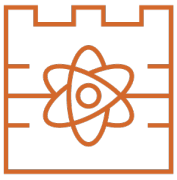
POLITECHNIKA KRAKOWSKA
im. Tadeusza Kościuszki
Wydział Inżynierii Materiałowej i Fizyki
Katedra Fizyki



Kierunek studiów: Fizyka Techniczna
Specjalność: Modelowanie Komputerowe

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

Dawid Paszkot

130334

Klasyfikacja gwiazd na podstawie własności fizycznych wykorzystując techniki uczenia maszynowego i głębokiego

Stars classification based on their physical properties using machine and deep learning techniques

Promotor pracy dyplomowej:
dr Radosław Kycia

Recenzent pracy dyplomowej:
dr hab. prof. PK Sebastian Kubis

Kraków, rok akad. 2023/24

Dawid Paszkot
imię i nazwisko

130334
nr albumu

Inżynierii Materiałowej i Fizyki
wydział PK

Fizyka Techniczna
kierunek studiów

stacjonarne I stopnia
forma studiów i poziom kształcenia

OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa inżynierska przydzielony fragment pracy dyplomowej, w przypadku pracy dyplomowej dwuautorskiej, pt.:

Klasyfikacja gwiazd na podstawie własności fizycznych wykorzystując techniki uczenia maszynowego i głębokiego

została napisana przeze mnie samodzielnie. Jednocześnie oświadczam, że ww. praca:

1) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2021r. poz. 1062)oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
2) nie była wcześniej podstawą żadnej innej procedury związanej z nadawaniem tytułów zawodowych, stopni lub tytułów naukowych.

Jednocześnie wyrażam zgodę na:

1) poddanie mojej pracy kontroli za pomocą systemu Antyplagiat oraz na umieszczenie tekstu pracy w bazie danych uczelni, w celu ochrony go przed nieuprawnionym wykorzystaniem. Oświadczam, że zostałem/am* poinformowany i wyrażam zgodę, by system Antyplagiat porównywał tekst mojej pracy z tekstem innych prac znajdujących się w bazie danych uczelni, z tekstami dostępnymi w zasobach światowego Internetu oraz z bazą porównawczą systemu Antyplagiat,
2) to, aby moja praca pozostała w bazie danych uczelni przez okres wynikający z przepisów prawa. Oświadczam, że zostałem poinformowany i wyrażam zgodę, że tekst mojej pracy stanie się elementem porównawczej bazy danych uczelni, która będzie wykorzystywana, w tym także udostępniana innym podmiotom, na zasadach określonych przez uczelnię, w celu dokonywania kontroli antyplagiatowej prac dyplomowych/doktorskich, a także innych tekstów, które powstaną w przyszłości

.....
podpis

3) Wyrażam zgodę na udostępnianie mojej pracy dyplomowej w Akademickim Systemie Archiwizacji Prac na PK do celów naukowo-badawczych z poszanowaniem przepisów ustawy o prawie autorskim i prawach pokrewnych (Dz.U. z 2021r. poz. 1062).

TAK/~~NIE*~~

.....
podpis

Jednocześnie przyjmuję do wiadomości, że w przypadku stwierdzenia popełnienia przeze mnie czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzej pracy, lub ustalenia naukowego, Rektor PK stwierdzi nieważność postępowania w sprawie nadania mi tytułu zawodowego (art. 77 ust. 5 ustawy z dnia 18 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce, (Dz.U. z 2021r., poz. 478, z późn. zm.))

.....
podpis

*niepotrzebne skreślić

Spis treści

Abstrakt

W pracy zastosowano techniki uczenia maszynowego nadzorowanego oraz głębokiego do klasyfikacji gwiazd na olbrzymy lub karły na podstawie ich właściwości fizycznych. Użyto danych z platformy *Kaggle*, która zajmuję się nauką o danych i uczeniu maszynowym, a także społecznością skupiającą analityków danych. Przy wykorzystaniu języka programistycznego Python oraz jego licznych bibliotek takie jak Pandas, scikit-learn, Tensorflow, Keras, Seaborn stworzono, przetrenowano oraz sprawdzono algorytmy uczenia maszynowego nadzorowanego oraz głębokiego. Dokonano porównania i określono skuteczność modeli w zadaniu klasyfikacji gwiazd. Najwydajniejszym algorytmem uczenia nadzorowanego był model Maszyn wektorów nośnych, w przypadku uczenia głębokiego wyniki były bardzo podobne do uczenia nadzorowanego.

Abstract

In the paper, supervised and deep learning techniques were applied for the classification of stars into giants or dwarfs based on their physical properties. Data from the *Kaggle* platform, specializing in data science and machine learning, as well as a community of data analysts, were used. Using the Python programming language and its various libraries such as Pandas, Scikit-learn, Tensorflow, Keras, and Seaborn, machine learning models were created, trained, and evaluated. The effectiveness of both supervised and deep learning algorithms was compared, and their performance in the task of star classification was determined. The most efficient supervised learning algorithm was the Support Vector Machines model, in the case of deep learning the results were very similar to supervised learning.

Wstęp

0.1 Cel pracy

Celem pracy jest przegląd i wdrożenie metod uczenia maszynowego oraz uczenia głębokiego w dziedzinie astrofizyki w celu wygenerowania najlepszego modelu do klasyfikacji gwiazd na podstawie ich wartości spektralnych znalezionych w zestawie danych pochodzących z platformy *Kaggle*.

0.2 Zakres pracy

Zakres pracy obejmuje część teoretyczną opisującą gwiazdy, ich właściwości oraz klasyfikacje. Następnie opisano uczenie maszynowe, w jaki sposób ocenia trenowane modeli, metody walidacji oraz opisano wykorzystywane algorytmy uczenia nadzorowanego. Opisane zostało również uczenie głębokie. Część teoretyczną zakończono opisem środowiska programistycznego *VS Code* oraz *Jupyter* wykorzystywanego do stworzenia modeli w języku *Python*.

Część praktyczna zawiera analizę eksploracyjną zestawu danych oraz opis implementacji opisanych algorytmów i sposób w jaki zostały wygenerowane modele do klasyfikacji gwiazd.

0.3 Metodyka pracy

Używając zbioru danych z platformy *Kaggle* dotyczących właściwości spektralnych gwiazd przeprowadzono analizę eksploracyjną oraz zaimplementowano algorytmy uczenia nadzorowanego i głębokiego wykorzystując język programowania Python oraz środowisko *VS Code* z rozszerzeniem *Jupyter*. W celu selekcji najwydajniejszych modeli zastosowano metryki oceniające model oraz przygotowano zestawienie wyników w formie tabularnej i graficznej.

Część I

Część teoretyczna

Rozdział 1

Gwiazdy

1.1 Definicja gwiazd

Gwiazdy to astronomiczne obiekty, które są złożone głównie z gorącego, jonizowanego gazu, przede wszystkim wodoru i helu. Proces termojądrowy, który zachodzi w jądrze, polega na łączeniu atomów wodoru, tworząc hel i uwalniając ogromne ilości energii w postaci światła i innych form promieniowania elektromagnetycznego.

Gwiazdy różnią się między sobą wielkością, masą, temperaturą i jasnością. Klasyfikacja gwiazdowa opiera się na ich widmie, które dostarcza informacji o temperaturze i składzie chemicznym. Gwiazdy są również sklasyfikowane na podstawie jasności absolutnej, czyli jasności, jaką miałyby, gdyby były umieszczone w stałej odległości od obserwatora.

Gwiazdy przechodzą przez cykl życia, zaczynając od formowania się w gęstych obszarach molekularnych, przechodząc przez etap główny, aż do faz końcowych, takich jak białe karły, neutronowe gwiazdy lub czarne dziury. Więcej na ten temat można znaleźć w książkach [1] [2]

1.2 Klasyfikacja gwiazd

- **System klasyfikacji widmowej MK** [2] - często używany system do dwuwymiarowej klasyfikacji widm gwiazdowych. Został stworzony przez W.W. Morgana i P.C. Keenana, skąd wywodzi się jego nazwa MK. W tym systemie każda gwiazda ma przypisany typ widmowy oraz klasę jasności. Typy widmowe informują o temperaturze gwiazdy są określane zgodnie z systemem harwardzkim, w którym typy widmowe dzieli się według temperatur i są oznaczone malejąco literami:

  $$\mathbf{O}, \mathbf{B}, \mathbf{A}, \mathbf{F}, \mathbf{G}, \mathbf{K}, \mathbf{M}$$

  Każdy z powyższych typów widmowych dzieli się na 10 podtypów oznaczanych cyframi w zakresie od 0 do 9, gdzie cyfra 0 oznaczana najwyższą temperaturę, a 9 - najniższą.

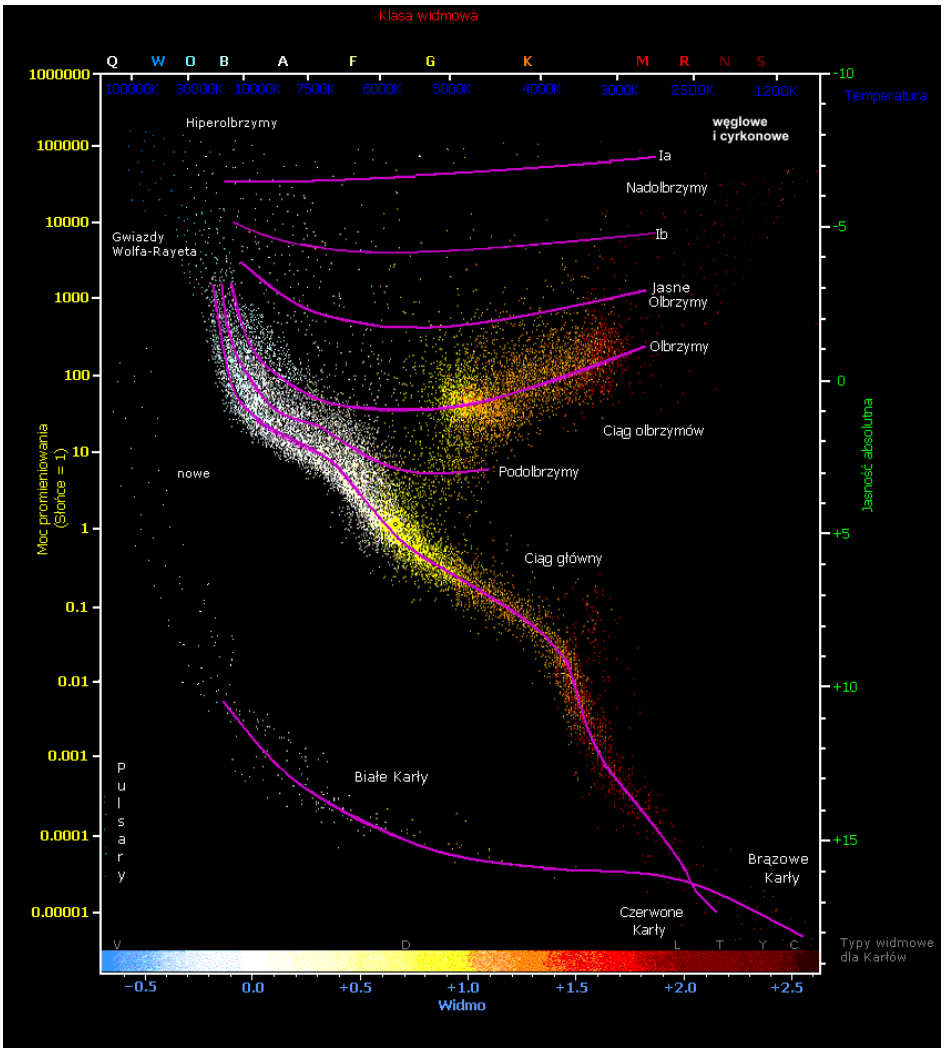
  Cyframi rzymskimi od I do VII oznacza się klasę jasności. W tym przypadku I oznacza jasność największą, a VII oznacza jasność najmniejszą.

- **Diagram Hertzsprunga-Russella** [2] - pokazany na Rys.1.1, jest to narzędzie graficzne, które przedstawia zależność pomiędzy jasnością, a temperaturą powierzchniową gwiazd. Na tym diagramie gwiazdy układają się w charakterystyczne obszary, co pozwala na określenie ich etapu ewolucyjnego. Gwiazdy grupują się w wyraźnie określonych obszarach. Większość gwiazd znajduje się na tzw. ciągu głównym, obejmującym jasne i gorące gwiazdy typu O oraz chłodne, słabe gwiazdy typu M. Są to typowe i najliczniejsze gwiazdy spotykane w kosmosie. Gwiazdą ciągu głownego jest np.: Słońce.

Ponad ciągiem głównym, w zakresie typów widmowych G-M, wyraźnie widoczny jest ciąg jasnych gwiazd, znany jako ciąg olbrzymów, czyli gwiazd średniej wielkości które są w końcowym stadium swojego życia.

W obszarze najwyższych jasności absolutnych występują nieliczne nadolbrzymy, czyli gwiazdy o masie od 10 do 50 razy większej niż słońce, w skali astronomicznej mają krótki czas życia od 10 do 50 milionów lat, dlatego obserwowane są w ramionach galaktyk spiralnych, galaktykach nieregularnych oraz gromadach otwartych.

Znacznie poniżej ciągu głównego, w przedziale typów widmowych A-F, nieregularny obszar zajmują bardzo słabe gwiazdy nazywane białymi karłami, czyli gwiazdami o małej lub średniej wielkości, które zakończyły już swój proces reakcji jądrowych, po przejściu stadium czerwonego olbrzyma i odrzuceniu zewnętrznych warstw pozostaje jądro, które nie produkuje już energii i stopniowo ochładza się przez promieniowanie zgromadzonego ciepła. Zakończenie reakcji termojądrowych we wnętrzu sprawia, że gwiazdy zapadają się pod własnym ciężarem, osiągając ogromną gęstość.



Rysunek 1.1: Diagram Hertzsprunga-Russella) - Źródło [3].

1.3 Jak mierzy się gwiazdę

Jasności gwiazd [2] - jednym z podstawowych parametrów gwiazd są ich jasności. Jasność ciała niebieskiego m odpowiadająca obserwowanemu strumieniowi promieniowania F czyli ilość energii przechodzącej przez jednostkową powierzchnię w jednostce czasu. Jest zdefiniowana jako:

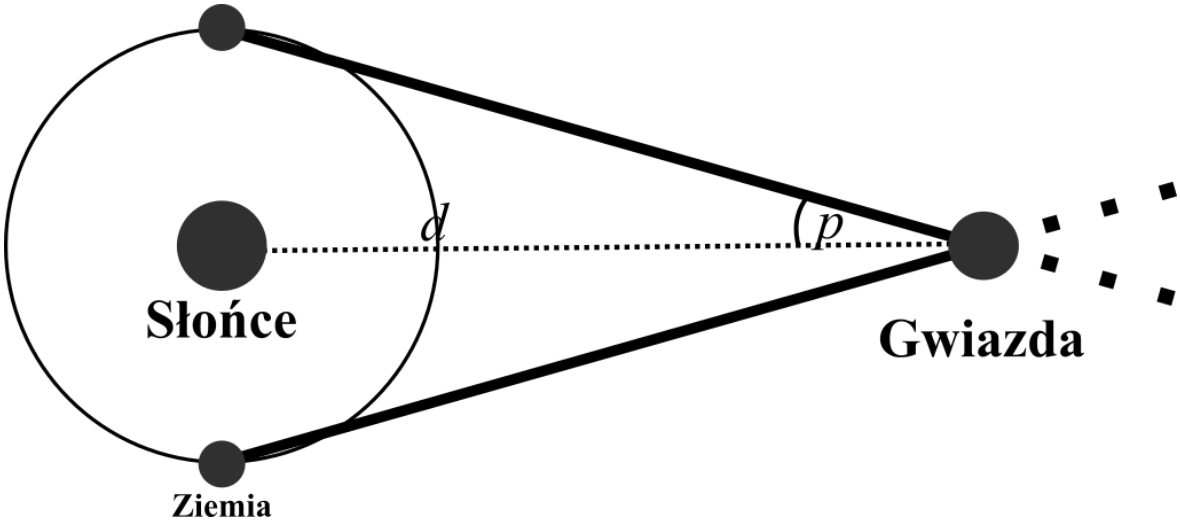
$$m = -2.5 \log F + const, \tag{1.1}$$

Jednostką jasności jest *wielkość gwiazdowa*, nazywana też *magnitudo*, Stałą *const* jest odpowiedzialna na wyznaczenie punktu zerowego skali logarytmicznej jasności, jej znajomość nie jest konieczna, ponieważ przy obliczaniu różnicy jasności między dwoma obiektami ta wartość się uprości.

Wielkość strumienia promieniowania zależy od mocy promieniowania badanego ciała oraz od jego odległości, do neutralizowania wpływu odległości używa się *jasności absolutnej*. Jest to strumień dobiegającego do obserwatora promieniowania gdyby odległość gwiazdy r była stała i oddalona o 10 parseków. Wtedy *jasność absolutna* M wyraża się:

$$M = m + 5 - 5 \log r. \tag{1.2}$$

Paralaksa trygonometryczna [1] - żeby móc zmierzyć jasność gwiazdy potrzebne jest ustalenie odległości między obserwatorem, a gwiazdą. W tym celu używa się metody paralaksy trygonometrycznej. Jest to technika pomiaru odległości do gwiazd na podstawie zmiany kąta paralaksy, czyli zmiany kąta, pod jakim gwiazda widziana jest z dwóch różnych punktów obserwacyjnych w ciągu roku, którą przedstawiono na Rys.1.2. Gwiazdy wydają się poruszać na tle bardziej odległych gwiazd, ponieważ Ziemia przemieszcza się wokół Słońca, tworząc tzw. roczny ruch paralaktyczny.



Rysunek 1.2: Paralaksa - Opracowanie własne na podstawie [4].

W kolejnym rozdziale opisane zostanie Uczenie maszynowe.

Rozdział 2

Uczenie maszynowe

Uczenie maszynowe [5] to dziedzina sztucznej inteligencji, która koncentruje się na tworzeniu systemów komputerowych zdolnych do uczenia się i doskonalenia swoich umiejętności bez bezpośredniego programowania. W przeciwieństwie do tradycyjnych metod programowania, w których programista określa konkretne instrukcje, w uczeniu maszynowym algorytmy są zdolne do ulepszania swojego działania na podstawie zbioru danych.

Aby zastosować uczenie maszynowe, konieczne jest zebranie odpowiednich danych, wybranie odpowiedniego modelu, przetrenowanie go na dostarczonych danych, a następnie ocena jego skuteczności na nowych danych. Wraz z postępem tej dziedziny, uczenie maszynowe znajduje coraz szersze zastosowanie w różnych dziedzinach, takich jak medycyna, finanse, przemysł, transport, czy analiza danych.

2.1 Rodzaje uczenia maszynowego

- **Uczenie nadzorowane** [5] - Algorytmy są uczone na podstawie wejściowych danych oraz odpowiadających im poprawnych wyjść (etykiet). Celem jest nauczenie modelu przewidywania poprawnych etykiet dla nowych, nieznanych danych.
- **Uczenie nienadzorowane** [5] - Algorytmy są uczone na podstawie danych, ale bez etykiet. Celem jest znalezienie wzorców, struktur lub ukrytych zależności w danych.
- **Uczenie przez wzmacnianie** [5] - Model jest trenowany poprzez interakcję ze środowiskiem. Algorytm podejmuje decyzje, a następnie otrzymuje informację zwrotną w postaci nagrody lub kary, co pozwala mu uczyć się, jakie akcje prowadzą do pożądanych rezultatów.

2.2 Ocena wydajności modelu

Do oceny algorytmów w uczeniu maszynowym używa się wielu metod jedną z nich jest tablica pomyłek, zwana również macierzą błędów *(ang. confusion matrix)*, to narzędzie używane do oceny wydajności algorytmów klasyfikacyjnych. Jest szczególnie przydatna w przypadku problemów z dwiema lub więcej klasami. Na rys.2.1 przedstawiony jest jej układ.

Rysunek 2.1: Tablica pomyłek.

2.2.1 Dokładność

Dokładność (ang. Accuracy)[5] mierzy ogólną skuteczność modelu, czyli stosunek liczby poprawnych klasyfikacji do całkowitej liczby przypadków.

$$\text{Dokładność} = \frac{\text{Liczba poprawnych klasyfikacji}}{\text{Całkowita liczba przypadków}},$$

2.2.2 Precyzja

Precyzja (*ang. Precision*)[5] mierzy stosunek liczby prawdziwie pozytywnych przypadków do sumy prawdziwie pozytywnych i fałszywie pozytywnych przypadków.

$$\text{Precyzja} = \frac{\text{Prawdziwie Pozytywne}}{\text{Prawdziwie Pozytywne} + \text{Fałszywie Pozytywne}},$$

2.2.3 Czułość

Czułość (*ang. Recall*) [5] nazywany też wskaźnikiem trafności (*ang. True Positive rate*), to miara która ocenia zdolność modelu do poprawnego zidentyfikowania wszystkich rzeczywistych pozytywnych przypadków w zestawie danych.

$$\text{Czułość} = \frac{\text{Prawdziwie Pozytywne}}{\text{Prawdziwie Pozytywne} + \text{Fałszywie Negatywne}},$$

2.2.4 Miara F1

Miara F1 (*ang. F1-score*) [5] to średnia harmoniczna precyzji i czułości. Uzyskuję się ją łącząc precyzję i czułość. Miara F1 będzie wyrażona w zakresie od 0 do 1, gdzie 1 to oznacza całkowite dopasowanie modelu.

$$\text{Miara F1} = 2 \times \frac{\text{Precyzja} \times \text{Czułość}}{\text{Precyzja} + \text{Czułość}}.$$

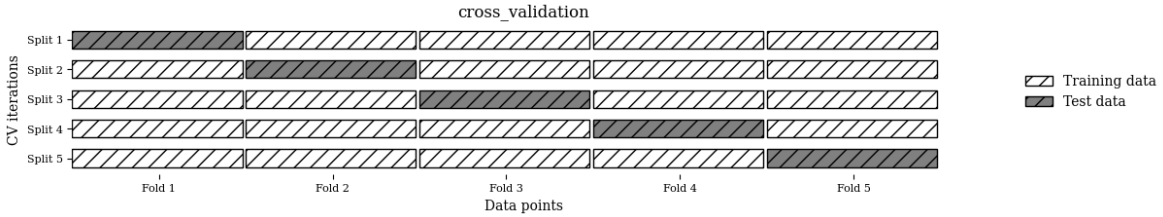
2.3 Metody walidacji

2.3.1 Walidacja krzyżowa

Walidacja krzyżowa [13] - metoda statystyczna oceny wydajności modelu, która jest bardziej stabilna i szczegółowa niż podział na zbiór treningowy i testowy. W tym przypadku dane są wielokrotnie dzielone, a następnie trenowane są różne modele. Najczęściej wykorzystuję się metodę k-krotnej walidacji krzyżowej, gdzie k to liczba która określa na ile części zostaną podzielone dane, najczęściej stosuję się 5 lub 10.

W przypadku pięciokrotnej walidacji krzyżowej dane są początkowo podzielone na pięć równych części, nazywanych kawałkami (ang. fold). Budowana jest sekwencja modeli, gdzie pierwszy model jest trenowany z użyciem pierwszego kawałka jako zbioru testowego, a pozostałe kawałki (2–5) służą jako zbiór treningowy. Model jest tworzony na danych z kawałków 2–5, a następnie jego dokładność jest oceniana na kawałku 1. Proces ten jest powtarzany, tworząc kolejne modele z użyciem kolejnych kawałków jako zbiorów testowych i odpowiednich danych treningowych.

Dla każdego z pięciu podziałów danych na zbiory treningowy i testowy obliczana jest dokładność modelu. Ostatecznie zbiera się pięć wartości dokładności, co pozwala na bardziej obiektywne ocenianie wydajności modelu.



Rysunek 2.2: Podział danych na 5 kawałków w walidacji krzyżowej.

Przy użyciu protego kodu pokazanego na Rys.2.3 wygenerowano Rys.2.2, który pokazuję jak wygląda walidacja krzyżowa, opisana powyżej.

```
import mglearn
mglearn.plots.plot_cross_validation()
```

Rysunek 2.3: Kod użyty do wygenerowania wizualizacji walidacji krzyżowej.

2.3.2 Przeszukiwanie siatki

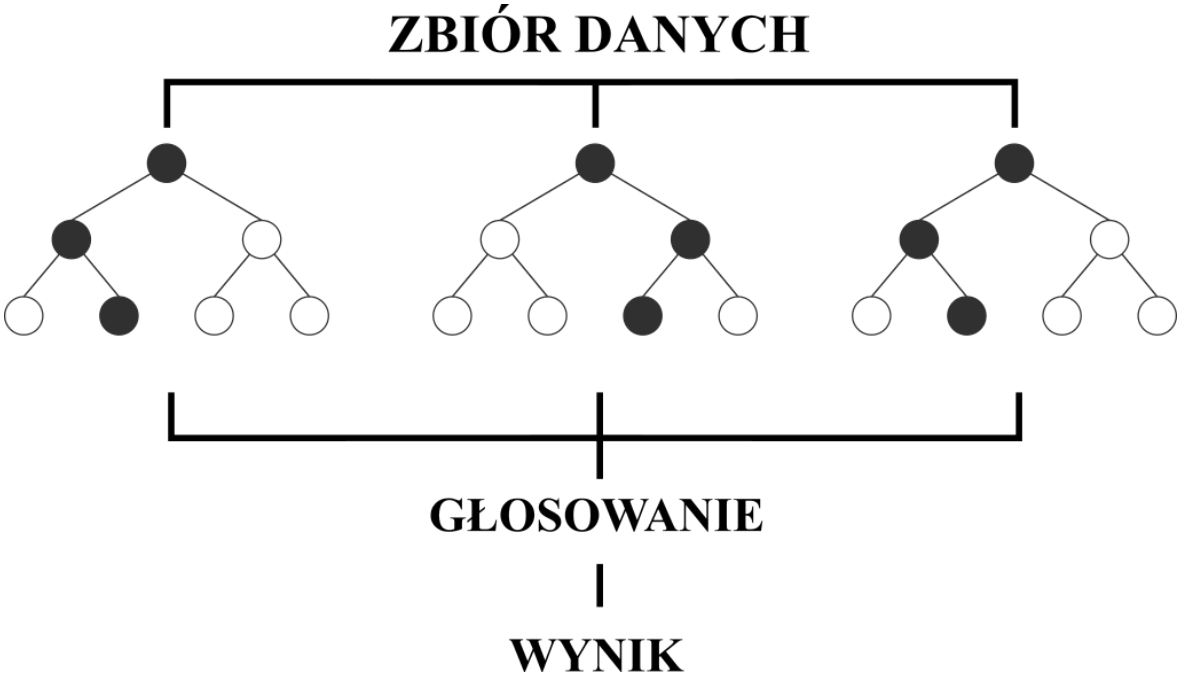
Przeszukiwanie siatki [5]- koncepcyjnie prosta metoda znalezienia najlepszych hiperparametrów dla danego algorytmu poprzez sprawdzenie wszystkich kombinacji zadanych parametrów. Komputer tworzy siatkę wszystkich możliwych kombinacji, a następnie ocenia każdą wariacje tych wartości, aby uzyskać optymalny zestaw hiperparametrów.

2.4 Używane algorytmy

2.4.1 Las losowy

Las losowy *(ang. Random Forrest)* [6] - należy do klasy algorytmów zespołowych. Został wprowadzony przez Leo Breimana i Adele Cutler. Las Losowy jest skutecznym i wszechstron-

nym algorytmem, stosowanym zarówno do problemów klasyfikacji, jak i regresji. Zaczyna się losowego wyboru podzbiorów danych treningowych z pełnego zestawu treningowego.

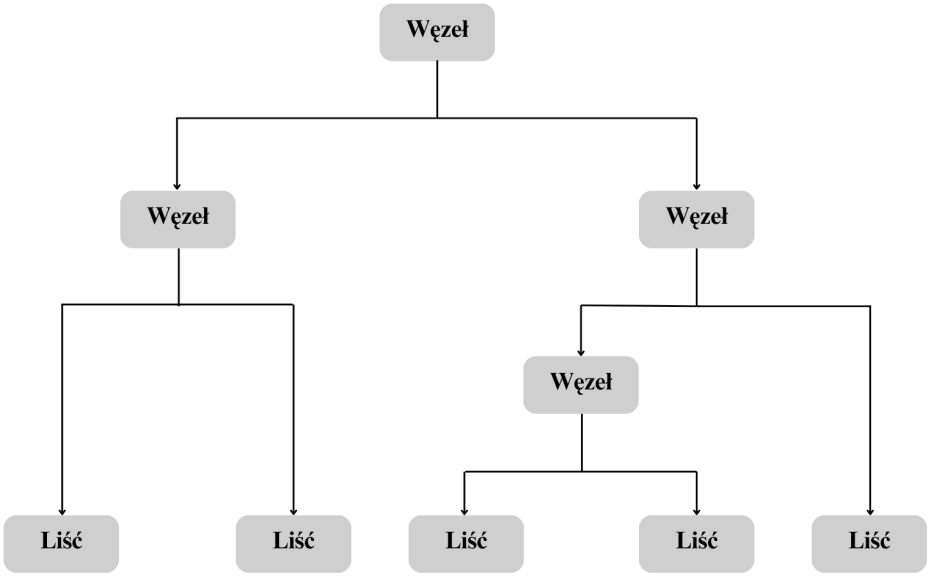


Rysunek 2.4: Graficzna reprezentacja algorytmu lasu losowego - Opracowanie własne na podstawie [5].

Rys 2.4 pokazuję jak dla każdego drzewa tworzonego w lesie, wybieramy inne przypadki oraz, opcjonalnie, inne cechy. Dla każdego podzbioru danych tworzymy drzewo decyzyjne, proces ten polega na podziale danych na podstawie cech, aby jak najlepiej separować klasy lub przewidzieć wartości. Następnie każde drzewo w lesie przewiduje wynik dla swojego podzbioru danych treningowych. W przypadku klasyfikacji, ostateczna prognoza jest wynikiem głosowania drzew. Klasa, która uzyska najwięcej głosów, staje się przewidywaną klasą. W przypadku regresji, prognozy drzew są uśredniane, aby uzyskać ostateczną prognozę.

2.4.2 Drzewo decyzyjne

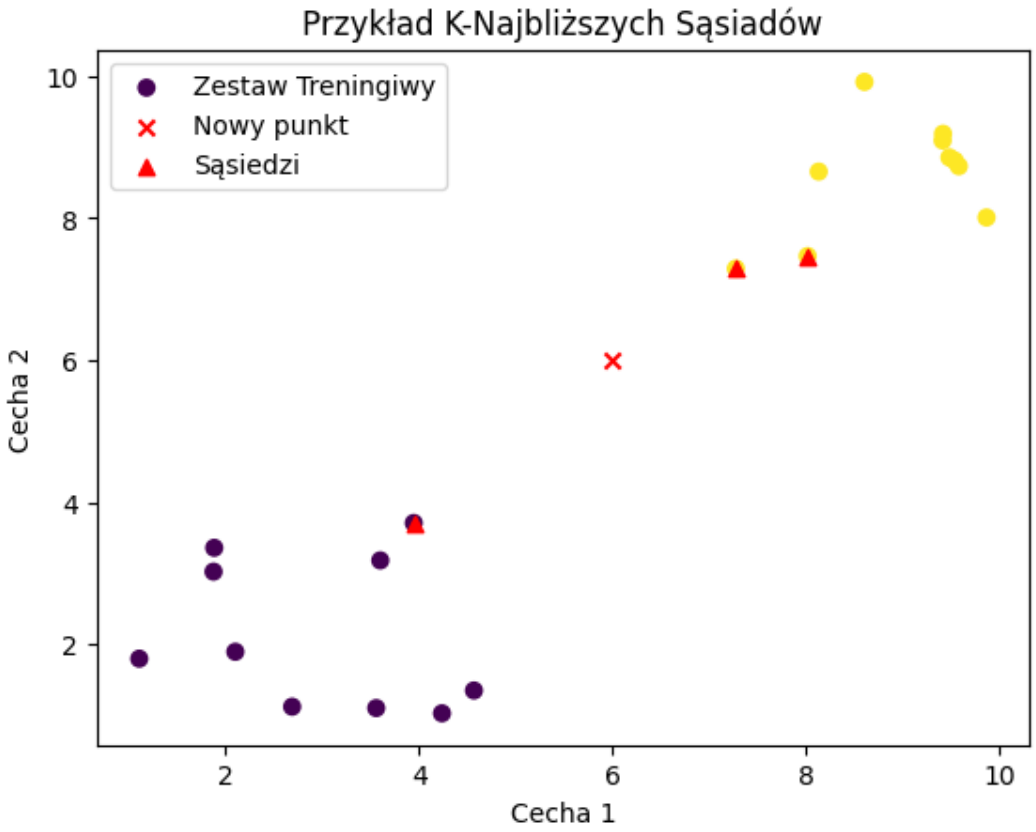
Drzewo decyzyjne (*ang. Decision Tree*) [7] [9] - algorytm uczenia maszynowego wykorzystywany do zadań klasyfikacji i regresji. Regresja jest techniką używaną do modelowania predykcyjnego, więc drzewa decyzyjne znajdują zastosowanie zarówno w klasyfikacji danych, jak i prognozowaniu, co może nastąpić w przyszłości. Struktura drzewa decyzyjnego, Rys.2.5, przypomina schemat blokowy, zaczynając od węzła korzenia z konkretnym pytaniem dotyczącym danych. To pytanie prowadzi do gałęzi zawierających potencjalne odpowiedzi. Następnie te gałęzie prowadzą do węzłów decyzyjnych (węzłów wewnętrznych), które formułują kolejne pytania prowadzące do dalszych wyników. Ten proces trwa aż dane osiągną węzeł terminalny (zwany także ”liściem”) i zakończą się.



Rysunek 2.5: Przykładowe drzewo decyzyjne - opracowanie własne na podstawie [8].

2.4.3 K-Najbliższych sąsiadów

K-Najbliższych sąsiadów *(ang. K-Nearest Neighbors)* [12] [13] - to prosty algorytm w uczeniu maszynowym model tak naprawdę tylko przechowuje zbiór danych treningowych. Przy dokonywaniu predykcji dla nowego punktu, model znajduję najbliższe punkty danych w zbiorze - "jego najbliższych sąsiadów" i na tej podstawie podejmuję decyzje. Zbliżone do siebie wartości cech między sąsiadami a nowym punktem to podobne cechy, więc algorytm zakłada, że punkt musi należeć do podobnej klasy. W przypadku klasyfikacji, model decyduje o przynależności nowego punktu do konkretnej klasy, biorąc pod uwagę etykiety sąsiadów tego punktu. W przypadku regresji, algorytm przewiduje wartość dla nowego punktu danych, uśredniając (w przypadku regresji) lub biorąc medianę (w przypadku klasyfikacji) wartości prognozowanych.



Rysunek 2.6: Wizualizacja działania algorytmu k-NN - Opracowanie własne.

Rys.2.6 pokazuje wizualizacje algorytmu k-NN, można zauważyć jak punkty z dwóch klas, nowy punkt, oraz trzej najbliżsi sąsiedzi tego punktu. Model decyduje o przynależności nowego punktu do konkretnej klasy, w której większość najbliższych sąsiadów należy. W tym przypadku nowy punkt został przypisany do Klasy 1.

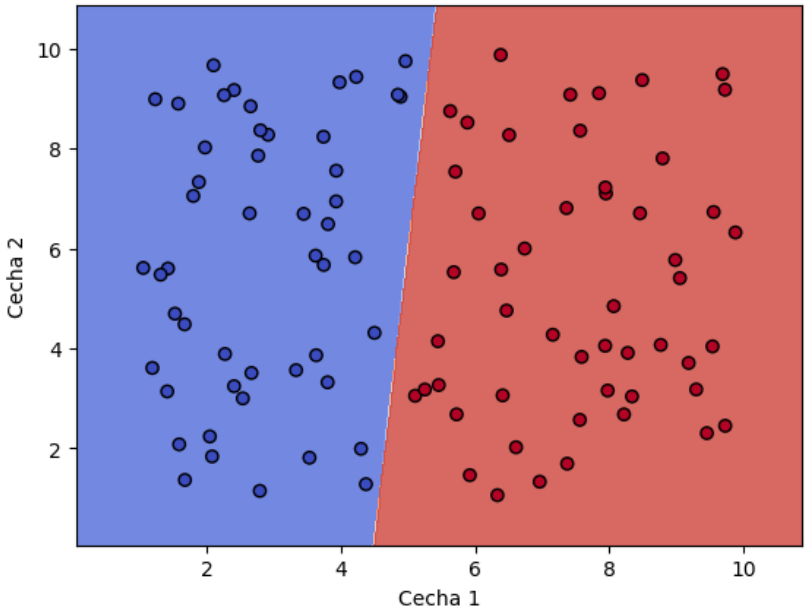
2.4.4 Maszyny wektorów nośnych

Maszyny wektorów nośnych (*ang. Support Vector Machines*) [11] - to algorytm znalezienia hiperpłaszczyzny, która skutecznie separuje przestrzeń wielowymiarową, zawierającą charakterystyczne cechy zbioru danych.

W przypadku zadania klasyfikacji, SVM poszukuje hiperpłaszczyzny, która najskuteczniej oddziela różne klasy danych. Nie jest to jedynie kwestia separacji klas, ale także optymalizacji marginesu hiperpłaszczyzny. Margines ten stanowi odległość między hiperpłaszczyzną a najbliższym wektorem cech próbki w zbiorze uczącym.

Kluczowym elementem w procesie treningu SVM są tzw. wektory nośne, czyli te punkty danych, które leżą najbliżej hiperpłaszczyzny. Optymalizacja odległości między tymi wektorami a hiperpłaszczyzną jest kluczowym aspektem algorytmu.

W trakcie treningu, SVM optymalizuje parametry modelu, dążąc do znalezienia hiperpłaszczyzny o maksymalnym marginesie. Jednocześnie minimalizuje błąd klasyfikacji, co przyczynia się do lepszej zdolności modelu do generalizacji na nowe dane.



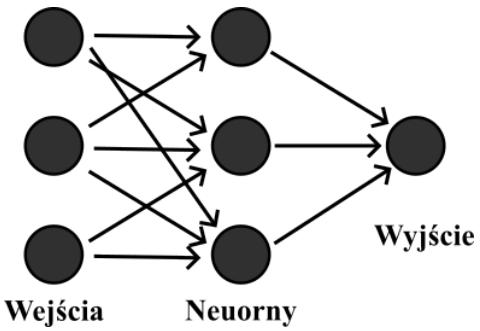
Rysunek 2.7: Wizualizacja działania algorytmu SVM - Opracowanie własne.

2.4.5 Perceptron

Perceptron [15] [16] - jedna z najprostszych form sztucznych sieciach neuronowych, pokazany na Rys.2.8. Neurony połączone są z wejściami za pomocą wag. Wejścia są reprezentowane przez węzły, a neurony decydują o aktywacji na podstawie ważonej sumy wejść i progu. Wagi są dostosowywane w procesie uczenia, gdzie sieć jest trenowana na podstawie zestawu wejść i odpowiadających im oczekiwanych wyników.

Ważnym elementem perceptronu jest niezależność między neuronami - każdy neuron decyduje o swojej aktywacji niezależnie od innych. Wagi, które przypisane są do każdego neuronu, są osobne, jedynym wspólnym elementem są wejścia.

Proces uczenia polega na porównywaniu wyników aktywacji neuronów z oczekiwanymi wynikami. Dla nieprawidłowo aktywowanych neuronów, wagi są modyfikowane, aby poprawić przyszłe odpowiedzi. Korygowanie wag opiera się na różnicy między wynikiem neuronu a oczekiwanym wynikiem, pomnożonej przez wartość wejścia. Wartość ta jest modyfikowana przez współczynnik uczenia, który kontroluje tempo uczenia się sieci.



Rysunek 2.8: Schemat algorytmu perceptronu - Opracowanie własne na podstawie[16].

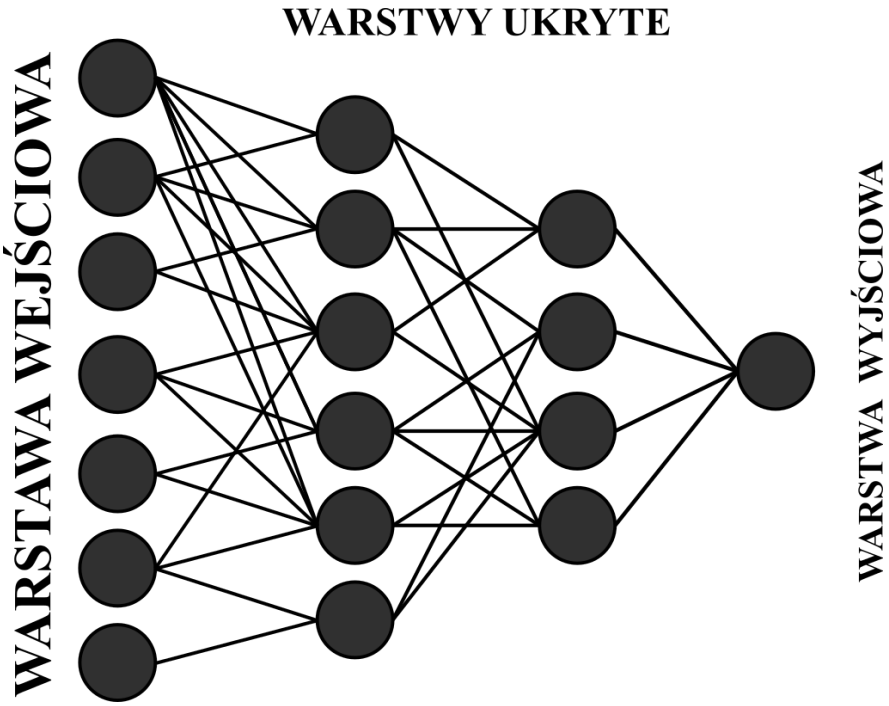
W następnym rozdziale opisane zostanie Uczenie głębokie.

Rozdział 3

Uczenie głębokie

Uczenie głębokie [14] to kolejna dziedzina sztucznej inteligencji, w której trochę inaczej patrzy się na dane. Kolejno uczy się następne warstwy modelu, który stopniowo przybliża się do poprawnej klasyfikacji, stąd nazwa "głębokie", która odnosi się do wielowarstwowości modelu. Liczba warstw danego modelu nazywana jest też *głębokością modelu*.

W uczeniu głębokim w większości przypadków warstwy uczone są z pomocą modeli tak zwanych *sieci neuronowych*.

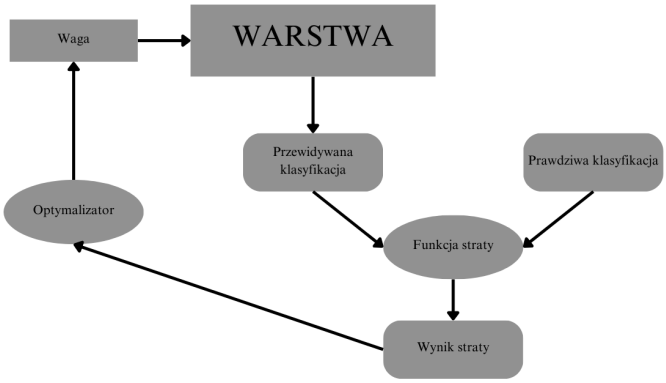


Rysunek 3.1: Przykład warstw modelu uczenia głębokiego - Opracowanie własne na podstawie [17].

Rys.3.1 pokazuje wizualną reprezentacje sieci neuronowej. Model importuje wartości cech z zestawu treningowego jako warstwę wejściową, następnie w warstwach ukrytych podczas trenowania tworzą się połączenia neuronowe, podobnie jak w mózgach organizmów żywych. Każda warstwa ukryta ma za zadanie wyodrębnić coraz bardziej skomplikowane i znaczące cechy z warstwy wejściowej. Po przejściu do warstwy wyjściowe model generuję odpowiedź lub przewidywanie zależnie do jakiego zadania został wytrenowany.

W prostym tłumaczeniu uczenie głębokie to tak naprawdę przypisywanie wartości do konkretnych cech, np.: do obrazka pokazującego pszczołę oznaczenie *pszczoła*, danym wcho-

dzącym do warstwy przypisuję się wagę, w uczeniu głębokim chodzi o to żeby przypisać wagi do wszystkich warstw tak żeby sieć dokonała poprawnego przewidywania.



Rysunek 3.2: Wyznaczanie wagi warstwy w uczeniu głębokim - Opracowanie własne na podstawie [18].

Na Rys.3.2 pokazano jak wygląda znajdowanie odpowiedniej wagi dla warstwy. Na początku wagi przypisywane są losowo, ale potem przewidywanie jest porównywane w funkcji straty z prawdziwą klasyfikacją i generowany jest wynik. Na postawie tego wyniku optymalizator dostosowuję wagę i przekazuję ją do warstwy. Czynność jest powtarza w każdej iteracji uczenia, aż do osiągnięcia zadowalającego wyniku.

Rozdział 4

Środowisko i narzędzia

4.1 Visual Studio Code i rozszerzenie Jupyter

Visual Studio Code [20]– darmowy edytor kodu źródłowego z kolorowaniem składni dla wielu języków, stworzony przez Microsoft, o otwartym kodzie źródłowym. Wspiera wiele języków, dodatkowo można rozszerzać możliwości edytora dzięki rozszerzeniom takim jak *Jupyter* [21].

Jest to interaktywne środowisko do programowania, analizy danych i tworzenia dokumentów zawierających kod, tekst oraz wyniki. Główną częścią *Jupyter* jest tzw. "notebook". Jest to dokument, który zawiera zarówno tekst, jak i kod. Może też zawierać multimedia, strony internetowe osadzone itd. Umożliwia *narracje kodu*. Można go używać do eksploracyjnej analizy danych, tworzenia interaktywnych prezentacji, pisania raportów, edukacji i wielu innych zastosowań.

4.2 Python i używane biblioteki

Python [22] - bardzo popularny, wysokopoziomowy język programowania, który wykorzystuję praktycznie w każdym aspekcie informatyki. Język jest znany ze swojej czytelności, prostoty składni, łatwości w nauce. pozwala na pisanie proceduralne, funkcyjne oraz obiektowe.

Python ma ogromną ilość bibliotek wprowadzonych do samego Pythona oraz napisanych przez społeczność programistyczną, pozwala na integrację z modułami napisanymi w językach takich jak C, co jest przydatne w przypadku wymagających obliczeniowo zadań.

Jest szeroko stosowany w wielu dziedzinach, takich jak rozwijanie oprogramowania internetowego, analiza danych, sztuczna inteligencja, uczenie maszynowe, automatyzacja.

- **Scikit-learn** [13] [24] - zapewniająca narzędzia do uczenia maszynowego, eksploracji danych i analizy statystycznej. Dostarcza szeroki zestaw algorytmów uczenia maszynowego, takich jak maszyny wektorów nośnych (SVM), drzewa decyzyjne, algorytmy k-najbliższych sąsiadów, regresja liniowa, regresja logistyczna i wiele innych.

- **Tensorflow** [14] [25] - otwarta biblioteka do uczenia maszynowego stworzona przez zespół Google Brain. Jest to narzędzie do tworzenia, trenowania i wdrażania modeli uczenia maszynowego, szczególnie w obszarze głębokiego uczenia.

- **Keras** [14] [26]- biblioteka używana w języku Python, dostarczająca wysokopoziomowe bloki do tworzenia modeli uczenia głębokiego uczenia, wykorzystuję się jaką jako nakładkę w tak zwanym stosie (*ang. stack*) oprogramowania razem z innymi bibliotekami takimi jak np.: Tensorflow.

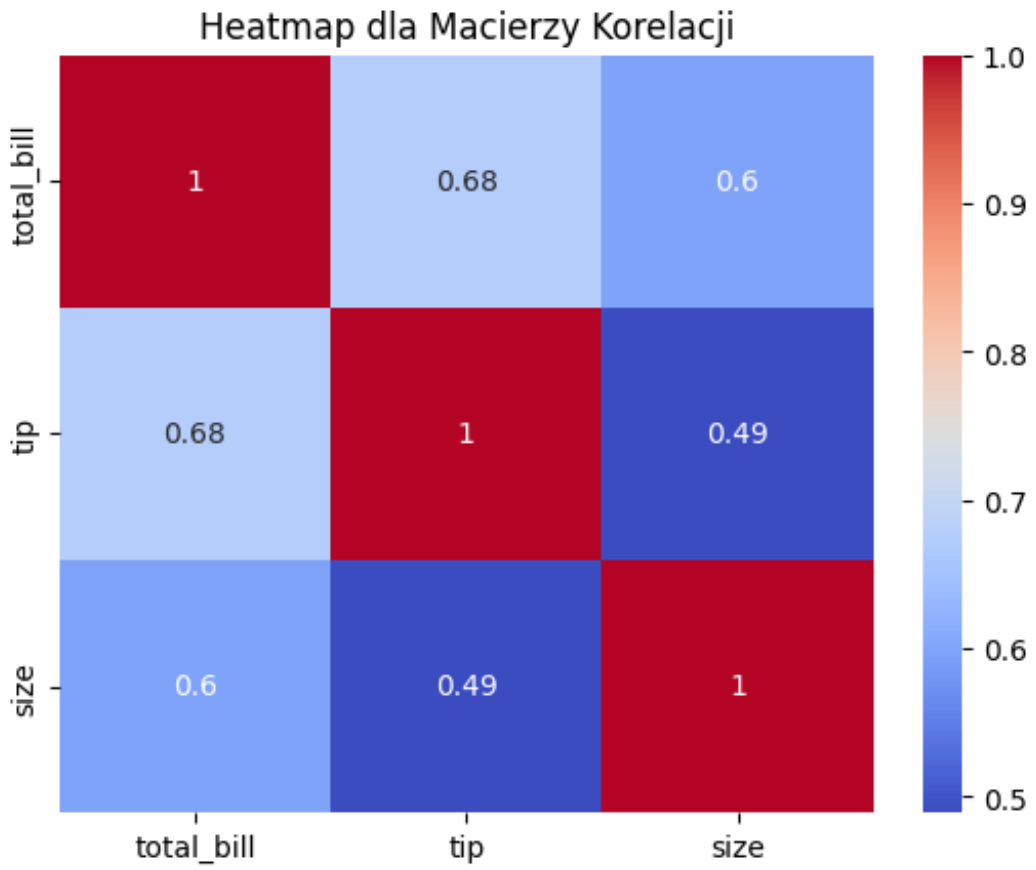
- **Matplotlib** [19] [27] - popularna biblioteka do tworzenia statycznych, interaktywnych i animowanych wykresów. Jest powszechnie używana w dziedzinach takich jak analiza danych, wizualizacja danych, eksploracja danych. Można ją wykorzystać do tworzenia wykresów 2D i 3D.
- **Seaborn** [28] - oparta na Matplotlib, biblioteka zapewniająca wysokopoziomowy interfejs do tworzenia atrakcyjnych i informacyjnych wykresów statystycznych. Często jest wykorzystywana do wizualizacji w analizaje danych oraz do prezentacji przeprowadzonej analizy. Przy użyciu kodu pokazanego na Rys.4.1, używając biblioteki Seaborn można stworzyć wykres ciepła (ang. heatmap), Rys.4.2dla macierzy korelacji, w tym wypadku przykładowego zestawu danych.

```
import seaborn as sns
import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

corr_matrix = tips.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Heatmap dla Macierzy Korelacji')
plt.show()
```

Rysunek 4.1: Kod ilustrujący mapę ciepła przykładowego zestawu danych przy użyciu biblioteki *Seaborn*.



Rysunek 4.2: Macierz korelacji przy użyciu mapy ciepła wykorzystując przykładowy zestaw danych z biblioteki Seaborn - Opracowanie własne.

Część II

Część praktyczna

Rozdział 5

Przegląd i działania na zbiorze danych

5.1 Analiza eksploracyjna

Pierwszym zadaniem jest import, analiza eksploracyjna oraz przygotowanie danych, które będą używane do uczenia modelu. Dobre dane są kluczowe dla skutecznego uczenia maszynowego. W tej pracy zestaw pochodzi z platformy **Kaggle** [29]

Kaggle to platforma dla pasjonatów analizy danych oraz uczenia maszynowego, badaczy i profesjonalistów, która oferuje różnorodne zbiory danych, które użytkownicy mogą eksplorować, analizować i wykorzystywać w swoich projektach.

Używane dane to zbiór właściwości kilkunastu tysięcy gwiazd zapisane w formacie *csv* Przy pomocy języka programowania Python oraz biblioteki *Padnas* dane zostają wczytane w środowisku *Visual Studio Code* z rozszerzeniem *Jupyter*.

Fragment kodu na Rys.5.1 importuję bibliotekę *Pandas* i nadaje jej alias *pd* dla wygody użytkowania w kodzie, następnie odczytuję i przechowuję dane w obiekcie typu *dataframe* w zmiennej o nazwie *data*. Ostatnia linia w skrawku pozwala wyświetlić dane w komórce pokazującej wyniki wykonanego kodu pokazane na Rys. 5.2

```
import pandas as pd
data = pd.read_csv('dataset/Star39552_balanced.csv')
data
```

Rysunek 5.1: Fragment kodu importujący biliotekę *Pandas* i przypisanie zmiennej data to importowanego pliku z danymi.

	Vmag	Plx	e_Plx	B-V	SpType	Amag	TargetClass
0	10.00	31.66	6.19	1.213	K7V	22.502556	1
1	8.26	3.21	1.00	1.130	K0III	15.792525	0
2	8.27	12.75	1.06	0.596	F9V	18.797552	1
3	6.54	5.23	0.76	1.189	K1III	15.132508	0
4	8.52	0.96	0.72	0.173	B8V	13.431356	1
...	...	...	...	...	...	...	...
39547	5.83	0.17	0.52	0.474	B7Iab	6.982245	0
39548	7.05	18.12	0.92	0.424	F5V	18.340790	1
39549	9.21	3.89	1.46	0.227	A1IV	17.159748	1
39550	9.01	2.13	1.46	1.467	M5III	15.651898	0
39551	9.12	3.82	0.79	0.480	F5V	17.030317	1

39552 rows × 7 columns

Rysunek 5.2: Komórka wynikowa pokazująca dane zapisane w zmiennej *data*.

W celu przeprowadzenia analizy eksploracyjnej na zmiennej *data* zostaje użyta metoda

data.info(), która służy do uzyskania podstawowych informacji o ramce danych (DataFrame). Na Rys.5.3 przestawiono wynik tej metody na używanym zbiorze. Wynik dostarcza nam

```
   data.info()
 ✓ 0.0s
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 39552 entries, 0 to 39551
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Vmag         39552 non-null  float64
 1   Plx          39552 non-null  float64
 2   e_Plx        39552 non-null  float64
 3   B-V          39552 non-null  float64
 4   SpType       39552 non-null  object
 5   Amag         39552 non-null  float64
 6   TargetClass  39552 non-null  int64
dtypes: float64(5), int64(1), object(1)
memory usage: 2.1+ MB
```

Rysunek 5.3: Komórka z kodem oraz wynikowa metody *data.info()*.

informacji o ilości i nazwie kolumn, w tym przypadku 39552 indywidualne gwiazdy, a także typie danych jakie te kolumny zawierają. Dostarczona też jest informacja o ilości indywidualnych wpisów w tabeli oraz zużycie pamięci na komputerze przez zbiór danych. Nazwy kolumn w zbiorze oznaczają:

- **Vmag - Visual magnitude**[1] - Wielkość wizualna - nazywana też jasnością widmową, jest to miara jasności gwiazdy widzianej z Ziemi.
- **Plx - Distance to Earths**[1] - Odległość do Ziemi, wyrażona w paralaksie trygonometrycznej.
- **e_Plx - Standard error of Plx** - Błąd standardowy paralaksy.
- **B-V - Color Index** - Indeks Barwowy - miara koloru gwiazdy. Jest to różnica między jej jasnością w niebieskim (B) a widzialnym (V) zakresie.
- **SpType - Spectral Type**[1] - Typ Spektralny - To klasyfikacja gwiazdy oparta na cechach obecnych w jej spektrum.
- **Amag - Absolute magnitude**[1] - Wielkość Absolutna - wielkość pozorna gwiazdy, jaką miałaby, gdyby była oddalona od Ziemi o 10 parseków (32,6 lat świetlnych).
- **TargetClass** - Klasa Docelowa - mówi o tym czy dana gwiazda jest Karłem (0) czy Gigantem (1).

5.2 Wstępne opracowanie danych

W większości przypadków w uczeniu maszynowym dane są reprezentowane jako liczby zmiennoprzecinkowe (float64) ze względu na potrzebę precyzyjnego obliczania wartości wag, gradientów i innych parametrów w trakcie treningu modelu. Dane w kolumnie *SpType* są wyrażone jako obiekt, z tego względu nie będą brane pod uwagę przy tworzeniu i trenowaniu modeli. Poniższy fragment kodu pokazany na Rys.5.4 przestawia procedurę usunięcia.

```
data = data.drop('SpType', axis=1)
```

Rysunek 5.4: Fragment kodu usuwający kolumnę *SpType*.

Następnym krokiem jest sprawdzenie czy zbiór danych posiada puste wartości i zduplikowanie wiersze, przy użyciu kodu pokazanego na Rys.5.5. Pierwsza linia zwraca liczbę brakujących wartości (*NaN*) w każdej kolumnie. Druga linia zwraca *True*, jeśli istnieją zduplikowane wiersze.

```
# Sprawdzanie czy dane nie maja wartosci null/Nan
print(f'Wartosci null w kolumach?\n{data.isnull().sum()}')

# Sprawdanie czy wystepuja duplikaty
print(f'Duplikaty w kolumnach?\n{data.duplicated().any()}')
```

Rysunek 5.5: Fragment kodu sprawdzający czy w zestawie danych istnieją duplikaty i puste wartości.

```
Wartości null w kolumach?
Vmag           0
Plx            0
e_Plx          0
B-V            0
Amag           0
TargetClass    0
dtype: int64
```

(a) Komórka wynikowa dla sprawdzania czy występują puste wartości.

```
Duplikaty w kolumnach?
False
```

(b) Komórka wynikowa dla sprawdzania duplikatów

Rysunek 5.6: Komórki wynikowe dla dwóch linii kodu w celu walidacji zbioru danych.

Na Rys. 5.6a można zobaczyć, że żadna kolumna w zbiorze danych nie zawiera pustych lub brakujących wartości. Tak samo Rys. 5.6b pokazuję że nie występują duplikaty wierszy.

5.3 Przygotowanie danych do uczenia maszynowego

Po analizie i odpowiednim sprawdzeniu czy zbiór danych będzie dobrze działał z algorytmami uczenia maszynowego należy przejść do przygotowania danych do wykorzystania w uczeniu.

```
# Dzielenie danych
x = data.drop("TargetClass", axis=1).values
y = data['TargetClass'].values
```

Rysunek 5.7: Fragment kodu odpowiedzialny za podział danych na właściwości gwiazdy oraz na klasę docelową.

```
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=73)
```

Rysunek 5.8: Fragment kodu odpowiedzialny za podział danych na zestawy treningowe oraz testowe.

Zaczynając od oddzielenia kolumny *TargetClass* od całego zbioru. Rys. 5.7pokazuje fragment kodu, który dzieli zbiór danych na dane przestawiające właściwości gwiazdy oraz na docelową klasę gwiazdy.

Następnie za pomocą kodu przestawionego na na Rys.5.8 dane dzielne są na zestaw treningowy oraz testowy używając funkcji *train_test_split* importowaną z modułu *sklearn.model_selection*. Funkcja ta dzieli dane na dwa zestawy - treningowy i testowy.

Argumenty:

- x: Dane wejściowe.
- y: Etykiety lub wyniki, które chcemy przewidywać.
- test_size: Określa, jaki procent danych zostanie przypisany do zestawu testowego (w tym przypadku 25%).
- random_state: Ziarno generatora liczb pseudolosowych. Ustawiając to ziarno, można uzyskać tę samą podział danych przy każdym uruchomieniu programu, co jest przydatne do reprodukowalności wyników.

Zwraca wartości:

- x_train: Dane wejściowe dla zestawu treningowego.
- x_test: Dane wejściowe dla zestawu testowego.
- y_train: Etykiety dla zestawu treningowego.
- y_test: Etykiety dla zestawu testowego.

Kolejnym krokiem jest użycie *StandardScaler*, który jest używany do standaryzacji danych, co może poprawić wydajność algorytmów, które są wrażliwe na różnice w skali danych. *StandardScaler* przekształca dane w taki sposób, że średnia każdej cechy wynosi 0, a odchylenie standardowe wynosi 1. Fragment kodu na Rys.5.9 przestawia proces standaryzacji na używanym zestawie danych.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)
```

Rysunek 5.9: Fragment kodu standaryzujący zestawy danych.

Pierwsza linia importuję obiekt *StandardScaler* z modułu *sklearn.preprocessing*, następnie tworzony jest obiekt w zmiennej o nazwie *scaler* oraz jest dostosowywany do danych treningowych. W trakcie tego dopasowania, obliczane są średnia i odchylenie standardowe dla każdej cechy w zestawie treningowym. Potem dane treningowe i testowe są przekształcane przy użyciu dopasowanego scaler'a. Przekształcenie to polega na odjęciu średniej i podzieleniu przez odchylenie standardowe.

Tak przygotowane dane można użyć w implementacji algorytmów uczenia maszynowego i głębokiego.

Rozdział 6

Implementacja uczenia maszynowego

6.1 Tworzenie funkcja do tworzenia i trenowania modeli

W celu zapewniania możliwe podobnych warunków do porównania modeli oraz żeby ułatwić i uprościć kod źródłowy została stworzona funkcja która wykonuję operacje tworzenia, trenowania, optymalizacji oraz testowania wielu modeli.

```
from sklearn.model_selection import GridSearchCV

def create_train_model(name:str, model, params:any):
    print(f'# Przetwarzanie modelu... {name}')

    current_model = model

    # Przeszukiwanie siatki w celu znalezienia najlepszych parametrow
    model_grid = GridSearchCV(current_model, params, cv=5, n_jobs=-1)
    model_grid.fit(x_train, y_train)
    print(f'# Najlepsze parametry:\n{model_grid.best_params_}')

    # Testowanie modelu na danych testowych
    y_pred = model_grid.predict(x_test)

    # Ewaulacja skutecznosci najlepszego modelu
    eval_perf_model(y_test, y_pred)
    gen_classification_report(y_test, y_pred)
    print('-----------------------------------')

    return model_grid
```

Rysunek 6.1: Kod przedstawiający funkcje odpowiedzialną za tworzenie, trenowanie, optymalizacje i testowanie algorytmów uczenia maszynowego

Na Rys.6.1 kod przestawia funkcje *create_train_model*, która przyjmuję 3 argumenty: *name, model, params*, odpowiednio są to nazwa - dowolny tekst, który reprezentuje nazwę modelu, model - obiekt z biblioteki *sklearn*, który reprezentuje model oraz parametry - są to parametry które są przekazywane do konkretnego modelu, zależne od wybranego algorytmu.

Do znalezienia najlepszych hiperparametrów wykorzystuję się funkcję *GridSearchCV* importowaną z modułu *sklearn.model_selection*, następnie model dokonuję przewidywanie na

zestawie testowym. Dane z przewidywaniami są wykorzystane do ewaluacji algorytmu za pomocą dwóch dodatkowych funkcji opisanych w rozdziale 6.2. Na koniec sama funkcja zwraca wytrenowany model, w celu dalszego testowania lub dalszej implementacji

6.2 Funkcje do oceny wydajności modelu

W celu oceny wydajności modelu oraz wygenerowania raportu klasyfikacji zostały napisane dwie funkcje odpowiedzialne za te operacje pokazane na Rys.6.2.

```
# Ewaluacja i raporty
from sklearn.metrics import accuracy_score, precision_score, recall_score
    , f1_score
from sklearn.metrics import classification_report

# Ocena wydajnosci modelu
def eval_perf_model(y_test_data, y_pred_data):
    print('# Wydajnosc testowanego modelu:')
    acc_score = accuracy_score(y_test_data, y_pred_data)
    print('Accuracy:\t', acc_score)
    prec_score = precision_score(y_test_data, y_pred_data)
    print('Precision:\t', prec_score)
    rec_score = recall_score(y_test_data, y_pred_data)
    print('Recall:\t\t', rec_score)
    fscore = f1_score(y_test_data, y_pred_data)
    print('F1:\t\t', fscore)

    return acc_score, prec_score, rec_score, fscore

# Generowanie raportu klasyfikacji
def gen_classification_report(y_test_data, y_pred_data):
    target_names = ['Class 0 - D', 'Class 1 - G']
    report = classification_report(y_test_data, y_pred_data, target_names
        =target_names)
    print (f'# Raport klasyfikacji:\n{report}')

    return report
```

Rysunek 6.2: Kod przedstawiający funkcje odpowiedzialne za generowanie metryki oraz raportu klasyfikacji.

Pierwsza funkcja *eval_perf_model* przyjmuję dwa argumenty: docelowe klasy z zestawu testowego oraz wartości przewidziane przez model używając zestawu testowego. Używając modułu *sklearn.metrics* funkcja generuję metryki opisane w rozdziale XX, które są stosowane do oceny wydajności modelu.

Funkcja *gen_classification_report* przyjmuje dokładnie te same argumenty jak funkcja *eval_perf_model* i odpowiada za wygenerowanie raportu klasyfikacji. Raport ten zawiera różne miary i informacje dotyczące wydajności modelu. Przykładowy wynik został zaprezentowany na Rys.6.3

```
              precision    recall  f1-score   support

           0       0.85      0.90      0.88       200
           1       0.78      0.69      0.73       100

    accuracy                           0.83       300
   macro avg       0.82      0.80      0.81       300
weighted avg       0.83      0.83      0.83       300
```

Rysunek 6.3: Przykładowa ocena modelu uczenia maszynowego.

Metryki są podzielone na odpowiednie klasy, dodatkowo kolumna *support* to liczba przypadków w danej klasie. *Macro avg* (pl. Średnia makro) to średnia z miar dla wszystkich klas, która traktuje każdą klasę równo, niezależnie od liczby obserwacji w każdej klasie. *Weighted avg* to średnia ważona z miar, gdzie waga to liczba przypadków w danej klasie.

6.3 Las losowy

Fragment kodu przestawiony na Rys.6.4 implementacje lasu losowego, pod zmienną *rf_params* kryje się zestaw hiperparametrów, które będą testowane podczas trenowania modelu przy użyciu przeszukiwaniu siatki.

```
from sklearn.ensemble import RandomForestClassifier
rf_params = {
    'n_estimators': (100, 200, 300),
    'max_depth': (2, 3),
    'criterion': ('gini', 'entropy'),
}

rf = create_train_model('Las losowy', RandomForestClassifier(), rf_params
    )
```

Rysunek 6.4: Implementacja algorytmu lasu losowego.

Na Rys.6.5 pokazano wyniki metryk używane przy testowaniu modelu oraz wybrane hiperparametry, które uzyskały najlepszy wynik dla danego modelu. Dzięki raportowi klasyfikacji można zauważyć, ze model lepiej sobie radzi z klasyfikacją klasy 0 czyli gwiazd karłowatych.

```
# Przetwarzanie modelu... Las losowy
# Najlepsze parametry:
{'criterion': 'gini', 'max_depth': 3, 'n_estimators': 200}
# Wydajnosc testowanego modelu:
Accuracy:    0.8785396440129449
Precision:   0.8391978122151322
Recall:              0.9351889475822837
F1:          0.8845969059287019
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.93      0.82      0.87      4966
 Class 1 - G       0.84      0.94      0.88      4922

    accuracy                           0.88      9888
   macro avg       0.88      0.88      0.88      9888
weighted avg       0.88      0.88      0.88      9888

-----------------------------------
```

Rysunek 6.5: Komórka wynikowa dla algorytmu lasu losowego.

6.4 Drzewo decyzyjne

Na Rys6.6 przestawiono implementacje oraz hiperparametry które będą testowe dla modelu drzewa decyzyjnego. Na Rys.6.7 przestawiono wyniki oraz raport klasyfikacji oraz parametry z najlepszymi wynikami. W porównaniu do Lasu losowego precyzja dla obydwu typów gwiazd jest zbliżona.

```
from sklearn.tree import DecisionTreeClassifier
dt_params = {
    'max_depth': (1,2,3,),
    'criterion': ('gini','entropy')
}

# Funkcja do Tworzenia modelu
dr = create_train_model('Drzewo decyzyjne', DecisionTreeClassifier(),
    dt_params)
```

Rysunek 6.6: Implementacja algorytmu drzewa decyzyjnego.

```
# Przetwarzanie modelu... Drzewo decyzyjne
# Najlepsze parametry:
{'criterion': 'gini', 'max_depth': 3}
# Wydajnosc testowanego modelu:
Accuracy:         0.8727750809061489
Precision:        0.8441021788129226
Recall:           0.9130434782608695
F1:               0.8772203786843645
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.91      0.83      0.87      4966
 Class 1 - G       0.84      0.91      0.88      4922

    accuracy                           0.87      9888
   macro avg       0.88      0.87      0.87      9888
weighted avg       0.88      0.87      0.87      9888

-----------------------------------
```

Rysunek 6.7: Komórka wynikowa dla algorytmu Drzewa decyzyjnego

6.5 K-Najbliższych sąsiadów

Kod przedstawiony na Rys.6.8 implementuje K-Najbliższych sąsiadów oraz hierparametry które będą testowe.

```
from sklearn.neighbors import KNeighborsClassifier
knn_params = {
    'n_neighbors': [2, 3, 4, 5, 6, 7],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

knn = create_train_model('K-Najblizszych sasiadow', KNeighborsClassifier
    (), knn_params)
```

Rysunek 6.8: Implementacja algorytmu K-Najbliższych sąsiadów.

Na Rys.6.9 przestawiono parametry które zostały wybrane metodą przeszukiwania siatki żeby osiągnąć najlepsze wyniki. Podobnie jak przypadku drzewa decyzyjnego model radzi sobie bardzo dobrze w klasyfikacji obydwu rodzajów gwiazd.

```
# Przetwarzanie modelu... K-Najblizszych sasiadów
# Najlepsze parametry:
{'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'uniform'}
# Wydajnosc testowanego modelu:
Accuracy:         0.8741909385113269
Precision:        0.8624359479700433
Recall:           0.8890694839496139
F1:               0.8755502200880352
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.89      0.86      0.87      4966
 Class 1 - G       0.86      0.89      0.88      4922

    accuracy                           0.87      9888
   macro avg       0.87      0.87      0.87      9888
weighted avg       0.87      0.87      0.87      9888

-----------------------------------
```

Rysunek 6.9: Komórka wynikowa dla algorytmu K-Najbliższych sąsiadów.

6.6 Maszyny wektorów nośnych

Na Rys.6.10 przestawiono fragment kodu, który odpowiada za implemetację algorytmu Maszyn wektorów nośnych oraz hiperparametry, które będą sprawdzane metodą przeszukiwania siatki.

```
from sklearn.svm import SVC
svm_params = {
'kernel' : ('linear', 'rbf'),
"gamma": ('scale', 'auto')
}

# Funkcja do Tworzenia modelu
svm = create_train_model('Maszyny wektorów śnonych', SVC(), svm_params)
```

Rysunek 6.10: Implementacja algorytmu Maszyn wektorów nośnych

Na Rys.6.11 można zobaczyć, że najlepsze parametry to *gamma* = *auto* oraz *kernel* = *rbf*. Model SVM poradził sobie z klasyfikacją zbioru bardzo dobrze wszystkie mierzone metryki są jedne z najwyższych.

```
# Przetwarzanie modelu... Maszyny wektorow nosnych
# Najlepsze parametry:
{'gamma': 'auto', 'kernel': 'rbf'}
# Wydajnosc testowanego modelu:
Accuracy:         0.8831917475728155
Precision:        0.8546413104876671
Recall:           0.9221861032100772
F1:               0.8871298739372618
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.92      0.84      0.88      4966
 Class 1 - G       0.85      0.92      0.89      4922

    accuracy                           0.88      9888
   macro avg       0.89      0.88      0.88      9888
weighted avg       0.89      0.88      0.88      9888

-----------------------------------
```

Rysunek 6.11: Komórka wynikowa dla algorytmu Maszyn wektorów nośnych.

6.7 Perceptron

```
from sklearn.linear_model import Perceptron
pr_params = {
    'alpha': [0.000001,0.00001,0.0001],
    'penalty': ['l1', 'l2']
}
# Funkcja do Tworzenia modelu
pr = create_train_model('Perceptron', Perceptron(), pr_params)
```

Rysunek 6.12: Implementacja algorytmu Perceptronu.

Na Rys.6.12 zaprezentowano kod wykorzystywany do stworzenia, trenowania i oceny algorytmu Perceptronu. Kod wybierze najlepsze parametry ze zmiennej *pr_params* dzięki funkcji przeszukiwania siatki.

```
# Przetwarzanie modelu... Perceptron
# Najlepsze parametry:
{'alpha': 0.0001, 'penalty': 'l1'}
# Wydajnosc testowanego modelu:
Accuracy:         0.7487864077669902
Precision:        0.6718353538201297
Recall:           0.9683055668427468
F1:               0.7932756324900134
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.94      0.53      0.68      4966
 Class 1 - G       0.67      0.97      0.79      4922

    accuracy                           0.75      9888
   macro avg       0.81      0.75      0.74      9888
weighted avg       0.81      0.75      0.74      9888

-----------------------------------
```

Rysunek 6.13: Komórka wynikowa dla algorytmu Perceptronu.

Rys.6.13 przestawia wyniki dla modelu perceptronu, jak można zauważyć model bardzo dobrze radzi sobie z klasyfikacją klasy 0 czyli gwiazd karłowatych, niestety bardzo słabo radzi sobie przy klasyfikacji gigantów. W porównaniu do innych testowanych modeli perceptron ma najwyższą czułość.

6.8 Walidacja krzyżowa

Wykorzystując walidację krzyżową, opisaną w rozdziale 2.3.1 można zminimalizować ryzyko przetrenowania modelu do konkretnego zestawu danych, Na Rys.6.14 fragment kodu przestawia implementacje dla testowanych algorytmów. Kod przetestuję każdy algorytm oraz wyświetli metrykę dokładności dla każdego modelu.

```
from sklearn.model_selection import KFold, cross_val_score

models = [
    ('Las losowy', rf),
    ('Drzewo Decyzyjne', dr),
    ('K-Naj.Sasiad.', knn),
    ('Maszyn.wkt.nosn.', svm),
    ('Perceptron', pr)
]
kfold = KFold(n_splits=5, shuffle=True, random_state=73)
print('# Algorytm #\t# Dokladnosc #\t# Odchyl. standar. #')
for name, model in models:
    scores = cross_val_score(model, x_train, y_train, cv=kfold, scoring='
        accuracy', n_jobs=-1)
    score = scores.mean()
     std_dev = np.std(scores)
    list_scores.append([model, score])
    print(f"{name}:\t{score}\t{std_dev}")
```

Rysunek 6.14: Kod implementujący walidację krzyżową na wytrenowanych modelach uczenia maszynowego.

Wyniki walidacji krzyżowej przestawiono na Rys.6.15, jak można zauważyć najlepiej poradził sobie algorytm Maszyn wektorów nośnych.

```
# Algorytm #            # Dokladnosc #         # Odchyl. standar. #
Las losowy:             0.8788430972108301     0.00379475834920271
Drzewo Decyzyjne:       0.8713256788919285     0.0045439058269208885
K-Naj.Sasiad.:          0.8744608399094451     0.006036683290974837
Maszyn.wkt.nosn.:       0.8805960501391181     0.005702562575630652
Perceptron:             0.8585496972884101     0.011509833070125227
```

Rysunek 6.15: Komórka wynikowa dla walidacji krzyżowej.

6.9 Porównanie wyników uczenia nadzorowanego

Po trenowaniu i testowaniu modeli na tym samym zestawie danych, można przejść do porównania i wybrania najlepszego modelu. Na Rys.6.16 pokazano tablice pomyłek dla każdego użytego modelu. Tabela 6.1 pokazuję metryki ze wszystkich trenowanych modeli, wyniki są bardzo zbliżone dla Lasu losowego oraz Maszyn wektorów nośnych. W tym przypadku Perceptron wyszedł najgorzej.

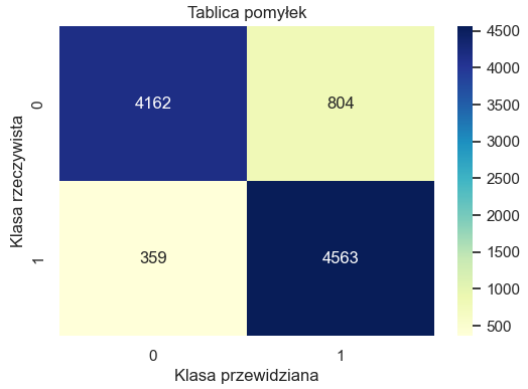
Porównując wyniki modeli uczenia nadzorowanego Las losowy oraz Maszyny wektorów nośnych osiągnęły podobne wyniki. Jednak na Rys.6.15 widać, że model Maszyn wektorów nośnych poradził sobie nieco lepiej przy walidacji krzyżowej. Dokładne wyniki przestawiono w Tab.6.2.

Metryka	Używany Model				
	Las losowy	Drzewo decyz.	K-Najb.sąsid.	Maszyn.wkt.nośn.	Perceptron
Dokładność	0.88	0.87	0.87	0.88	0.75
Precyzja	0.85	0.84	0.86	0.85	0.67
Czułość	0.93	0.91	0.89	0.92	0.97
Miara F1	0.89	0.88	0.88	0.89	0.79

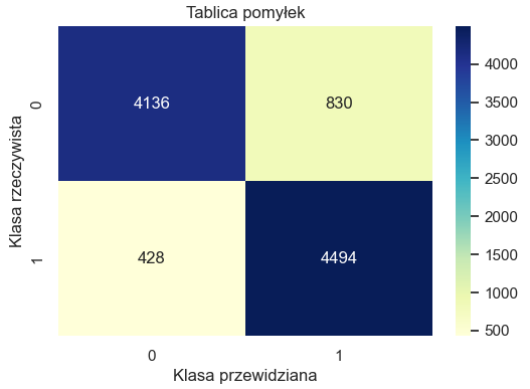
Tabela 6.1: Wyniki trenowania i testowania używanych modeli.

Używany Model	Dokładność	Odchylenie standardowe
Las losowy	0.8788	0.0038
Drzewo decyzyjne	0.8713	0.0045
K-Najb.sąsid.	0.8745	0.0060
Maszyn.wkt.nośn.	0.8806	0.0057
Perceptron	0.8585	0.0115

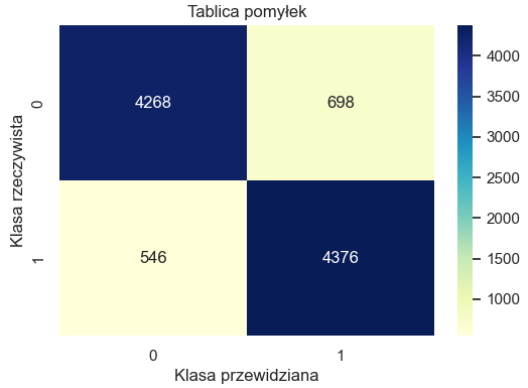
Tabela 6.2: Wyniki walidacji krzyżowej dla trenowanych modeli.



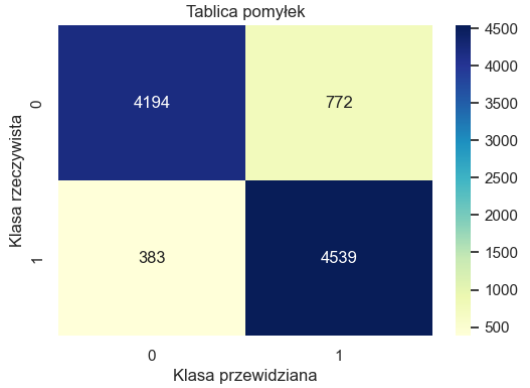
(a) Tablica pomyłek dla Lasu losoweg.o



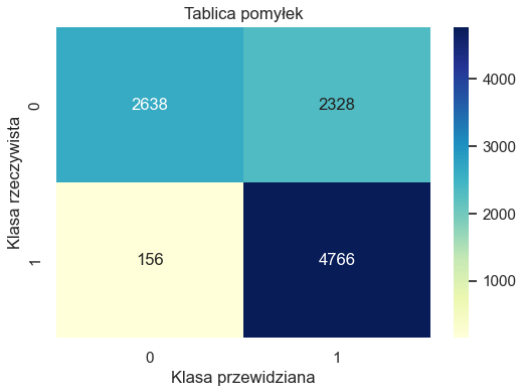
(b) Tablica pomyłek dla Drzewa decyzyjneg.o



(c) Tablica pomyłek dla K-Najbliższych sąsiadów.



(d) Tablica pomyłek dla Maszyny wektorów nośnych.



(e) Tablica pomyłek dla Percetronu.

Rysunek 6.16: Tablice pomyłek dla modeli uczenia nadzorowanego.

6.10 Przewidywanie na nowym zestawie danych

Używając wytrenowanego modelu Maszyn wektorów nośnych została przeprowadzone przewidywania na drugim, mniejszym zestawie danych. Fragment kodu przedstawiony na Rys.6.17 pokazuję import oraz przygotowanie danych do działa z modelem. Dane zostały podzielone, podobnie jak w przypadku danych treningowych, na parametry opisujące gwiazdy oraz na etykiety klasyfikujące te gwiazdy.

```
new_data = pd.read_csv('dataset/Star3642_balanced.csv')

# new_data.info()
new_data = new_data.drop('SpType', axis=1)

# Sprawdzanie czy dane nie ąmaj świartoci null/Nan
print(f'śWartoci null w kolumach?\n{data.isnull().sum()}')

# Sprawdanie czy ęąwystpuj duplikaty
print(f'Duplikaty w kolumnach?\n{data.duplicated().any()}')

x_new = new_data.drop("TargetClass", axis = 1).values
y_new = new_data['TargetClass'].values

# Normalizacja
scaler.fit(x_new)
x_new = scaler.transform(x_new)
```

Rysunek 6.17: Fragment kodu przygotowujący nowy zestaw danych do testowania modeli.

Na Rys.6.18 przestawiono fragment kodu po wykonaniu dokonuję przewidywania na nowym zestawie danych oraz generuję podsumowanie, podobne do tych używanych przy porównywaniu modeli.

```
model = svm
print('Przewidywanie na nowym zestawie danych...')
y_new_pred = model.predict(x_new)

eval_perf_model(y_new, y_new_pred)
gen_classification_report(y_new, y_new_pred)
```

Rysunek 6.18: Fragment kodu implementujący przewidywanie oraz ocenę modeli na nowych danych.

```
Przewidywanie na nowym zestawie danych...
# Wydajnosc testowanego modelu:
Accuracy:         0.899505766062603
Precision:        0.8943089430894309
Recall:           0.9060955518945635
F1:               0.900163666121113
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.90      0.89      0.90      1821
 Class 1 - G       0.89      0.91      0.90      1821

    accuracy                           0.90      3642
   macro avg       0.90      0.90      0.90      3642
weighted avg       0.90      0.90      0.90      3642
```

Rysunek 6.19: Wyniki przewidywania dla nowego zestawu danych.

Rys.6.19 pokazuję metrykę przewidywania algorytmu Maszyn wektorów nośnych na no-

wym zestawie danych. Algorytm bardzo dobrze poradził sobie z zadaniem, 90% przypadków została sklasyfikowana poprawnie, jest to bardzo dobry wynik.

Rozdział 7

Implementacja głębokiego uczenia

Rys.7.1 pokazuję implementację głębokiego uczenia.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

from tensorflow.keras.metrics import Accuracy, Precision, Recall

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(5,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy', Precision(), Recall()])

model.summary()

model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split
    =0.2)

y_test_pred = model.predict(X_test)
y_test_pred_binary = (y_test_pred > 0.5).astype(int)

eval_perf_model(y_test, y_test_pred_binary)
gen_classification_report(y_test, y_test_pred_binary)
```

Rysunek 7.1: Implementacja głębokiego uczenia w języku Python.

Przy wykorzystaniu biblioteki Keras (opisania w Roz.4) stworzono model sieci neuronowej, pierwszą warstwą jest warstwa wejściowa która ma 64 neurony oraz przyjmuję dane z zestawu danych, w tym przypadku jest to 5 cech gwiazdy. Następnie są dwie warstwy ukryte które mają kolejno 32 oraz 16 neuronów.

Ostatnią warstwą jest warstwa wyjściowa, która ma tylko jeden neuron z aktywatorem *sigmoid* który jest najbardziej odpowiedni do zadania klasyfikacji binarnej.

```
model. compile ( optimizer ='adam ', loss='binary_crossentropy ', metrics
    =[ 'accuracy ', Precision (), Recall ()])
```

Rysunek 7.2: Kod odpowiedzialny za kompilacje modelu sieci neuronowej.

Na Rys.7.2 pokazano fragment kodu jest odpowiedzialny za kompilowanie modelu, jako funkcje stary wykorzystuję się *binary_crossentropy*, która jest odpowiednia do klasyfikacji binarnej. Podczas trenowania modelu zbierane są metryki takie jak: dokładność, precyzja, czułość.

Następnie model jest trenowany i dokonywane są przewidywania na zestawie testowym, a wyniki konwertowane to postaci całkowitej (*ang.integer*).

Na końcu model jest oceniany przy użyciu funkcji oceny trenowanych modeli (opisane w Roz.6.2), wyniki ewaluacji przestawiono na Rys.7.3.

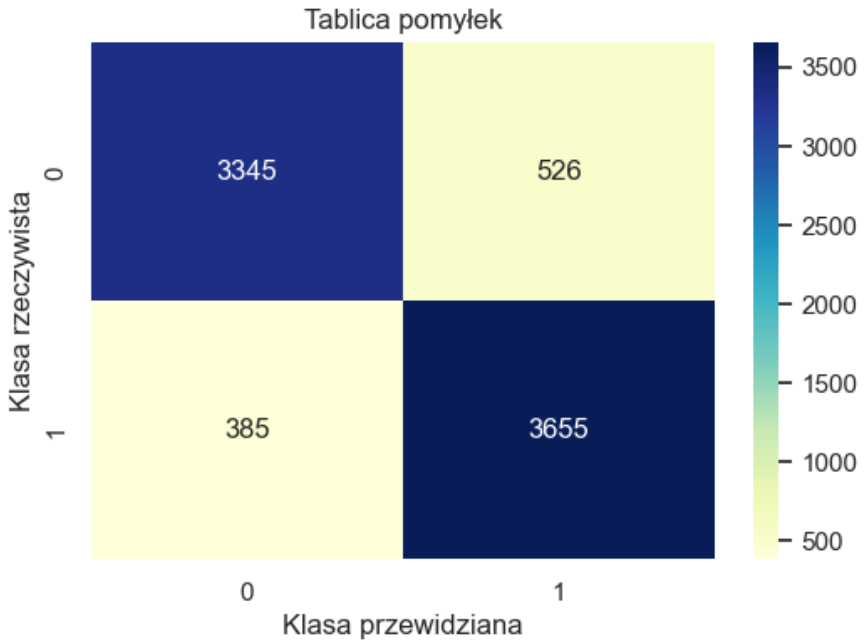
```
# Wydajnosc testowanego modelu:
Accuracy:         0.8848438882568576
Precision:        0.8741927768476441
Recall:           0.9047029702970297
F1:               0.8891862303855979
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.90      0.86      0.88      3871
 Class 1 - G       0.87      0.90      0.89      4040

    accuracy                           0.88      7911
   macro avg       0.89      0.88      0.88      7911
weighted avg       0.89      0.88      0.88      7911
```

Rysunek 7.3: Ewaluacja i raport klasyfikacji modelu uczenia głębokiego.

Rys.7.4 przedstawia tablice pomyłek dla modelu uczenia głębokiego.



Rysunek 7.4: Tablica pomyłek dla uczenia głębokiego.

Następnie dokonano przewidywania na nowym zestawie danych, wyniki na Rys.7.5

```
# Wydajnosc testowanego modelu:
Accuracy:        0.899780340472268
Precision:       0.8969465648854962
Recall:          0.9033498077979132
F1:              0.9001367989056087
# Raport klasyfikacji:
              precision    recall  f1-score   support

 Class 0 - D       0.90      0.90      0.90      1821
 Class 1 - G       0.90      0.90      0.90      1821

    accuracy                           0.90      3642
   macro avg       0.90      0.90      0.90      3642
weighted avg       0.90      0.90      0.90      3642
```

Rysunek 7.5: Wyniki ewaluacji przewidywania na nowym zestawie danych.

Jak można zauważyć model poradził sobie bardzo dobrze, szczególnie na nowym zestawie danych. Wszystkie mierzone metryki są na poziomie 90%, co jest bardzo zadowalającym wynikiem.

Rozdział 8

Podsumowanie i wnioski

Wykorzystując techniki uczenia maszynowego i głębokiego zostały stworzone modele do klasyfikacji gwiazd na olbrzymy lub karły wykorzystując dane z platformy *Kaggle*. Zostało przetestowanych 5 algorytmów uczenia maszynowego nadzorowanego oraz została stworzona siec neutronowa przy użyciu uczenia głębokiego.

Do znalezienia najlepszych hiperparametrów wykorzystano metodę przeszukiwania siatki używając funkcji *GridSearchCV* będącą, oraz przeprowadzono walidacje krzyżową dla każdego wytrenowanego modelu.

Spośród algorytmów uczenia nadzorowanego Las losowy oraz Maszyny wektorów nośnych osiągnęły najlepsze, ale też zbliżone wyniki. Później wykorzystano drugi, mniejszy zestaw danych do weryfikacji wytrenowanego modelu Maszyny wektorów nośnych, który osiągnął zadowalające wyniki.

Wykorzystując techniki uczenia głębokiego stworzono model sieci neuronowej, która została wytrenowana na tym samym zestawie danych, oraz przeprowadzono weryfikacje na drugim zestawie danych.

Uczenie maszynowe nadzorowane oraz głębokie przyniosły bardzo podobne wyniki, modele potrafią klasyfikować gwiazdy z bardzo wysoką dokładnością i precyzją. Wyniki uzyskane przez modele są bardzo zadowalające, szczególnie modele uczenia nadzorowanego Maszyn wektorów nośnych, Lasu losowego oraz sieć neuronowa, wszystkie wymienione potrafią bardzo dobrze klasyfikować gwiazdy na olbrzymy lub karły na podstawie ich właściwości fizycznych.

Bibliografia

[1] Caroll, B.W., Ostlie, D.A *An Introduction to Modern Astrophysics*, Pearson Education, 2007

[2] Marcin Kubiak, *Gwiazdy i materia międzygwiazdowa*, Wydawnictwo Naukowe PWN, 1994

[3] http://www.atlasoftheuniverse.com/hr.html (Dostęp 2024-02-12)

[4] https://centrumnaukiec1.pl/aktualnosci/paralaksa——z-kciukiem-w-gwiazdy (Dostęp 2024-02-12)

[5] Sebastian Raschka, *Python Machine Learning*, wydanie trzecie, Packt Publishing, 2016

[6] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (Dostęp 2024-02-12)

[7] https://scikit-learn.org/stable/modules/tree.html (Dostęp 2024-02-12)

[8] https://www.ibm.com/topics/decision-trees (Dostęp 2024-02-12)

[9] Aleksandra Król-Nowak, Katarzyna Kotarba, *Podstawy uczenia maszynowego*, Wydawnictwa AGH, 2022

[10] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html (Dostęp 2024-02-12)

[11] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC (Dostęp 2024-02-12)

[12] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html (Dostęp 2024-02-12)

[13] Andreas C. Müller & Sarah Guido, *Introduction to Machine Learning with Python*, O'Reilly Media, 2017

[14] François Chollet, *Deep Learning with Python*, Manning Publications Co, 2018

[15] https://scikit-learn.org/stable/modules/linear_model.html#perceptron (Dostęp 2024-02-12)

[16] Stephen Marsland, *MACHINE LEARNING An Algorithmic Perspective*, wydanie drugie, CRC Press Taylor & Francis Group, 2015

[17] https://cdn.bulldogjob.com/system/photos/files/000/006/189/original/image6.png (Dostęp 2024-02-12)

[18] https://drek4537l1klr.cloudfront.net/chollet/Figures/01fig09.jpg (Dostęp 2024-02-12)

[19] Wes McKinney, *Python for Data Analysis*, O'Reilly Media, 2012

[20] https://code.visualstudio.com/ (Dostęp 2024-02-12)

[21] https://jupyter.org/ (Dostęp 2024-02-12).

[22] https://www.python.org/ (Dostęp 2024-02-12)

[23] https://pandas.pydata.org/ (Dostęp 2024-02-12)

[24] https://scikit-learn.org/stable/ (Dostęp 2024-02-12)

[25] https://www.tensorflow.org/ (Dostęp 2024-02-12)

[26] https://keras.io/ (Dostęp 2024-02-12)

[27] https://matplotlib.org/ (Dostęp 2024-02-12)

[28] https://seaborn.pydata.org/ (Dostęp 2024-02-12)

[29] https://www.kaggle.com/datasets/vinesmsuic/star-categorization-giants-and-dwarfs (Dostęp 2024-02-12)

Spis rysunków

Spis tabel

Dodatek A

Kod źródłowy projektu zastosowany w pracy

```
# ------------------ MACHINE LEARNING ------------------

# Import Library & Data
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
data = pd.read_csv('dataset/Star39552_balanced.csv')


# data.info()

data = data.drop('SpType', axis=1)


# Sprawdzanie czy dane nie ąmaj świartoci null/Nan
print(f'śWartoci null w kolumach?\n{data.isnull().sum()}')

# Sprawdanie czy ęąwystpuj duplikaty
print(f'Duplikaty w kolumnach?\n{data.duplicated().any().sum()}')

from sklearn.model_selection import train_test_split

# Dzielenie danych
x = data.drop("TargetClass", axis=1).values
y = data['TargetClass'].values

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25,
    random_state=73)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

# Ewaluacja i raporty
from sklearn.metrics import accuracy_score, precision_score, recall_score
   , f1_score
```

```
from sklearn.metrics import classification_report, confusion_matrix

# Ocena wydajnosci modelu
def eval_perf_model(y_test_data, y_pred_data):
    print('# śćWydajno testowanego modelu:')
    acc_score = accuracy_score(y_test_data, y_pred_data)
    print('Accuracy:\t', acc_score)
    prec_score = precision_score(y_test_data, y_pred_data)
    print('Precision:\t', prec_score)
    rec_score = recall_score(y_test_data, y_pred_data)
    print('Recall:\t\t', rec_score)
    fscore = f1_score(y_test_data, y_pred_data)
    print('F1:\t\t', fscore)

    return acc_score, prec_score, rec_score, fscore

# Generowanie raportu klasyfikacji
def gen_classification_report(y_test_data, y_pred_data):
    target_names = ['Class 0 - D', 'Class 1 - G']
    report = classification_report(y_test_data, y_pred_data, target_names
        =target_names)
    print (f'# Raport klasyfikacji:\n{report}')

    return report

# Tworzenie tablicy pomylek
def create_heatmap(y_test_data, y_pred_data):
    conf_matrix = confusion_matrix(y_test_data, y_pred_data)

    # Wyswietlenie macierzy łpomyek
    print("Tablica łpomyek:\n", conf_matrix)

    # żUycie Seaborn do stworzenia heatmap
    sns.set(style="whitegrid")
    plt.figure(figsize=(6, 4))
    sns.heatmap(conf_matrix, annot=True, cmap="YlGnBu", fmt=".0f",
        xticklabels=[0, 1], yticklabels=[0, 1])
    plt.title('Tablica pomylek')
    plt.xlabel('Klasa przewidziana')
    plt.ylabel('Klasa rzeczywista')
    plt.show()

# Funkcja do tworzenia i trenowania modelu
# Jedna funkcja która żumoliwa ąpodobn impementacja żróynych algorytmów
# Zwraca najlepsze parametry
from sklearn.model_selection import GridSearchCV

def create_train_model(name:str, model, params:any):
    print(f'# Przetwarzanie modelu... {name}')

    current_model = model

    # Przeszukiwanie siatki w celu znalezienia najlepszych parametrow
    model_grid = GridSearchCV(current_model, params, cv=5, n_jobs=-1)
    model_grid.fit(x_train, y_train)
    print(f'# Najlepsze parametry:\n{model_grid.best_params_}')
```

```
    # Testowanie modelu na danych testowych
    y_pred = model_grid.predict(x_test)

    # Ewaulacja skutecznosci najlepszego modelu
    eval_perf_model(y_test, y_pred)
    gen_classification_report(y_test, y_pred)
    print('-----------------------------------')
    create_heatmap(y_test, y_pred)

    return model_grid


# Random Forrest
from sklearn.ensemble import RandomForestClassifier
rf_params = {
    'n_estimators': (100, 200, 300),
    'max_depth': (2, 3),
    'criterion': ('gini', 'entropy'),
}

rf = create_train_model('Las losowy', RandomForestClassifier(), rf_params
    )


#KNN
from sklearn.neighbors import KNeighborsClassifier
knn_params = {
    'n_neighbors': [2, 3, 4, 5, 6, 7],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

knn = create_train_model('K-Najblizszych sąsiadOw', KNeighborsClassifier
    (), knn_params)

# SVM
from sklearn.svm import SVC
svm_params = {
'kernel' : ('linear', 'rbf'),
"gamma": ('scale', 'auto')
}

# Funkcja do Tworzenia modelu
svm = create_train_model('Maszyny wektorow nosnych', SVC(), svm_params)


# Decision tree
from sklearn.tree import DecisionTreeClassifier
dt_params = {
    'max_depth': (1,2,3,),
    'criterion': ('gini','entropy')
}

# Funkcja do Tworzenia modelu
dr = create_train_model('Drzewo decyzyjne', DecisionTreeClassifier(),
    dt_params)
```

```
# Perceptron
from sklearn.linear_model import Perceptron
pr_params = {
    'alpha': [0.000001,0.00001,0.0001],
    'penalty': ['l1', 'l2']
}
# Funkcja do Tworzenia modelu
pr = create_train_model('Perceptron', Perceptron(), pr_params)


# Walidacja krzyzowa
from sklearn.model_selection import KFold, cross_val_score

list_scores = []
models = [
    # ('Regr.logist.', lr),
    ('Las losowy', rf),
    ('Drzewo Decyzyjne', dr),
    ('K-Naj.ąSsiad.', knn),
    ('Maszyn.wkt.śnon.', svm),
    ('Perceptron', pr)
]
kfold = KFold(n_splits=5, shuffle=True, random_state=73)
print('# Algorytm #\t# Dokladnosc #\t# Odchyl. standar. #')
for name, model in models:
    scores = cross_val_score(model, x_train, y_train, cv=kfold, scoring='
        accuracy', n_jobs=-1)
    score = scores.mean()
    std_dev = np.std(scores)
    print(f"{name}:\t{score}\t{std_dev}")


# Przewidywanie na nowym zestawie danych
new_data = pd.read_csv('dataset/Star3642_balanced.csv')

# new_data.info()
new_data = new_data.drop('SpType', axis=1)

# Sprawdzanie czy dane nie ąmaj świartoci null/Nan
print(f'śWartoci null w kolumach?\n{data.isnull().sum()}')

# Sprawdanie czy ęąwystpuj duplikaty
print(f'Duplikaty w kolumnach?\n{data.duplicated().any()}')

x_new = new_data.drop("TargetClass", axis = 1).values
y_new = new_data['TargetClass'].values

# Normalizacja
scaler.fit(x_new)
x_new = scaler.transform(x_new)

model = svm
print('Przewidywanie na nowym zestawie danych...')
y_new_pred = model.predict(x_new)

eval_perf_model(y_new, y_new_pred)
```

```
gen_classification_report(y_new, y_new_pred)

print('--- END ---')


# ------------------ DEEP LEARNING ------------------

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

from tensorflow.keras.metrics import Accuracy, Precision, Recall

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(5,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy', Precision(), Recall()])

model.summary()
# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split
    =0.2)

y_test_pred = model.predict(X_test)
y_test_pred_binary = (y_test_pred > 0.5).astype(int)

eval_perf_model(y_test, y_test_pred_binary)
gen_classification_report(y_test, y_test_pred_binary)

new_data = pd.read_csv('dataset/Star3642_balanced.csv')

# new_data.info()
new_data = new_data.drop('SpType', axis=1)

# Sprawdzanie czy dane nie ąmaj świartoci null/Nan
print(f'śWartoci null w kolumach?\n{data.isnull().sum()}')

# Sprawdanie czy ęąwystpuj duplikaty
print(f'Duplikaty w kolumnach?\n{data.duplicated().any()}')

x_new = new_data.drop("TargetClass", axis = 1).values
y_new = new_data['TargetClass'].values

# Normalizacja
scaler.fit(x_new)
x_new = scaler.transform(x_new)
```