stworzona przy pomocy biblioteki Pandas ramka danych. Kolumna X odpowiada za oś x, kolumna Y odpowiada za oś y, natomiast kolumna Klasa odpowiada za etykietę klasy.

Używając biblioteki **Scikit-learn** jesteśmy w stanie stworzyć model drzewa decyzyjnego, który wykorzystuje dane pojawiające się w Rys. 2.1. Natomiast przy użyciu biblioteki **Mlxtend** [9] jesteśmy w stanie pokazać nasz zbiór danych na wykresie po zastosowaniu modelu drzewa decyzyjnego na Rys. 2.2.

```
from sklearn.tree import DecisionTreeClassifier
from mlxtend.plotting import plot_decision_regions

trees = DecisionTreeClassifier()

trees.fit(x, y)

plot_decision_regions(x, y, clf=trees, legend=1)
```

W powyższym kodzie, w pierwszych dwóch linijkach importujemy funkcje z biblioteki Scikit-learn oraz Mlxtend, następnie tworzymy obiekt *trees* typu *DecisionTreeClassifier*. Metoda *fit()* trenuje nasz model na podanych przez nas danych *(x, y)* ze zbioru danych na Rys. 2.1, natomiast funkcja *plot_decision_regions* rysuje wykres pokazany w Rys. 2.2.

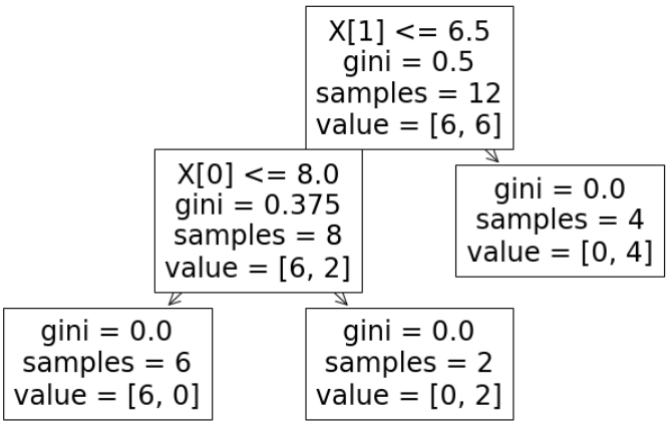


Rysunek 2.2: Wykres zbioru danych z Rys. 2.1 po zastosowaniu algorytmu drzewa decyzyjnego przestawia oś X i Y, na której mamy wyznaczone obszary pojawiania się dwóch różnych klas, gdzie dla klasy 1 jest to obszar pomarańczowy, a dla klasy 0 jest to obszar niebieski.

Jak można zauważyć na Rys. 2.2 nasz wykres dzieli płaszczyznę *xy* na dwie części, w której znajdują się dwie różne klasy. Jesteśmy w stanie zobaczyć, gdzie znajdują się granice wyznaczające nasze obszary decyzyjne. Do szczegółowego pokazania jak została wyznaczona nasza granica oraz w jaki sposób graficznie wygląda nasz model decyzyjny, skorzystamy z biblioteki Scikit-learn.

```
from sklearn import tree

tree.plot_tree(trees)
```

W powyższym kodzie używamy funkcji *plot_tree*, która odpowiada za powstanie Rys. 2.3.



Rysunek 2.3: Model drzewa decyzyjnego zastosowany przy wyznaczaniu wykresu z Rys. 2.2.

Rys. 2.3 przestawia drzewo składające się z korzenia drzewa, który rozrasta się na węzły potomne. Gałąź składa się z czterech części. Pierwsza część określa regułę, jaka zostaje zastosowana na danej gałęzi, w tym przypadku sprawdzamy nasze punkty na X[1], czyli osi y, czy są większe od 6,5, jeżeli tak to te punkty należą to klasy 1. Druga określa rodzaj kryterium rozgałęzienia potrzebne do wyznaczenia przyrostu informacji [7]. Trzecia podaje nam informacje do ilu przykładów (ang. *samples*) odnosi się nasza gałęź. Ostatnia część informuje nas, ile przykładów pochodzi z klasy 0 i 1.

Przyrost informacji pojawiający się na Rys. 2.3 odpowiada za rozdzielnie gałęzi, a konkretnie, żeby takie rozdzielenie przydzieliło nam jak najwięcej klas do próbek danych. W drzewie decyzyjnym istnieją trzy rodzaje jakości podziału pokazane w [7].

Najwyżej na Rys. 2.3 znajduje się tak zwany korzeń drzewa decyzyjnego, od którego zaczyna się podejmowanie decyzji przez nasz algorytm. Korzeń drzewa rozrasta się na kolejne gałęzie, aż do czasu, gdzie wszystkie punkty zbioru zostaną przypisane do przewidzianej klasy. Jednak musimy zauważyć, że im więcej rozgałęzień, tym bardziej nasz model jest podatny na przetrenowanie poprzez zbyt dokładne podzielenie naszych danych.

Przetrenowanie jest to zjawisko, w którym model świetnie sprawdza się przy danych uczących, czyli takich, które zostały użyte do jego trenowania, jednak przy innych danych model będzie radził sobie znacznie gorzej.

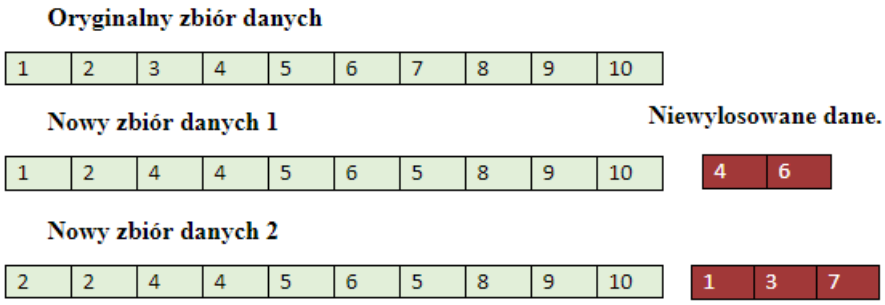
2.2.2 Agregacja pasków startowych

Agregacja pasków startowych [10] należy do metod zespołowych *(ang. Ensamble learning)*. Jest to technika uczenia maszynowego, która łączy kilka modeli w celu stworzenia jednego wydajnego modelu.

Celem tego algorytmu jest stworzenie nowego losowego zbioru danych, który będzie losowany z oryginalnych danych. Istnieje szansa, że dana próbka nie zostanie wylosowana w nowym zbiorze. Wzór, który określa tę szansę, znajdziemy w (2.1).

$$P = (1 - \frac{1}{m})^m, \tag{2.1}$$

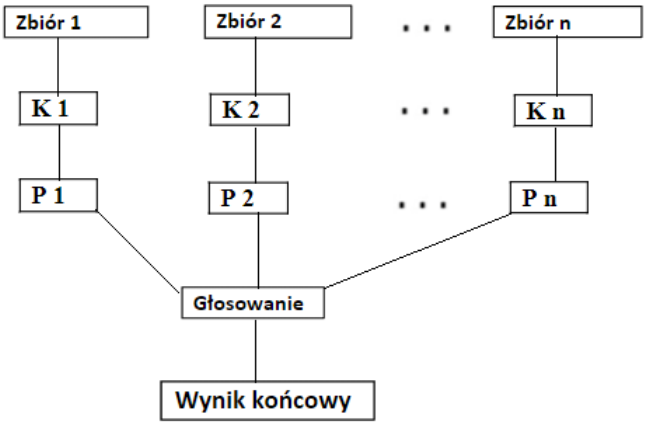
gdzie P = prawdopodobieństwo niewylosowania wiersza, m = ilość wierszy oryginalnego zbioru danych.



Rysunek 2.4: Rysunek przybliżający losowanie nowych danych w agregacji pasków startowych.

Sytuacje, w jaki sposób wygląda losowanie nowego zboru danych, przybliża Rys. 2.4. Możemy zauważyć, że oryginalny zbiór danych zawierał 10 różnych liczb, jednak podczas tworzenia nowego zbioru była szansa niewylosowania danej liczby. W pierwszym przypadku nie została wylosowana liczba 4 i 6, a w drugim liczba 1, 3 i 7.

W algorytmie agregacji pasków startowych możemy wybrać, ile takich zbiorów chcemy stworzyć. Do działania tego algorytmu będzie nam potrzebny algorytm klasyfikacji, czyli taki który będzie tworzył prognozy, dla danego nowego zbioru. Najczęściej takim algorytmem jest drzewo decyzyjne. Następną rzeczą będzie sumowanie tych prognoz za pomocą głosowania większościowego lub ze średniej z prognoz do stworzenia jednego ostatecznego modelu.



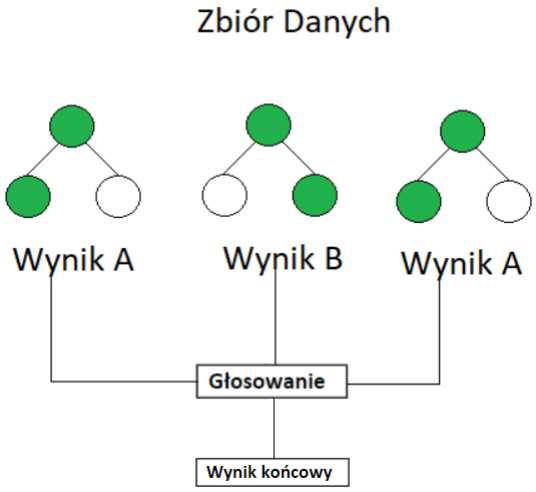
Rysunek 2.5: Rysunek pokazujący przykładowe działanie algorytmu agregacji pasków startowych.

Rys. 2.5 przestawia, jak działa model agregacji pasków startowych. Każdy utworzony losowy zbiór przechodzi przez algorytm kwalifikacyjny K i przewiduje predykcje P. Następnie zostaje przeprowadzane głosowanie większościowe lub średniej z prognoz, które tworzy nam wynik końcowy.

2.2.3 Las losowy

Las losowy [11] *(ang. random forest)* jest to kolejna metoda uczenia zespołowego. Algorytm ten jest bardzo podobny do agregacji pasków startowych, gdyż również i w tej metodzie

zostaną stworzone losowe zbiory danych. Istnieje jednak kilka różnic. W tym przypadku mamy z góry określony, jaki algorytm klasyfikacyjny zostanie zastosowany i jest to drzewo decyzyjne. Możemy zobaczyć działanie lasu losowego na Rys. 2.6.



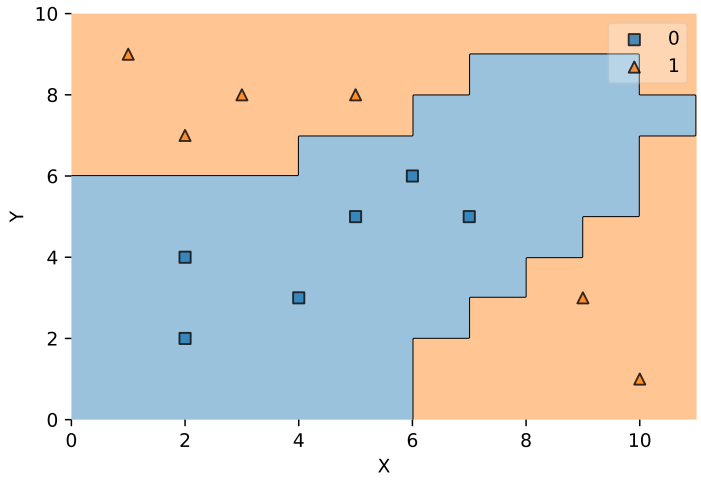
Rysunek 2.6: Rysunek pokazujący przykładowe działanie algorytmu lasu losowego.

Na Rys. 2.6 nasz zbiór danych zostaje podzielony na trzy drzewa decyzyjne, a następnie każde z nich przewiduje wynik algorytmu. Kolejnym krokiem jest głosowanie większościowe, które ma na celu wybrać najlepszy wynik, w tym przypadku były to wynik A.

2.2.4 Maszyna wektorów nośnych

Maszyna wektorów nośnych *(ang. Support Vector Machines)* [12] - jest to algorytm, którego użycie można zastosować w klasyfikacji, bądź regresji. Model ten ma na celu stworzenie hiperpłaszczyzny, która będzie rozdzielać przestrzeń wielowymiarową, na której znajdują się odpowiednie cechy zbioru danych. Celem takiej operacji jest określenie klasy danych próbek.

Jednak znaleziona hiperpłaszczyzna naszego algorytmu ma nie tylko oddzielać klasę. Ma również za zadanie znaleźć optymalny margines hiperpłaszczyzny, czyli odległość tej płaszczyzny od najbliższego wektora cech próbki w zbiorze uczącym. Można to zauważyć, porównując dwa wykresu z Rys. 2.2 oraz Rys. 2.7.



Rysunek 2.7: Przykładowe zastosowanie modelu maszyny wektorów nośnych na danych z Rys. 2.1.

Na Rys. 2.2 widzimy, że oś Y została podzielona w punkcie 6,5, natomiast Rys. 2.7 podzielił naszą oś w punkcie 6, ponieważ jest to optymalne. Podobna sytuacja zachodzi na osi X. Można również zauważyć, że margines hiperpłaszczyzny został zastosowany w innych punktach, przez co nasz obszar decyzyjny klasy 0 ma większą powierzchnię. O algorytmie maszyny wektorów nośnych możemy się przeczytać więcej w [7, 13].

2.2.5 K-najbliższych sąsiadów

Klasyfikator k-najbliższych sąsiadów *(ang. k-nearest neighbor classifier)* [14] - KNN należy do algorytmów grupy leniwych, czyli takich, które szukają rozwiązania wtedy, gdy pojawia się próbka testująca.

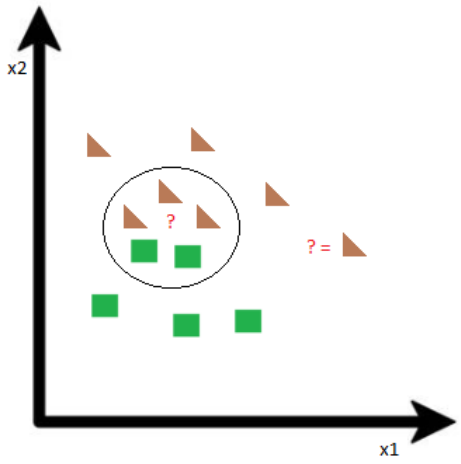
Podczas tworzenia KNN musimy wybrać wartość parametru k, czyli ilość najbliższych sąsiadów naszej próbki oraz metrykę odległości. Po wybraniu tych parametrów model zacznie szukać najbliższych sąsiadów próbki, a następnie dzięki głosowaniu większościowemu określi jej klasę. [7]

Metryka odległości jest wybierana w celu zbadania dystansu pomiędzy dwoma próbkami. Najczęściej wybieraną metryką jest metryka Euklidesowa, którą wyliczymy ze wzoru (2.2).

$$d\left(p,q\right)=\sqrt{\sum_{i=1}^{n}\left(q_i-p_i\right)^2}, \tag{2.2}$$

gdzie $d(p,q)$ jest odległością pomiędzy dwoma punktami p i q.

Żeby lepiej zrozumieć działanie algorytmu, możemy spojrzeć na Rys. 2.8.



Rysunek 2.8: Przykładowe działanie algorytmu KNN.

Na Rys. 2.8. możemy zauważyć, że algorytm chce dopasować nową próbkę. W tym celu stara się sprawdzić, najbliższe pięć etykiet klasy wokół naszej próbki. Następnie zostanie przeprowadzone głosowanie większościowe, które wskaże, że nasza próbka powinna mieć etykietę klasy trójkąta.

2.3 Metody walidacji modeli uczenia maszynowego – walidacja krzyżowa i GridSearch.

2.3.1 GridSearchCV

Każdy algorytm ma swoje hiperparametry, czyli pewną konfigurację modelu jak na przykład ustalenie parametru k w KNN. W zależności od wybranych hiperparmetórw będzie wyznaczany inny model, dlatego należy sprawdzić różne ich rodzaje, a do wyznaczenia najlepszych z nich istnieje wiele funkcji, takich jak Random Search.

Jednak w tej pracy zostanie wykorzystana funkcja GridSearchCV pochodząca z biblioteki Scikit-learn. Ma ona za zadanie sprawdzenie jakości klasyfikacji możliwych kombinacji hiperparametrów, które podał użytkownik. Możemy przeprowadzić przykładowe użycie funkcji GridSearchCV.

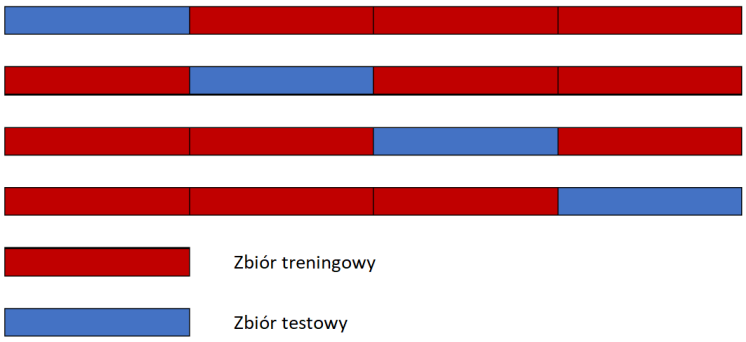
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

trees = DecisionTreeClassifier()

parameters = {'max_depth' : (1,2),
            'criterion' : ('gini','entropy' )
}

grid = GridSearchCV(trees, param_grid = parameters)
grid.fit(x, y)
```

Powyższy kod pokazuje jak należy używać funkcji GridSerachCV. Początkowo trzeba zaimportować tę funkcję z biblioteki Scikit-learn. Potrzebny jest również model, dla którego będziemy szukać najlepszej kombinacji hiperparametrów. W tym przypadku skorzystamy z

modelu oraz z danych pokazanych w Rys. 2.1. Następnym krokiem, który należy wykonać, jest wpisanie hiperparametrów, jakie chcemy sprawdzić i zapisanie ich do zmiennej *parameters*. Kolejno używamy funkcji GridSerachCV, gdzie musimy wpisać dwa parametry. Pierwszy określa algorytm, z jakiego będziemy korzystać, a drugi parametr *param_grid* określa, jakie hiperparametry chcemy sprawdzić. Całość funkcji zapisujemy do zmiennej *grid*, która następnie będzie trenować nasze modele poprzez funkcje $fit()$.

2.3.2 Walidacja krzyżowa

Walidacja krzyżowa [15] jest to metoda statystyczna stosowana przy ocenie jakości modelu uczenia maszynowego. Walidacja krzyżowa to procedura ponownego próbkowania, a najważniejszym jej parametrem jest parametr k, który odpowiada na ile grup, zostanie podzielona próbka danych. Metoda ta ma na celu ocenić predykcje modelu na różnych zbiorach i polega na podzieleniu zestawu danych na nowe zestawy i utworzeniu z nich zbiorów danych testowych i treningowych.



Rysunek 2.9: Walidacja krzyżowa z parametrem 4 k.

Na Rys. 2.9. są pokazane cztery takie same zbiory danych, a każdy z nich jest podzielony na cztery części. Jedna z tych części jest przydzielana do zbioru testowego, a trzy do zbioru treningowego. W każdym z czterech zbiorów danych nasz zbiór testowy jest przydzielany do innej części. Dzięki takiemu zabiegowi na jednym zbiorze danych nasz model będzie mógł przeprowadzić cztery testy dokładność klasyfikacji.

Rozdział 3

Narzędzia i środowisko

3.1 Jupyter Notebook

Rysunek 3.1: Project Jupyter. Źródło [16]

Jupyter Notebook [16] jest to narzędzie do narracji kodu umożliwiające łączenie kodu, opisów i elementów multimedialnych. W Jupyter Notebook możemy budować i edytować kod w blokach. Wynik takiego kodu zostanie wyświetlony zaraz pod blokiem, co może ułatwić prace, gdy korzystamy z wielu wykresów. Kiedyś podczas tworzenia nowego pliku w Jupyter Notebook można było wybierać pomiędzy wersjami języka programowania Python, do wyboru był Python w wersji drugiej i trzeciej. Obecnie nie ma możliwości wyboru wersji drugiej, dlatego w pracy została wykorzystana wersja trzecia.

3.2 Python i jego biblioteki

Rysunek 3.2: Język programowania Python. Źródło [17]

Python jest to obecnie jeden z najbardziej popularnych języków programowania. Zalicza się go jako język programowania wysokiego poziomu. Python został stworzony na początku lat 90, przez Guido van Rossuma, jednak wiele innych osób również miało wkład w rozwój tego języka. Główną zaletą Pythona jest duża ilość bardzo rozbudowanych bibliotek. Biblioteki takie jak Scpi [18], TensorFlow [19], Numpy [20], czy Scikit-learn [21] są potężnymi narzędziami, które możemy zastosować do uczenia maszynowego. Biblioteki python, które okazały się przydatne przy tworzeniu tej pracy, znajdują się poniżej.

3.2.1 Scikit-learn

Rysunek 3.3: Biblioteka Scikit-learn. Źródło [21]

Scikit-learn [21] jest to biblioteka open source dla jezyka programowania Python. Biblioteka ta skupia się głównie na uczeniu maszynowym i zawiera różne algorytmy klasyfikacji, regresji i grupowania, a także sposoby potrzebne do przetwarzania danych. Przy powstaniu niniejszej pracy Scikit-learn zapewnił dostęp do potrzebnych algorytmów, a także pozwolił odpowiednio przetworzyć dane. Można przestawić bibliotekę Scikit-learn za pomocą implementacji jednego z modeli.

```
from sklearn.ensemble import ExtraTreesClassifier

forest_extra = ExtraTreesClassifier()
forest_extra.fit(x_train,y_train)
forest_extra_pred = forest_extra.predict(x_test)
```

Powyższy kod przestawia przykładowe stworzenie modelu extra tree [22]. W pierwszej linijce kodu importujemy model z biblioteki Scikit-learn, następnie tworzymy obiekt *forest_extra* typu *ExtraTreesClassifier*. Metoda *.fit()* trenuje nasz model na podanych przez nas danych (*x_train, y_train*). Wyjaśnianie tych danych znajdziemy w następnym rozdziale. Natomiast metoda *.predict()* służyć nam będzie, żeby przewidzieć wyniki dla danych znajdujących się w zmiennej *x_test*. W podobny sposób będziemy korzystać z tej biblioteki w rozdziale 5.

3.2.2 Pandas

Rysunek 3.4: Biblioteka Pandas. Źródło [23]

Pandas [23] jest to biblioteka napisana w Pythonie przy pomocy biblioteki Numpy [24]. Pandas służy zwykle do pozyskiwania danych, ich obróbki lub przygotowania. Potrafi także wczytywać dane z wielu źródeł, takich jak strony internetowe lub pliki z rozszerzeniem *.csv*, czyli formatem do przechowywania danych w plikach tekstowych. Do wczytania naszych danych potrzebna będzie funkcja *read_csv()*, która służy do odczytywania plików *.csv*. Tak wczytane dane przyjmą typ danych DataFrame, czyli odpowiednik tabeli w Excelu [25].

	X	Y	Klasa
0	2	4	0
1	2	2	0
2	7	5	0

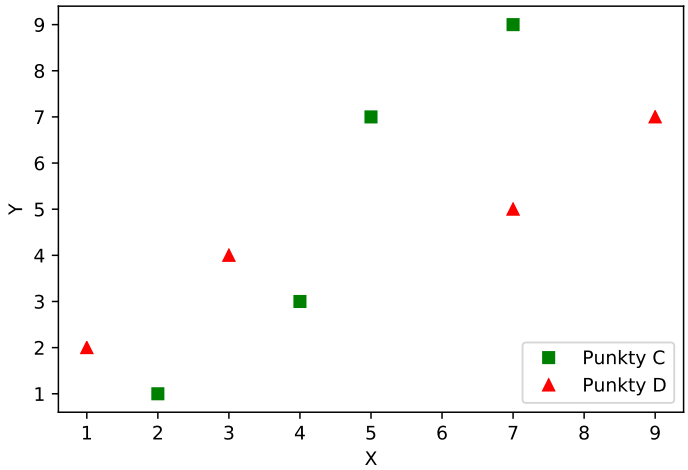
Rysunek 3.5: Przykładowe pokazanie zbioru danych przy użyciu biblioteki Pandas.

Rys. 3.5 pokazuje typ danych DataFrame. Ten typ danych przyjmuje wygląd tablicy i składa się z wierszy oraz kolumn, które jesteśmy w stanie przekształcać dzięki bibliotece pandas.

3.2.3 Matplotlib

Rysunek 3.6: Biblioteka Matplotlib. Źródło [26]

Matplotlib [26] jest to biblioteka, która służy do tworzenia lub manipulacji wykresów. Biblioteka ta została napisana przez Johna Huntera.



Rysunek 3.7: Wykres stworzony przy użyciu biblioteki Matplotlib.

```
import matplotlib.pyplot as plt

C = [2,4,5,7]
D = [1,3,7,9]

plt.plot(C,D,'bs',label = "Punkty C",color='green')
plt.plot(D,C,'g^',label = "Punkty D",color='red')
```

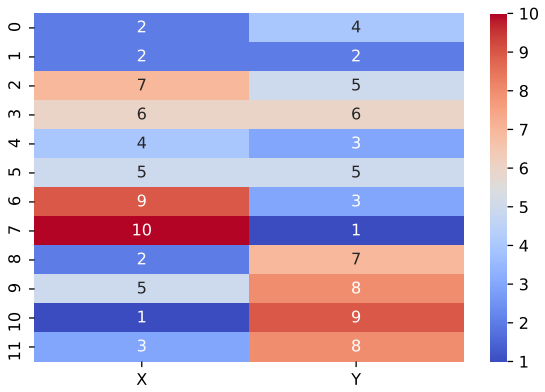
```
plt.ylabel('Y')
plt.xlabel('X')
plt.legend(loc="lower right")

plt.show
```

Powyżej znajduje się przykładowy kod, jaki można napisać w celu stworzenia wykresu. Jak można zauważyć pierwszą czynnością było zaimportowanie modułu *pyplot* z biblioteki Matplotlib. Następnie korzystamy z funkcji *plot*. Parametry, jakie możemy ustawić w funkcji *plot* to po pierwsze, współrzędne, które chcemy przestawić, w tym przypadku są to zbiory C i D. Kolejnym parametrem, który możemy określić, będzie wybór kształtu, jaki mają przyjąć nasze punkty. Parametr *label* odpowiada za nazwę naszych punktów, a parametr *color* za ich kolor. W kolejnych linijkach ustalamy nazwę naszych osi przy pomocy funkcji *ylabel* oraz *xlabel*, a także miejsce naszej legendy wykresu, w tym przypadku jest to dolny prawy róg. Na sam koniec funkcja *show* odpowiada za wyświetlenie naszego wykresu.

3.2.4 Seaborn

Rysunek 3.8: Biblioteka Seaborn . Źródło [27]

Biblioteka **Seaborn** [27] została stworzona na bazie biblioteki Matplotlib i ma ona na celu stworzenie niestandardowych wykresów. Jednym z takich wykresów jest *heatmap* z biblioteki Seaborn, która za pomocy kolorów pokazuje nam największe wartości.



Rysunek 3.9: Przykładowy wykres *heatmap* zastosowany na zbiorze danych z Rys 2.1

Na wykresie 3.9 możemy zauważyć, że największe wartości mają kolor czerwony a najniższe kolor niebieski.

```
 import seaborn

seaborn.heatmap(x, cmap="coolwarm", annot=True)
```

Powyższy kod przestawia stworzenie mapy ciepła (typ wykresu *heatmap*). W tym celu importujemy bibliotekę Seaborn, a następnie wywołujemy funkcje *heatmap*, która zawiera trzy parametry: x, czyli nasz zestaw danych, *cmap*, który określa kolor wykresu oraz *annot*, który wyświetla wartości naszych danych w tabelkach.

Część II

Część Praktyczna

Rozdział 4

Przygotowanie danych oraz metryka

4.1 Opis danych

Pierwszą czynności potrzebną do wykonania projektu było pobranie danych ze strony [28]. Dane te mają nazwę HTRU2 i jest to zbiór danych opisujący potencjalne pulsary zebrane do "High Time Resolution Universe Survey (South)" [29]. Następnie plik z danymi w formacie *.csv* został wczytany do środowiska Jupyter Notebook za pomocą biblioteki Pandas.

```
import pandas as pd

dataframe = pd.read_csv('HTRU_2.csv')
```

Na powyższym kodzie importujemy bibliotekę Pandas jako pd. Jest to częsty zabieg mający na celu skrócenie kodu. Następnie wykorzystujemy funkcję *read_csv()* do odczytania naszego pliku *HTRU_2.csv* zawierającego nasze dane. Nasze wczytane dane zapisujemy do zmiennej dataframe typu danych DataFrame opisanego w sekcji 3.2.2.

	Mean of integradet profile	Standard devation of the integrated profile	Excess kurtosis integraded profile	Skewness integrated profile	Mean of the DM-SNR curve	Standard devation DM-SNR curve	Excess kurtosis of the DM-SRR curve	Skewness of the DM-SNR curve	Class
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	0
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	0
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	0
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	0
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	0
...	...	...	...	...	...	...	...	...	...
17893	136.429688	59.847421	-0.187846	-0.738123	1.296823	12.166062	15.450260	285.931022	0
17894	122.554688	49.485605	0.127978	0.323061	16.409699	44.626893	2.945244	8.297092	0
17895	119.335938	59.935939	0.159363	-0.743025	21.430602	58.872000	2.499517	4.595173	0
17896	114.507812	53.902400	0.201161	-0.024789	1.946488	13.381731	10.007967	134.238910	0
17897	57.062500	85.797340	1.406391	0.089520	188.306020	64.712562	-1.597527	1.429475	0

17898 rows × 9 columns

Rysunek 4.1: Pokazanie fragmentu danych w zmiennej dataframe w środowisku Jupyter Notebook.

Na Rys. 4.1 pokazany został fragment naszych danych w zmiennej dataframe. Jeszcze więcej możemy się dowiedzieć o naszych danych przy pomocy funkcji *.info()*.

```
dataframe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17898 entries, 0 to 17897
Data columns (total 9 columns):
 #   Column                                        Non-Null Count  Dtype
---  ------                                        --------------  -----
 0   Mean of integradet profile                    17898 non-null  float64
 1   Standard devation of the integrated profile   17898 non-null  float64
 2   Excess kurtosis integraded profile            17898 non-null  float64
 3   Skewness integrated profile                   17898 non-null  float64
 4   Mean of the DM-SNR curve                      17898 non-null  float64
 5   Standard devation DM-SNR curve                17898 non-null  float64
 6   Excess kurtosis of the DM-SRR curve           17898 non-null  float64
 7   Skewness of the DM-SNR curve                  17898 non-null  float64
 8   Class                                         17898 non-null  int64
dtypes: float64(8), int64(1)
memory usage: 1.2 MB
```

Rysunek 4.2: Kod funkcji *.info()* oraz jej wynik.

Rys. 4.2 dostarcza nam informacji, jakie typy danych znajdują się w naszej kolumnie, ilość pamięci, jakie zużywa nasz zbiór danych, a także ilość wierszy oraz kolumn. Widzimy, że nasze dane składają się z 17898 wierszy, czyli liczbie naszych potencjalnych pulsarów oraz z 9 kolumn. Nazwy naszych kolumn wyglądają następująco:

- **Mean of integradet profile** - Średnia uśrednionego profilu pulsara
- **Standard devation of the integrated profile** - Odchylenie standardowe uśrednionego profilu pulsara
- **Excess kurtosis integraded profile** – Kurtoza uśrednionego profilu pulsara
- **Skewness integrated profile** - Współczynnik skośności uśrednionego profilu pulsara
- **Mean of the DM-SNR curve** - Średnia krzywej DM-SNR
- **Standard devation DM-SNR curve** - Odchylenie standardowe krzywej DM-SNR
- **Excess kurtosis of the DM-SRR curve** - Kurtoza krzywej DM-SNR
- **Skewness of the DM-SNR curve** - Współczynnik skośności krzywej DM-SNR
- **Class** - Klasa

Możemy zauważyć, że istnieją cztery funkcję statystyczne odnoszące się do dwóch zagadnień. Te cztery funkcje to:

- **Średnia arytmetyczna** [30] liczona jest poprzez podzielenie sumy wszystkich liczb należących do naszego zbioru danych, a następnie podzielenie jej przez liczbę naszych danych. Średnią możemy wyznaczyć ze wzoru (4.1).

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{4.1}$$

  gdzie $\bar{x}$ = średnia, n = liczba danych, x_i = indywidualna dana.

- **Odchylenie standardowe** [31] informuje nas o rozrzucie wyników dookoła średniej. Im bardziej nasze wyniki są przybliżone do średniej, tym mniejsze będzie odchylenie standardowe. Natomiast, jeśli nasze wyniki znacznie różnią się od siebie, nasze odchylenie standardowe będzie wysokie. Odchylenie standardowe wyznaczamy ze następującego wzoru:

$$\hat{\sigma} = \frac{1}{N}\sum_{i=1}^{n}(x - \bar{x})^2, \tag{4.2}$$

gdzie σ = odchylenie standardowe, x_i = indywidualna dana, $\bar{x}$ = średnia, N = liczba danych.

- **Kurtoza** [32] informuje nas, czy większość uzyskanych przez nas danych orbituje wokół naszej średniej. Jeżeli kurtoza przyjmie wartość powyżej 0, oznacza to, że większość naszych danych jest zbliżona do średniej. Jeżeli nasze dane będą znacznie odchylone od średniej, kurtoza przyjmie wartość poniżej zera. Kurtoze wyznaczymy ze wzoru (4.3).

$$K = \frac{1}{N}\sum_{i=1}^{N}\frac{(x_i - \bar{x})^4}{\sigma^4}, \tag{4.3}$$

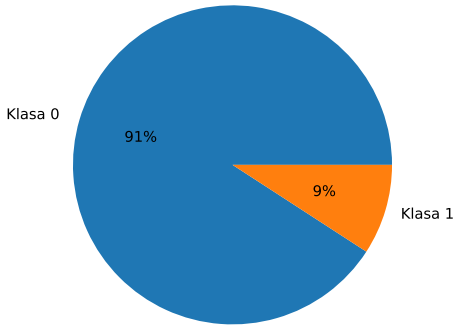
gdzie K = kurtoza, σ = odchylenie standardowe, x_i = indywidualna dana, $\bar{x}$ = średnia, N = liczba danych.

- **Skośność** [33] określa miarę asymetrii naszych danych, czyli czy w naszym zbiorze danych występuje więcej wyników poniżej, czy powyżej średniej. Mamy trzy możliwości ułożenia się współczynnika skośności. Jeżeli nasza wartość będzie bardzo blisko 0, informuje nas to o braku lub minimalnej asymetrii. Jeżeli współczynnik skośności przyjmie wartość powyżej 0, oznaczać to będzie, że nasze dane posiadają prawostronną asymetrię rozkładu, czyli większość naszych wyników będzie poniżej średniej. Ostatni przypadek to pojawienie się lewostronnej asymetrii rozkładu co oznacza, że większość naszych wyników będzie miała wartość powyżej średniej. Współczynnik skośności obliczymy ze wzoru:

$$SKE = \frac{1}{N}\sum_{i=1}^{N}\frac{(x_i - \overline{x})^3}{\sigma^3}, \tag{4.4}$$

gdzie SKE = współczynnik skośności, σ = odchylenie standardowe, x_i = indywidualna dana, $\bar{x}$ = średnia, N = liczba danych.

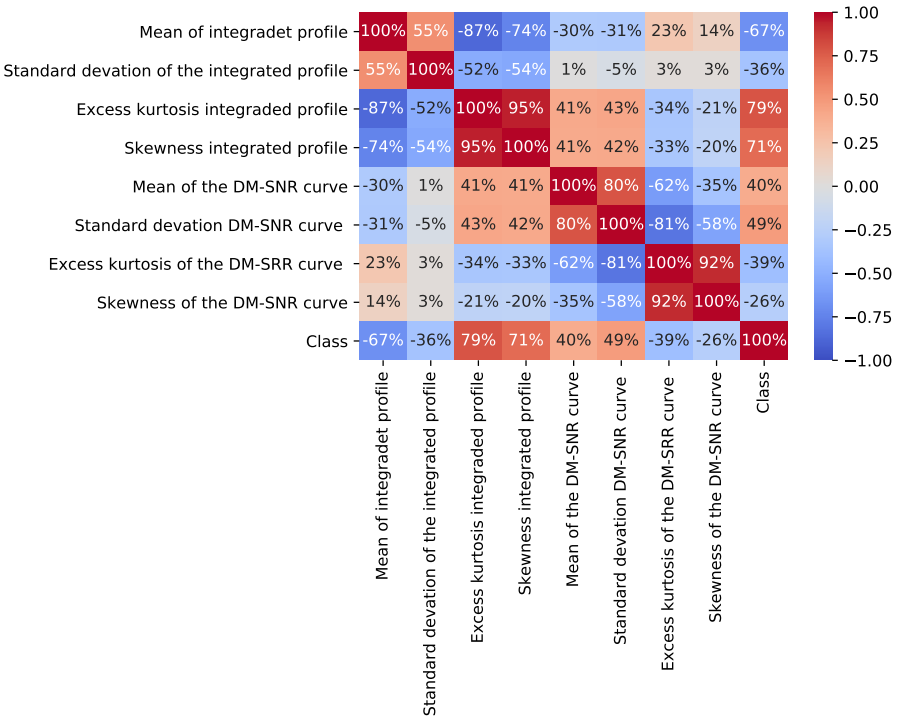
Ostatnia nasza kolumna odnosi się do klasy. Ma ona wartość 0 lub 1, gdzie klasa o wartości 0 oznacza, że dochodzące sygnały nie są od pulsarów, natomiast klasa 1 oznacza, że sygnały zostały wysłane przez pulsary. Rys. 4.3 pokazuje nam stosunek klasy 0 do klasy 1.



Rysunek 4.3: Proporcja sygnałów pochodzących z pulsarów do sygnałów niepochodzących od pulsarów.

Jak możemy zauważyć, aż 91% naszych danych nie pochodzi od pulsarów. W związku z tym możemy wywnioskować, że poszukiwania pulsarów nie należą do łatwych, o czym mówi praca [34].

Dzięki pomocy biblioteki Seaborn możemy również pokazać korelacje naszych kolumn, co przestawia Rys. 4.4.



Rysunek 4.4: Korelacja pomiędzy kolumnami zbioru danych dataframe.

Macierz korelacji przestawia jak dwie zmienne losowe, są ze sobą powiązane. Można zauważyć, że aż 95% powiązania występuje pomiędzy współczynnikiem skośności uśrednionego profilu a kurtozą uśrednionego profilu, podobna sytuacja zachodzi w krzywej DM-SNR gdzie

powiązanie między tymi zmiennymi wynosi 92%. Oznacza to, że współczynnik skośności oraz kurtoza mają bardzo wysoką korelację.

Najmniejsze powiązanie występuje pomiędzy kurtozą uśrednionego profilu a średnią arytmetyczną uśrednionego profilu pulsara.

4.2 Przygotowanie danych

Podstawową czynnością, jaką należy wykonać na naszym zbiorze, jest sprawdzenie, czy nie brakuje w nim danych, ponieważ brakujące dane mogą wpłynąć na dokładność naszego algorytmu. W przypadku, gdy w naszym zbiorze danych wiersz zawiera brakującą wartość wyświetli się wartość numeryczna NaN (ang. *not a number*). Biblioteka Pandas posiada odpowiednią funkcję, która sprawdzi, czy podana przez nas zmienna posiada jakiekolwiek NaN. Nazwa tej funkcji to *isnull()* i zwraca ona wartość *False* albo *True*

```
print(dataframe.isnull().values.any())
```

Powyższy kod przestawia sprawdzenie, czy istnieje jakakolwiek wartość NaN w naszym zbiorze danych. Wynik naszej funkcji *isnull()* ma wartość *False*, co oznacza, że żaden wiersz w zbiorze *dataframe* nie ma brakującej wartości.

Nastepną czynnością, jaką należy wykonać będzie oddzielenie kolumny *Class* od reszty. W tym przypadku użyjemy funkcji *iloc*, która w zmiennej x zapisze pierwszych 8 kolumn, a w zmiennej y ostatnią kolumnę, czyli *Class*. Kod przestawiający tę czynność został przedstawiony poniżej.

```
x = dataframe.iloc[:,0:8]
y = dataframe.iloc[:,8:]
```

Kolejną rzeczą, którą należy zastosować, będzie podzielenie danych na część testową oraz treningową. Część treningowa odpowiada za nauczenie naszego modelu zdolności prawidłowego wyznaczania klasy, poprzez znajomość innych wartości z kolumn naszego zbioru. Część testowa ma służyć jako sprawdzenie jakości naszego modelu.

Wprowadzony został taki podział, ponieważ sprawdzenie jakości algorytmu powinno się przeprowadzać na nieznanych dotąd modelowi danych. Ma to na celu ustalić, czy nasz model jest w stanie prawidłowo przewidywać przyszłe obserwacje. Więcej informacji o tej operacji możemy znaleźć w książce [7]. Kod, który pokazuje podzielenie naszych danych, jest pokazany poniżej.

```
 from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25,
random_state=1)
```

Do przeprowadzenia podziału zbioru danych pokazanego na powyższym kodzie została użyta funkcja *train_test_split*, która została zaimportowana z biblioteki Scikit-learn. Nasza zmienna x została podzielona na x_train oraz na x_test, natomiast zmienna y została podzielona na y_train i y_test. X_train oraz y_train należą do części treningowej, natomiast x_test i y_test do części testowej. Nasz zbiór danych został podzielony w stosunku 75% na część treningową i 25% nas część testową, za co odpowiada w naszym kodzie parametr *test_size*. Ostatni parametr w naszej funkcji *train_test_split* to *random_state*, który odpowiada za tasowanie zestawu danych przed podziałem.

Następnie przeprowadzona zostanie operacja normalizacji za pomocą funkcji *StandardScaler* z biblioteki Scikit-learn. Ma ona na celu sprawienie, że dane po zastosowaniu tej funkcji

będą miały wartość średnią równą 0, a odchylenie standardowe równe 1. Celem takiego zabiegu jest zmienić różne zakresy naszych danych, ponieważ jeżeli ich zakres będzie różny może to prowadzić do złego przetworzenia naszych danych. Zastosowanie funkcji *StandardScaler* znajduje się poniżej.

```
from sklearn.preprocessing import StandardScaler

scalar = StandardScaler()

x_train = scalar.fit_transform(x_train)
x_test = scalar.transform(x_test)
```

W powyższym kodzie importujemy funkcję *StandardScaler* z biblioteki Scikit-learn. Następnie tworzymy obiekt *scalar* typu *StandardScaler*, którego używamy do transformacji zmiennych x_train, x_test za pomocą metody $transform$.

4.3 Metryka

Do przeprowadzenia metryki służącej do ewaluacji naszych modeli skorzystamy z **tablicy pomyłek**, która jest metodą oceny klasyfikacji naszych klas. Ma ona zastosowanie przy podziale klas na przewidywaną klasę pozytywną oraz przewidywaną klasę negatywną. Celem tablicy jest sprawdzenie, czy przeprowadzone przewidywania są prawidłowe lub fałszywe.

		Klasa rzeczywista	
Klasa predykowana		**pozytywna**	**negatywna**
	pozytywna	prawdziwie pozytywna (TP)	fałszywie pozytywna (FP)
	negatywna	fałszywie negatywna (FN)	Prawdziwie negatywna (TN)

Rysunek 4.5: Tablica pomyłek.

Rys. 4.5 przestawia nam tablice pomyłek składa się ona z czterech rzeczy:

- ***True Positive* (TP)** - prawdziwie pozytywny jest to sytuacja, gdy rzeczywista klasa to Prawda (P), a predykowana klasa jest również Prawda (P).
- ***True Negative* (TN)** - prawdziwie negatywny jest to sytuacja, gdy rzeczywista klasa to Fałsz (F), a predykowana klasa jest również Fałsz (F).
- ***False Positive* (FP)** - fałszywie pozytywny jest to sytuacja, gdy rzeczywista klasa to Fałsz (F), a predykowana klasa to Prawda (P).
- ***False Negative* (FN)** - fałszywie negatywny jest to sytuacja, gdy rzeczywista klasa to Prawda (P), a predykowana klasa to Fałsz (F).

Dzięki tablicy pomyłek jesteśmy w stanie wyznaczyć miary, które pozwalają nam ocenić jakość modelu.

4.3.1 Dokładność

Dokładność (ang. *accuracy*) odpowiada za wyznaczenie stosunku ilości poprawnie przewidzianych wartości klas zbioru danych do wszystkich przewidzianych wartości. Dokładność możemy wyznaczyć ze wzoru (4.5).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{4.5}$$

Oznacza to, jeżeli zastosujemy nasz wzór na 100 wierszach zbioru danych i w 85 wierszach wyznaczymy prawidłową klasę oraz 25 wierszach fałszywą, nasza dokładność wyniesie 0,85.

Dokładność możemy wyznaczyć poprzez zaimportowanie funkcji *accuracy_score* z biblioteki Scikit-learn.

```
from sklearn.metrics import accuracy_score

accuracy_score(y_test, y_prediction)
```

Funkcja *accuracy_score* potrzebuje dwóch parametrów. Parametr *y_test* jest częścią testową i zawiera prawidłowe klasy wierszy, natomiast *y_prediction* zawiera przewidziane klasy przez nasz model. Następnie funkcja *accuracy_score* porównuje te dwa parametry i podaje wynik.

4.3.2 Miara F1

Posługując się tablicą pomyłek, możemy wyznaczyć **precyzje** (ang. *precision*). Precyzja jest miarą tego, w jakim stopniu możemy ufać przewidzianym wartościom. Czyli jakie jest prawdopodobieństwo prawidłowego przewidzenia wartości klasy, wyznaczymy ją ze następującego wzoru:

$$Precision = \frac{TP}{TP + FP}. \tag{4.6}$$

Jeśli na 100 wierszy w zbiorze danych klasa 1 stanowi 50 wierszy, natomiast nasz model przewidział 80 wierszy, nasza precyzja wyniesie 0.625.

Innym sposobem miary jest **pokrycie** (ang. *recall*). Pokrycie jest podobne do precyzji, jednak ma ono za zadanie znaleźć, czy odpowiednia ilość przewidzianej klasy zgadza się z klasą prawdziwą. Pokrycie wyznaczymy ze wzoru (4.7).

$$Recall = \frac{TP}{TP + FN}. \tag{4.7}$$

Oznacza to, że jeżeli na 100 przypadków klasy 0 wyznaczyliśmy 90 przypadków klasy 0, nasze pokrycie wyniesie 0.90.

Istnieje miara, która jest w stanie sprawdzać jednakowo precyzje oraz pokrycie, taka miara nazywa się F1. **Miara F1** jest to średnia harmoniczna z pokrycia oraz z precyzji. Dzięki niej uzyskujemy jeden pomiar i ma on przedział pomiędzy 0 a 1, gdzie wynik 1 jest najlepszy. Wzór wygląda następująco:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}. \tag{4.8}$$

Kod potrzebny do wyznaczenia miary F1 jest bardzo podobny do wyznaczenia dokładności i w tym przypadku skorzystamy z biblioteki Scikit-learn w celu zaimportowania funkcji $f1\_score$.

```
from sklearn.metrics import f1_score

f1_score(y_test, y_prediction)
```

Podobnie jak w przypadku funkcji *accuracy_score* mamy dwa parametry, które zostały wyjaśnione w sekcji 4.3.1.

Rozdział 5

Wybór najlepszego algorytmu

5.1 Drzewo decyzyjne

Wszystkie hiperparametry drzewa decyzyjnego są pokazane na stronie [8], jednak w naszej analizie skupimy się na kryterium wyznaczania przyrostu informacji oraz na maksymalnej głębokości drzewa. Maksymalna głębokość drzewa, czyli liczba poziomów rozgałęzień drzewa decyzyjnego, nie powinna być duża, ponieważ może to doprowadzić do przetrenowania, które zostało wyjaśnione w rozdziale 2.

```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score ,f1_score
import statistics

tree = DecisionTreeClassifier()

parameters = {'max_depth' : (1,2,3,),
            'criterion' : ('gini','entropy' )
}

grid = GridSearchCV(tree, param_grid = parameters, cv = 5)
grid.fit(x_train, y_train)

cv_test_score = grid.cv_results_['mean_test_score']
cv_test_std = grid.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)
```

Jak widać na powyższym kodzie hiperparametry, które chcemy sprawdzić znajdują się w zmiennej *parameters*. Następnie używamy ich razem z algorytmem drzewa decyzyjnego *tree* w funkcji GridSearchCV, która zostanie zapisana do zmiennej *grid*. Następnie ta zmienna trenuje nasz model przy użyciu metody $fit()$.

Za pomocą funkcji GridSearchCV zostanie również przeprowadzana walidacja krzyżowa z parametrem $k = 5$, za co odpowiada w funkcji parametr *cv*. Wyniki tej walidacji zosta-

ną uśrednione i zapisane jako wynik dokładności w *cv_score_mean* i jako wynik odchylenia standardowego w *cv_score_std*.

Kolejno model będzie chciał przewidzieć klasy zmiennej testującej *x_test* i zapisze wyniki w zmiennej *y_prediction*, która zostanie użyta do sprawdzenia jakości naszego modelu przy pomocy funkcji $f1\_score$ oraz *accuracy_score*.

Tabela 5.1: Wyniki walidacji krzyżowej drzewa decyzyjnego.

	Dokładność	Odchylenie standardowe
Wyniki	0.9765	0.0023

Tabela 5.2: Wyniki najlepszych hiperparametrów modelu drzewa decyzyjnego dla części testowej.

	Dokładność	Miara F1
Wyniki	0.9792	0.8800

5.2 Las losowy

Hiperparamery lasu losowego są podobne do poprzedniego modelu z racji podobieństw tych algorytmów. Pojawia się jednak też parametr *n_estimators*, czyli ilość nowych zbiorów danych wyjaśnionych w rozdziale 2.

Na stronie [10] znajdują się wszystkie hiperparametry, natomiast w naszej analizie do sprawdzenia będzie ilość losowych zbiorów danych, maksymalna głębokość drzewa oraz kryterium wyznaczania przyrostu informacji.

```
from sklearn.ensemble import RandomForestClassifier

forest_param = {
            'n_estimators' : (100,200,300),
            'max_depth' : (2,3),
            'criterion' : ('gini','entropy'),
}

forest_cls = RandomForestClassifier()
grid_forest = GridSearchCV(forest_cls, param_grid = forest_param, cv = 5)
grid_forest.fit(x_train, y_train)

cv_test_score = grid.cv_results_['mean_test_score']
cv_test_std = grid.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_forest.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)
```

Hiperparametry do sprawdzenia znajdują się w zmiennej $forest\_param$, a funkcja GridSearchCV zostaje zapisana do zmiennej $grid\_forest$.

Również w tym przypadku zostaje przeprowadzona walidacja krzyżowa z parametrem $k = 5$. Następne czynności są takie same jak w poprzednim modelu.

Tabela 5.3: Wyniki walidacji krzyżowej modelu lasu losowego.

	Dokładność	Odchylenie standardowe
Wyniki	0.9755	0.0023

Tabela 5.4: Wyniki najlepszych hiperparametrów modelu lasu losowego dla części testowej.

	Dokładność	Miara F1
Wyniki	0.9789	0.8756

5.3 Maszyna wektorów nośnych

W tym algorytmie będziemy sprawdzać dwa hiperparametry [12]. Pierwszym z nich jest jądro (*ang. kernel*). Odpowiada ono za stronę matematyczną naszego algorytmu i ma na celu mapowanie danych w przestrzeni wielowymiarowej. W naszym algorytmie będziemy sprawdzać dwa rodzaje jądra. Drugim hiperparametrem, jaki będziemy sprawdzać to gamma, która jest współczynnikiem jądra. O jądrach i ich współczynniku możemy dowiedzieć się w [13] oraz [7].

```
from sklearn import svm

svm_cls = svm.SVC()

svm_param = {
    'kernel' : ('linear', 'rbf'),
    "gamma": ('scale', 'auto')
}


grid_svm = GridSearchCV(svm_cls, param_grid = svm_param, cv = 5)
grid_svm.fit(x_train, y_train)

cv_test_score = grid_svm.cv_results_['mean_test_score']
cv_test_std = grid_svm.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_svm.predict(x_test)
acc_score = accuracy_score(y_test, y_prediction)
f1score = f1_score(y_test, y_prediction)
```

Zmienna *svm_param* zawiera hiperparametry dla maszyny wektorów nośnych. Reszta kodu przebiega podobnie jak we wcześniejszych algorytmach.

Tabela 5.5: Wyniki walidacji krzyżowej modelu maszyny wektorów nośnych.

	Dokładność	Odchylenie standardowe
Wyniki	0.9780	0.0027

Tabela 5.6: Wyniki najlepszych hiperparametrów modelu maszyny wektorów nośnych dla części testowej.

	Dokładność	Miara F1
Wyniki	0.9815	0.8915

5.4 K-najbliższych sąsiadów

Najważniejszym hiperparametrem algorytmu K-najbliższych sąsiadów jest ilość sąsiadów naszej próbki testującej. W naszym algorytmie będziemy sprawdzać różne liczby tych sąsiadów i będą one zapisane w zmiennej *neigh_param*. Całość hiperparametrów znajdziemy na stronie [14].

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score ,f1_score
from sklearn.neighbors import KNeighborsClassifier

neigh = KNeighborsClassifier()

neigh_param = {
    'n_neighbors': ( 2,3,4,5,6,7)
}

grid_neigh  = GridSearchCV(neigh, param_grid = neigh_param)
grid_neigh.fit(x_train, y_train)

y_prediction = grid_neigh.predict(x_test)
acc_score = accuracy_score(y_test, y_prediction)
f1score = f1_score(y_test, y_prediction)
```

Tabela 5.7: Wyniki walidacji krzyżowej modelu K-najbliższych sąsiadów.

	Dokładność	Odchylenie standardowe
Wyniki	0.9773	0.0022

Tabela 5.8: Wyniki najlepszych hiperparametrów modelu K-najbliższych sąsiadów dla części testowej.

	Dokładność	Miara F1
Wyniki	0.9803	0.8848

5.5 Agregacja pasków startowych

Agregacja pasków startowych należy do metod zespołowych, dlatego możemy skorzystać z tego modelu i użyć jednego z naszych wcześniejszych modeli, by poprawić jego jakość.

Najważniejszym hiperparametrem agregacji pasków startowych jest *base_estimator*, czyli jaki algorytm początkowy będzie wykorzystywany do stworzenia predykcji naszych losowych zbiorów danych. W tym przypadku będziemy korzystać z drzewa decyzyjnego, ponieważ ten model jest podatny na przetrenowanie a parametr *n_estimators* powinien zminimalizować to ryzyko.

Wszystkie hiperparametry możemy znaleźć na stronie [10], natomiast w naszej analizie skupimy się na ilości nowych zbiorów danych $n\_estimators$.

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score ,f1_score
from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier(criterion="entropy", max_depth = 3)

bagging_param={
    'n_estimators' : (100,200,300),

}

bagging = BaggingClassifier(base_estimator= tree)
grid_bagging = GridSearchCV(bagging, param_grid = bagging_param)
grid_bagging.fit(x_train, y_train)

y_prediction = grid_bagging.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)
```

Tabela 5.9: Wyniki walidacji krzyżowej modelu agregacji pasków startowych.

	Dokładność	Odchylenie standardowe
Wyniki	0.9843	0.0021

Tabela 5.10: Wyniki najlepszych hiperparametrów modelu agregacji pasków startowych dla części testowej.

	Dokładność	Miara F1
Wyniki	0.9803	0.8860

5.6 Potok oraz walidacje krzyżowa

Potok *(ang. pipeline)* ma za zadanie stworzyć sekwencje zadań. Jest to w pewnym rodzaju opakowanie, w które możemy umieścić nasz model wraz z takimi funkcjami jak *StandardScaler*. O funkcji pipeline możemy się dowiedzieć w [7]. Do naszego potoku chcemy dodać najlepszy dostępny model, dlatego porównamy wyniki walidacji krzyżowej wszystkich modeli.

Tabela 5.11: Wyniki walidacji krzyżowej modeli. Średnia dokładności oraz odchylenie standardowe.

	Dokładność	Odchylenie standardowe
Drzewo decyzyjne	0.9765	0.0023
Las losowy	0.9755	0.0023
Maszyna wektorów nośnych	0.9780	0.0027
K-najbliższych sąsiadów	0.9773	0.0022
Agregacja pasków startowych	0.9843	0.0021

Z Tabeli 5.11 wynika, że naszym najlepszym modelem jest agregacja pasków startowych. Model ten ma nieznacznie lepsze wyniki, niż maszyna wektorów nośnych. Następnie opakujemy ten model w potok wraz z jego najlepszymi hiperparametrami.

```
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import make_pipeline

tree = DecisionTreeClassifier(criterion="entropy", max_depth = 3)

pipe = make_pipeline(StandardScaler(),
                     BaggingClassifier(base_estimator = tree, n_estimators = 100))
```

Początkowo importujemy wszystkie funkcje potrzebne w powyższym kodzie. Następnie tworzymy obiekt *pipe* przy pomocy metody *make_pipeline*, w którym znajduje się transformacja danych *StandardScaler* oraz model agregacji pasków startowych z najlepszymi wyznaczonymi hiperparametrami.

Rozdział 6

Podsumowanie i wnioski

Przy użyciu metod uczenia nadzorowanego zostały stworzone algorytmy, których celem była weryfikacja prawdziwych pulsarów w zbiorze danych z HTRU2. Po odpowiednim przygotowaniu danych funkcja GridSearchCV znalazła, które hiperparametry są najlepsze dla danego modelu. Przeprowadzona została również walidacja krzyżowa dla każdego modelu. Porównane zostały wyniki algorytmów w celu znalezienia najlepszego z nich. Najlepszym algorytmem okazał się model agregacji pasków startowych. Następnie model ten został opakowany w potok.

Wyniki przeprowadzonej analizy okazały się zadowalające, ponieważ algorytm z dużym prawdopodobieństwem jest w stanie ustalić, czy sygnały dochodzące z kosmosu, które przypominają pulsary, rzeczywiście nimi są.

Bibliografia

[1] Michael Kramer, D. R. Lorimer, Duncan Lorimer, *Handbook of Pulsar Astronomy* , Wydanie czwarte, Cambridge University Press, 2012.

[2] `https://www.nasa.gov/mission_pages/chandra/multimedia/vela2012.html`(Dostęp: 19.01.2023)

[3] `https://www.manchester.ac.uk/discover/news/slowest-ever-pulsar-star-discovered-by-phd-student/`(Dostęp: 19.01.2023)

[4] S. R. Kulkarni, Dispersion measure: Confusion, Constants & Clarity , arXiv, 2020.

[5] `https://astronet.pl/autorskie/pulsary-i-gwiazdy-neutronowe/`(Dostęp: 26.01.2023)

[6] `https://www.nasa.gov/feature/goddard/2017/messier-1-the-crab-nebula`(Dostęp: 19.01.2023)

[7] S. Raschka, V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2* , Wydanie trzecie, Packt Publishing, 2019.

[8] `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`(Dostęp: 06.01.2023)

[9] `http://rasbt.github.io/mlxtend/`

[10] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html`(Dostęp: 21.01.2023)

[11] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`(Dostęp: 24.01.2023)

[12] `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`(Dostęp: 24.01.2023)

[13] Andreas C. Müller, Sarah Guido, *Machine learning, Python i data science. Wprowadzenie* , Helion, 2021.

[14] `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`(Dostęp: 24.01.2023)

[15] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html`(Dostęp: 26.01.2023)

[16] `https://jupyter.org/`(Dostęp: 06.01.2023)

[17] `https://www.python.org/`(Dostęp: 06.01.2023)

[18] https://pypi.org/project/scpi/(Dostęp: 26.01.2023)

[19] https://www.tensorflow.org/?hl=pl(Dostęp: 26.01.2023)

[20] https://numpy.org/(Dostęp: 26.01.2023)

[21] https://scikit-learn.org/stable/(Dostęp: 26.01.2023)

[22] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html(Dostęp: 26.01.2023)

[23] https://pandas.pydata.org/(Dostęp: 26.01.2023)

[24] Ivan Idris, *Numpy Beginner's Guide* , Wydanie drugie, Packt Publishing, 2013.

[25] https://www.microsoft.com/pl-pl/microsoft-365/excel(Dostęp: 26.01.2023)

[26] https://matplotlib.org/stable/gallery/misc/logos2.html(Dostęp: 16.01.2023)

[27] https://seaborn.pydata.org/(Dostęp: 16.01.2023)

[28] https://archive.ics.uci.edu/ml/datasets/HTRU2(Dostęp: 07.01.2023)

[29] M. J. Keith et al., 'The High Time Resolution Universe Pulsar Survey - I. System Configuration and Initial Discoveries',2010, Monthly Notices of the Royal Astronomical Society, vol. 409, pp. 619-627. DOI: 10.1111/j.1365-2966.2010.17325.x

[30] https://www.naukowiec.org/wiedza/statystyka/srednia-arytmetyczna_716.html(Dostęp: 30.01.2023)

[31] https://www.naukowiec.org/wiedza/statystyka/odchylenie-standardowe_703.html(Dostęp: 30.01.2023)

[32] https://www.naukowiec.org/wiedza/statystyka/kurtoza_698.html(Dostęp: 30.01.2023)

[33] https://www.naukowiec.org/wiedza/statystyka/skosnosc_714.html(Dostęp: 30.01.2023)

[34] R. J. Lyon, 'Why Are Pulsars Hard To Find?', PhD Thesis, University of Manchester, 2016.

Dodatek A

Całość kodu zastosowanego w projekcie

```
# Wczytanie danych oraz sprawdzenia czy są jakieś NaN

import pandas as pd

dataframe = pd.read_csv('HTRU_2.csv',index_col=None)

x = dataframe.iloc[:,0:8]
y = dataframe.iloc[:,8:]

dataframe.info()
print(dataframe .isnull().values.any())

# Podział danych i przeprowadzenie transformacji

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.25,
random_state=1)

scalar = StandardScaler()
x_train = scalar.fit_transform(x_train)
x_test = scalar.transform(x_test)

# Wyznaczenie najlepszych hiperparametrów dla drzewa decyzyjnego
oraz walidacja krzyżowa

from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score ,f1_score
import statistics

tree = DecisionTreeClassifier()

parameters = {'max_depth' : (1,2,3,),
```

```
            'criterion' : ('gini','entropy' )
}

grid = GridSearchCV(tree, param_grid = parameters, cv = 5)
grid.fit(x_train, y_train)

cv_test_score = grid.cv_results_['mean_test_score']
cv_test_std = grid.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)


print('Wynik średniej walidacji krzyżowej : ' + str(cv_score_mean))
print('Wynik odchylenia standardowego walidacji krzyżowej  : ' + str(cv_score_std ))
print('Wynik dokładności najlepszego  modelu : ' + str(acc_score))
print('Wynik F1 najlepszego  modelu : ' + str(f1score))
print("Najlepsze hiperparametry to " + str(grid.best_params_))

# Wyznaczenie najlepszych hiperparametrów dla lasu losowego
oraz walidacja krzyżowa

from sklearn.ensemble import RandomForestClassifier

forest_param = {
            'n_estimators' : (100,200,300),
            'max_depth' : (2,3),
            'criterion' : ('gini','entropy'),
}

forest_cls = RandomForestClassifier()
grid_forest = GridSearchCV(forest_cls, param_grid = forest_param, cv = 5)
grid_forest.fit(x_train, y_train)

cv_test_score = grid_forest.cv_results_['mean_test_score']
cv_test_std = grid_forest.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_forest.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)

print('Wynik średniej walidacji krzyżowej : ' + str(cv_score_mean))
print('Wynik odchylenia standardowego walidacji krzyżowej  : ' + str(cv_score_std ))
print('Wynik dokładności najlepszego modelu : ' + str(acc_score))
print('Wynik F1 najlepszego  modelu : ' + str(f1score))
```

```
print("Najlepsze hiperparametry to " + str(grid_forest.best_params_))

# Wyznaczenie najlepszych hiperparametrów dla maszyny wektorów nośnych
oraz walidacja krzyżowa

from sklearn import svm

svm_cls = svm.SVC()

svm_param = {
    'kernel' : ('linear', 'rbf'),
    "gamma": ('scale', 'auto')
}


grid_svm = GridSearchCV(svm_cls, param_grid = svm_param, cv = 5)
grid_svm.fit(x_train, y_train)

cv_test_score = grid_svm.cv_results_['mean_test_score']
cv_test_std = grid_svm.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_svm.predict(x_test)
acc_score = accuracy_score(y_test, y_prediction)
f1score = f1_score(y_test, y_prediction)


print('Wynik średniej walidacji krzyżowej : ' + str(cv_score_mean))
print('Wynik odchylenia standardowego walidacji krzyżowej  : ' + str(cv_score_std ))
print('Wynik dokładności najlepszego  modelu : ' + str(acc_score))
print('Wynik F1 najlepszego  modelu : ' + str(f1score))
print("Najlepsze hiperparametry to " + str(grid_svm.best_params_))

# Wyznaczenie najlepszych hiperparametrów dla K-najbliższych sąsiadów
oraz walidacja krzyżowa

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score ,f1_score
from sklearn.neighbors import KNeighborsClassifier

neigh = KNeighborsClassifier()

neigh_param = {
    'n_neighbors': ( 2,3,4,5,6,7)
}


grid_neigh  = GridSearchCV(neigh, param_grid = neigh_param, cv = 5)
grid_neigh.fit(x_train, y_train)
```

```
cv_test_score = grid_neigh.cv_results_['mean_test_score']
cv_test_std = grid_neigh.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_neigh.predict(x_test)
acc_score = accuracy_score(y_test, y_prediction)
f1score = f1_score(y_test, y_prediction)

print('Wynik średniej walidacji krzyżowej : ' + str(cv_score_mean))
print('Wynik odchylenia standardowego walidacji krzyżowej  : ' + str(cv_score_std ))
print('Wynik dokładności najlepszego  modelu : ' + str(acc_score))
print('Wynik F1 najlepszego  modelu : ' + str(f1score))
print("Najlepsze hiperparametry to " + str(grid_neigh.best_params_))

# Wyznaczenie najlepszych hiperparametrów dla agregacji pasków startowych
oraz walidacja krzyżowa

from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier(criterion="entropy", max_depth = 3)

bagging_param={
    'n_estimators' : (100,200,300),
}


bagging = BaggingClassifier(base_estimator= tree)
grid_bagging = GridSearchCV(bagging, param_grid = bagging_param, cv = 5)
grid_bagging.fit(x_train, y_train)

cv_test_score = grid_bagging.cv_results_['mean_test_score']
cv_test_std = grid_bagging.cv_results_['std_test_score']
cv_score_mean = statistics.mean(cv_test_score)
cv_score_std = statistics.mean(cv_test_std)

y_prediction = grid_bagging.predict(x_test)
acc_score = accuracy_score(y_test,y_prediction)
f1score = f1_score(y_test, y_prediction)

print('Wynik średniej walidacji krzyżowej : ' + str(cv_score_mean))
print('Wynik odchylenia standardowego walidacji krzyżowej  : ' + str(cv_score_std ))
print('Wynik dokładności najlepszego modelu : ' + str(acc_score))
print('Wynik F1 najlepszego  modelu : ' + str(f1score))
print("Najlepsze hiperparametry to " + str(grid_bagging.best_params_))

# Stworzenie potoku

from sklearn.preprocessing import StandardScaler
```

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import make_pipeline

tree = DecisionTreeClassifier(criterion="entropy", max_depth = 3)

pipe = make_pipeline(StandardScaler(),
                     BaggingClassifier(base_estimator = tree, n_estimators = 100))
```