Machine Learning w Pythonie – krótki wstęp.
W skrócie, uczenie maszynowe to sposób, w jaki komputery uczą się na podstawie dostarczonych danych, rozpoznawania wzorców i reguł, na podstawie, których mogą podejmować pewne decyzje. Możemy przyjąć podział algorytmów uczenia maszynowego na:

Uczenie nadzorowane,
Uczenie nienadzorowane,
Uczenie ze wzmocnieniem.

W poniższym przykładzie, przyjrzymy się problemowi związanym z uczeniem nadzorowanym - w tej metodzie posiadamy dane, które składają się z wejść (cech) oraz odpowiadających im oczekiwanych wyjść (etykiet, klas). Celem jest nauczenie modelu przewidywania odpowiednich wyjść na podstawie nowych, nieznanych wcześniej danych. Przykłady algorytmów: regresja liniowa, regresja logistyczna, drzewa decyzyjne,lasy losowe, maszyny wektorów nośnych (SVM), sieci neuronowe.

W uczeniu nadzorowanym możemy mieć do czynienia z problemami regresji, lub klasyfikacji. Problem regresji – problem przewiduje wartości ciągłe, a klasyfikacja przynależność do klasy (prawdopodobieństwo przynależności do klasy)

Przykład zastosowania biblioteki pandas oraz sklearn w problemie klasyfikacji:

Do przechowywania danych użyjemy DataFrame z biblioteki pandas.
Pandas jest szeroko stosowaną biblioteką języka Python, która umożliwia łatwą manipulację i analizę danych.
DataFrame jest dwuwymiarowa strukturą danych, przypominająca tabelę w bazie danych.
DataFrame składa się z wierszy i kolumn, gdzie każda kolumna może przechowywać dane różnych typów (np. liczby, ciągi znaków, daty).
Kolumny posiadają swoje nazwy.

Możemy pobrać interesujące nas dane z internetu a następnie zapisać je do DataFrame:

https://archive.ics.uci.edu/ml/datasets.php

https://archive.ics.uci.edu/ml/machine-learning-databases/iris/

Do budowy modeli użyjemy sklearn:
https://scikit-learn.org/stable/

Niech będą to dane dotyczące Kosaćców (Irisów)
Pobrane dane określają:
gatunek irysa - class
długość działki kielicha – sepal_length
szerokość działki kielicha sepal_width
długość płatka – petal_length
szerokość płatka – petal_width
gatunek irysa - class
-- Iris-setosa
-- Iris-versicolour
-- Iris-virginica


```
import pandas as pd
url = "./iris.csv"
df = pd.read_csv(url)
print(df.head(10))
```

Jeżeli pobieramy dane bezpośrednio z internetu (brak nagłówków):

```
import ssl
# Wyłączenie weryfikacji certyfikatu SSL -problem z ssl
ssl._create_default_https_context = ssl._create_unverified_context
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
# Nagłówki kolumn:
headers = ["sepal_length", "sepal_width", "petal_length", "petal_width",
"class"]
# Pobieraniei przypisanie nagłówków:
df = pd.read_csv(url,names=headers)
```

Metoda head zwraca n wierszy(domyślnie 5).

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Za pomocą
df.describe()
możemy uzyskać proste statystyki:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

kilka innych przydatnych funkcji:

```
print(df.shape)
print(df.columns)
```

df.info()

(150, 5)

Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'], dtype='object')

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 # Column   Non-Null Count Dtype
--- ------   -------------- -----
 0 sepal_length 150 non-null   float64
 1 sepal_width   150 non-null   float64
 2 petal_length 150 non-null   float64
 3 petal_width   150 non-null   float64
 4 class   150 non-null   object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB

Zbudujemy, wytrenujemy oraz klasyfikator na podstawie otrzymanych danych.
Najprostszy schemat budowy i oceny modelu wygląda tak:
1.Podział danych
2.Inicjalizacja modelu
3.Trening modelu
4. Predykcja na nowych danych
5. Ocena modelu

Pierwszym etapem jest podział danych na zbiór treningowy i testowy.
Zbiór testowy posłuży do końcowej walidacji modelu.

Rozdzielamy dane, na zmienne objaśniające i zmienną objaśnianą :

```
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
```

Metoda iloc wycina dane z ramki w kolejności – wiersze, kolumny.
„:” służy wycinania elementów od do. np.:

```
subset = df.iloc[0:1, 0:5]
```

subset:
   sepal_length sepal_width petal_length petal_width   class
0   5.1   3.5   1.4   0.2 Iris-setosa
Sam „:” zwraca wszystko, „:-1” - od zerowego indeksu, do ostatniego indeksu – bez niego
X – wszystkie wiersze, kolumny bez zmiennej objaśnianej – class.
y – wszystkie wiersze, z jedną ostatnią kolumną (class).

Do podziału na zbiory treningowe i testowe użyjemy:

```
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=2023)
```

test_size – rozmiar zbioru testowego (25%)
random_state – ziarno generatora liczb pseudolosowych, zapewnia te same wyniki.

W przykładzie użyjemy, trochę przewrotnie model KNN.
Główne założenie algorytmu jest takie, że podobne dane mają tendencję do występowania w pobliżu siebie w przestrzeni cech. Dlatego, aby przewidzieć klasę lub wartość dla nowych, nieoznaczonych danych, model KNN poszukuje k najbliższych sąsiadów w treningowym zbiorze danych i wykorzystuje ich etykiety lub wartości, aby dokonać predykcji dla nowych danych.
Dlaczego przewrotnie ? KNN za każdą predykcją nowych danych porównuje cechy do posiadanych danych wyliczanych za pomocą określonej metryki – nie ma tutaj opcji trenowania. Dane są po prostu przechowywane w pamięci (KNN może stosować różne algorytmy optymalizacyjne w wyszukiwaniu najbliższych sąsiadów(BallTree, KDTree)

2, Inicjalizacja modelu:

```
from sklearn.neighbors import KNeighborsClassifier
```

```
knn = KNeighborsClassifier(n_neighbors=5)
```

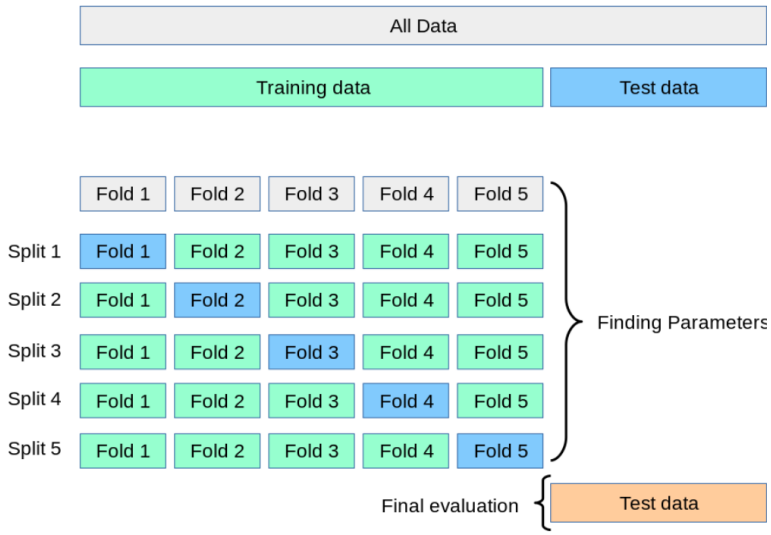
n_neighbors – liczba najbliższych sąsiadów, domyślnie 5.

3.„Trenowanie” modelu:

```
knn.fit(X_train, y_train)
```

4. Predykcja i 5. Ocena modelu:
Moglibyśmy przetestować model na zbiorze testowym i dokonać oceny.
Zanim jednak przejdziemy do predykcji i oceny modelu na zbiorze testowym, zobaczmy inny sposób oceny modelu - walidację krzyżową.

1.Podział zbioru danych na k równych części.
2.Użycie w każdej iteracji, jednego podzbioru jako zbiór testowy, a pozostałe podzbiory jako zbiór treningowy.
3.Trenowanie modelu na zbiorze treningowym.
4. Predykcja i ocena modelu na zbiorze testowym i zapisanie wyników.
5.Powtórzenie kroków 2-4 dla każdej kombinacji podzbiorów.
6.Uśrednienie wyników z poszczególnych iteracji, aby uzyskać ogólną ocenę modelu



Do podziału używamy Kfold, do oceny cross_val_score:

```
from sklearn.model_selection import train_test_split, KFold, cross_val_score
```

```
kfold = KFold(n_splits=5,random_state=2023,shuffle=True)

scores = cross_val_score(knn, X_train, y_train, cv=kfold, scoring="accuracy")
```

n_splits – na ile pozbiorów zrobić podział.

Metoda cross_val_score dokonuje oceny przy użyciu walidacji krzyżowej. W cv moglibyśmy wpisać po prostu liczbę na ile podzbiorów chcemy podzielić zbiór danych. Aby jednak móc powtórzyć wyniki, należy zastosować Kfold z podanym ziarnem random_state, oraz shuffle = True(mieszanie danych).
Jako metrykę oceny wybraliśmy dokładność(accuracy

```
print("Wyniki sprawdzianu krzyżowego:")
print(scores)
print(f"Średnia dokładność: {scores.mean()}")
```

Wyniki sprawdzianu krzyżowego:
[0.91304348 1. 0.95454545 0.90909091 0.90909091]
Średnia dokładność: 0.93715415019762856

Walidacja krzyżowa jest często wykorzystywana do dopasowania hiperparametrów, w naszym wypadku mogą to być miary odległości oraz liczba najbliższych sąsiadów.

Przetestujmy 2 miary: Euklidesową oraz Manhattan

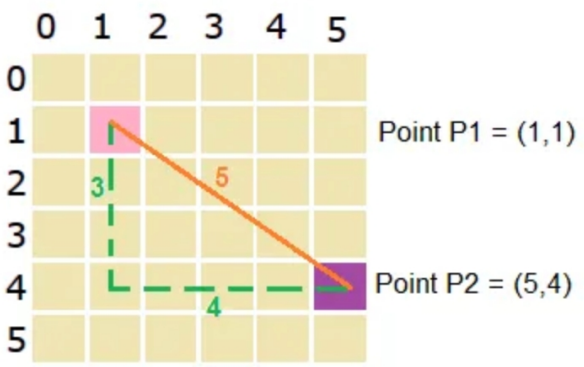
Euclidean:

$$D_e = \left(\sum_{i=1}^{n} (p_i - q_i)^2 \right)^{1/2}$$

Manhattan:

$$D_m = \sum_{i=1}^{n} |p_i - q_i|$$

Gdzie, n – liczba wymiarów,
pi , qi – punkty w danym wymiarze.



Euclidean distance = $\sqrt{(5-1)^2 + (4-1)^2} = 5$

Manhattan distance = $|5-1| + |4-1| = 7$

Oraz liczbę sąsiadów od 1 do 20:

```
from sklearn.model_selection import train_test_split, KFold, cross_val_score,
GridSearchCV
```

```
param_grid = {
    'n_neighbors': list(range(1, 21)),  # liczba sąsiadów od 1 do 20
    'metric': ['euclidean', 'manhattan']
}
```

```
grid_search = GridSearchCV(knn, param_grid, cv=KFold(n_splits=5,
random_state=2023,shuffle=True))

grid_search.fit(X_train, y_train)
```

```
results = pd.DataFrame(grid_search.cv_results_)

print(results)
results.to_csv("results.csv")
```

W param grid ustawiamy parametry, które chcemy przetestować.

GridSeaerchCV testuje nasz model dla parametrów, które podaliśmy, wykorzystując podział KFold.

Odczyt wyników w terminalu jest dość nieczytelny. Możemy zapisać go do pliku .csv, lub wyświetlić interesujące nas wartości.

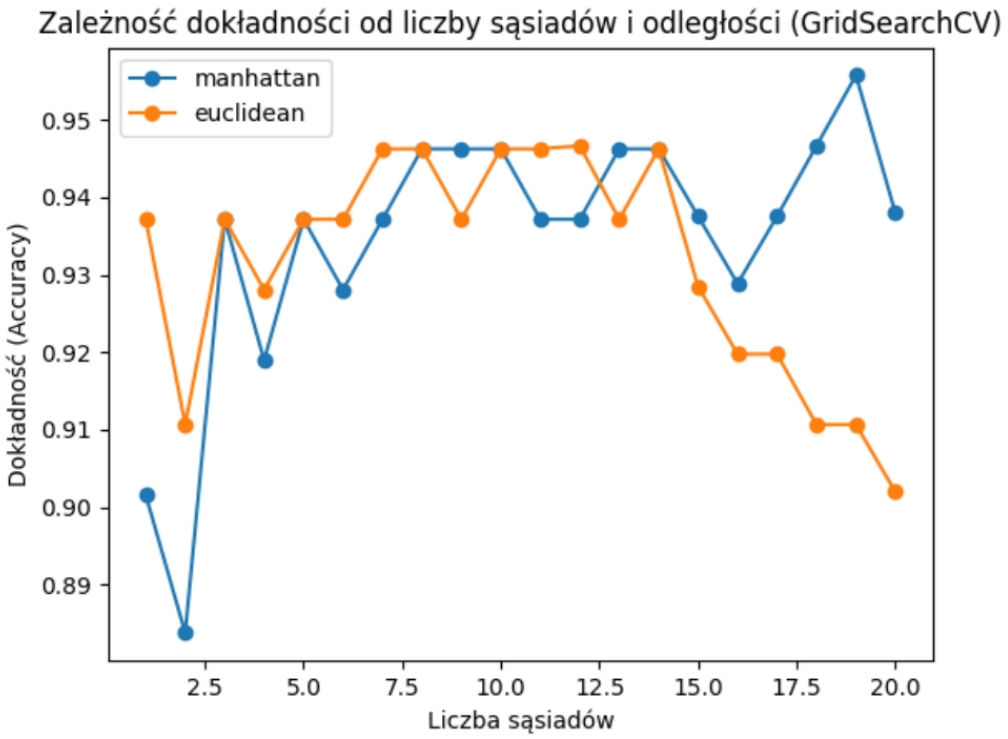
```
print(results["mean_test_score"])
```

Najlepsze parametry, wynik, oraz najlepszy model:

```
print("Najlepsze parametry: ", grid_search.best_params_)
print("Najlepszy wynik: ", grid_search.best_score_)

best_model = grid_search.best_estimator_
```

Na wykresie przedstawiłem poszczególne wyniki średnich wyników dokładności w zależności od liczby sąsiadów oraz metryki odległości:



Tak znaleziony model, możemy w końcu przetestować na zbiorze testowym, np. za pomocą dokładności:

```
from sklearn.metrics import accuracy_score
```

```
best_model = grid_search.best_estimator_
best_predict = best_model.predict(X_test)
print("Dokładność modelu na zbiorze testowym: ", accuracy_score(y_test,
best_predict))
```

Jak również na zbiorze testowym (również jak na zbiorze treningowym):

```
best_predict_train = best_model.predict(X_train)
print("Dokładność na zbiorze treningowym: ", accuracy_score(y_train,
best_predict_train ))
best_predict = best_model.predict(X_test)
print("Dokładność na zbiorze testowym: ", accuracy_score(y_test, best_predict))
```

Macierz pomyłek :

```
from sklearn.metrics import, accuracy_score, confusion_matrix
```

```
cm_train = confusion_matrix(y_train, best_predict_train)
print("Macierz pomyłek dla zbioru treningowego:")
print(cm_train)
report = classification_report(y_train, best_predict_train)
print(report)
```

Zadanie:
Zainicjalizuj wstępny model, wytrenuj, dokonaj tuningu, wybierz i przetestuj najlepszy model dla klasyfikatora innego niż podany w przykładzie (**inny algorytm**) na tych samych danych (irysy). Przygotuj raport w PDF z wynikami oraz kodem.