Haskell na produkcji

czyli czy warto sięgać po niestandardowe narzędzia

Tomasz Tylec

11 listopada 2018

IF Research Polska (Gdańsk) / Mazars-Zettafox (Paris) | *tomasz.tylec@ifresearch.pl*

Outline

Reguły klasyfikujące

Definicja

Regułą nazywamy predykat, wyrażenie logiczne, które każdemu elementowi w zbiorze danych przypisuje wartość *prawda* lub *fałsz*.

Przykład

Dane:

index	A	B	C
0	low	high	low
1	low	low	low
2	high	low	high
3	medium	low	high

Reguły

Definicja

Regułą nazywamy predykat, wyrażenie logiczne, które każdemu elementowi w zbiorze danych przypisuje wartość *prawda* lub *fałsz*.

Przykład

Dane:

index	A	B	C
0	low	high	low
1	low	low	low
2	high	low	high
3	medium	low	high

Reguła: $A \in \{\text{low}, \text{medium}\} \wedge B \in \{\text{low}\}$

Rozważać będziemy reguły postaci:

$$c_1 \wedge c_2 \wedge \ldots \wedge c_n,$$

gdzie c_i jest postaci $X \in \{\ldots\}$, lub $X \in (a, b)$.

Definicja

Regułą nazywamy predykat, wyrażenie logiczne, które każdemu elementowi w zbiorze danych przypisuje wartość *prawda* lub *fałsz*.

Przykład

Dane:

index	A	B	C
0	low	high	low
1	low	low	low
2	high	low	high
3	medium	low	high

Reguła: $A \in \{\text{low}, \text{medium}\} \ \wedge \ B \in \{\text{low}\}$

Rozważać będziemy reguły postaci:

$$c_1 \ \wedge \ c_2 \ \wedge \ \ldots \ \wedge \ c_n,$$

gdzie c_i jest postaci $X \in \{\ldots\}$, lub $X \in (a, b)$.

Algebra reguł

Reguły tworzą zbiór częściowo uporządkowany

$r \leq q$ jeśli dla każdego wiersza dla którego q jest prawdziwe, r też jest prawdziwe.

Przykłady:

- $A \in \{\text{low}\} \wedge B \in \{\text{low}\} \leq A \in \{\text{low}, \text{medium}\} \wedge B \in \{\text{low}\}$
- ale $A \in \{\text{low}\}$ oraz $B \in \{high\}$ jest są w relacji porządku.

Algebra reguł

Reguły tworzą zbiór częściowo uporządkowany

$r \leq q$ jeśli dla każdego wiersza dla którego q jest prawdziwe, r też jest prawdziwe.

Przykłady:

- $A \in \{\text{low}\} \wedge B \in \{\text{low}\} \leq A \in \{\text{low}, \text{medium}\} \wedge B \in \{\text{low}\}$
- ale $A \in \{\text{low}\}$ oraz $B \in \{high\}$ jest są w relacji porządku.

Reguły można ze sobą łączyć

Jeśli r, q są regułą to $r \wedge q$ też jest regułą. Wtedy $r \wedge q \leq r$ oraz $r \wedge q \leq q$.

Rozważać będziemy reguły postaci:

$$c_1 \wedge c_2 \wedge \ldots \wedge c_n,$$

gdzie c_i jest postaci $X \in \{\ldots\}$, lub $X \in (a, b)$.

Jeśli r, q są regułą to $r \vee q$ definiujemy jako najmniejszą (w sensie $\leq$) regułę spełniającą: $r \leq r \vee q$ oraz $q \leq r \vee q$.

Zadanie

Mając do dyspozycji zbiór sklasyfikowanych danych, znaleźć niewielki zbiór *najlepszych* reguł, względem pewnej funkcji oceniającej (*score function*).

Metoda I: przeszukiwanie drzewa

Zaczynając od zbioru najogólniejszych nietrywialnych reguł $a_1, a_2, \ldots, a_k$, tworzymy reguły postaci:

- $a_2 \wedge a_5$,
- $a_5 \wedge a_{10} \wedge a_{143}$,
- $a_{242} \wedge a_{743} \wedge a_{342} \wedge a_{1233}$,
- itd.

wybierając tylko te najlepsze.

Metoda I: przeszukiwanie drzewa

Zaczynając od zbioru najogólniejszych nietrywialnych reguł $a_1, a_2, \ldots, a_k$, tworzymy reguły postaci:

- $a_2 \wedge a_5$,
- $a_5 \wedge a_{10} \wedge a_{143}$,
- $a_{242} \wedge a_{743} \wedge a_{342} \wedge a_{1233}$,
- itd.

wybierając tylko te najlepsze.

index	A	B	C
0	low	hig	low
1	low	low	low
3	high	low	high
4	medium	low	high

$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$

$\mathrm{Dom}(B) = \{\text{low, high}\}$

$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$

$a_1 = A \in \{\text{low, medium}\}, a_3 = B \in \{\text{low}\}$
$r = a_1 \wedge a_3 = A \in \{\text{low, medium}\} \wedge B \in \{\text{low}\}$

Metoda I: przeszukiwanie drzewa

Zaczynając od zbioru najogólniejszych nietrywialnych reguł $a_1, a_2, \ldots, a_k$, tworzymy reguły postaci:

- $a_2 \wedge a_5$,
- $a_5 \wedge a_{10} \wedge a_{143}$,
- $a_{242} \wedge a_{743} \wedge a_{342} \wedge a_{1233}$,
- itd.

wybierając tylko te najlepsze.

index	A	B	C
0	low	hig	low
1	low	low	low
3	high	low	high
4	medium	low	high

$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$

$\mathrm{Dom}(B) = \{\text{low, high}\}$

$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$

$a_1 = A \in \{\text{low, medium}\}, a_3 = B \in \{\text{low}\}$

$r = a_1 \wedge a_3 = A \in \{\text{low, medium}\} \wedge B \in \{\text{low}\}$

Metoda I: przeszukiwanie drzewa

Zaczynając od zbioru najogólniejszych nietrywialnych reguł $a_1, a_2, \ldots, a_k$, tworzymy reguły postaci:

- $a_2 \wedge a_5$,
- $a_5 \wedge a_{10} \wedge a_{143}$,
- $a_{242} \wedge a_{743} \wedge a_{342} \wedge a_{1233}$,
- itd.

wybierając tylko te najlepsze.

index	A	B	C
0	low	hig	low
1	low	low	low
3	high	low	high
4	medium	low	high

$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$

$\mathrm{Dom}(B) = \{\text{low, high}\}$

$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$

$a_1 = A \in \{\text{low, medium}\}$, $a_3 = B \in \{\text{low}\}$

$r = a_1 \wedge a_3 = A \in \{\text{low, medium}\} \wedge B \in \{\text{low}\}$

Metoda I: przeszukiwanie drzewa

Zaczynając od zbioru najogólniejszych nietrywialnych reguł $a_1, a_2, \ldots, a_k$, tworzymy reguły postaci:

- $a_2 \wedge a_5$,
- $a_5 \wedge a_{10} \wedge a_{143}$,
- $a_{242} \wedge a_{743} \wedge a_{342} \wedge a_{1233}$,
- itd.

wybierając tylko te najlepsze.

index	A	B	C
0	low	hig	low
1	low	low	low
3	high	low	high
4	medium	low	high

$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$

$\mathrm{Dom}(B) = \{\text{low, high}\}$

$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$

$a_1 = A \in \{\text{low, medium}\}, a_3 = B \in \{\text{low}\}$

$r = a_1 \wedge a_3 = A \in \{\text{low, medium}\} \wedge B \in \{\text{low}\}$

Metoda II: przeszukiwanie podzbiorów

Każdy element zbioru danych x_i odpowiada regule $r(x_i)$– takiej, która wybiera dokładnie ten element. Tworzymy reguły postaci:

- $r(x_{10}) \vee r(x_{31})$,
- $r(x_1) \vee r(x_{45}) \vee r(x_{321}) \vee r(x_{1421}) \vee r(x_{2352}) \vee r(x_{31002})$
- itp.

wybierając najlepsze.

Metoda II: przeszukiwanie podzbiorów

Każdy element zbioru danych x_i odpowiada regule $r(x_i)$– takiej, która wybiera dokładnie ten element. Tworzymy reguły postaci:

- $r(x_{10}) \lor r(x_{31})$,
- $r(x_1) \lor r(x_{45}) \lor r(x_{321}) \lor r(x_{1421}) \lor r(x_{2352}) \lor r(x_{31002})$
- itp.

wybierając najlepsze.

index	A	B	C
0	low	high	low
1	low	low	low
3	low	low	high
4	medium	low	high

$$\mathbf{Dom}(A) = \{\text{low, medium, high}\}$$
$$\mathbf{Dom}(B) = \{\text{low, high}\}$$
$$\mathbf{Dom}(C) = \{\text{low, medium, high}\}$$

$$a_1 = A \in \{\text{low}\} \land B \in \{\text{low}\} \land C \in \{\text{low}\}$$
$$a_2 = A \in \{\text{medium}\} \land B \in \{\text{low}\} \land C \in \{\text{high}\},$$
$$r = a_1 \land a_2 = A \in \{\text{low, medium}\} \land B \in \{\text{low}\} \land C \in \{\text{low, high}\}$$

Metoda II: przeszukiwanie podzbiorów

Każdy element zbioru danych x_i odpowiada regule $r(x_i)$– takiej, która wybiera dokładnie ten element. Tworzymy reguły postaci:

- $r(x_{10}) \lor r(x_{31})$,
- $r(x_1) \lor r(x_{45}) \lor r(x_{321}) \lor r(x_{1421}) \lor r(x_{2352}) \lor r(x_{31002})$
- itp.

wybierając najlepsze.

index	A	B	C
0	low	high	low
1	low	low	low
3	low	low	high
4	medium	low	high

$$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$$
$$\mathrm{Dom}(B) = \{\text{low, high}\}$$
$$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$$

$a_1 = A \in \{\text{low}\} \land B \in \{\text{low}\} \land C \in \{\text{low}\}$
$a_2 = A \in \{\text{medium}\} \land B \in \{\text{low}\} \land C \in \{\text{high}\}$,
$r = a_1 \land a_2 = A \in \{\text{low, medium}\} \land B \in \{\text{low}\} \land C \in \{\text{low, high}\}$

Metoda II: przeszukiwanie podzbiorów

Każdy element zbioru danych x_i odpowiada regule $r(x_i)$– takiej, która wybiera dokładnie ten element. Tworzymy reguły postaci:

- $r(x_{10}) \lor r(x_{31})$,
- $r(x_1) \lor r(x_{45}) \lor r(x_{321}) \lor r(x_{1421}) \lor r(x_{2352}) \lor r(x_{31002})$
- itp.

wybierając najlepsze.

index	A	B	C
0	low	high	low
1	low	low	low
3	low	low	high
4	medium	low	high

$$\mathrm{Dom}(A) = \{\text{low, medium, high}\}$$
$$\mathrm{Dom}(B) = \{\text{low, high}\}$$
$$\mathrm{Dom}(C) = \{\text{low, medium, high}\}$$

$a_1 = A \in \{\text{low}\} \land B \in \{\text{low}\} \land C \in \{\text{low}\}$
$a_2 = A \in \{\text{medium}\} \land B \in \{\text{low}\} \land C \in \{\text{high}\}$,
$r = a_1 \land a_2 = A \in \{\text{low, medium}\} \land B \in \{\text{low}\} \land C \in \{\text{low, high}\}$

Metoda II: przeszukiwanie podzbiorów

Każdy element zbioru danych x_i odpowiada regule $r(x_i)$– takiej, która wybiera dokładnie ten element. Tworzymy reguły postaci:

- $r(x_{10}) \vee r(x_{31})$,
- $r(x_1) \vee r(x_{45}) \vee r(x_{321}) \vee r(x_{1421}) \vee r(x_{2352}) \vee r(x_{31002})$
- itp.

wybierając najlepsze.

index	A	B	C
0	low	high	low
1	low	low	low
3	low	low	high
4	medium	low	high

$$\mathbf{Dom}(A) = \{\text{low, medium, high}\}$$
$$\mathbf{Dom}(B) = \{\text{low, high}\}$$
$$\mathbf{Dom}(C) = \{\text{low, medium, high}\}$$

$$a_1 = A \in \{\text{low}\} \wedge B \in \{\text{low}\} \wedge C \in \{\text{low}\}$$
$$a_2 = A \in \{\text{medium}\} \wedge B \in \{\text{low}\} \wedge C \in \{\text{high}\},$$
$$r = a_1 \wedge a_2 = A \in \{\text{low, medium}\} \wedge B \in \{\text{low}\} \wedge C \in \{\text{low, high}\}$$

Problemy

Obie metody wymagają:

- szybkiego liczenia funkcji *score* (wiele reguł do sprawdzenia);
- efektywnej implementacji algebry reguł (wiele reguł do skonstruowania);
- stosowania heurystycznych optymalizacji.

Problemy

Obie metody wymagają:

- szybkiego liczenia funkcji *score* (wiele reguł do sprawdzenia);
- efektywnej implementacji algebry reguł (wiele reguł do skonstruowania);
- stosowania heurystycznych optymalizacji.

W konsekwencji:

- kod będzie się często zmieniał na etapie tworzenia prototypu
- nie wiadomo z góry, która reprezentacja reguł będzie optymalna
- może istnieć więcej niż jeden sposób heurystycznej optymalizacji przeszukania
- może nie istnieć globalnie najefektywniejsze rozwiązanie

Z drugiej strony, opis algorytmu jest bardzo prosty.

Implementacja w Haskellu

Czym jest Haskell?

General-purpose, purely functional programming language with non-strict semantics and strong static typing [wikipedia]

Czym jest Haskell?

General-purpose, purely functional programming language with non-strict semantics and strong static typing [wikipedia]

- *strongly typed:* wszystko ma określony typ; kompilator statycznie sprawdza zgodność typów; brak domyślnych konwersji między typami.

```
pi  :: Double
sin :: Double -> Double
```

podobnie jak w matematyce: $\pi \in \mathbb{R}$, $sin : \mathbb{R} \rightarrow \mathbb{R}$

Czym jest Haskell?

General-purpose, purely functional programming language with non-strict semantics and strong static typing [wikipedia]

- *strongly typed:* wszystko ma określony typ; kompilator statycznie sprawdza zgodność typów; brak domyślnych konwersji między typami.

```
pi  :: Double
sin :: Double -> Double
```

  podobnie jak w matematyce: $\pi \in \mathbb{R}$, $sin : \mathbb{R} \to \mathbb{R}$

- *pure:* brak zmiennych i efektów ubocznych

```
a = 5
a = 3         -- to 5 czy 3?!

f :: Int -> Int -> Double
f x y = ...   -- wartość f zależy *tylko* od x i y
```

Czym jest Haskell?

General-purpose, purely functional programming language with non-strict semantics and strong static typing [wikipedia]

- *strongly typed:* wszystko ma określony typ; kompilator statycznie sprawdza zgodność typów; brak domyślnych konwersji między typami.

```haskell
pi  :: Double
sin :: Double -> Double
```

  podobnie jak w matematyce: $\pi \in \mathbb{R}$, $sin : \mathbb{R} \to \mathbb{R}$

- *pure:* brak zmiennych i efektów ubocznych

```haskell
a = 5
a = 3          -- to 5 czy 3?!

f :: Int -> Int -> Double
f x y = ...    -- wartość f zależy *tylko* od x i y
```

- *lazy evaluation:* wartości są liczone dopiero wtedy, gdy są potrzebne:

```haskell
-- lista kwadratów wszystkich liczb naturalnych
squares = [x*x | x <- [0..]]
take 5 squares    -- pierwszych 5 wartości
-- [0, 1, 4, 9, 16]
```

Typy i klasy typów

```haskell
type Rule a = Map Text a
```

- ułatwia prototypowanie: kompilator sam podpowiada co i gdzie jeszcze zmienić;
- mniej błędów w wersji finalnej;
- porządkują proces myślowy.

Typy i klasy typów

```haskell
type Rule a = Map Text a
```

- ułatwia prototypowanie: kompilator sam podpowiada co i gdzie jeszcze zmienić;
- mniej błędów w wersji finalnej;
- porządkują proces myślowy.

Łatwo możemy tworzyć nietrywialne typy:

```haskell
data Tree a = Node a [Tree a]
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```haskell
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Zbiory jako reguły:

```
instance RuleType (Set a) where
  a .<=. b = a `isSubsetOf` b
  a  /\  b = a `intersect` b
  a  \/  b = a `union` b
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Zbiory jako reguły:

```
instance RuleType (Set a) where
  a .<=. b = a `isSubsetOf` b
  a  /\  b = a `intersect` b
  a  \/  b = a `union` b

instance Rule a => RuleType (Rule a) where
  a .<=. b = ...
  a  /\  b = ...
  a  \/  b = ...
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Zbiory jako reguły:

```
instance RuleType (Set a) where
  a .<=. b = a `isSubsetOf` b
  a  /\  b = a `intersect` b
  a  \/  b = a `union` b

instance Rule a => RuleType (Rule a) where
  a .<=. b = ...
  a  /\  b = ...
  a  \/  b = ...

instance Rule a => RuleType [a] where
  a .<=. b = all $ zipWith (.<=.) a b
  a  /\  b = all $ zipWith (/\) a b
  a  \/  b = all $ zipWith (\/) a b
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Zbiory jako reguły:

```
instance RuleType (Set a) where
  a .<=. b = a `isSubsetOf` b
  a  /\  b = a `intersect` b
  a  \/  b = a `union` b

instance Rule a => RuleType (Rule a) where
  a .<=. b = ...
  a  /\  b = ...
  a  \/  b = ...

instance Rule a => RuleType [a] where
  a .<=. b = all $ zipWith (.<=.) a b
  a  /\  b = all $ zipWith (/\) a b
  a  \/  b = all $ zipWith (\/) a b
```

Reprezentacja bitowa:

```
instance RuleType Bool where
  a .<=. b = b `implies` a
  a  /\  b = a && b
```

Klasa reguł

Więcej niż jeden typ może być regułą:

```
class RuleType a where
  (.<=.) :: a -> a -> Bool  -- poset
  (/\)   :: a -> a -> a     -- and for rules
  (\/)   :: a -> a -> a     -- sup for rules
```

Zbiory jako reguły:

```
instance RuleType (Set a) where
  a .<=. b = a `isSubsetOf` b
  a  /\  b = a `intersect` b
  a  \/  b = a `union` b

instance Rule a => RuleType (Rule a) where
  a .<=. b = ...
  a  /\  b = ...
  a  \/  b = ...

instance Rule a => RuleType [a] where
  a .<=. b = all $ zipWith (.<=.) a b
  a  /\  b = all $ zipWith (/\) a b
  a  \/  b = all $ zipWith (\/) a b
```

Reprezentacja bitowa:

```
instance RuleType Bool where
  a .<=. b = b `implies` a
  a  /\  b = a && b

instance RuleType BitArray where
  a .<=. b = not b | a
  a  /\  b = a & b
```

Parametryczny polimorfizm + purity

```haskell
searchRules :: RuleType a, WithScore a =>
            -> (a -> Bool)  -- f. przycinająca
            -> Best a       -- najlepsze reguły
            -> Tree a       -- przestrzeń do przeszukania
            -- codomain
            -> Best a       -- nowe najlepsze reguły
```

Implementacja algorytmu wykorzystuje tylko ogólne cechy argumentów, nie konkretne reprezentacje, tu:

- algebra reguł ($\leq$, $\wedge$);
- dodawanie nowego elementu do zbioru najlepszych reguł.

Parametryczny polimorfizm + purity

```
searchRules :: RuleType a, WithScore a =>
            -> (a -> Bool)  -- f. przycinająca
            -> Best a       -- najlepsze reguły
            -> Tree a       -- przestrzeń do przeszukania
            -- codomain
            -> Best a       -- nowe najlepsze reguły
```

Implementacja algorytmu wykorzystuje tylko ogólne cechy argumentów, nie konkretne reprezentacje, tu:

- algebra reguł ($\leq$, $\wedge$);
- dodawanie nowego elementu do zbioru najlepszych reguł.

Dzięki temu:

- łatwo eksperymentować z różnymi reprezentacji obiektu (korzystamy z dwóch, dwie czy trzy inne były przetestowane trakcie implementacji);
- łatwiej testować poprawność algorytmu (można np. wybrać najprostszą reprezentację reguł);
- bardziej przejrzysty kod.

Parametryczny polimorfizm + purity

```
searchRules :: RuleType a, WithScore a =>
            -> (a -> Bool)  -- f. przycinająca
            -> Best a       -- najlepsze reguły
            -> Tree a       -- przestrzeń do przeszukania
            -- codomain
            -> Best a       -- nowe najlepsze reguły
```

Implementacja algorytmu wykorzystuje tylko ogólne cechy argumentów, nie konkretne reprezentacje, tu:

- algebra reguł ($\leq$, $\wedge$);
- dodawanie nowego elementu do zbioru najlepszych reguł.

Dzięki temu:

- łatwo eksperymentować z różnymi reprezentacji obiektu (korzystamy z dwóch, dwie czy trzy inne były przetestowane trakcie implementacji);
- łatwiej testować poprawność algorytmu (można np. wybrać najprostszą reprezentację reguł);
- bardziej przejrzysty kod.

```
searchRules :: RuleType a, WithScore a => Int -> (a -> Bool) -> Best a -> Tree a -> Best a
searchRules 0 _ _ bestK (Node r _) = update r bestK
searchRules d prune bestK (Node r cs) = foldl' (searchRules (d-1) prune) updated children
  where
    children = filter interesting cs
    updated  = update r bestK
    interesting (Node r _) = not $ prune r
```

Lazy evaluation

Możemy oddzielić tworzenie przestrzeni reguł do przeszukania od samego algorytmu przeszukującego:

```
hSpace :: RuleType r => [r] -> r -> Tree r
hSpace [] r = Node r []
hSpace pool r = Node r $ filter notNull . map subForest . init . tails $ pool
  where
    subForest (u:us) = hSpace us (r /\ u)
    notNull (Node d _) = not . isNull $ d
```

tworzy w praktyce nieskończone drzewo reguł do przeszukania. Możemy je transformować:

```
  space = hSpace pool trivial
  fmap toScored space  -- infinite tree with scores!
```

Wersja Pythonowa?

- liczenie funkcji *score*, z wykorzystaniem pandas/numpy/numexpr było stosunkowo szybkie;
- ale implementacja algebry reguł była okrutnie powolna…
- …lub bardzo nieczytelna, jeśli chcieć ją zoptymalizować (pythran, etc.);
- mało wydajne zrównoleglanie kodu;
- przez większą złożoność kodu, znacznie trudniejsze eksperymentowanie z różnymi wariantami algorytmu.

Wersja Pythonowa?

- liczenie funkcji *score*, z wykorzystaniem pandas/numpy/numexpr było stosunkowo szybkie;
- ale implementacja algebry reguł była okrutnie powolna…
- …lub bardzo nieczytelna, jeśli chcieć ją zoptymalizować (pythran, etc.);
- mało wydajne zrównoleglanie kodu;
- przez większą złożoność kodu, znacznie trudniejsze eksperymentowanie z różnymi wariantami algorytmu.

Statystyki

- liczba linii kodu: ok 1500 (Haskell) vs 3800 (Python)
- w tym kod bezpośrednio związany z implementacją algorytmu: 540 (Haskell) vs 3000 (Python)
- wersja w Haskellu średnio ok. 10 razy szybsza od pythonowej (single core) i znacznie lepiej zrównoleglająca się (do 32 rdzeni, w pythonie: kilka do kilkunastu).

Integracja z ekosystemem Pythona

Największy problem

Haskell ma ubogi ekosystem bibliotek odczytujących dane z różnych formatów do których łatwo można eksportować z pandas + brak standardowej biblioteki typu pandas

Największy problem

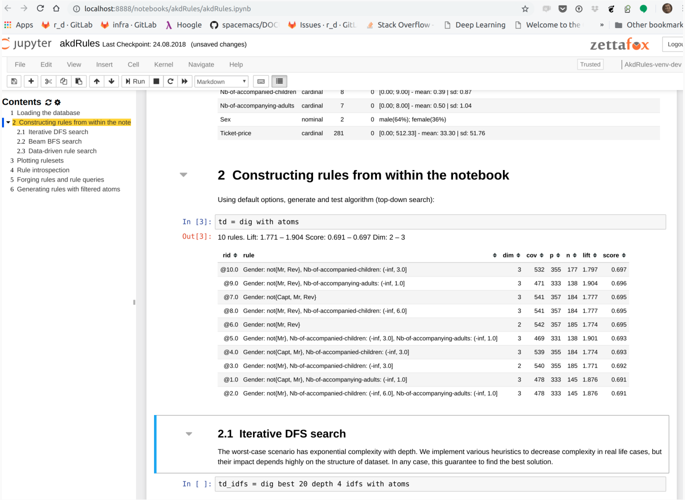
Haskell ma ubogi ekosystem bibliotek odczytujących dane z różnych formatów do których łatwo można eksportować z pandas + brak standardowej biblioteki typu pandas

ARFF

Własna implementacja formatu ARFF (od Weka):

- prosty format tekstowy – niezależny od specyficznych sposób binarnego kodowania (pandas miewa z tym problemy);
- w przeciwieństwie do CSV kolumny mają ściśle określony typ i dziedzinę;
- efekt końcowy: czytanie plików ARFF niewiele wolniejsze od czytania np. HDF5 w pandas;
- niestety konieczna była implementacja ARFF również w Pythonie...

Dedykowany jupyter kernel



jupyter akdRules Last Checkpoint: 24.08.2018 (unsaved changes)

Contents

Nb-of-accompanied-children	cardinal	8	0	[0.00; 9.00] - mean: 0.39 | sd: 0.87
Nb-of-accompanying-adults	cardinal	7	0	[0.00; 8.00] - mean: 0.50 | sd: 1.04
Sex	nominal	2	0	male(64%); female(36%)
Ticket-price	cardinal	281	0	[0.00; 512.33] - mean: 33.30 | sd: 51.76

2 Constructing rules from within the notebook

Using default options, generate and test algorithm (top-down search):

```
In [3]: td = dig with atoms
```

Out[3]: 10 rules. Lift: 1.771 – 1.904 Score: 0.691 – 0.697 Dim: 2 – 3

rid	rule	dim	cov	p	n	lift	score
@10.0	Gender: not{Mr, Rev}, Nb-of-accompanied-children: (-inf, 3.0]	3	532	355	177	1.797	0.697
@9.0	Gender: not{Mr, Rev}, Nb-of-accompanying-adults: (-inf, 1.0]	3	471	333	138	1.904	0.696
@7.0	Gender: not{Capt, Mr, Rev}	3	541	357	184	1.777	0.695
@8.0	Gender: not{Mr, Rev}, Nb-of-accompanied-children: (-inf, 6.0]	3	541	357	184	1.777	0.695
@6.0	Gender: not{Mr, Rev}	2	542	357	185	1.774	0.695
@5.0	Gender: not{Mr}, Nb-of-accompanied-children: (-inf, 3.0], Nb-of-accompanying-adults: (-inf, 1.0]	3	469	331	138	1.901	0.693
@4.0	Gender: not{Capt, Mr}, Nb-of-accompanied-children: (-inf, 3.0]	3	539	355	184	1.774	0.693
@3.0	Gender: not{Mr}, Nb-of-accompanied-children: (-inf, 3.0]	2	540	355	185	1.771	0.692
@1.0	Gender: not{Capt, Mr}, Nb-of-accompanying-adults: (-inf, 1.0]	3	478	333	145	1.876	0.691
@2.0	Gender: not{Mr}, Nb-of-accompanied-children: (-inf, 6.0], Nb-of-accompanying-adults: (-inf, 1.0]	3	478	333	145	1.876	0.691

2.1 Iterative DFS search

The worst-case scenario has exponential complexity with depth. We implement various heuristics to decrease complexity in real life cases, but their impact depends highly on the structure of dataset. In any case, this guarantee to find the best solution.

```
In [ ]: td_idfs = dig best 20 depth 4 idfs with atoms
```

- znajomy interfejs
- pozwala wykorzystać algorytmy wyszukujące reguły do edycji istniejących reguł – przydatne gdy dodaje się wiedzę domenową.

Podsumowanie

Podsumowanie

Dzięki dopasowaniu specyfiki problemu i cech narzędzia, udało się uzyskać:

- wydajniejszy proces tworzenia narzędzia
- kod łatwiejszy w utrzymaniu
- łatwość tworzenia eksperymentalnych wariantów
- większą szybkość działania
- łatwiejsze testowanie
- większa satysfakcja z pracy

Choć bywały trudniejsze momenty:

- brak wsparcia w ekosystemie
- nie jest to droga dla początkujących
- trzeba uważać na przesadną abstrakcję

Dziękuję za uwagę