Wstęp do uczenia maszynowego

Mikołaj Słupiński

15 listopada 2025

Spis treści

1. Algorytm k-Nearest Neighbors (kNN)
2. Podział danych: train/test/validation
3. Metryki oceny modeli
4. Overfitting i underfitting
5. Cross-validation
6. DBSCAN - klasteryzacja oparta na gęstości

k-Nearest Neighbors (kNN)

k-Nearest Neighbors (kNN) - Wprowadzenie

k-Nearest Neighbors (kNN) to jeden z najprostszych algorytmów uczenia maszynowego.

Główna idea:

- Klasyfikacja obiektu na podstawie najbliższych sąsiadów
- ”Powiedz mi, kim są twoi sąsiedzi, a powiem ci, kim jesteś”
- Nie wymaga uczenia modelu - **lazy learning**

k-Nearest Neighbors (kNN) - Wprowadzenie

Zastosowania:

- Klasyfikacja (np. rozpoznawanie obrazów, diagnoza medyczna)
- Regresja (przewidywanie wartości ciągłych)
- Systemy rekomendacyjne

kNN - Zasada działania

Algorytm kNN dla klasyfikacji:

1. Wybierz liczbę sąsiadów k (hiperparametr)
2. Dla nowej obserwacji $\boldsymbol{x}$:
   - Oblicz odległość do wszystkich punktów w zbiorze treningowym
   - Znajdź k najbliższych sąsiadów
   - Przypisz klasę na podstawie głosowania większościowego

kNN - Zasada działania

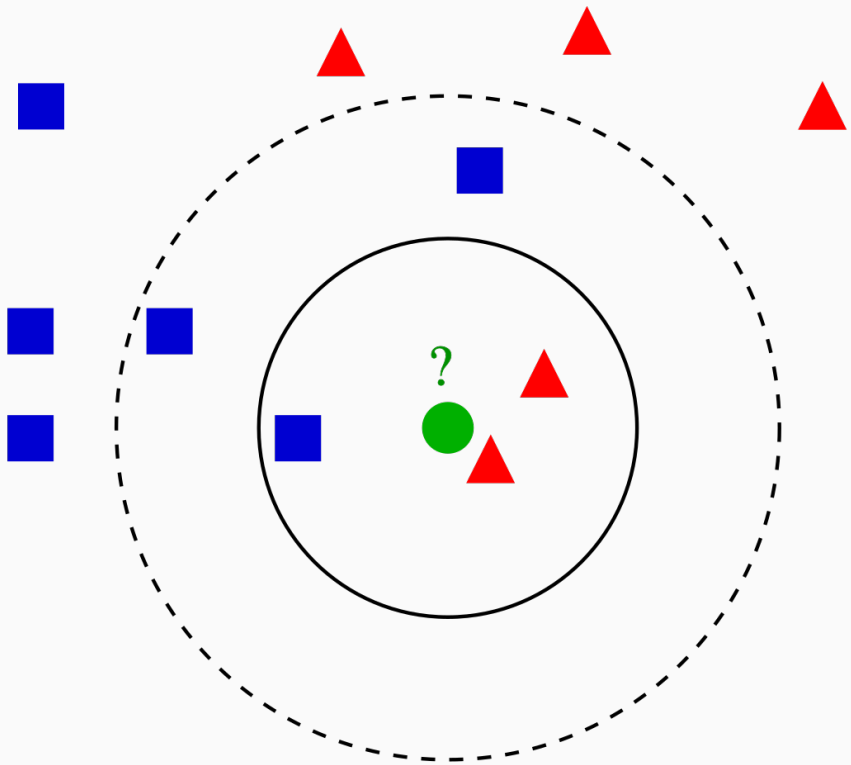
Algorytm kNN dla klasyfikacji:

Formalnie:

$$\hat{y} = \arg\max_{c \in \mathcal{Y}} \sum_{i \in N_k(\boldsymbol{x})} \mathbb{I}(y_i = c)$$

gdzie $N_k(\boldsymbol{x})$ to zbiór k najbliższych sąsiadów punktu $\boldsymbol{x}$.

kNN - Zasada działania



kNN - Metryki odległości

Wybór metryki odległości jest kluczowy dla działania algorytmu.

Popularne metryki:

1. Odległość euklidesowa (L2):

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{j=1}^{d} (x_j - x'_j)^2}$$

kNN - Metryki odległości

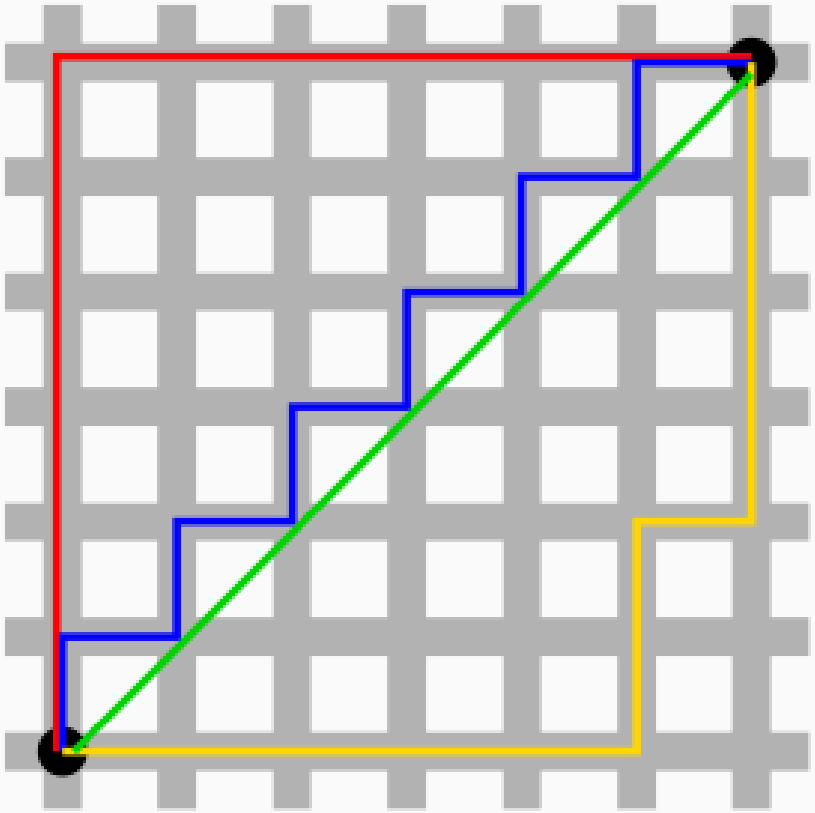
Wybór metryki odległości jest kluczowy dla działania algorytmu.

Popularne metryki:

2. Odległość Manhattan (L1):

$$d(x, x') = \sum_{j=1}^{d} |x_j - x'_j|$$

kNN - Metryki odległości



kNN - Metryki odległości

Wybór metryki odległości jest kluczowy dla działania algorytmu.

Popularne metryki:

3. Odległość Minkowskiego:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \left(\sum_{j=1}^{d} |x_j - x'_j|^p \right)^{1/p}$$

kNN - Wybór parametru k

Parametr k ma kluczowe znaczenie:

Małe k (np. k=1):

- Model jest bardzo wrażliwy na szum w danych
- Ryzyko przeuczenia (overfitting)
- Granice decyzyjne są bardzo nieregularne

kNN - Wybór parametru k

Parametr k ma kluczowe znaczenie:

Duże k:

- Model jest bardziej odporny na szum
- Ryzyko niedouczenia (underfitting)
- Granice decyzyjne są bardziej gładkie

kNN - Wybór parametru k

Parametr k ma kluczowe znaczenie:

Praktyka:

- Często wybiera się k jako nieparzystą liczbę (uniknięcie remisów)
- Typowe wartości: $k \in \{3, 5, 7, 9, 11\}$
- Optymalną wartość dobiera się za pomocą walidacji krzyżowej

kNN - Zalety i wady

Zalety:

- Prosty w implementacji i zrozumieniu
- Nie wymaga fazy uczenia
- Może modelować skomplikowane granice decyzyjne
- Działa dobrze dla małych zbiorów danych
- Może być używany do klasyfikacji i regresji

kNN - Zalety i wady

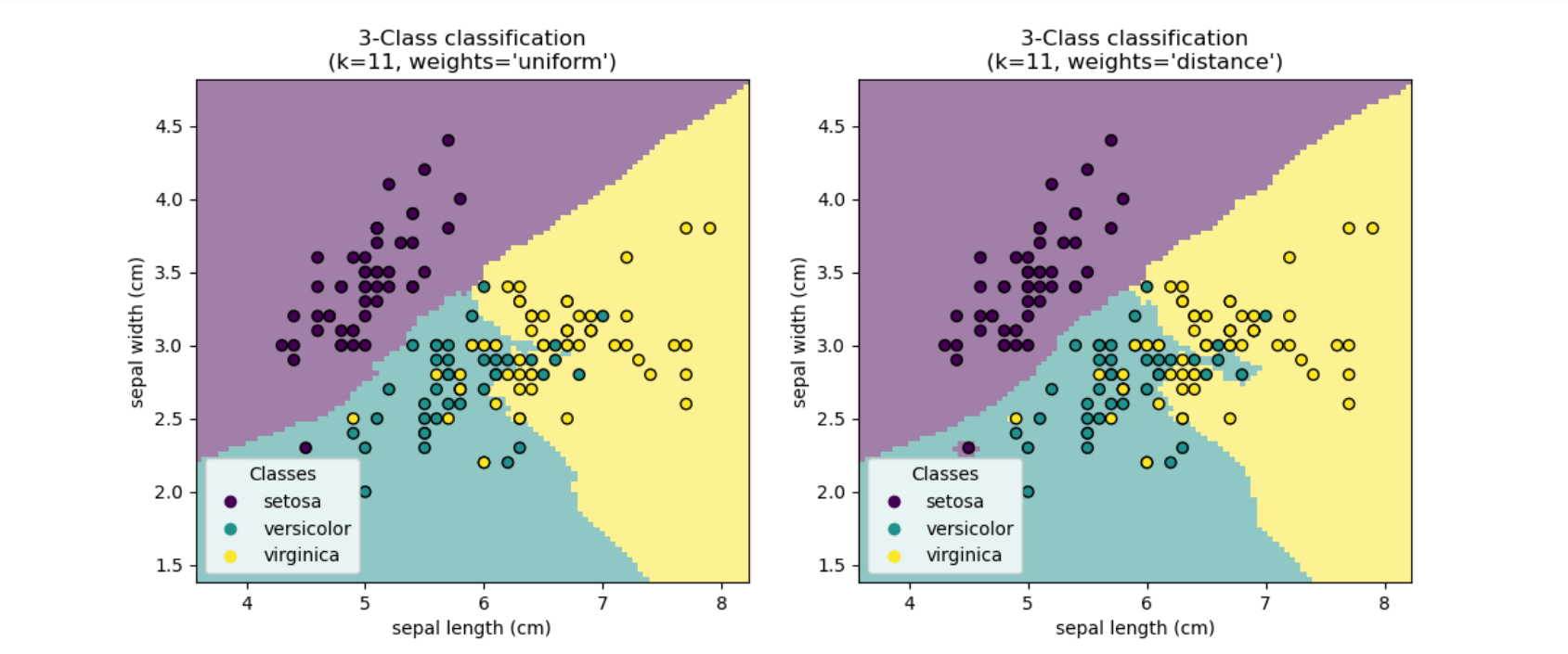
Wady:

- Kosztowny obliczeniowo podczas predykcji ($O(n \cdot d)$)
- Wymaga dużo pamięci (przechowuje cały zbiór treningowy)
- Wrażliwy na skalę cech - wymaga normalizacji
- Słabo radzi sobie z wysoką wymiarowością (*curse of dimensionality*)
- Wrażliwy na niezbalansowane klasy

kNN - Przykład w Pythonie

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

# Wczytaj dane
X, y = load_iris(return_X_y=True)

# Normalizacja danych
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Tworzenie i uczenie modelu
knn = KNeighborsClassifier(n_neighbors=5,
                           metric='euclidean')
knn.fit(X_scaled, y)

# Predykcja
y_pred = knn.predict(X_scaled)
```

kNN - Przykład w Pythonie



Podział danych

Podział danych - Motywacja

Dlaczego dzielimy dane?

Podział danych - Motywacja

Problem: Jeśli trenujemy i testujemy na tych samych danych, otrzymamy zbyt optymistyczną ocenę modelu.

Podział danych - Motywacja

Cel: Ocenić, jak dobrze model generalizuje na **nowych, niewidzianych danych**.

Podział danych - Motywacja

Rozwiązanie: Podział danych na rozłączne podzbiory:

- **Zbiór treningowy (training set)** - do uczenia modelu
- **Zbiór walidacyjny (validation set)** - do doboru hiperparametrów
- **Zbiór testowy (test set)** - do finalnej oceny modelu

Podział train/test

Podstawowy podział: train/test

- **Training set (70-80%):** dane używane do uczenia modelu
- **Test set (20-30%):** dane używane jedynie do finalnej oceny

Podział train/test - Kluczowe zasady

Kluczowe zasady:

- Zbiór testowy NIE może być używany podczas uczenia
- Podział powinien być losowy, ale **powtarzalny** (fixed random seed)
- W przypadku niezbalansowanych klas używamy **stratified split**

Problem z doborem hiperparametrów

Problem: Jak dobrać hiperparametry (np. k w kNN)?

- Nie możemy użyć zbioru testowego!
- Potrzebujemy osobnego zbioru walidacyjnego

Podział train/validation/test

Rozszerzony podział: train/validation/test

- **Training set (60-70%):** uczenie modelu
- **Validation set (10-20%):** dobór hiperparametrów i wybór modelu
- **Test set (10-20%):** finalna ocena wybranego modelu

Typowy workflow

Typowy workflow:

1. Trenuj różne modele/konfiguracje na *train*
2. Oceniaj je na *validation*
3. Wybierz najlepszy model
4. Jeden raz oceń wybrany model na *test*

Ważne zasady

Uwaga: Zbiór testowy używamy tylko **raz**, na samym końcu!

Dlaczego?

- Wielokrotne użycie zbioru testowego prowadzi do **przecieku informacji**
- Model zostaje pośrednio ”dostrojony” do danych testowych
- Tracisz miarodajną ocenę generalizacji

Podział danych - Przykład w Pythonie

```python
from sklearn.model_selection import train_test_split

# Podział train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Podział train na train/validation
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.25,
    random_state=42, stratify=y_train
)

print(f"Train size: {len(X_train)}")      # 60%
print(f"Validation size: {len(X_val)}")   # 20%
print(f"Test size: {len(X_test)}")        # 20%
```

Parametr stratify: zachowuje proporcje klas w każdym podzbiorze

Metryki oceny modeli

Metryki oceny - Wprowadzenie

Po czym poznać dobry model?

Potrzebujemy miar jakości, które pozwolą nam:

- Ocenić jakość predykcji modelu
- Porównać różne modele
- Wybrać najlepsze hiperparametry

Metryki oceny - Wybór metryki

Wybór metryki zależy od:

- Typu problemu (klasyfikacja/regresja)
- Zbalansowania klas
- Kosztów różnych typów błędów
- Wymagań biznesowych

Macierz pomyłek (Confusion Matrix)

Podstawowe narzędzie do oceny klasyfikacji binarnej.

		Predykcja	
		Pozytywna	Negatywna
Rzeczywistość	Pozytywna	TP	FN
	Negatywna	FP	TN

Gdzie:

- **TP (True Positive):** Poprawnie sklasyfikowane jako pozytywne
- **TN (True Negative):** Poprawnie sklasyfikowane jako negatywne
- **FP (False Positive):** Błędnie sklasyfikowane jako pozytywne (błąd I rodzaju)
- **FN (False Negative):** Błędnie sklasyfikowane jako negatywne (błąd II rodzaju)

Przykład: Detekcja spamu

Scenariusz: Model wykrywający spam w 100 emailach

		Predykcja		
		Spam	Nie-spam	**Suma**
Rzeczywistość	Spam	35	5	40
	Nie-spam	8	52	60
	Suma	43	57	100

Interpretacja:

- **TP = 35:** Spam poprawnie rozpoznany jako spam
- **TN = 52:** Normalne emaile poprawnie rozpoznane
- **FP = 8:** Normalne emaile błędnie oznaczone jako spam (irytujące!)
- **FN = 5:** Spam który przeszedł przez filtr (niebezpieczne!)

Accuracy (Dokładność)

Definicja:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Odsetek poprawnych predykcji.

Kiedy używać?

- Dobra metryka dla **zbalansowanych klas**
- Gdy wszystkie błędy mają podobną wagę
- Najprostsza i najczęściej używana metryka

Accuracy - Przykład dla zbalansowanych klas

Przykład:

- Mamy 100 przykładów: 50 pozytywnych, 50 negatywnych
- Model poprawnie klasyfikuje 45 pozytywnych i 47 negatywnych
- Accuracy = $\frac{45+47}{100} = 0.92$ (92%)

Model działa dobrze - 92% przykładów jest poprawnie sklasyfikowanych.

Accuracy - Problem z niezbalansowanymi klasami

Problem z niezbalansowanymi klasami:

- Mamy 100 przykładów: 5 pozytywnych, 95 negatywnych
- Model zawsze przewiduje klasę negatywną
- Accuracy = $\frac{0+95}{100} = 0.95$ (95%)

Wniosek: Wysoka accuracy (95%), ale model jest bezużyteczny!
Nie wykrył ani jednego pozytywnego przykładu.

$\Rightarrow$ Dla niezbalansowanych klas potrzebujemy innych metryk!

Precision (Precyzja)

Definicja:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Interpretacja: Jaka część pozytywnych predykcji była poprawna?

"Czy możemy ufać pozytywnym predykcjom?"

Kiedy jest ważna?

- Gdy **False Positive** jest kosztowny
- Przykład: Filtr spamu - nie chcemy blokować ważnych emaili
- Przykład: Rekomendacje produktów - nie chcemy irytować użytkownika złymi sugestiami

Recall/Sensitivity (Czułość)

Definicja:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Interpretacja: Jaka część rzeczywistych pozytywów została wykryta?

"Czy wykryliśmy wszystkie pozytywne przypadki?"

Kiedy jest ważny?

- Gdy **False Negative** jest kosztowny
- Przykład: Diagnoza chorób - nie chcemy przegapić chorego pacjenta
- Przykład: Detekcja oszustw - lepiej sprawdzić więcej transakcji

Specificity (Swoistość)

Definicja:

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Interpretacja: Jaka część rzeczywistych negatywów została poprawnie sklasyfikowana?

"Jak dobrze rozpoznajemy przypadki negatywne?"

Kiedy jest ważna?

- Komplementarna do Recall
- Ważna w diagnostyce medycznej (poprawne wykluczenie choroby)
- Często używana razem z Sensitivity w analizie medycznej

F1-Score - Definicja

F1-Score - harmoniczna średnia Precision i Recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

F1-Score - Dlaczego harmoniczna średnia?

Dlaczego harmoniczna średnia?

- Średnia arytmetyczna: $\frac{0.9+0.1}{2} = 0.5$
- Średnia harmoniczna: $\frac{2 \cdot 0.9 \cdot 0.1}{0.9+0.1} = 0.18$

Harmoniczna średnia ”karze” za niską wartość jednej z metryk.

F_β Score

Ogólniejsza wersja - F_β score:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

gdzie β kontroluje wagę Recall względem Precision.

Przykład - Diagnoza medyczna

Problem: Wykrywanie rzadkiej choroby (1% populacji)

Zbiór testowy: 1000 pacjentów

- 10 chorych
- 990 zdrowych

Porównamy dwa modele wykrywające tę chorobę.

Model A - Zawsze ”zdrowy”

Model A: Zawsze przewiduje ”zdrowy”

Wyniki:

- Accuracy = 990/1000 = **99%**
- Recall = 0/10 = **0%** (nie wykrył żadnego chorego!)
- Precision = undefined (brak pozytywnych predykcji)

Wniosek: Model bezużyteczny, mimo wysokiej dokładności!

Model B - Rzeczywisty klasyfikator

Model B: TP=8, FP=20, FN=2, TN=970

Wyniki:

- Accuracy = (8+970)/1000 = 97.8%
- Precision = 8/(8+20) = 28.6%
- Recall = 8/(8+2) = 80%
- F1-Score = 42.1%

Wniosek: Model B jest lepszy, mimo niższej accuracy!
Wykrywa 80% chorych pacjentów.

Krzywa ROC

ROC (Receiver Operating Characteristic):

Wykres True Positive Rate vs False Positive Rate

Definicje:

$$TPR = \frac{TP}{TP + FN} \quad \text{(Recall)}$$

$$FPR = \frac{FP}{FP + TN} = 1 - \text{Specificity}$$

AUC (Area Under Curve)

AUC: Pole pod krzywą ROC

Wartości:

- Zakres: [0, 1]
- AUC = 0.5: model losowy
- AUC = 1.0: model idealny
- AUC > 0.7: zazwyczaj uznawane za przyzwoite

Interpretacja AUC

Interpretacja AUC:

Prawdopodobieństwo, że model przypisze wyższy score losowo wybranemu przykładowi pozytywnemu niż negatywnemu.

Innymi słowy: Jeśli losowo wybierzemy jeden pozytywny i jeden negatywny przykład, AUC mówi nam jak prawdopodobne jest, że model poprawnie oceni który jest pozytywny.

Metryki - Przykład w Pythonie

```python
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, f1_score,
                             confusion_matrix, roc_auc_score)

# Oblicz metryki
acc = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='binary')
recall = recall_score(y_true, y_pred, average='binary')
f1 = f1_score(y_true, y_pred, average='binary')

# Macierz pomyłek
cm = confusion_matrix(y_true, y_pred)
print(f"Confusion Matrix:\n{cm}")

# AUC wymaga prawdopodobieństw, nie klas
y_proba = model.predict_proba(X_test)[:, 1]
auc = roc_auc_score(y_true, y_proba)

print(f"Accuracy: {acc:.3f}, Precision: {precision:.3f}")
print(f"Recall: {recall:.3f}, F1: {f1:.3f}, AUC: {auc:.3f}")
```

Metryki dla klasyfikacji wieloklasowej

Jak rozszerzyć metryki na więcej niż 2 klasy?

Problem: Precision, Recall, F1-Score są zdefiniowane dla klasyfikacji binarnej.

Rozwiązanie: Strategie uśredniania wyników z poszczególnych klas

- Macro-average
- Weighted-average
- Micro-average

Macro-average

1. Macro-average:

- Oblicz metrykę dla każdej klasy osobno
- Uśrednij wyniki (każda klasa ma równą wagę)
- Dobra dla zbalansowanych klas

Przykład:

- Klasa A: Precision = 0.8
- Klasa B: Precision = 0.6
- Klasa C: Precision = 0.9
- Macro-Precision = (0.8 + 0.6 + 0.9) / 3 = 0.77

Weighted-average i Micro-average

2. Weighted-average:

- Jak macro, ale ważone liczebnością klas
- Dobra dla niezbalansowanych klas

3. Micro-average:

- Zsumuj TP, FP, FN dla wszystkich klas
- Oblicz jedną globalną metrykę
- Dla accuracy = micro-F1

Overfitting i Underfitting

Overfitting i Underfitting - Wprowadzenie

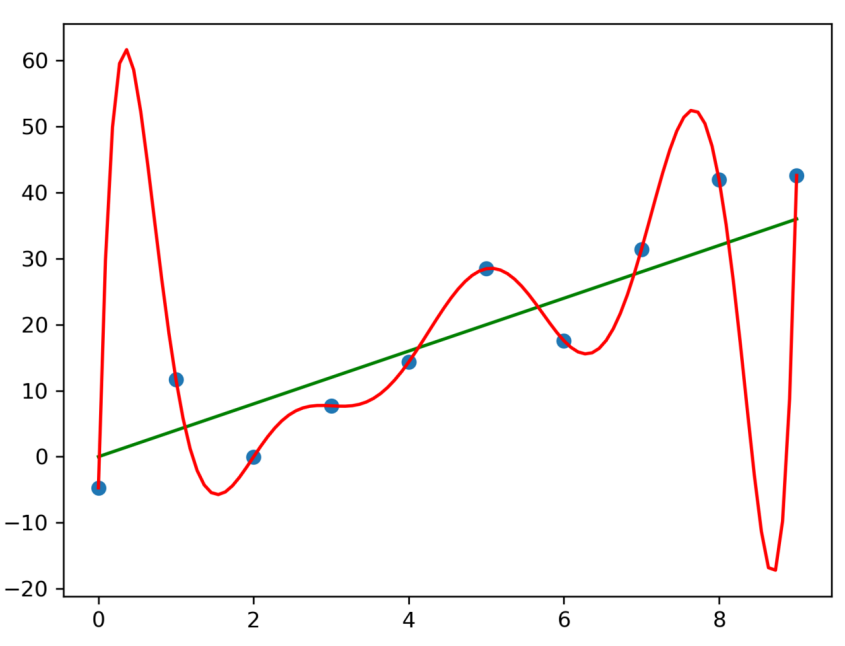
Kluczowy problem w uczeniu maszynowym:

Jak dobrze model generalizuje na nowe dane?

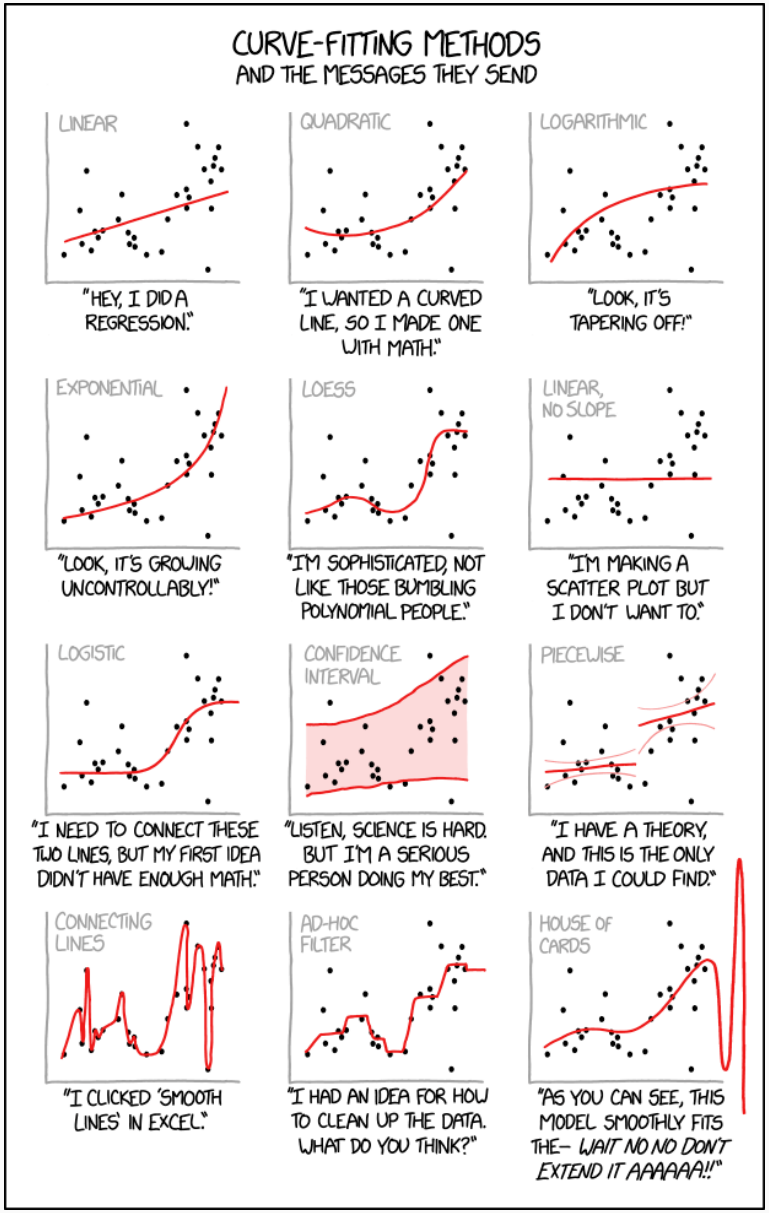
Dwa ekstremalne przypadki:

- Underfitting (niedouczenie)
- Overfitting (przeuczenie)

Overfitting i Underfitting - Wprowadzenie



Overfitting i Underfitting - Wprowadzenie



Underfitting i Overfitting

1. Underfitting (niedouczenie):

   - Model jest zbyt prosty
   - Słabo radzi sobie zarówno na zbiorze treningowym, jak i testowym
   - Wysokie bias, niskie variance

2. Overfitting (przeuczenie):

   - Model jest zbyt skomplikowany
   - Świetnie radzi sobie na zbiorze treningowym, słabo na testowym
   - Nauczył się szumu w danych treningowych zamiast prawdziwych wzorców
   - Niskie bias, wysokie variance

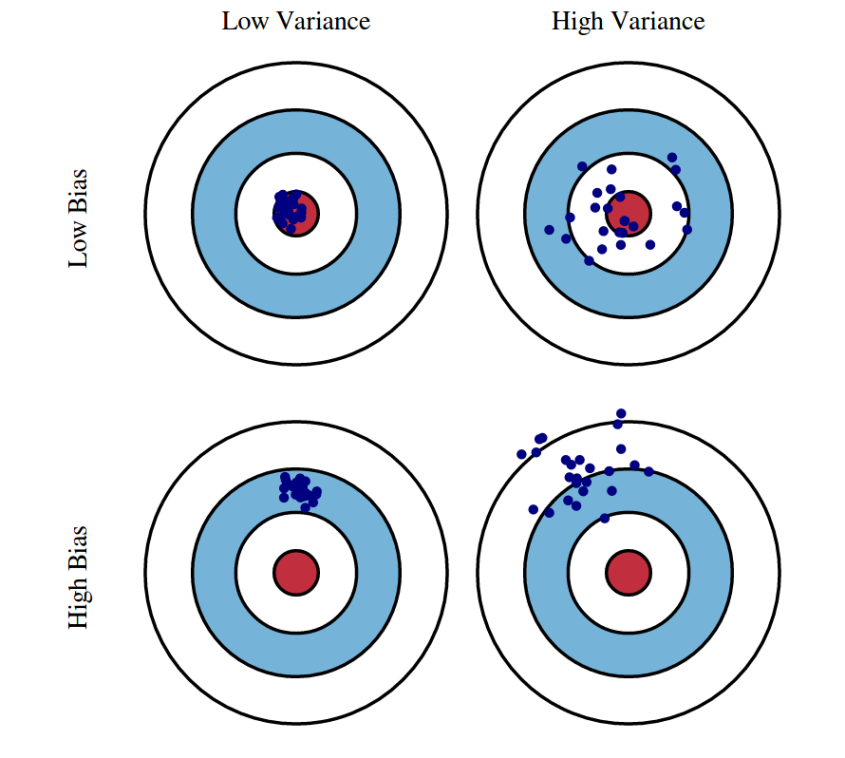
Bias-Variance Tradeoff

Błąd modelu można rozłożyć na trzy składowe:

$$\mathbb{E}[\text{Error}] = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Każda składowa ma inną przyczynę i wymaga innego podejścia do redukcji.

Bias-Variance Tradeoff



Rysunek 1: Caption

Bias (Obciążenie)

Bias (obciążenie):

- Błąd wynikający z błędnych założeń modelu
- Wysoki bias $\rightarrow$ model zbyt prosty $\rightarrow$ underfitting

Przykład:

- Próba dopasowania linii prostej do danych kwadratowych
- Model nie ma wystarczającej złożoności, by uchwycić prawdziwy wzorzec

Variance (Wariancja)

Variance (wariancja):

- Wrażliwość modelu na zmianę danych treningowych
- Wysoka wariancja $\rightarrow$ model zbyt skomplikowany $\rightarrow$ overfitting

Przykład:

- Model dopasowuje się idealnie do danych treningowych, włącznie z szumem
- Małe zmiany w danych treningowych powodują duże zmiany w modelu

Irreducible Error i Tradeoff

Irreducible Error:

- Szum w danych, którego nie da się wyeliminować
- Nie zależy od wyboru modelu

Tradeoff:

- Zmniejszenie bias zwykle zwiększa variance i odwrotnie
- Cel: znaleźć balans minimalizujący całkowity błąd
- Optymalna złożoność modelu leży gdzieś pomiędzy

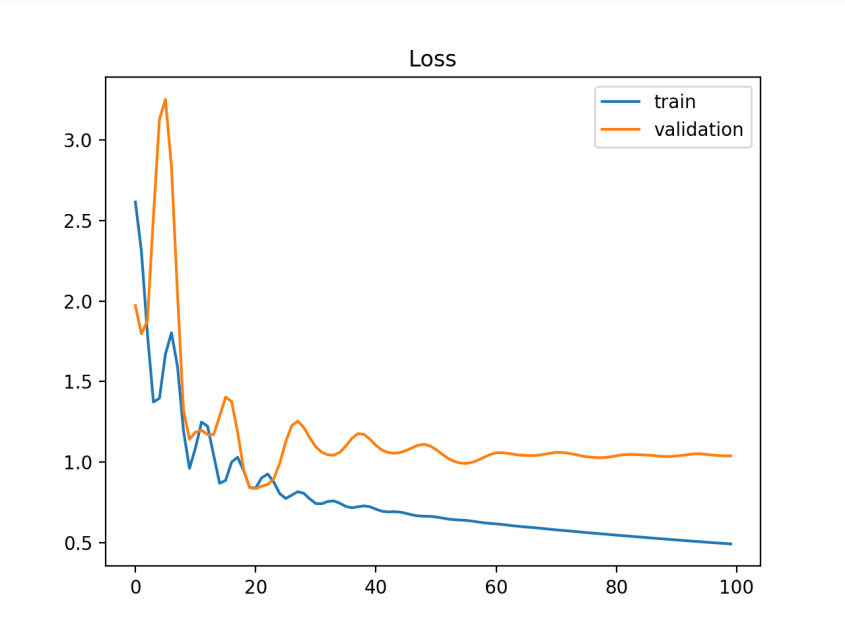
Krzywa uczenia (Learning Curve)

Learning curve pokazuje błąd modelu w funkcji rozmiaru zbioru treningowego.

Cel:

- Zrozumieć jak model radzi sobie z rosnącą ilością danych
- Zdiagnozować overfitting lub underfitting
- Ocenić czy warto zbierać więcej danych

Krzywa uczenia (Learning Curve)



Rysunek 2: Caption

Błąd treningowy w funkcji danych

Błąd treningowy: rośnie ze wzrostem liczby przykładów

Dlaczego?

- Z małą liczbą danych łatwo dopasować model idealnie
- Z większą liczbą danych trudniej uzyskać zerowy błąd
- Model musi znaleźć kompromis między różnorodnymi przykładami

Błąd walidacyjny w funkcji danych

Błąd walidacyjny: maleje ze wzrostem liczby przykładów

Dlaczego?

- Więcej danych $\rightarrow$ lepsza generalizacja
- Model uczy się prawdziwych wzorców zamiast szumu
- W pewnym momencie osiąga plateau

Po osiągnięciu plateau dodanie kolejnych danych nie poprawia wyników.

Diagnoza z krzywej uczenia

Jak rozpoznać problemy?

Overfitting:

- Duża różnica między błędem treningowym a walidacyjnym
- Krzywe nie zbiegają się

Underfitting:

- Oba błędy wysokie i zbliżone
- Krzywe zbiegają się, ale na wysokim poziomie błędu

Rozpoznawanie problemów

Symptomy overfitting:

- Bardzo niski błąd treningowy, wysoki błąd testowy
- Duża różnica między training accuracy a validation accuracy
- Skomplikowany model (wiele parametrów, wysoka głębokość)

Symptomy underfitting:

- Wysoki błąd treningowy i testowy
- Błędy są podobne
- Model nie może uchwycić wzorców w danych

Idealny model: Niskie błędy treningowy i testowy, niewielka różnica.

Jak walczyć z overfitting? - Więcej danych

1. Więcej danych treningowych

- Najprostsze i najskuteczniejsze rozwiązanie
- Większy zbiór danych $\rightarrow$ trudniej nauczyć się szumu
- Data augmentation dla obrazów, tekstu

Przykłady data augmentation:

- Obrazy: rotacja, odbicie, przycinanie, zmiana jasności
- Tekst: synonimizacja, tłumaczenie tam i z powrotem
- Audio: zmiana tempa, dodanie szumu

Jak walczyć z overfitting? - Regularyzacja

2. Regularizacja

- L1, L2 regularization - kara za duże wagi
- Dropout (w sieciach neuronowych) - losowe wyłączanie neuronów
- Early stopping - zatrzymanie treningu przed przeuczeniem

Działanie:

- Ogranicza złożoność modelu
- Preferuje prostsze rozwiązania
- Zmusza model do uczenia się solidnych wzorców

Jak walczyć z overfitting? - Uproszczenie modelu

3. Uproszczenie modelu

- Zmniejszenie liczby cech (feature selection)
- Zmniejszenie głębokości drzewa decyzyjnego
- Zwiększenie k w kNN
- Zmniejszenie liczby warstw lub neuronów w sieci

Zasada:

- Prostszy model ma mniejszą zdolność do zapamiętania szumu
- Zmniejsza variance, ale może zwiększyć bias

Jak walczyć z overfitting? - Cross-validation

4. Cross-validation

- Lepsze wykorzystanie dostępnych danych
- Bardziej wiarygodna ocena generalizacji
- Pomaga wykryć overfitting wcześniej

Korzyści:

- Każdy przykład jest raz w zbiorze walidacyjnym
- Średnia z wielu podziałów daje stabilniejszą ocenę
- Redukuje wpływ losowości podziału danych

Jak walczyć z underfitting? - Złożoność modelu

1. Zwiększenie złożoności modelu

- Więcej parametrów, głębsze drzewo decyzyjne
- Zmniejszenie k w kNN
- Przejście do bardziej ekspresyjnego modelu

Przykłady:

- Model liniowy $\rightarrow$ model nieliniowy
- Płytkie drzewo $\rightarrow$ głębsze drzewo
- Mała sieć neuronowa $\rightarrow$ większa sieć

Jak walczyć z underfitting? - Feature Engineering

2. Inżynieria cech (Feature Engineering)

- Dodanie nowych, istotnych cech
- Transformacje cech (wielomiany, interakcje)

Przykłady transformacji:

- $x \rightarrow x^2, x^3$ (cechy wielomianowe)
- $x_1, x_2 \rightarrow x_1 \cdot x_2$ (interakcje)
- Logarytmy, pierwiastki, funkcje trygonometryczne

Jak walczyć z underfitting? - Regularyzacja

3. Zmniejszenie regularyzacji

- Mniejsza kara za złożoność modelu
- Dłuższe uczenie (jeśli używamy early stopping)
- Zmniejszenie współczynnika regularyzacji (np. λ)

Uwaga:

- Zbyt słaba regularyzacja może prowadzić do overfitting
- Trzeba znaleźć balans

Jak walczyć z underfitting? - Jakość danych

4. Sprawdzenie jakości danych

Pytania do zadania:

- Czy cechy są informacyjne dla tego problemu?
- Czy dane są poprawne i kompletne?
- Czy problem jest w ogóle rozwiązywalny z tymi danymi?

Możliwe problemy:

- Brakujące istotne cechy
- Błędy w etykietach
- Szum dominuje nad sygnałem

kNN - Overfitting (k=1)

kNN jako przykład bias-variance tradeoff:

k = 1 (overfitting):

- Model ma bardzo niski bias (może dopasować dowolny wzorzec)
- Wysoka wariancja (wrażliwy na szum)
- Granica decyzyjna bardzo nieregularna
- Training accuracy często = 100%

Model ”zapamiętuje” każdy punkt treningowy, nawet jeśli jest to szum.

kNN - Underfitting (k=n)

k = n (underfitting):

- Model ma wysoki bias (zbyt uproszczony)
- Niska wariancja (stabilny)
- Wszystkie predykcje takie same (klasa większościowa)
- Niskie accuracy

Optymalne k:

- Znajduje się gdzieś pomiędzy k=1 a k=n
- Trzeba dobrać eksperymentalnie!
- Używamy walidacji krzyżowej do znalezienia najlepszego k

Cross-validation

Cross-validation - Motywacja

Problem z pojedynczym podziałem train/test:

- Ocena modelu zależy od szczęśliwego losowania
- Niewykorzystane dane (test set nie służy do uczenia)
- Przy małych zbiorach danych tracimy cenne przykłady
- Wysoka wariancja oszacowania błędu

Rozwiązanie: Cross-validation

- Wykorzystaj dane do uczenia i testowania w różnych konfiguracjach
- Uśrednij wyniki z wielu podziałów
- Otrzymaj bardziej wiarygodne oszacowanie błędu generalizacji

k-Fold Cross-Validation

Najpopularniejsza metoda walidacji krzyżowej.

Algorytm:

1. Podziel dane na k równych części (folds)
2. Dla $i = 1, 2, \ldots, k$:
   - Użyj fold i jako zbioru testowego
   - Użyj pozostałych $k - 1$ folds jako zbioru treningowego
   - Wytrenuj model i oceń go
3. Uśrednij wyniki z k iteracji

k-Fold Cross-Validation - Wyniki

Wynik:

$$\text{CV Score} = \frac{1}{k} \sum_{i=1}^{k} \text{Score}_i$$

Typowe wartości k:

- $k = 5$ lub $k = 10$ (najczęściej)
- Trade-off: większe k = mniejsze bias, większe variance i koszt obliczeniowy

Zalety:

- Każdy przykład jest raz w zbiorze testowym
- Bardziej wiarygodna ocena niż pojedynczy podział

Stratified k-Fold Cross-Validation

Problem: W standardowym k-fold klasy mogą być nierównomiernie rozłożone między folds.

Rozwiązanie: Stratified k-Fold

- Zachowuje proporcje klas w każdym fold
- Każdy fold jest reprezentatywny dla całego zbioru
- Szczególnie ważne dla niezbalansowanych klas

Przykład:

- Zbiór: 80% klasa A, 20% klasa B
- Standardowy k-fold: losowy podział
- Stratified k-fold: każdy fold ma 80% A i 20% B

Zalecenie: Dla klasyfikacji prawie zawsze używaj stratified k-fold!

Leave-One-Out Cross-Validation (LOOCV)

Ekstremalny przypadek k-fold, gdzie $k = n$.

Algorytm:

- Dla każdej obserwacji:
  - Użyj jej jako zbioru testowego (1 przykład)
  - Pozostałe $n - 1$ przykładów jako zbiór treningowy
  - Wytrenuj model i oceń
- Uśrednij wyniki z n iteracji

LOOCV - Zalety i wady

Zalety:

- Maksymalne wykorzystanie danych
- Nieobciążone oszacowanie błędu (niskie bias)
- Deterministyczne (brak losowości)

Wady:

- Bardzo kosztowne obliczeniowo (n iteracji)
- Wysoka wariancja oszacowania
- Modele trenowane na bardzo podobnych zbiorach

W praktyce: k-fold z k=5 lub k=10 jest lepszym kompromisem.

Cross-validation - Ocena modelu

1. Ocena modelu:
   - Bardziej wiarygodne oszacowanie błędu generalizacji
   - Ocena stabilności modelu

Cross-validation - Dobór hiperparametrów

2. **Dobór hiperparametrów (Hyperparameter Tuning):**
   - Testuj różne wartości hiperparametrów
   - Dla każdej konfiguracji wykonaj CV
   - Wybierz konfigurację z najlepszym CV score

Cross-validation - Wybór modelu

3. Wybór modelu:

- Porównaj różne algorytmy (kNN vs SVM vs Random Forest)
- Wybierz model z najlepszym CV score

Uwaga: Po wyborze modelu i hiperparametrów za pomocą CV, wytrenuj finalny model na **całym** zbiorze treningowym i oceń na **oddzielnym** zbiorze testowym!

Cross-validation - Przykład w Pythonie

```python
from sklearn.model_selection import (cross_val_score,
                                     StratifiedKFold)
from sklearn.neighbors import KNeighborsClassifier

# Model
knn = KNeighborsClassifier(n_neighbors=5)

# 5-fold cross-validation
cv_scores = cross_val_score(knn, X, y, cv=5,
                            scoring='accuracy')
print(f"CV scores: {cv_scores}")
print(f"Mean CV score: {cv_scores.mean():.3f} "
      f"+/- {cv_scores.std():.3f}")

# Stratified k-fold
skf = StratifiedKFold(n_splits=5, shuffle=True,
                      random_state=42)
cv_scores_stratified = cross_val_score(knn, X, y,
                                       cv=skf,
                                       scoring='f1_macro')
```

Grid Search with Cross-Validation

```python
from sklearn.model_selection import GridSearchCV

# Definiuj siatkę hiperparametrów
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'metric': ['euclidean', 'manhattan', 'minkowski'],
    'weights': ['uniform', 'distance']
}

# Grid search z 5-fold CV
grid_search = GridSearchCV(
    KNeighborsClassifier(),
    param_grid,
    cv=5,
    scoring='f1_macro',
    n_jobs=-1  # wykorzystaj wszystkie rdzenie
)

grid_search.fit(X_train, y_train)

print(f"Best params: {grid_search.best_params_}")
print(f"Best CV score: {grid_search.best_score_:.3f}")

# Najlepszy model
best_model = grid_search.best_estimator_
```

Nested Cross-Validation

Problem: Używanie tego samego CV do wyboru hiperparametrów i oceny modelu prowadzi do zbyt optymistycznych wyników.

Rozwiązanie: Nested CV

- **Zewnętrzna pętla (outer loop):** ocena modelu
  - Dzieli dane na train/test
- **Wewnętrzna pętla (inner loop):** dobór hiperparametrów
  - Na zbiorze treningowym z outer loop
  - Wykonuje Grid Search z CV

Koszt: Jeśli outer = 5-fold i inner = 5-fold, trenujemy $5 \times 5 = 25$ modeli dla każdej konfiguracji hiperparametrów!

Kiedy używać? Gdy mamy mało danych i chcemy najbardziej wiarygodnego oszacowania.

DBSCAN

DBSCAN - Wprowadzenie

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Rodzaj algorytmu: Klasteryzacja (unsupervised learning)

Główna idea:

- Klastry to obszary o wysokiej gęstości punktów
- Oddzielone obszarami o niskiej gęstości
- Punkty w rzadkich obszarach = szum (outliers)

DBSCAN - Różnica od kNN

Różnica od kNN:

- kNN = klasyfikacja (supervised)
- DBSCAN = klasteryzacja (unsupervised, brak etykiet)

DBSCAN - Zalety

Zalety:

- Nie trzeba podawać liczby klastrów z góry
- Radzi sobie z klastrami dowolnego kształtu
- Automatycznie wykrywa outliers

DBSCAN - Parametry

Parametry algorytmu:

1. ε **(epsilon):** Promień sąsiedztwa punktu
   - Definiuje, jak daleko szukamy sąsiadów

2. **MinPts:** Minimalna liczba punktów w sąsiedztwie
   - Ile punktów musi być w promieniu ε, aby utworzyć klaster

DBSCAN - Typy punktów

Typy punktów:

- **Core point (punkt rdzeniowy):** Ma $\geq$ MinPts punktów w swoim ε-sąsiedztwie (włącznie z sobą)
- **Border point (punkt brzegowy):** Ma $<$ MinPts punktów w sąsiedztwie, ale należy do sąsiedztwa core point
- **Noise point (szum):** Nie jest ani core, ani border

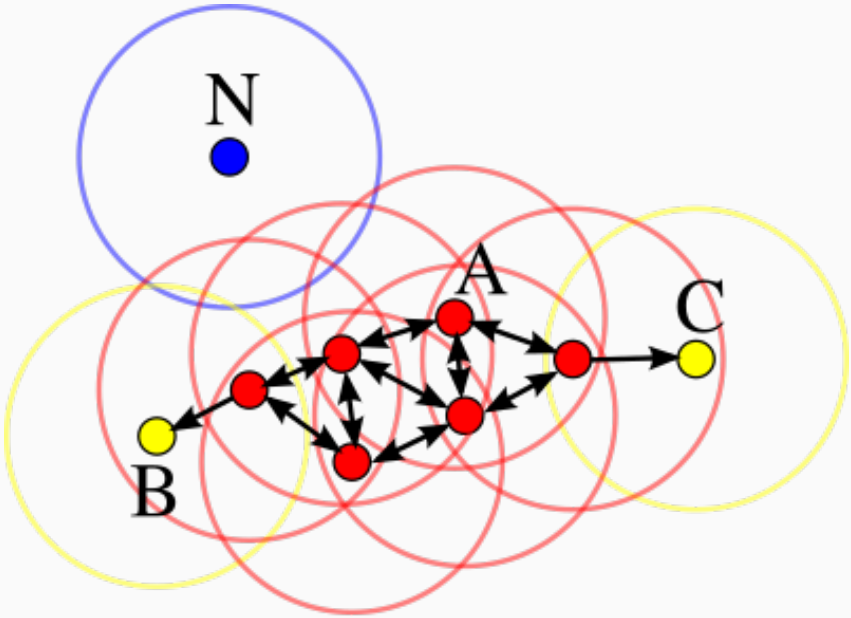
DBSCAN - Zasada działania

Algorytm (uproszczony):

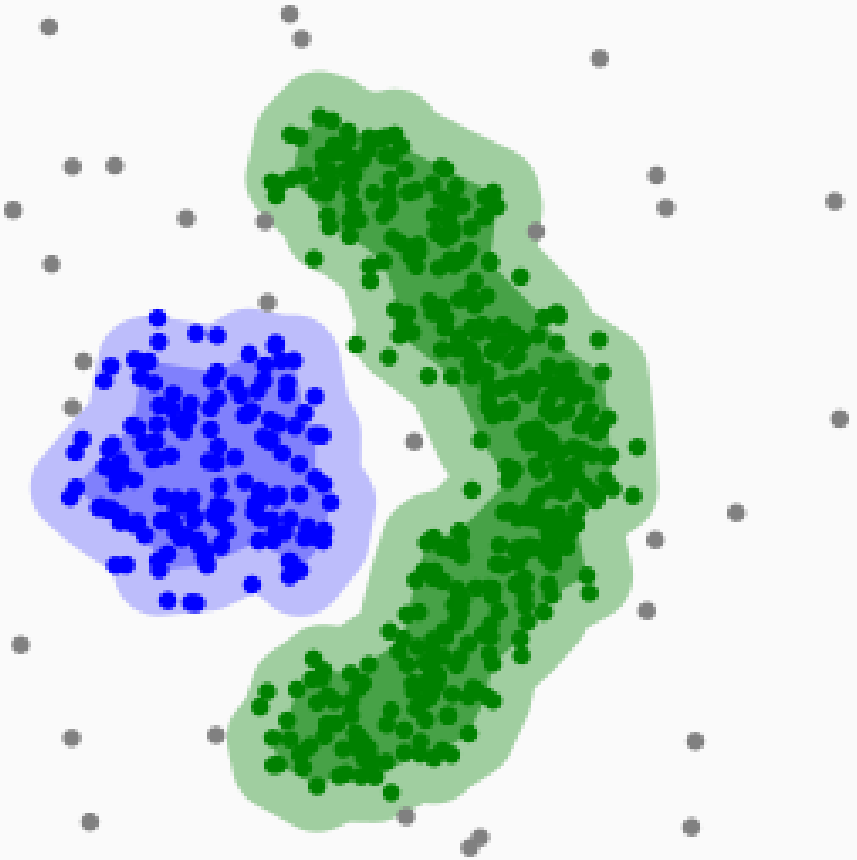
1. Dla każdego punktu p:
   - Znajdź wszystkie punkty w odległości $\leq \varepsilon$
   - Jeśli $|\text{sąsiedztwo}| \geq \text{MinPts}$, p jest core point
2. Utwórz klastry:
   - Dla każdego nieodwiedzonego core point rozpocznij nowy klaster
   - Dodaj do klastra wszystkie punkty gęsto osiągalne (density-reachable)
   - Punkty brzegowe należą do klastra pierwszego core point, który je osiągnie
3. Punkty nieprzypisane do żadnego klastra = szum

Gęsta osiągalność (density-reachable): Punkt q jest gęsto osiągalny z p, jeśli istnieje ścieżka punktów rdzeniowych prowadząca od p do q.

DBSCAN - Zasada działania



DBSCAN - Zasada działania



DBSCAN - Zalety

Zalety:

- Nie wymaga podania liczby klastrów z góry
- Wykrywa klastry o dowolnych kształtach (nie tylko kuliste)
- Odporne na outliers (identyfikuje je jako szum)
- Deterministyczny (z ustalonymi parametrami)

DBSCAN - Wady

Wady:

- Wrażliwy na dobór parametrów ε i MinPts
- Trudności z klastrami o różnych gęstościach
- Słabo radzi sobie w wysokich wymiarach (curse of dimensionality)
- Kosztowny obliczeniowo dla dużych zbiorów (można przyspieszyć strukturami indeksującymi)

DBSCAN - Zastosowania

Zastosowania:

- Analiza danych przestrzennych (GIS)
- Segmentacja obrazów
- Wykrywanie anomalii

DBSCAN - Przykład w Pythonie

```python
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
import numpy as np

# Dane (unsupervised - nie ma y!)
X = np.random.randn(300, 2)

# Normalizacja (DBSCAN wrażliwy na skalę!)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5, metric='euclidean')
clusters = dbscan.fit_predict(X_scaled)

# Wyniki
print(f"Liczba klastrów: {len(set(clusters)) - 1}")
# -1 oznacza szum
print(f"Liczba punktów szumu: {sum(clusters == -1)}")
print(f"Etykiety klastrów: {set(clusters)}")
```

Porównanie: kNN vs DBSCAN

Cecha	kNN	DBSCAN
Typ uczenia	Supervised	Unsupervised
Zadanie	Klasyfikacja	Klasteryzacja
Wymaga etykiet?	Tak	Nie
Główny parametr	k	ε, MinPts
Wykrywa outliers?	Nie bezpośrednio	Tak
Kształt klastrów	Nie dotyczy	Dowolny
Koszt predykcji	$O(n)$	$O(n \log n)$
Normalizacja	Wymagana	Wymagana

Wspólne cechy:

- Oparte na odległościach/sąsiedztwie
- Wrażliwe na skalę cech (wymagają normalizacji)
- Problemy z wysoką wymiarowością

Podsumowanie

Omówione zagadnienia:

1. **kNN:** Prosty algorytm klasyfikacji oparty na sąsiadach
2. **Podział danych:** Train/validation/test dla rzetelnej oceny modelu
3. **Metryki:** Accuracy, Precision, Recall, F1-Score, AUC
   - Różne metryki dla zbalansowanych i niezbalansowanych klas
4. **Overfitting/Underfitting:** Bias-variance tradeoff
5. **Cross-validation:** Bardziej wiarygodna ocena generalizacji
6. **DBSCAN:** Klasteryzacja oparta na gęstości z wykrywaniem szumu

Kluczowa lekcja:

- Nie ma uniwersalnego algorytmu ani metryki
- Zawsze dostosowuj metodę do problemu
- Walidacja jest kluczowa dla sukcesu modelu!

Źródła

1. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*
2. Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*
3. *https://scikit-learn.org/stable/documentation.html*
4. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*
5. *https://github.com/janchorowski/ml_uwr*
6. Ester, M., et al. (1996). A density-based algorithm for discovering clusters. *KDD-96*