Odkrywanie konceptu jadalnego grzyba z wykorzystaniem algorytmu ID3

Projekt

Komputerowe wspomaganie decyzji

Mateusz Olejarz

Grzegorz Paryś

1. Opis danych wejściowych

Dane na których będziemy pracować pochodzą ze strony http://archive.ics.uci.edu/ml/datasets/Mushroom.
Przedstawiony zbiór danych reprezentuje 8124 grzyby, z których każdy został sklasyfikowany jako jadalny, niejadalny lub o nieznanej jadalności i niezalecany. Te o nieznanej jadalności uznane zostały jako trujące i znajdują sie w grupie niejadalnych. W zbiorze znajduje się 4208 grzybów jadalnych, którym nadana została etykieta e oraz 3916 trujących o etykiecie p.

Zbiór opiera się na 22 atrybutach:

- Kształt czapeczki – *cap-shape*
- Powierzchnia czapeczki – *cap-surface*
- Kolor czapeczki – *cap-color*
- Zasinienie – *bruises*
- Zapach – *odor*
- Przyczepienie blaszek – *gill-atachement*
- Rozmieszczenie blaszek – *gill-spacing*
- Rozmiar blaszek – *gill-size*
- Kolor blaszek – *gill-color*
- Kształt łodygi – *stalk-shape*
- Korzeń łodygi – *stalk-root*
- Powierzchnia ponad pierścieniem – *stalk-surface-above-ring*
- Powierzchnia pod pierścieniem – *stalk-surface-below-ring*
- Kolor łodygi nad pierścieniem – *stalk-color-above-ring*
- Kolor łodygi pod pierścieniem – *stalk-color-below-ring*
- Typ osłony – *veil-type*
- Kolor osłony – *veil-color*
- Liczba pierścieni – *ring-number*
- Typ pierścienia – *ring-type*
- Kolor wydruku zarodników – *spore-print-color*
- Populacja – *population*
- Siedlisko – *habitat*

Dane przedstawione są za pomocą łańcuchów tekstu, co rozwiązujemy za pomocą **preprocessing** z biblioteki **sklearn.**

```
# Zamieniamy dane z tekstu na liczby, w przeciwnym wypadku nie zadziała DecisionTreeClassifier
le = preprocessing.LabelEncoder()
for column in mushroom_data.columns:
    mushroom_data[column] = le.fit_transform(mushroom_data[column])
for column in mushroom_delete_column.columns:
    mushroom_delete_column[column] = le.fit_transform(mushroom_delete_column[column])
for column in mushroom_get_column_mode.columns:
    mushroom_get_column_mode[column] = le.fit_transform(mushroom_get_column_mode[column])
```

Wśród wartości znajduje się 2480 brakujących informacji, co sygnalizowane jest przez znak „?” na miejscu brakującego atrybutu. Przeprowadzamy badanie 3 przypadków.

- Pozostawienie znaków zapytania przy zamienianiu na liczby
- Zastąpienie ich poprzez wartość najczęściej występującą
- Usunięcie wierszy ze znakami zapytania

Chcieliśmy także użyć innych statystyk, takich jak wartość średnia czy mediana, jednak okazało się to niemożliwe.

```
mushroom_missing_left_x = mushroom_data.iloc[:, 1:23]
mushroom_column_mode_x = mushroom_get_column_mode.iloc[:, 1:23]
mushroom_delete_column_x = mushroom_delete_column.iloc[:, 1:23]
mushroom_missing_left_y = mushroom_data.iloc[:, 0]
mushroom_column_mode_y = mushroom_get_column_mode.iloc[:, 0]
mushroom_delete_column_y = mushroom_delete_column.iloc[:, 0]
```

Zbiory danych podzieliliśmy na zbiory uczące oraz trenujące:

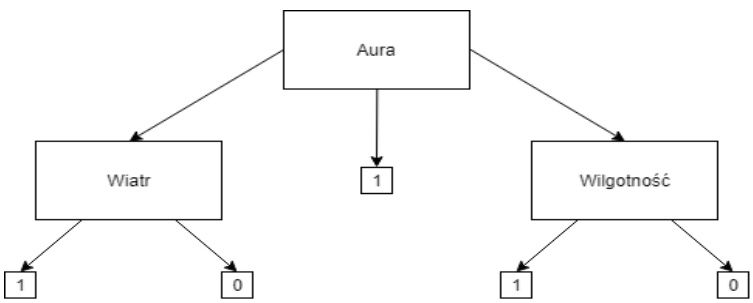
```
missing_left_train_x, missing_left_test_x, missing_left_train_y, missing_left_test_y = \
    train_test_split(mushroom_missing_left_x, mushroom_missing_left_y, train_size=0.8, random_state=1)
mode_train_x, mode_test_x, mode_train_y, mode_test_y = \
    train_test_split(mushroom_column_mode_x, mushroom_column_mode_y, train_size=0.8, random_state=1)
deleted_train_x, deleted_test_x, deleted_train_y, deleted_test_y = \
    train_test_split(mushroom_delete_column_x, mushroom_delete_column_y, train_size=0.8, random_state=1)
```

2. Opis stosowanego narzędzia

Do rozwiązania problemu klasyfikacji w projekcie użyliśmy drzewa decyzyjnego. Jest to etykietowane drzewo (spójny skierowany graf acykliczny), które korzysta z graficznego przedstawienia decyzji i ich możliwych konsekwencji. Są one skonstruowane tak, aby pomóc w podejmowaniu decyzji. W ich skład wchodzą:

- Korzeń – odpowiada wszystim możliwym decyzjom
- Węzły wewnętrzne – odpowiadają testom przeprowadzanym na wartościach atrybutów przykładów
- Liście – etykiety kategorii

Przykładowe drzewo decyzyjne:



Zalety drzew decyzyjnych:

- Są proste do zrozumienia i interpretacji
- Brak konieczności przygotowywania danych
- Są w stanie obsługiwać zarówno dane numeryczne jak i kategoryczne
- Można je poprawić za pomocą metod statystycznych
- Szybko wykonują operacje na dużych ilościach danych

Ogólny algorytm zstępującego budowania drzewa decyzyjnego:

Ważnym elementem budowy drzewa jest podjęcie decyzji, czy obecnie rozpatrywany węzeł ma być liściem. Jeśli tak, to należy przypisać my odpowiednią etykietę kategorii. W przeciwnym wypadku tworzymy kolejny węzeł z testem, którego wynikom będą odpowieadały gałęzie prowadzące do poddrzew budowanych w ten sam sposób. Różne rozwiązanie problemu podjęcia decyzji liść-węzeł oraz wyboru odpowiedniego testu są przyczyną powstania wielu różnych algorytmów konstruwoania drzew decyzyjnych.

```
function buduj-drzewo(P,d,S)
{
// podjęcie decyzji czy węzeł jest końcowy - liść
  if kryterium-stopu(P,S) then
     utwórz liść l;
     d1:=kategoria(P,d);
     return l;
  endif;

//wybranie najlepszego  testu dzielącego i rekurencyjny podział na pod-węzły
  utwórz węzeł n;
  t_n:=wybierz-test(P,S);
  d:=kategoria(P,d);
  for all r∈R_tn do
    n[r]:=buduj-drzewo(P_tnr,d,S-{t_n});
  endfor
  return n
}
```

agumenty wejściowe funkcji:

- P – zbiór etykietowanych przykładów pojęcia c,
- d – domyślna etykieta kategorii,
- S – zbiór możliwych testów

argument zwracany:

- węzeł-korzeń drzewa decyzyjnego reprezentującego hipotezę przybliżającą (aproksymującą) pojęcie c na zbiorze przykładów P

W zależności od postaci funkcji *kryterium-stopu*, *kategoria* i *wybierz-test* powstać mogą różne algorytmy, które będą konstruować drzewa o różnych wielkościach i własnościach.

Testy

Efektywność drewa zależy w dużym stopniu od wyboru testu, to od niego zależy złożoność końcowa drzewa. Dla różnych atrybutów istnieją różne rodzaje testów.

a) Atrybuty nominalne

Dla tej grupy stosuje się test sprawdzający wartość atrybutu. Wyróżniamy trzy rodzaje testów:

- *testy tożsamościowe* - wynikiem testu jest wartość atrybutu - dzieli przykłady na tyle zbiorów, ile jest wartości atrybutów
- *testy równościowe* - sprawdza wartość atrybutu w stosunku do zadanej wartości - dzieli przykłady uczące na dwa zbiory - tych których wartość jest równa podanej wartości i pozostałe
- *testy przynależnościowe* - sprawdza przynależność wartości atrybutu do określonego zbioru wartości - dzieli przykłady na dwa zbiory - te, których wartości należą do podanego zbioru wartości i pozostałe

b) Atrybuty porządkowe

Można dla nich stosować te same rodzaje testów co dla atrybutów nominalnych, jednakże tracona jest zależność porządku między atrybutami, co może prowadzić do skomplikowania drzewa.

c) Atrybuty ciągłe

Przy tej grupie atrybutów stosuje się testy przynależnościowe, których podzbiorem dziedziny jest pewnie przedział

Zarówno dla atrybutów porządkowych jak i ciągłych stosuje się również *testy nierównościowe* bazujące na relacji porządku na wartościach atrybutów. Przykłady dzieli się na dwa zbiory pod względem wartości atrybutu - mniejszej lub większej od podanej.

Przycinanie Drzew

Jest to technika stosowana aby uniknąć przeuczenia, zwiększyć dokładność klasyfikacji dla danych rzeczywistych oraz do uproszczenia budowy drzewa. Polega ono na zastąpieniu poddrzewa liściem reprezentującym kategorię naczęściej występującą w usuwanym poddrzewie. Kategoria ta nazwyana jest kategowią większościową.

Przycinanie dokonywane jest na podstawie zbioru przykładów nazywanego zbiorem przycinania. Algorytm przycinania wygląda następująco:

```
przytnij_drzewo (T - drzewo do przycięcia, P - zbiór przycinania)
{
  1. dla wszystkich węzłów n drzewa T wykonaj:
  2.  zastąp n liściem l z etykietą większościowej kategorii w zbiorze P_T,n
      jeśli nie powiększy to szacowanego na podstawie P błędu rzeczywistego
drzewa T;
  3. koniec dla
  4. zwróć T
}
```

Wyróżnia się dwa rodzaje przycinania ze względu na postać zbioru przycinania:

1. Zbiór przycinania jest równy zbiorowi trenującemu - do przycinania na podstawie zbioru trenującego sotsuje się heurystyki.
2. Zbiór przycinania jest różny od zbioru trenującego - podejście to stosowane jest gdy dostępna jest wystarczająca ilość przykładów trenujących. Wówczas przeprowadza się budowę drzewa na np. 2/3 z nich, pozostawiając 1/3 jako zbiór przycinania.

Opis stosowanego przez nas algorytmu ID3

Algorytm ID3 stworzony został w 1986r. przez Rossa Quinlana, a jego cechą charakterystyczną jest wybór atrybutów dla których przeprowadza się takie testy, aby drzewo było jak najprostsze i jak najefektywniejsze. Aby obliczyć który z atrybutów da największy przyrost informacji oblicza się entropię.

Entropia definiowana jest jako średnia ilość informacji, przypadająca na znak symbolizujący zajście zdarzenia z pewnego zbioru. Zdarzenia w tym zbiorze mają przypisane prawdopodobieństwa wystąpienia.

Wzór na entropię:

$$H(x) = \sum_{i=1}^{n} p(i) \log \frac{1}{p(i)} = -\sum_{i=1}^{n} p(i) \log p(i)$$

gdzie p(i) to prawdopodobieństwo zajścia zdarzenia i

Własności entropii:

- Nieujemna
- Maksymalna, gdy prawdopodobieństwa zajść zdarzeń są takie same
- Równa 0, gdy stany przyjmują wartość 0 lub 1
- Superpozycja – gdy dwa systemy są niezależne to entropia sumy systemów równa się sumie entropii

Treść algorytmu:

1. Oblicz entropię dla każdego atrybutu
2. Wybierz atrybut **A** z najniższą entropią
3. Podziel zbiór przykładów uczących ze względu na wartość atrybutu **A** na rozłączne podzbiory
4. Dodaj do drzewa krawędzie z warunkami:

   ```
   jeśli A=a1 to ... (poddrzewo 1)
   jeśli A=a2 to ... (poddrzewo 2)
   ...
   ```

5. Dla każdego poddrzewa wykonaj kroki od 1.
6. W każdej iteracji jeden atrybut jest usuwany. Algorytm zatrzymuje się, gdy do rozpatrzenia nie pozostanie juz żaden atrybut lub wszystkie przykłady w danym podrzewie mają tą samą wartość atrybutu decyzyjnego.

Problemy w stosowaniu algorytmu ID3:

- istnieje wiele obiektów opisanych takimi samymi atrybutami z taką samą decyzją - należy wyeliminować przykłady nie wnoszące nowych informacji
- istnieją obiekty z brakami danych - należy braki uzupełnić wprowadzając na miejsce przykładu z brakami przykłady z wszystkimi możliwymi kombinacjami wartości brakujących atrybutów
- nadmierne rozrastanie się drzewa – należy dokonać przycinania drzewa.
- bardzo duża baza danych – należy przeprowadzić algorytm na losowej próbce zamiast na całości danych

3. Uzyskana jakość

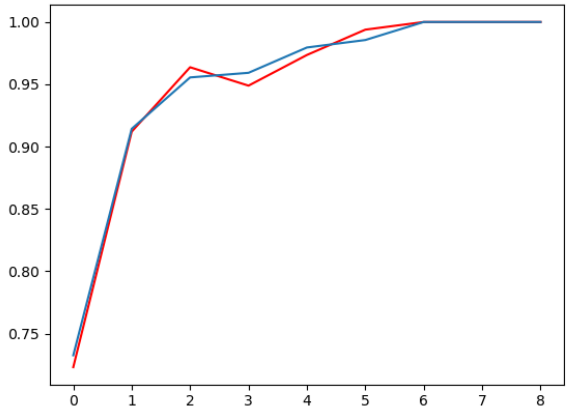
Jakość, w zależności od zastosowanego rozwiązania poradzenia sobie z wartościami brakującymi wynosiła:

- Przy pozostawnieniu znaków zapytania – **0.7015057573073517** – **1.0**
- Przy zastąpieniu znaków zapytania najczęściej występującymi wartościami – **0.8795394154118689** – **1.0**
- Przy usunięciu wierszy ze znakami zapytania – **0.5483076923076923** – **1.0**

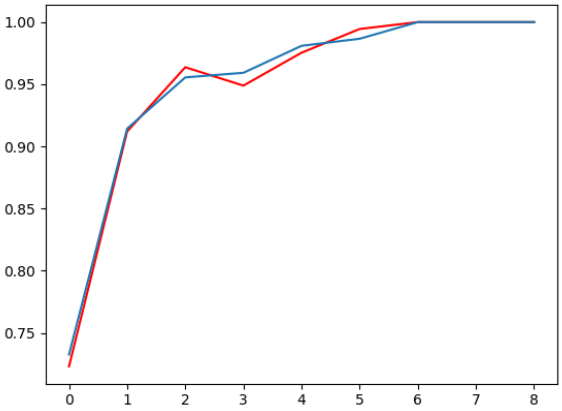
Usunięcie wierszy okazało się najgorszym rozwiązaniem, odjęło znaczną część informacji. Nie biorąc pod uwagę przypadku, w którym rozpoznawał dane, na których się uczył, jego skuteczność w najlepszym przypadku wynosiła jedynie **0.6984615384615385**

Wykresy poszczególnych rozwiązań:

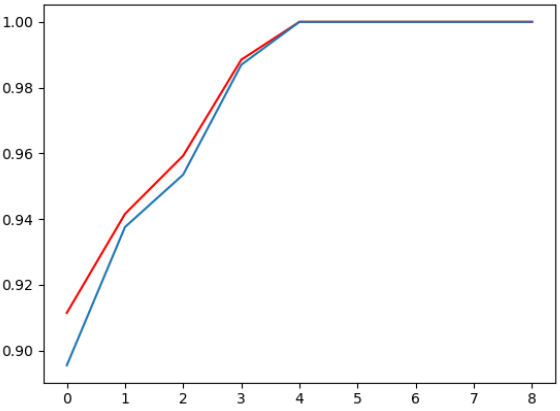
- Pozostawienie brakujących argumentów przy zamianie na typ liczbowy:



- Zastąpienie znaków, tymi występującymi najczęściej:



- Usunięcie wierszy z brakującymi danymi:



Niebieski *– wyniki przy zbiorze trenującym*

Czerwony *– wyniki przy zbiorze testowym*

4. Analiza zmian parametrów konfiguracyjnych

Korzystne okazało się zmienienie **splittera** z domyślnie *best* na *random,* szczególnie dla przypadku, w którym część wierszy została usunięta. Wzrost dokładności był na tyle duży, że pomimo spadku dokładnościu przy innych metodach, ogólna dokładność wzrosła o 3%.

5. Kod

Link do repozytorium: *https://github.com/olejarz-mateusz/DrzewoDycyzyjneID3*

Kod:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import tree
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
import seaborn as sn

# Otworzenie pliku z danymi
colNames=['target','cap-shape', 'cap-surface', 'cap-color', 'bruises',
'odor', 'gill-attachment', 'gill-spacing', 'gill-size', 'gill-color',
'stalk-shape', 'stalk-root', 'stalk-surface-above-ring', 'stalk-surface-
below-ring', 'stalk-color-above-ring', 'stalk-color-below-ring', 'veil-
type', 'veil-color', 'ring-number', 'ring-type', 'spore-print-color',
'population','habitat']
mushroom_data = pd.read_csv('agaricus-lepiota.data', sep=',',
names=colNames)

# Wyświetlenie informacji o zbiorze
print("Training dataset:")
print("patients_train_data:", mushroom_data.shape)

# Tworzymy z niego dwa kolejne zestawy w celu modyfikacji kolumny z
brakujacymi wartościami
mushroom_get_column_mode = mushroom_data.copy()
mushroom_delete_column = mushroom_data.copy()

# Brakujące wartości 11 kolumny zastępujemy poprzez usunięcie wierszy, w
których występuje znak zapytania lub zastąpienie ich najczęściej
występującą wartością. Niestety nie udało się wykonać uzupełnienia innymi
funkcjami statystycznymi, jak mediana lub średnia. Sprawdzimy też co stanie
się po zostawieniu znaków zapytania.
mode_value = mushroom_data.mode().iloc[:,11]

mushroom_get_column_mode.replace("?", np.nan, inplace=True)
mushroom_get_column_mode.replace(np.nan, mode_value[0], inplace=True)
mushroom_delete_column.replace("?", np.nan, inplace=True)
mushroom_delete_column.dropna(inplace=True)

# Zamieniamy dane z tekstu na liczby, w przeciwnym wypadku nie zadziała
DecisionTreeClassifier
le = preprocessing.LabelEncoder()
for column in mushroom_data.columns:
    mushroom_data[column] = le.fit_transform(mushroom_data[column])
for column in mushroom_delete_column.columns:
    mushroom_delete_column[column] =
```

```
le.fit_transform(mushroom_delete_column[column])
for column in mushroom_get_column_mode.columns:
    mushroom_get_column_mode[column] =
le.fit_transform(mushroom_get_column_mode[column])

mushroom_missing_left_x = mushroom_data.iloc[:, 1:23]
mushroom_column_mode_x = mushroom_get_column_mode.iloc[:, 1:23]
mushroom_delete_column_x = mushroom_delete_column.iloc[:, 1:23]
mushroom_missing_left_y = mushroom_data.iloc[:, 0]
mushroom_column_mode_y = mushroom_get_column_mode.iloc[:, 0]
mushroom_delete_column_y = mushroom_delete_column.iloc[:, 0]

missing_left_train_x, missing_left_test_x, missing_left_train_y,
missing_left_test_y = \
    train_test_split(mushroom_missing_left_x, mushroom_missing_left_y,
train_size=0.8, random_state=1)
mode_train_x, mode_test_x, mode_train_y, mode_test_y = \
    train_test_split(mushroom_column_mode_x, mushroom_column_mode_y,
train_size=0.8, random_state=1)
deleted_train_x, deleted_test_x, deleted_train_y, deleted_test_y = \
    train_test_split(mushroom_delete_column_x, mushroom_delete_column_y,
train_size=0.8, random_state=1)

left_model = tree.DecisionTreeClassifier(criterion='entropy', max_depth=10,
class_weight='balanced', random_state=1)
mode_model = tree.DecisionTreeClassifier(criterion='entropy', max_depth=10,
class_weight='balanced', random_state=1)
deleted_model = tree.DecisionTreeClassifier(criterion='entropy',
max_depth=10, class_weight='balanced', random_state=1)

left_model.fit(missing_left_train_x, missing_left_train_y)
mode_model.fit(mode_train_x, mode_train_y)
deleted_model.fit(deleted_train_x, deleted_train_y)

# Wyniki poszczególnych modeli na różnych danych testowych
print(left_model.score(missing_left_test_x, missing_left_test_y))
print(left_model.score(mode_test_x, mode_test_y))
print(left_model.score(deleted_test_x, deleted_test_y))
print(mode_model.score(missing_left_test_x, missing_left_test_y))
print(mode_model.score(mode_test_x, mode_test_y))
print(mode_model.score(deleted_test_x, deleted_test_y))
print(deleted_model.score(missing_left_test_x, missing_left_test_y))
print(deleted_model.score(mode_test_x, mode_test_y))
print(deleted_model.score(deleted_test_x, deleted_test_y))

# Wykres wyników poszczególnych modeli
test_scores_left = []
train_scores_left = []
for i in range(1, 10):
    graph1 = tree.DecisionTreeClassifier(criterion='entropy',
random_state=1, max_depth=i)
    graph1.fit(missing_left_train_x, missing_left_train_y)
    test_scores_left.append(graph1.score(missing_left_test_x,
missing_left_test_y))
    train_scores_left.append(graph1.score(missing_left_train_x,
missing_left_train_y))
plt.plot(test_scores_left, color='red')
plt.plot(train_scores_left)
plt.show()

test_scores_mode = []
```

```
train_scores_mode = []
for i in range(1, 10):
    graph2 = tree.DecisionTreeClassifier(criterion='entropy',
random_state=1, max_depth=i)
    graph2.fit(mode_train_x, mode_train_y)
    test_scores_mode.append(graph2.score(mode_test_x, mode_test_y))
    train_scores_mode.append(graph2.score(mode_train_x, mode_train_y))
plt.plot(test_scores_mode, color='red')
plt.plot(train_scores_mode)
plt.show()

test_scores_deleted = []
train_scores_deleted = []
for i in range(1, 10):
    graph2 = tree.DecisionTreeClassifier(criterion='entropy',
random_state=1, max_depth=i)
    graph2.fit(deleted_train_x, deleted_train_y)
    test_scores_deleted.append(graph2.score(deleted_test_x,
deleted_test_y))
    train_scores_deleted.append(graph2.score(deleted_train_x,
deleted_train_y))
plt.plot(test_scores_deleted, color='red')
plt.plot(train_scores_deleted)
plt.show()

# Macierze konfuzji poszególnych modeli
left_model_predictions = left_model.predict(missing_left_test_x)
left_model_cnf_matrix = confusion_matrix(missing_left_test_y,
left_model_predictions)
sn.heatmap(left_model_cnf_matrix)

mode_model_predictions = mode_model.predict(mode_test_x)
mode_model_cnf_matrix = confusion_matrix(mode_test_y,
mode_model_predictions)
sn.heatmap(mode_model_cnf_matrix)

deleted_model_predictions = deleted_model.predict(deleted_test_x)
deleted_model_cnf_matrix = confusion_matrix(deleted_test_y,
deleted_model_predictions)
sn.heatmap(deleted_model_cnf_matrix)
```