WEKA
Waikato Environment for Knowledge Analysis

Przemysław Klęsk
pklesk@wi.zut.edu.pl

Zakład Sztucznej Inteligencji
Wydział Informatyki, ZUT

Spis treści

Spis treści

WEKA

- **WEKA** (*Waikato Environment for Knowledge Analysis*) — projekt open-source zaimplementowany w Javie mający na celu zbudowanie środowiska ułatwiającego pracę z algorytmami **uczenia maszynowego** (ang. *machine learning*) do rozwiązywania problemów **eksploracji danych** (ang. *data mining*)
- **Autorzy:** Eibe Frank, Mark Hall i Len Trigg (Uniwersytet Waikato, Hamilton, Nowa Zelandia). Strona uniwersytetu: *http://www.cs.waikato.ac.nz*.
- **Strona główna projektu:** *http://www.cs.waikato.ac.nz/~ml/weka*
- Nazwa i logo pochodzą od nielotnego ptaka występującego unikalnie w Nowej Zelandii.

Najważniejsze cechy

- Możliwość modyfikowania i rozszerzania (GNU General Public Licence).
- Kilka punktów dostępowych:
  - eksplorator (*Explorer*) — lekkie GUI,
  - interfejs linii komend (*Simple Command Line Interface*),
  - eksperymentator (*Experimenter*),
  - graficzny modeler przepływów (*Knowledge Flow*) — odpowiednik eksploratora,
  - możliwość wywoływania algorytmów z zewnętrznych (własnych) programów Javy.
- Obsługa danych poprzez pliki tekstowe: *.arff*, *.exp* (WEKA), oraz .csv.
- Dodatkowo obsługa plików w formacie *C4.5* [1] oraz plików binarnych.
- Możliwość czytania danych przez JDBC lub z URLi.

[1] Popularny w środowiskach data-mining'u. Nazwa pochodzi od algorytmu budowania drzewa decyzyjnego.

Spis treści

Format .arff (ang. *Attribute Relation File Format*) — przykłady

Można znaleźć w: `.../Weka-3-6/data/`

```
@RELATION iris

@ATTRIBUTE sepallength REAL
@ATTRIBUTE sepalwidth  REAL
@ATTRIBUTE petallength  REAL
@ATTRIBUTE petalwidth REAL
@ATTRIBUTE class  {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.1,3.5,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,5.4,2.1,Iris-virginica
6.7,3.1,5.6,2.4,Iris-virginica

⋮
```

.arff — przykłady

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes

⋮
```

.arff — przykłady

```
@relation 'cpu'

@attribute MYCT real
@attribute MMIN real
@attribute MMAX real
@attribute CACH real
@attribute CHMIN real
@attribute CHMAX real
@attribute class real

@data
125,256,6000,256,16,128,198
29,8000,32000,32,8,32,269
29,8000,32000,32,8,32,220
29,8000,32000,32,8,32,172
29,8000,16000,32,8,16,132

⋮
```

.arff — szczegóły

- **Nazwa relacji** (zbioru/problemu):

  `@relation <relation-name>`

  Jeżeli w <relation-name> występują spacje, to należy nazwę obtoczyć apostrofami.

- Format dla **atrybutów**:

  `@attribute <attribute-name> <datatype>`

- Możliwe typy dla `<datatype>`:
  - rzeczywistoliczbowy: `numeric` lub `real` (w zależności od wersji WEKA),
  - wyliczeniowy: `<nominal-specification>`,
  - napisowy: `string`,
  - data: `date [<date-format>]`.
- Domyślny **format daty** wg ISO-8601: `"yyyy-MM-dd'T'HH:mm:ss"`
- **Braki w danych** można reprezentować poprzez: ?

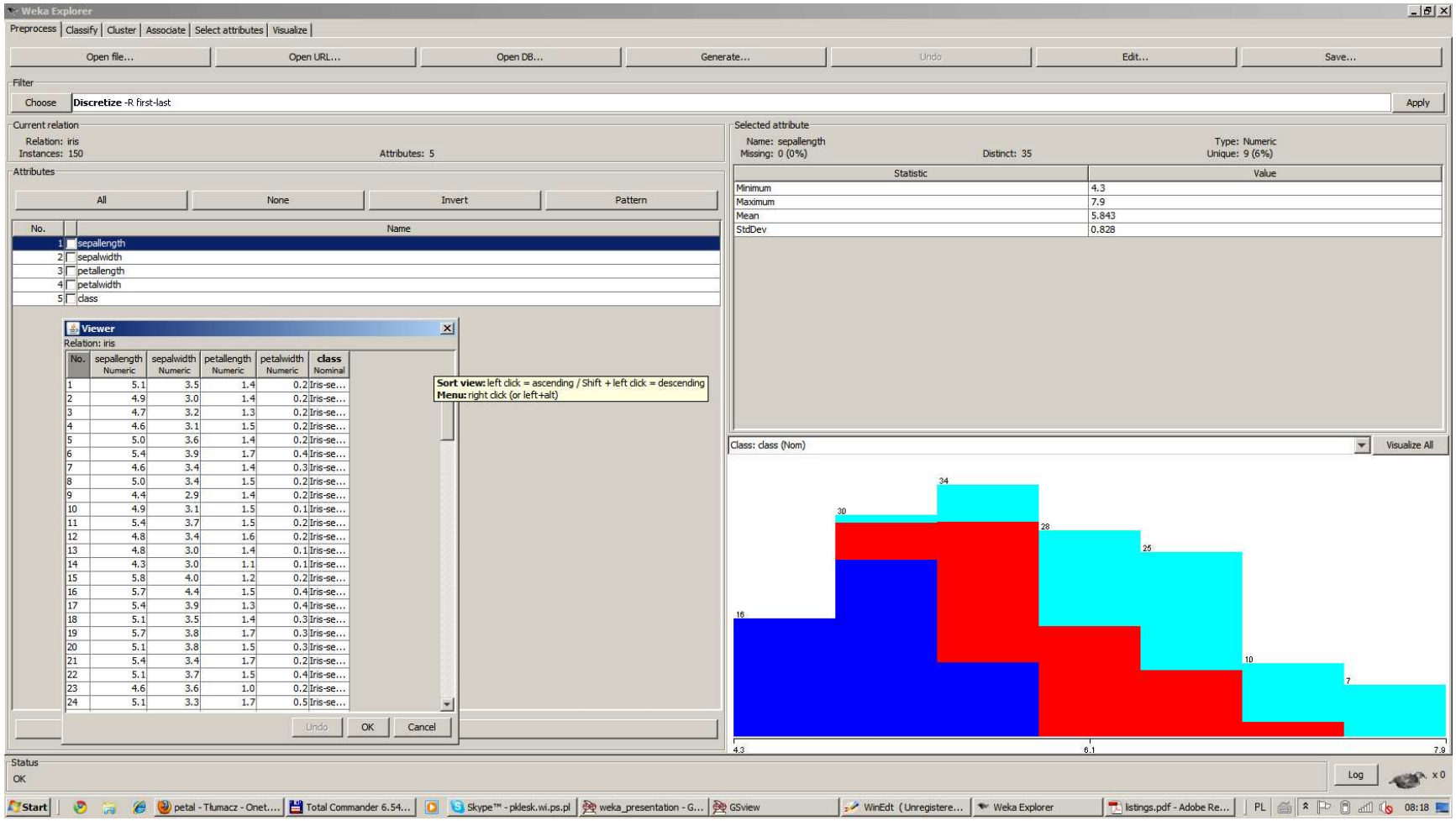
Format C4.5

- Często napotykany przy różnych bench-markowych zbiorach danych. M.in. w repozytorium **UCI Machine Learning Repository** *http://archive.ics.uci.edu/ml/*.
- Na zbiór danych składają się dwa pliki: *.names*, *.data*.
- Pliki *.names* w pierwotnym zamyśle powinny być strukturą z dziedzinami zmiennych (w praktyce często nieustrukturyzowane, ale informacyjne),
- Pliki *.data* — pliki z danymi; wartości oddzielone przecinkami, wiersze znakami nowej linii.
- Poprawne pliki w C4.5 można znaleźć np. na stronie Rossa Quinlama: *http://www.rulequest.com/Personal/* w najnowszym release algorytmu C4.5.

Spis treści

Widok Explorera

Główne zakładki (klasy metod): *Preprocess*, *Classify*, *Cluster*, *Associate*, *Select attributes*, *Visualize*.



Ważniejsze filtry w ramach *Preprocess*

NominalToBinary

Zamienia każdy atrybut wyliczeniowy o k wartościach na k atrybutów binarnych (przydatne dla atrybutów o bardzo licznych dziedzinach).

Discretize (w wersji z nadzorem — *supervised*)

Zamienia wskazane (w polu *attributeIndices*) atrybuty rzeczywistoliczbowe na dyskretne dzieląc oś na przedziały. Dostępne są dwie metody:

- Minimum Description Length (Fayyad, Irani) — rekurencyjne ustalanie szufladek podziałowych oceniane wskaźnikiem *information gain* (inaczej *Kullback-Leibler's divergence*),
- MDL z kryterium Konenenki.

Przykład dla zbioru *glass*: naiwny klasyfikator Bayesa bez i z uprzednią dyskretyzacją atrybutów.

Ważniejsze filtry w ramach *Preprocess*

Attribute Selection

Duża grupa metod do wyboru podzbioru atrybutów możliwie silnie skorelowanych ze zmienną wyjściową i możliwie słabo skorelowanych między sobą. Na selekcję składają się zawsze dwa elementy: **sposób oceniania** i **sposób przeszukiwania** (sposób przebiegania podzbiorów zbioru atrybutów, ekonomiczniejszy niż zachłanny).

Przykład dla zbioru *diabetes*: naiwny klasyfikator Bayesa bez i z uprzednią selekcją atrybutów.

Ważniejsze filtry w ramach *Preprocess*

Resample

Zaczerpnięcie próby (z, lub bez zwracania) ze zbioru danych. Może być przydatne przy dużych zbiorach danych do wstępnej analizy.

MultiFilter

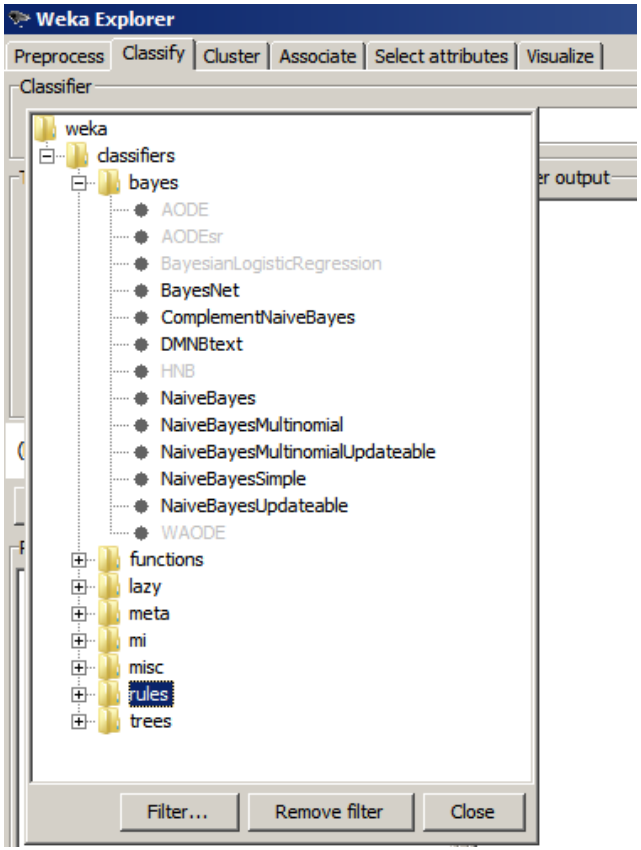
Pozwala tworzyć, zapisywać i uruchamiać sekwencje filtrów.

Niektóre filtry na szczególne okazje

- *Add* — pozwala dodać nowy atrybut/zmienną do zbioru danych. Dalsza edycja możliwa ręcznie lub poprzez inne filtry.
- *AddExpression* — pozwala dodać nowy atrybut wypełniony wartościami stanowiącymi wynik pewnego działania arytmetycznego na istniejących atrybutach.
- *AddID* — dodaje atrybut z numerem porządkowym ID.
- *Center* — środkuje zbiór danych na wszystkich atrybutach numerycznych z wyjątkiem wyjściowego.
- *Discretize* z pakietu `unsupervised` — pozwala na dyskretyzację bez zwracania uwagi na korelację ze zmienną wyjściową (m.in. na równoliczne lub równoszerokie przedziały).
- *FirstOrder* — zamienia wskazane n atrybutów rzeczywistoliczbowych na $n-1$ atrybutów powstałych jako różnice wartości (przydatne np. przy szeregach czasowych). Istnieją także: *TimeSeriesDelta*, *TimeSeriesTranslate*.
- *InterquartileRange* — znakuje rekordy stanowiące punkty odstające (ang. *outliers*), zostawia wynik w postaci dodatkowych dwóch kolumn.
- *MathExpression* — modyfikuje wskazany atrybut wg podanego wyrażenia arytmetycznego.
- *MergeTwoValues* — łączy dwie wartości ze zbioru wyliczeniowego w jedną.
- Konwersje: *NumericToBinary*, *NumericToNominal*, *StringToNominal*, *StringToWordVector*, ...
- Ujednolicanie: *Normalize*, *Standarize*.

Klasyfikatory (zakładka *Classify*)

- Duży zestaw (ponad 100) algorytmów do klasyfikacji i aproksymacji.
- Możliwość sprawdzenia zdolności do *uogólniania* poprzez wydzielenie zbioru testowego lub *krzyżową walidację*.
- Możliwość wizualizacji wyników (mały zakres).
- Możliwość zapisywania wynikowych modeli.



Ważniejsze klasyfikatory

Klasyfikator bez reguł (ang. *Zero-Rule Classifier*)
Java: `weka.classifiers.rules.ZeroR`

- Dla nowoprzychodzących obiektów odpowiada zawsze numerem klasy, która była *najczęstsza* w zbiorze uczącym.
- Nie używa żadnych atrybutów wejściowych do wnioskowania.
- Dobry punkt wyjścia do porównań z innymi algorytmami. Wskazuje minimalne prawdopodobieństwo poprawnego sklasyfikowania (lub maksymalne błędnego), względem którego inne algorytmy powinny być lepsze.

Ważniejsze klasyfikatory

Naiwny klasyfikator Bayesa (ang. *Naive Bayes Classifier*)
Java: `weka.classifiers.bayes.NaiveBayes`

- Klasyfikator probabilistyczny z dodanym tzw. *założeniem naiwnym*.
- Uczenie polega na odnotowaniu rozkładów częstości warunkowych dla poszczególnych wartości atrybutów w klasach (pod warunkiem klasy). Częstości te utożsamiane są z prawdopodobieństwami $P(X_i = j|Y = k)$.
- Dla nowoprzychodzących obiektów jako odpowiedź wybierana jest ta klasa, dla której jest największe prawdopodobieństwo warunkowe (pod warunkiem wektora X).

$$y^* = P\Big(Y = y|X = (x_1, x_2, \ldots, x_n)\Big) \tag{1}$$

$$= \arg\max_y \frac{P(X = (x_1, x_2, \ldots, x_n)|Y = y)P(Y = y)}{\sum_z P(X = (x_1, x_2, \ldots, x_n)|Y = z)P(Y = z)}. \tag{2}$$

- Założenie naiwne mówi, że zmienne wejściowe są niezależne[a]. Pozwala to przedstawić prawdopodobieństwo $P(X = (x_1, x_2, \ldots, x_n)|Y = y)$ dla całego wektora $(x_1, x_2, \ldots, x_n)$ jako iloczyn prawdopodobieństw.
- Ostatecznie (mianownik stały): $y^* = \arg\max_y \prod_{i=1}^n P(X_i = x_i|Y = y)P(Y = y)$.

[a] Właściwie lekko słabsze założenie: zmienne wejściowe są niezależne warunkowo w klasach.

Ważniejsze klasyfikatory

Perceptron wielowarstwowy (ang. *Multi-Layer Perceptron*)
Java: `weka.classifiers.functions.MultiLayerPerceptron`

- Sieć neuronowa złożona z kilku warstw neuronów o sigmoidalnych funkcjach aktywacji.
- Algorytm uczenia *on-line*: wsteczna propagacja błędu ze współczynnikiem rozpędu (ang. *momentum*). Wzór na poprawkę pewnej wagi w w kroku o numerze t:

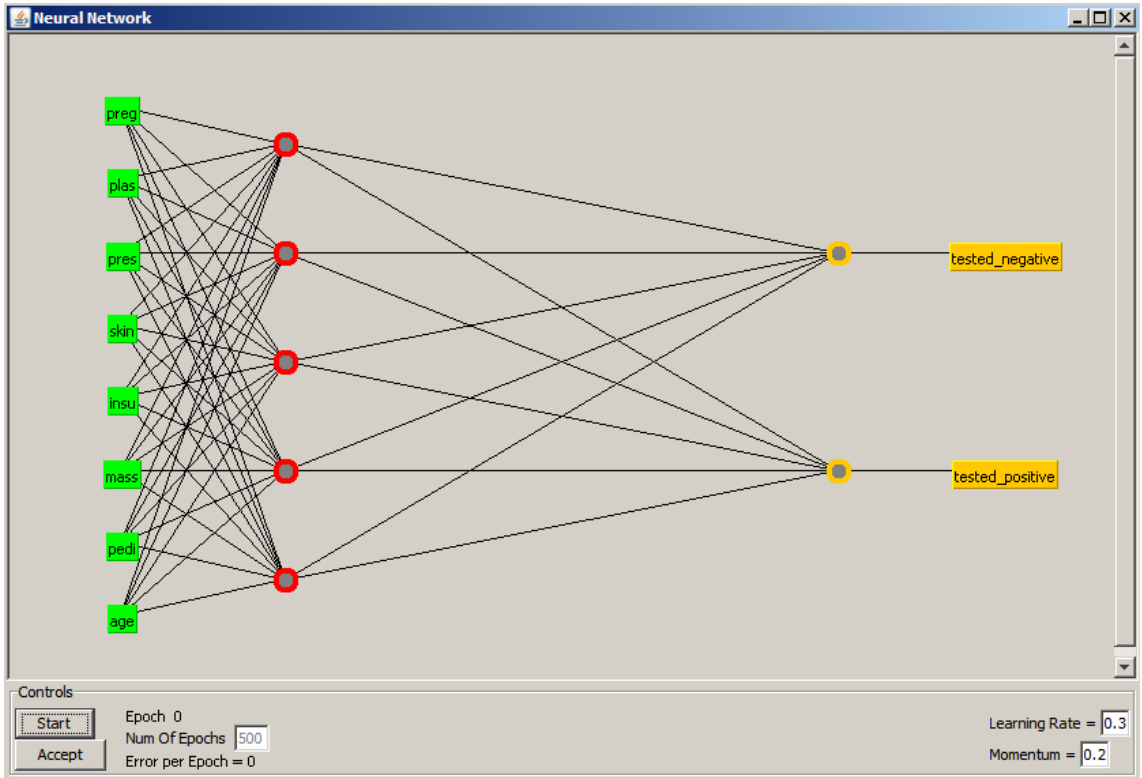
$$w(t+1) = w(t) - \eta \frac{\partial \frac{1}{2} e^2}{\partial w(t)} + \mu \left(w(t) - w(t-1) \right), \tag{3}$$

  gdzie: $\frac{1}{2}e^2$ błąd kwadratowy na aktualnie pokazanej próbce, η współczynnik uczenia, μ współczynnik rozpędu.
- Możliwość włączenia regularyzacji wygaszającej wagi (ang. *weight decay*) — bardzo często poprawia zdolność do uogólniania.

$$w(t+1) = w(t) - \eta \frac{\partial \frac{1}{2} e^2}{\partial w(t)} - \lambda w(t) + \mu \left(w(t) - w(t-1) \right), \tag{4}$$

  gdzie λ współczynnik regularyzujący (kara).
- Możliwość automatycznego i ręcznego budowania architektury sieci.

GUI do sieci neuronowych



Ważniejsze klasyfikatory

Sieć neuronowa RBF (ang. *Radial Basis Functions*)
Java: `weka.classifiers.functions.RBFNetwork`

- Sieć neuronowa złożona zwykle z dwóch warstw, pierwszej z neuronami o radialnej funkcji aktywacji, drugiej z jednym neuronem liniowym.

$$y_{\mathrm{RBF}}(\mathbf{x}) = w_0 + \sum_{k=1}^{K} w_k \phi(\mathbf{x}, \mathbf{c}_k, \sigma_k); \tag{5}$$

$$\phi(\mathbf{x}, \mathbf{c}, \sigma) = \exp\left(-\frac{\sum_{i=1}^{n} (x_i - c_i)^2}{2\sigma^2} \right). \tag{6}$$

  gdzie $\mathbf{c}_k$ określają współrzędne geometryczne środka k-tego neuronu, a σ_k wpływa na szerokość funkcji radialnej.
- Uczenie odbywa się zwykle *off-line* poprzez algorytm heurystyczny złożony z dwóch części: *klasteryzacji* — rozstawiającej odpowiednio środki i szerokości neuronów, *minimalizacji sumarycznego błędu kwadratowego* — sprowadza się do rozwiązania liniowego układu równań w celu znalezienia optymalnych wag w_k.
- Istnieje opcja regularyzacji (odpowiadająca równoważnie *weight decay* w MLP) pod nazwą *ridge*.

Ważniejsze klasyfikatory

Drzewa do klasyfikacji (ang. *Classification And Regression Trees*)
Java: `weka.classifiers.trees.SimpleCart`

- Algorytm składa się z dwóch części: budowania drzewa binarnego i przycinania (ang. *pruning*).
- Część budująca poczynając od korzenia rekurencyjne dodaje węzły. W każdym węźle na podstawie pewnego kryterium oceniającego rozkład prawdopodobieństwa klas (*indeks Gini'ego*, *entropia*) dobierany jest warunek decydujący o rozcięciu dziedziny wzdłuż jednej ze zmiennych na pewnej odciętej na dwie podkostki — w ten sposób powstają dwa węzły potomne.

$$\text{gini}(p) = \sum_i \sum_{j \neq i} p_i p_j = 1 - \sum_i {p_i}^2, \tag{7}$$

$$\text{entropy}(p) = -\sum_i p_i \log_2 p_i, \tag{8}$$

  gdzie $p_i = P(Y = i|\text{aktualny węzeł})$.
- Część przycinająca dba o zachowanie dobrych własności uogólniających klasyfikatora. Poprzez krzyżową walidację usuwane są nadmiarowe węzły z drzewa powodujące zbytnie dopasowanie do danych uczących.

Ważniejsze klasyfikatory

Przykład drzewa dla zbioru *iris*:

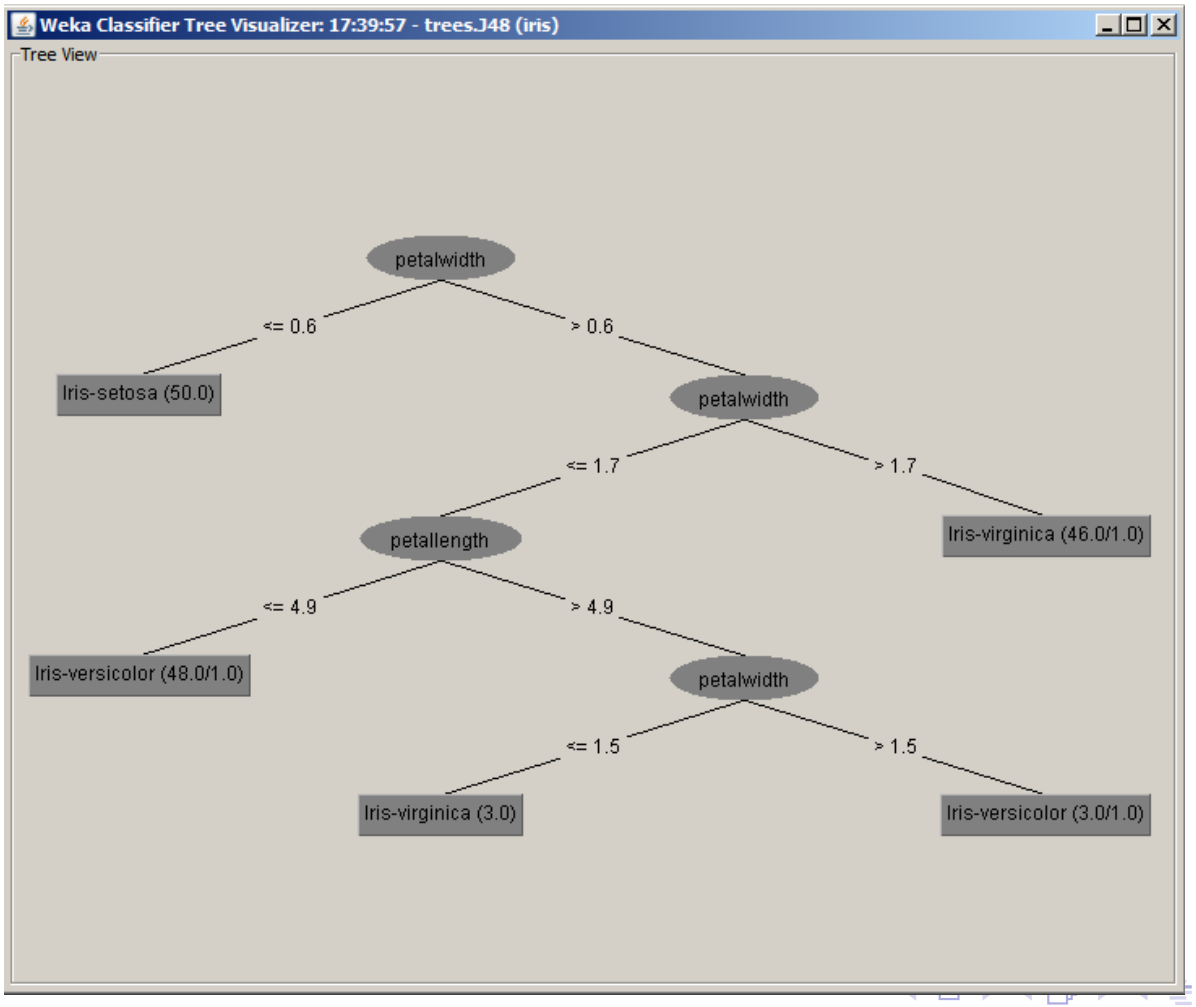
```
CART Decision Tree

petallength < 2.45: Iris-setosa(50.0/0.0)
petallength >= 2.45
|   petalwidth < 1.75
|   |   petallength < 4.95: Iris-versicolor(47.0/1.0)
|   |   petallength >= 4.95
|   |   |   petalwidth < 1.55: Iris-virginica(3.0/0.0)
|   |   |   petalwidth >= 1.55: Iris-versicolor(2.0/1.0)
|   petalwidth >= 1.75: Iris-virginica(45.0/1.0)

Number of Leaf Nodes: 5
Size of the Tree: 9
Correctly Classified Instances         143               95.3333 %
```

Drzewo z algorytmu J48 (*C4.5*)

Cechy C4.5 (Quinlan, 1993): kryterium *information gain*, pozwala na braki w danych, radzi sobie z ciągłymi i dyskretnymi zmiennymi.

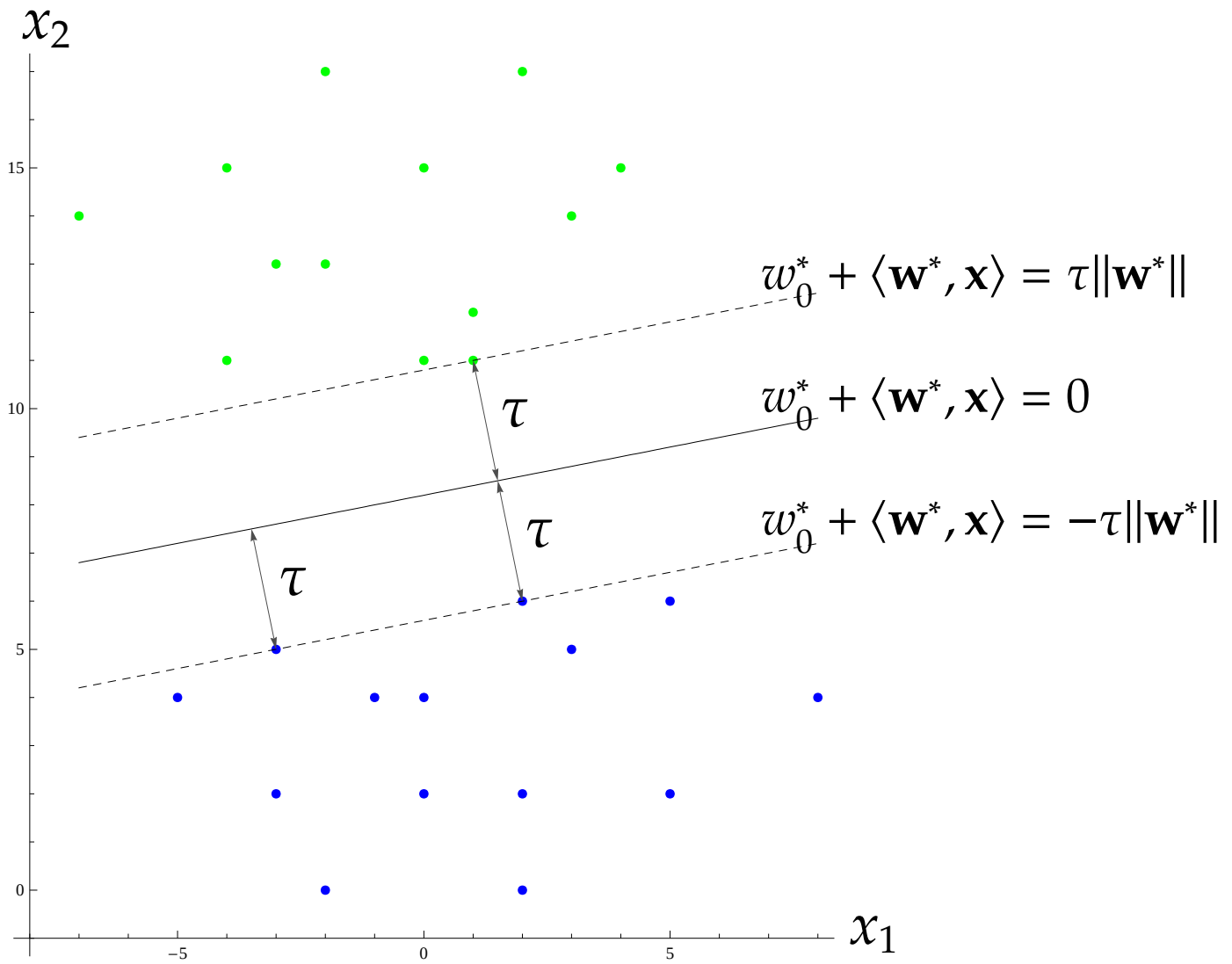


Ważniejsze klasyfikatory

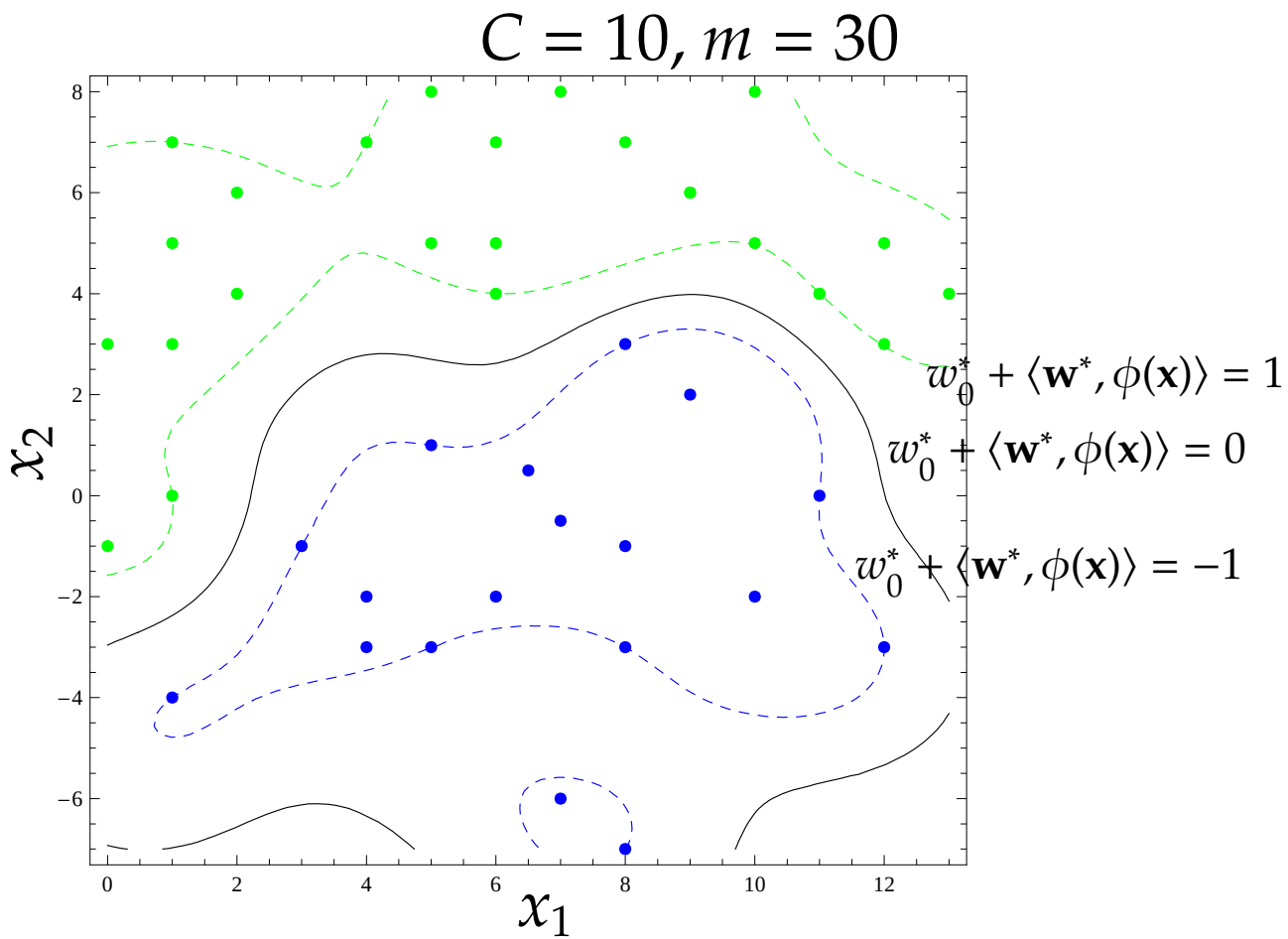
Klasyfikator SVM (ang. *Support Vector Machines*) strojony algorytmem SMO (ang. *Simple Minimal Optimization*)
Java: `weka.classifiers.functions.SMO`

- SVM (Vapnik) to klasyfikator poszukujący wielowymiarowej płaszczyzny oddzielającej klasy o **maksymalnym marginesie separacji**.
- Można wskazać pewne analogie do algorytmu perceptronu Rosenblatt'a.
- Polski odpowiednik nazwy — *maszyny punktów podparcia* należy kojarzyć z punktami, które definiują (podpierają) wynikową płaszczyznę. Są to punkty leżące w obszarze najtrudniejszym do sklasyfikowania.
- W przypadku kiedy dane nie są liniowo separowalne, klasyfikator optymalizuje kryterium, które stara się maksymalizować margines i jednocześnie minimalizować liczbę pomyłek.
- Możliwość **klasyfikacji nieliniowej** (o maksymalnym marginesie) poprzez sztuczkę z **przekształceniem jądrowym** (ang. *kernel trick*).
- Klasyfikacja z liczbą klas większą niż dwie, odbywa się poprzez zbudowanie kilku klasyfikatorów binarnych (1 kontra reszta).
- Rozwiązanie dla SVM wymaga **optymalizacji formy kwadratowej** z ograniczeniami. Metoda SMO realizuje to przez binaryzację zmiennych.

SVM liniowy dla zbioru w $\mathbb{R}^2$



SVM nieliniowy dla zbioru w $\mathbb{R}^2$ (przekształcenie jądrowe)



Ważniejsze klasyfikatory

Klasyfikator Głosujący Perceptron (ang. *Voted Perceptron*)
Java: `weka.classifiers.functions.VotedPerceptron`

- Głosujący perceptron (Freund i Schapire, 1998) to klasyfikator będący modyfikacją algorytmu perceptronu Rosenblatt'a.
- Algorytm przebiega sekwencyjnie po danych i stosuje podstawowy wzór na modyfikację wag (współczynników płaszczyzny), gdy dany punkt danych jest źle sklasyfikowany:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta \mathbf{x}_i y_i, \tag{9}$$

  gdzie $\eta \in (0, 1]$ to współczynnik uczenia.
- Algorytm zapamiętuje wszystkie pośrednie stany wektora wag, a także zlicza dla każdego z nich tzw. *liczbę przeżyć* tj. liczbę iteracji, w których kolejne punkty nie zmieniły stanu danego wektora.
- Ostateczny wektor wag — średnia ważona po zapamiętanych wektorach:

$$\mathbf{w}^* = \frac{\sum_j s_j \mathbf{w}_j}{\sum_j s_j}, \tag{10}$$

  gdzie s_j liczba przeżyć j-tego zapamiętanego wektora.

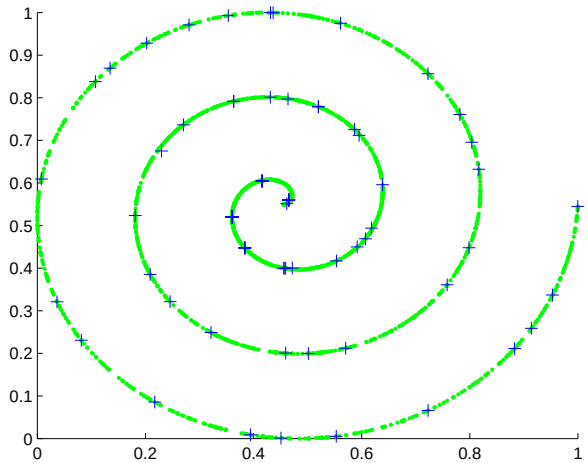
Aproksymacja w ramach zakładki *Classify*

- Przykłady problemów w WEKA: *cpu*, *cpu.with.vendor*.
- Przykłady problemów w UCI: *Parkinsons Telemonitoring*, *Wine Quality*, *Concrete Slump Test*, *Forest Fires*.
- Ważniejsze algorytmy do aproksymacji w WEKA:
  - *LinearRegression*,
  - *SimpleLinearRegression* (wybiera tylko jeden atrybut),
  - *SMOReg* (wersja SVM dla estymacji regresji),
  - *M5Rules* (pokrewny drzewkom CART, tyle że dla regresji; w każdym liściu budowany jest liniowy model — czyli cała powierzchnia kawałkami liniowa, możliwe nieciągłości),
  - *REPTree* (j.w. ale model kawałkami stały; y w liściach ustalany jako średnia z y_i dla danych w liściu).

Klasteryzacja — zakładka *Cluster*

Klasteryzacja (analiza skupień, kwantowanie wektorowe, grupowanie)

Zadanie uczenia bez nadzoru, polegające na zastąpieniu zbioru punktów (danych) w pewnej przestrzeni dużo mniejszym skończonym zbiorem tzw. *reprezentantów* lub *klastrów*, dla których średnia odległość do punktów danych jest możliwie najmniejsza i jednocześnie odległości międzyklastrowe są możliwie największe. Ważne jest dobre postawienie optymalizowanego kryterium.



Klasteryzacja

- Klasteryzacja bywa używana do stratnej kompresji.
- Ważniejsze algorytmy w WEKA: *Expectation Maximization* (`weka.clusterers.EM`), *K-means*[2] (`weka.clusterers.SimpleKMeans`).
- Wstępna normalizacja/standaryzacja zmiennych jest zalecana i wpływa na wynik.
- Możliwość podpinania do algorytmów innych metryk (odległości) niż Euklidesowa.
- Niektóre algorytmy w WEKA domyślnie wyszukują po jednym reprezentancie dla klasy (co stanowi pewne połączenie zadania bez nadzoru i z nadzorem).

[2] Tłumaczyć jako: *K*-średnie.

Reguły asocjacyjne — zakładka *Associate*

- Poszukiwanie w zbiorze transakcji zakupowych reguł postaci np.:

  jeżeli PIWO to CHIPSY i PIELUSZKI.

- Główny algorytm: *A priori* (Agrawal, 1994). Dwa parametry wejściowe: minimalne wsparcie i minimalne zaufanie. Dwie główne części algorytmu: znajdowanie *zbiorów częstych*, generowanie *reguł asocjacyjnych*.
- W przykładowych zbiorach WEKA — zbiór *supermarket*.

Spis treści

Experimenter — własności

- Pozwala zbiorczo uruchamiać wiele algorytmów na wielu zbiorach danych.
- Dwa podstawowe tryby: *simple* i *advanced* (z poziomu GUI o wyborze decyduje radio button).
- Dla eksperymentów możliwe dwie nastawy: *standard* (eksperyment na lokalnej maszynie) lub *remote* (zdalny z możliwością rozproszenia na kilka maszyn).
- Wyniki eksperymentów mogą lądować (*results destination*) w:
  - plikach .arff lub .csv — brak możliwości wznowienia długiego eksperymentu od miejsca błędu,
  - w bazie danych (JDBC).

Experimenter — przygotowanie eksperymentu

- Nastawy testowania: krzyżowa walidacja, podział danych na uczące/testowe z mieszaniem lub bez.
- Domyślnie zadanie klasyfikacji. Można wskazać regresję, o ile poszczególne zbiory danych pozwalają.
- Możliwość wczytywania całych katalogów (rekurencyjnie w głąb).
- *Iteration control* — powtórzenia, aby dostać statystycznie istotne wyniki. Np. 10 iteracji i domyślna 10-krotna krzyżowa walidacja oznacza 100 wykonań pewnego algorytmu. Przy wielu zbiorach i wielu algorytmach dodatkowe ustawienie: *data sets first* lub *algorithms first*.
- Budowanie listy algorytmów poprzez: *Add new* oraz filtrowanie typów algorytmów.

Experimenter — uruchomienie eksperymentu

- Zakładka *Run* — wyświetla log uruchomień.
- W razie błędów należy zwykle wrócić do zakładki *Setup* i zmienić pewne ustawienia lub np. usunąć nie pasujący do danych algorytm.
- W razie powodzenia log zakończy się wpisem:

  ```
  There were 0 errors
  ```

  a wyniki trafią do wskazanego wcześniej poprzez *results destination* wyjścia.

Experimenter — analiza wyników

- Zakładka *Analyse*.
- W panelu *Source* widnieje liczba rezultatów np. : `Got 200 results`.
- Należy wskazać źródło wyników: *File*, *Database*, *Experiment* (ostatnio przeprowadzony eksperyment).
- Porównania *Paired T-Tester* — **test *t*-studenta dla średnich**: czy średni wynik dwóch algorytmów jest istotnie różny czy też zaniedbywalnie różny?
- Możliwe różne pola porównań: procent poprawnych/błędnych, liczba poprawnych/błędnych, liczba instancji testowych, itp.
- **Notacja „v/ /*”** wskazuje odpowiednio wynik: istotnie lepszy/porównywalny/gorszy względem algorytmu bazowego tj. ustalonego jako pierwszy.
- Warto ustalić klasyfikator bezregułowy `ZeroR` jako pierwszy.
- **Formaty przedstawienia** wyników porównawczych (*Output format*): Plain-Text, CSV, GNU Plot, LaTeX, HTML, Significance only.

Spis treści

WEKA API — ogólnie

- Pozwala osadzać możliwości WEKA we własnych projektach/systemach napisanych w Javie.
- Możliwości:
  - ustawianie opcji (parametrów),
  - tworzenie zbiorów danych w pamięci RAM,
  - wczytywanie i zapisywanie danych (z różnych źródeł),
  - uruchamianie algorytmów,
  - wizualizacja,
  - serializacja.

Ustawianie opcji

- Poprzez gettery/settery danego algorytmu lub `setOptions(String[])` i `getOptions()`, o ile klasa implementuje interfejs `weka.core.OptionHandler`.
- Przykład:

```
import weka.filters.unsupervised.attribute.Remove;
...
String[] options = new String[2];
options[0] = "-R";
options[1] = "1";
Remove rm = new Remove();
rm.setOptions(options);
```

- Częsty błąd: wkładanie nazwy parametru i wartości po jeden indeks tablicy.
- Klasa pomocnicza: `weka.core.Utils`. Przykłady:

```
String valueString = Utils.getOption("R", options);
String[] options = Utils.splitOptions("-R 1");
```

Reprezentacja zbiorów danych

`weka.core.Instances`

Cała tabelka z danymi. Wiersze (obiekty) wyłuskuje się poprzez `instance(int)`, kolumny poprzez `attribute(int)`. Uwaga: numeracja od 0.

`weka.core.Instance`

Reprezentuje pojedynczy wiersz (obiekt) w danych. Poszczególne pola wyłuskuje się poprzez: `value(int)` lub `value(Attribute)`.

`weka.core.Attribute`

Reprezentuje kolumnę (atrybut) w danych. Trzyma informacje o typie: `type()` oraz `isNumeric()`, `isDate()`, itp. Jeżeli typ wyliczeniowy, to zestaw wartości w: `enumerateValues()`[a].

[a] Istnieje możliwość tworzenia atrybutów relacyjnych, wtedy na `weka.core.Attribute` składają się ponownie `weka.core.Instances`.

Wczytywanie danych z plików

- **Wczytywacz dedykowany** do formatu. Np.:

```
import weka.core.converters.CSVLoader;
import weka.core.Instances;
import java.io.File;
...
CSVLoader loader = new CSVLoader();
loader.setSource(new File("/some/where/some.data"));
Instances data = loader.getDataSet();
```

- **Wczytywacz ogólny** (patrzy na rozszerzenie pliku). Np.:

```
import weka.core.converters.ConverterUtils.DataSource;
import weka.core.Instances;
...
Instances data1 = DataSource.read("/some/where/dataset.arff");
Instances data2 = DataSource.read("/some/where/dataset.csv");
Instances data3 = DataSource.read("/some/where/dataset.xrff");
```

Zapisywanie danych do plików

- **Zapisywacz dedykowany** do formatu. Np.:

```
import weka.core.Instances;
import weka.core.converters.CSVSaver;
import java.io.File;
...
// data structure to save
Instances data = ...

CSVSaver saver = new CSVSaver();
saver.setInstances(data);
saver.setFile(new File("/some/where/data.csv"));
saver.writeBatch();
```

- **Zapisywacz ogólny** (patrzy na rozszerzenie pliku). Np.:

```
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSink;
...
Instances data = ... // data structure to save
DataSink.write("/some/where/data.arff", data);
DataSink.write("/some/where/data.csv", data);
```

Jawne wskazywanie atrybutu z klasą

Niektóre formaty, np. *.arff*, nie przechowują tej informacji. Po wczytaniu danych, należy jawnie wskazać atrybut z klasą. Nie wskazanie może skutkować: `weka.core.UnassignedClassException`.

Przykład:

```
// uses the first attribute as class attribute
if (data.classIndex() == -1)
data.setClassIndex(0);
...
// uses the last attribute as class attribute
if (data.classIndex() == -1)
data.setClassIndex(data.numAttributes() - 1);
```

Wczytywanie danych z bazy danych

- **weka.experiment.InstanceQuery** — pozwala wczytywać m.in. dane *rzadkie*, nie pozwala na odczyt „kawałkami”.

```
import weka.core.Instances;
import weka.experiment.InstanceQuery;
...
InstanceQuery query = new InstanceQuery();
query.setDatabaseURL("jdbc_url");
query.setUsername("the_user");
query.setPassword("the_password");
query.setQuery("select * from whatsoever");
// if your data is sparse, then you can say so, too:
// query.setSparseData(true);
Instances data = query.retrieveInstances();
```

- **weka.core.converters.DatabaseLoader** — pozwala na odczyt „kawałkami”. Przykład:

```
import weka.core.Instance;
import weka.core.Instances;
import weka.core.converters.DatabaseLoader;
...
DatabaseLoader loader = new DatabaseLoader();
loader.setSource("jdbc_url", "the_user", "the_password");
loader.setQuery("select * from whatsoever");
Instances structure = loader.getStructure();
Instances data = new Instances(structure);
Instance inst;
while ((inst = loader.getNextInstance(structure)) != null) data.add(inst);
```

Przykład wywołania filtra

```
import weka.core.Instances;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Remove;
...
String[] options = new String[2];
options[0] = "-R"; // "range"
options[1] = "1"; // first attribute
Remove remove = new Remove(); // new instance of filter
remove.setOptions(options); // set options
remove.setInputFormat(data); // inform filter about dataset
// **AFTER** setting options
Instances newData = Filter.useFilter(data, remove); // apply filter
```

Uwaga: metoda `setInputFormat(Instances)` pozwala filtrowi ustalić format danych wyjściowych. Wszystkie opcje muszą być ustawione do momentu wywołania jej. Opcje ustawione później są ignorowane.

Wywoływanie klasyfikatorów

Klasyfikatory uczone off-line

Przyjmują do nauki (i wymagają) cały zbiór uczący i zwracają końcowy stan klasyfikatora. Warunek: zbiór uczący musi zmieścić się w pamięci. Nazewnictwo WEKI: *batch-trainable*.

Klasyfikatory uczone on-line

Pozwalają na naukę przyrostową, próbka po próbce (obiekt po obiekcie). Po każdym takim kroku zmienia się stan klasyfikatora. Nazewnictwo WEKI: *incremental* lub *on-the-go*. Wszystkie tego typu algorytmy muszą implementować interfejs: `weka.classifiers.UpdateableClassifier`.

Przykład uczenia off-line

```
import weka.core.Instances;
import weka.classifiers.trees.J48;
...
Instances data = ... // from somewhere
String[] options = new String[1];
options[0] = "-U"; // unpruned tree
J48 tree = new J48(); // new instance of tree
tree.setOptions(options); // set the options
tree.buildClassifier(data); // build classifier
```

Przykład uczenia (i czytania) on-line

```
import weka.core.converters.ArffLoader;
import weka.classifiers.bayes.NaiveBayesUpdateable;
import java.io.File;
...
// load data
ArffLoader loader = new ArffLoader();
loader.setFile(new File("/some/where/data.arff"));
Instances structure = loader.getStructure();
structure.setClassIndex(structure.numAttributes() - 1);

// train NaiveBayes
NaiveBayesUpdateable nb = new NaiveBayesUpdateable();
nb.buildClassifier(structure);
Instance current;
while ((current = loader.getNextInstance(structure)) != null)
    nb.updateClassifier(current);
```

W momencie `nb.buildClassifier(structure)` odbywa się tylko niezbędna inicjalizacja potrzebnych do nauki wewnętrznych zmiennych klasyfikatora.

Testowanie — zdolność do *uogólniania*

- Klasą zarządzającą testowaniem jest `weka.classifiers.Evaluation`.
- Dwa modele testowania:
  - **model z krzyżową walidacją** (jeżeli liczba foldów równa liczbie obiektów to mamy walidację *leave-one-out*),
  - **model z wydzielonym zbiorem do testu**.
- W modelu z krzyżową walidacją obiekt `Evaluation` wywołuje wewnętrznie `buildClassifier(Instances)`. Implementacja tej metody powinna realizować na początku „wyzerowanie" klasyfikatora.
- W modelu ze zbiorem testowym należy wywołać `buildClassifier(Instances)` samodzielnie, przed przekazaniem klasyfikatora do obiektu `Evaluation`.
- W modelu z krzyżową walidacją ważną rolę pełni `java.util.Random` — *seed*, powtarzalność.

Krzyżowa walidacja — przykład

```
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
import weka.core.Instances;
import java.util.Random;
...
Instances data = ... // from somewhere
Evaluation eval = new Evaluation(data);
J48 tree = new J48();
eval.crossValidateModel(tree, data, 10, new Random(1));
System.out.println(
  eval.toSummaryString("\nSummary:\n\n", false));
System.out.println(
  eval.toMatrixString("\nConfusion matrix:\n\n", false));
```

Przykładowy wynik `eval.toSummaryString()`:

```
Results:

Correctly Classified Instances         144               96      %
Incorrectly Classified Instances         6                4      %
Kappa statistic                          0.94
Mean absolute error                      0.035
Root mean squared error                  0.1586
Relative absolute error                  7.8705 %
Root relative squared error             33.6353 %
Total Number of Instances              150
```

Przykładowy wynik `eval.toConfusionMatrix()`:

```
=== Confusion Matrix ===

  a  b  c   <-- classified as
 50  0  0 |  a = Iris-setosa
  0 49  1 |  b = Iris-versicolor
  0  2 48 |  c = Iris-virginica
```

Wydzielony zbiór testowy — przykład

```
import weka.core.Instances;
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
...
Instances train = ... // from somewhere
Instances test = ... // from somewhere
// train classifier
Classifier tree = new J48();
tree.buildClassifier(train);

// evaluate classifier and print some statistics
Evaluation eval = new Evaluation(train);
eval.evaluateModel(tree, test);
System.out.println(
  eval.toSummaryString("\nSummary:\n\n", false));
System.out.println(
  eval.toMatrixString("\nConfusion matrix:\n\n", false));
```

Serializacja — przykład

```
import weka.classifiers.Classifier;
import weka.classifiers.trees.J48;
import weka.core.converters.ConverterUtils.DataSource;
import weka.core.SerializationHelper;
...
// load data
Instances data = DataSource.read("/some/where/data.arff");
data.setClassIndex(inst.numAttributes() - 1);

// train and test J48 tree via cross-validation
Evaluation eval = new Evaluation(data);
J48 tree = new J48();
eval.crossValidateModel(tree, data, 10, new Random(1));
System.out.println(
  eval.toSummaryString("\nSummary:\n\n", false));

// display tree as text
System.out.println(tree.toString());

// !serialize model
SerializationHelper.write("c:/j48.model", tree);
```

Deserializacja — przykład

```
import weka.classifiers.Classifier;
import weka.classifiers.trees.J48;
import weka.core.converters.ConverterUtils.DataSource;
import weka.core.SerializationHelper;
...
// !deserialize model from binary file
J48 tree = (J48) SerializationHelper.read("c:/j48.model");

// display model as text (check-back)
System.out.println(tree.toString());
```

Spis treści

Prosta implementacja — szkic postępowania

1. Stworzyć klasę osadzoną w pakiecie `weka.classifiers`.
2. Odziedziczyć po klasie `weka.classifiers.Classifier`.
3. Dodać identyfikator wersji `public static final long serialVersionUID = ....`
4. Zaimplementować metody obsługowo-opisowe:

```
public String globalInfo();
public Enumeration listOptions();
public void setOptions(String[] options) throws Exception;
public String[] getOptions();
public Capabilities getCapabilities();
public String toString();
```

5. Zaimplementować metody główne:

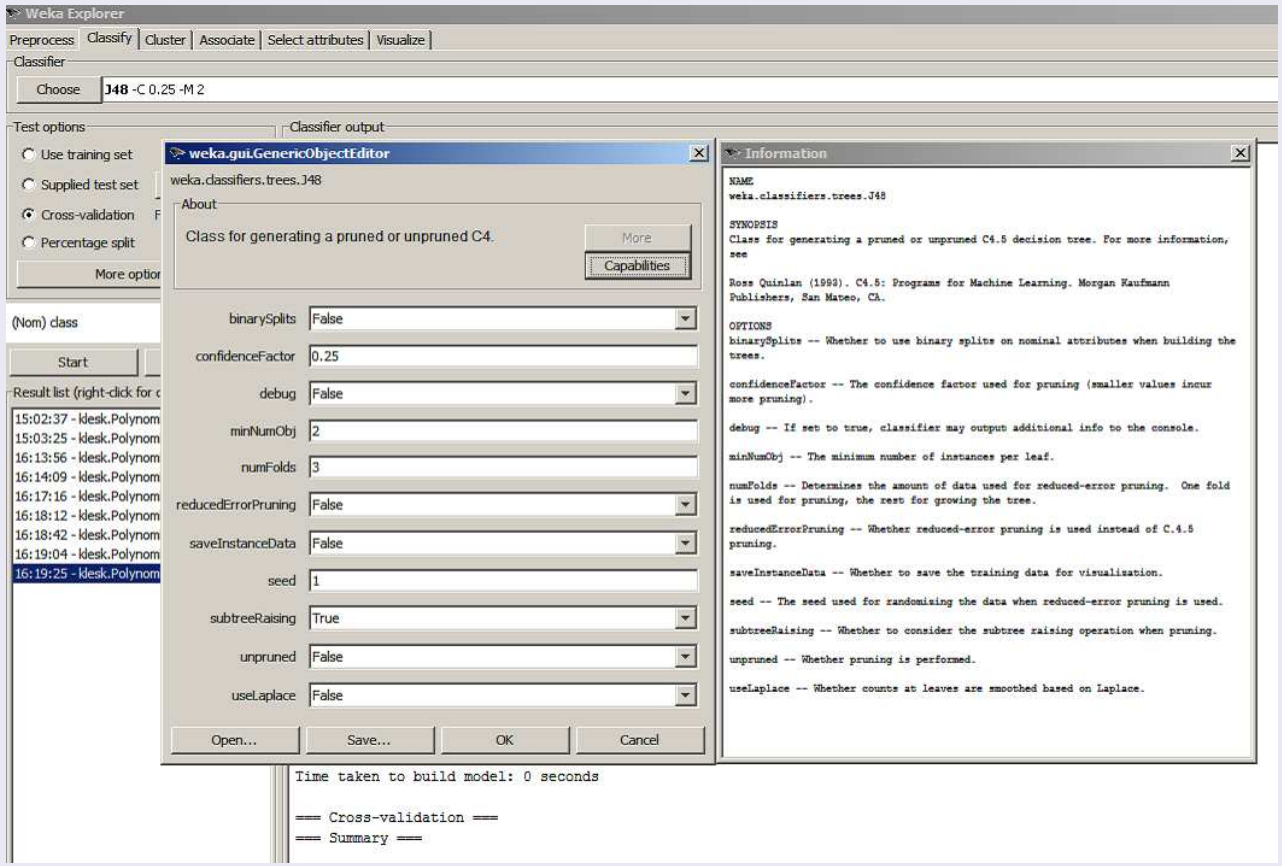
```
public void buildClassifier(Instances data) throws Exception;
public double classifyInstance(Instance instance) throws Exception;
```

6. Skompilować klasę i zamknąć w słoik, np. `kowalski.jar`.
7. Dodać wpis w pliku `weka.jar/weka/gui/GenericPropertiesCreator.props` w sekcji `weka.classifiers.Classifier=`
8. Uruchomić WEKA dodając do `CLASSPATH` potrzebne biblioteki. Np.: `java -classpath "weka.jar;kowalski.jar" weka.gui.GUIChooser`

Metody obsługowo-opisowe

`public String globalInfo()`

Ma zwracać ogólny opis metody wyświetlany w GUI po wyborze algorytmu pod przyciskiem *More*. W tym: opis parametrów, odwołania do literatury.



Metody obsługowo-opisowe

`Enumeration listOptions()`

Ma zwracać wyliczenie `Enumeration<Option>`. Wynik będzie użyty do wyświetlenia pomocy w linii komend. Ewentualne opcje nadklasy (nadklasyfikatora) powinny być również przekazane — `super.listOptions()`.

`void setOptions(String[] options) throws Exception`

Ma ustawiać w klasyfikatorze opcje (parametry) pochodzące z linii komend. Nazwa parametru i wartość powinny zawsze przychodzić w dwóch różnych pozycjach w tablicy. Wkazówka: użycie `Utils.getOption()`, np.:

```
tmpStr = Utils.getOption("lambda", options); //lambda coefficient
if (tmpStr.length() == 0) setLambda(0.0); //default lambda
else  setLambda(Integer.parseInt(tmpStr));
```

Metody obsługowo-opisowe

String[] getOptions()

Metoda wywoływana z poziomu GUI, gdy ustawienia są kopiowane do schowka. Nazwy parametrów powinny być dodane wraz z myślnikami. Np.:

```
List<String> result = new ArrayList<String>();
result.add("-lambda");
result.add("" + getLambda());
result.addAll(Arrays.asList(super.getOptions())); // superclass
return result.toArray(new String[result.size()]);
```

Metody obsługowo-opisowe

Capabilities getCapabilities()

Ustawia umiejętności klasyfikatorowi. Ta metoda będzie odpytywana z poziomu GUI m.in. przy badaniu, które algorytmy mogą być zastosowane do wczytanego zbioru danych. Np:

```
Capabilities result = super.getCapabilities();
//attributes capabilities
result.enable(Capability.NOMINAL_ATTRIBUTES);
result.enable(Capability.NUMERIC_ATTRIBUTES);
result.enable(Capability.DATE_ATTRIBUTES);
result.enable(Capability.MISSING_VALUES);
//class capabilities
result.enable(Capability.NOMINAL_CLASS);
result.enable(Capability.MISSING_CLASS_VALUES);
```

String toString()

Ma zwracać napis, który zostanie wyświetlony w GUI w głównym panelu po wykonaniu się algorytmu. Zwyczajowo przedstawia zbudowany model.

Spis treści

Regresja z regularyzacją L_2 (*ridge regression*)

Zminimalizować:

$$\sum_{i=1}^{I}\Bigg(\underbrace{w_0 + \sum_{k=1}^{K} w_k g_k(\mathbf{x}_i)}_{f(\mathbf{x}_i)} - y_i\Bigg)^2, \tag{11}$$

(gdzie I rozmiar zbioru danych, K liczba baz) przy ograniczeniach

$$\sum_{k=1}^{K} {w_k}^2 \leqslant \gamma. \tag{12}$$

Co jest równoważne[3] minimalizacji:

$$\sum_{i=1}^{I}\Bigg(w_0 + \sum_{k=1}^{K} w_k g_k(\mathbf{x}_i) - y_i\Bigg)^2 + \lambda \sum_{k=1}^{m} {w_k}^2, \tag{13}$$

dla pewnego λ.

[3] Idąc przez mnożniki Lagrange'a.

Regresja wielomianowa

Jeżeli przyjąć jako bazy g_k funkcje wielomianowe stopnia co najwyżej m, to w ogólności dla problemu $\mathbb{R}^n \rightarrow \mathbb{R}$ używamy funkcji (notacja z wieloindeksem):

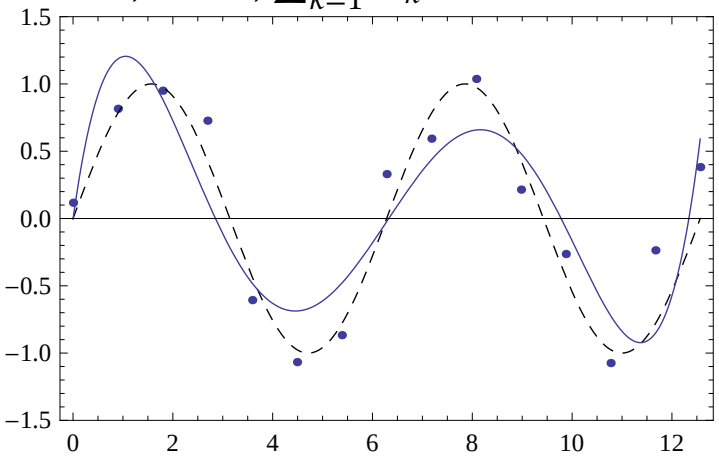
$$f(\underbrace{x_1, x_2, \ldots, x_n}_{\mathbf{x}}) = \sum_{j=0}^{m} \sum_{\substack{0 \leqslant q_1, q_2, \ldots, q_n \leqslant j \\ q_1 + q_2 + \cdots + q_n = j}} w_{q_1, q_2, \ldots, q_n} \cdot {x_1}^{q_1} {x_2}^{q_2} \cdots {x_n}^{q_n}. \tag{14}$$

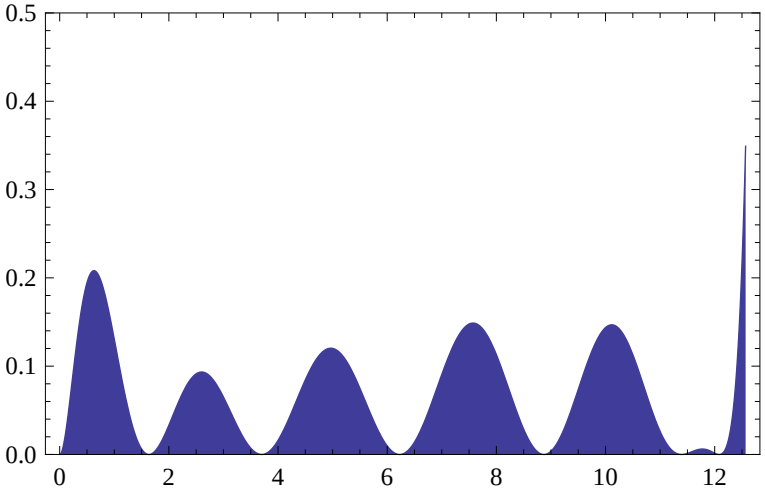
Liczba wszystkich termów wielomianu (baz) i tym samym liczba potrzebnych do nastrojenia współczynników $w_{q_1, q_2, \ldots, q_n}$ wynosi

$$\binom{n+m}{m}. \tag{15}$$

Regresja wielomianowa z regularyzacją L_2

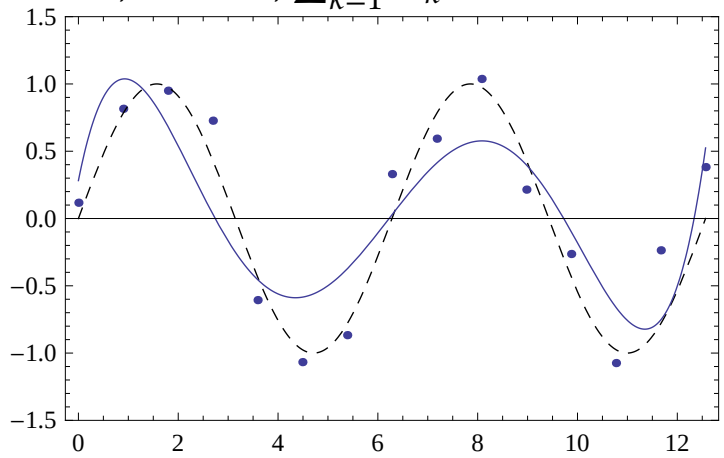
$m = 5, \lambda = 0, \sum_{k=1}^{m} w_k{}^2 \approx 10.74$

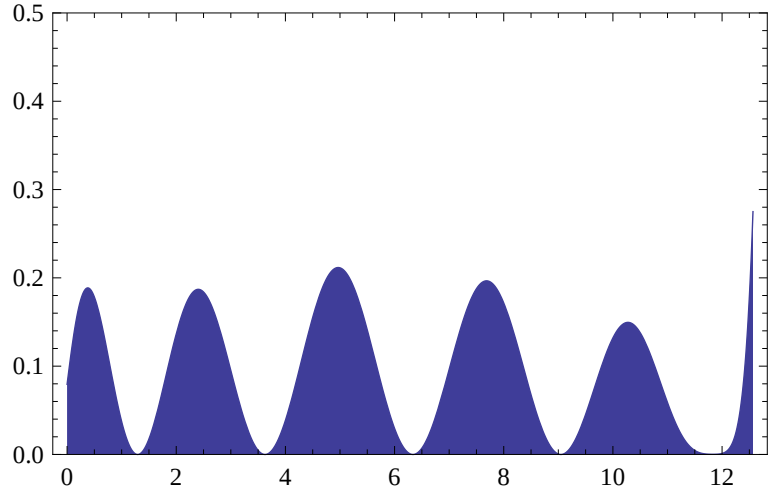




$$\int_0^{4\pi} \left(f_{5,0}(x) - \sin(x)\right)^2 dx \approx 0.84$$

$m = 5, \lambda = 0.1, \sum_{k=1}^{m} w_k{}^2 \approx 5.64$

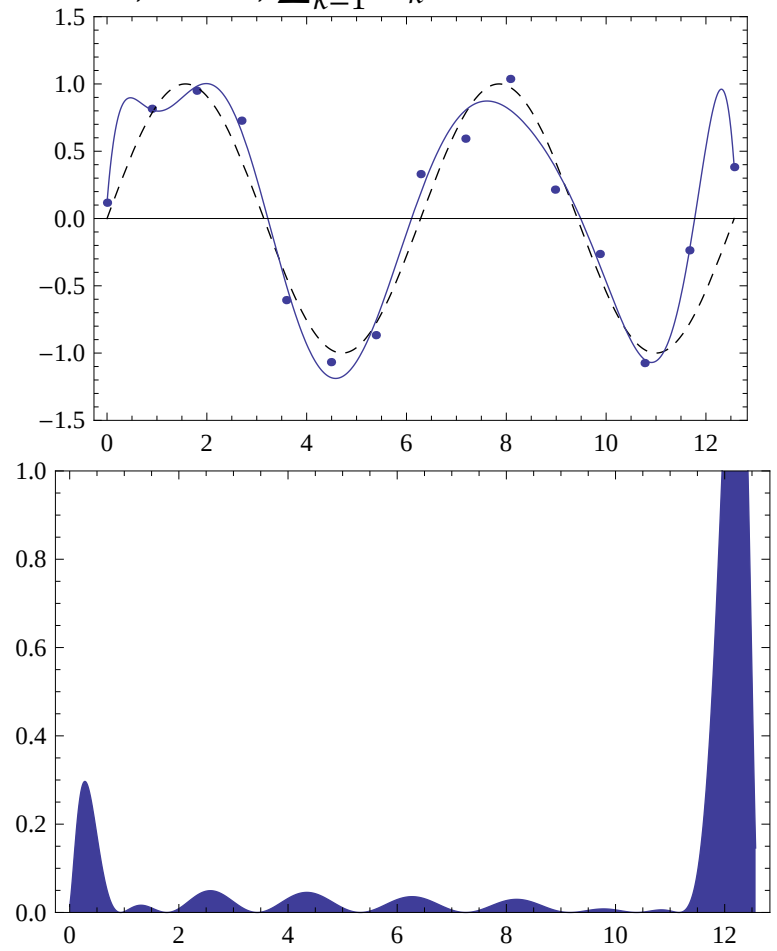




$$\int_0^{4\pi} \left(f_{5,0.1}(x) - \sin(x)\right)^2 dx \approx 1.14$$

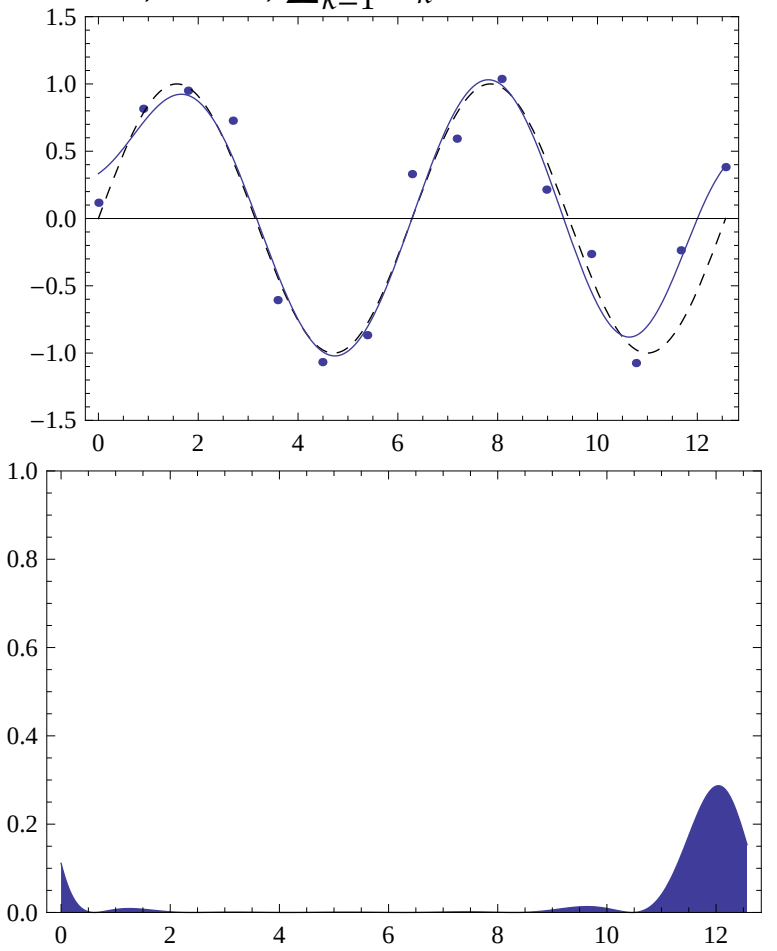
Regresja wielomianowa z regularyzacją L_2

$m = 10, \lambda = 0, \sum_{k=1}^{m} w_k{}^2 dx \approx 257.16$



$$\int_0^{4\pi} \left(f_{10,0}(x) - \sin(x)\right)^2 dx \approx 1.23$$

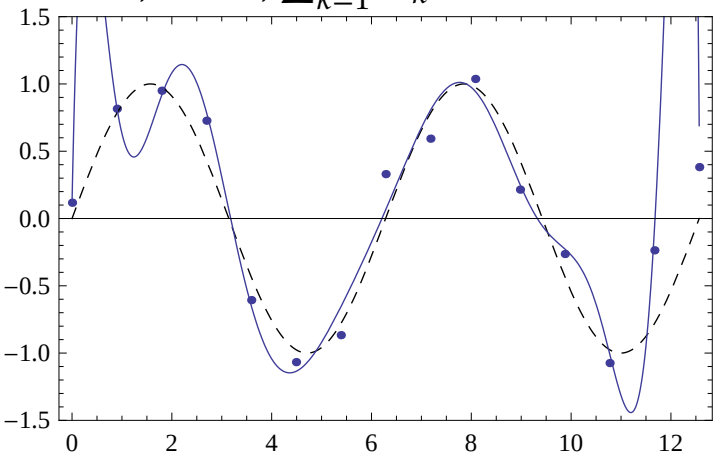
$m = 10, \lambda = 1, \sum_{k=1}^{m} w_k{}^2 dx \approx 0.17$

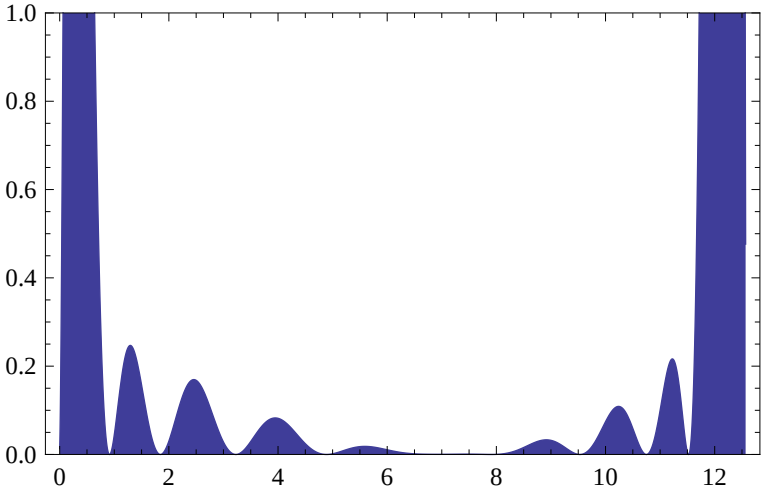


$$\int_0^{4\pi} \left(f_{10,1}(x) - \sin(x)\right)^2 dx \approx 0.36$$

Regresja wielomianowa z regularyzacją L_2

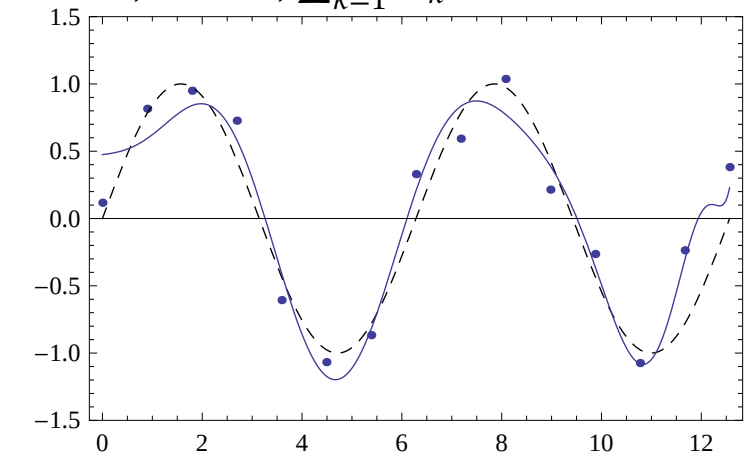
$m = 14, \lambda = 0, \sum_{k=1}^{m} w_k{}^2 \approx 6413$

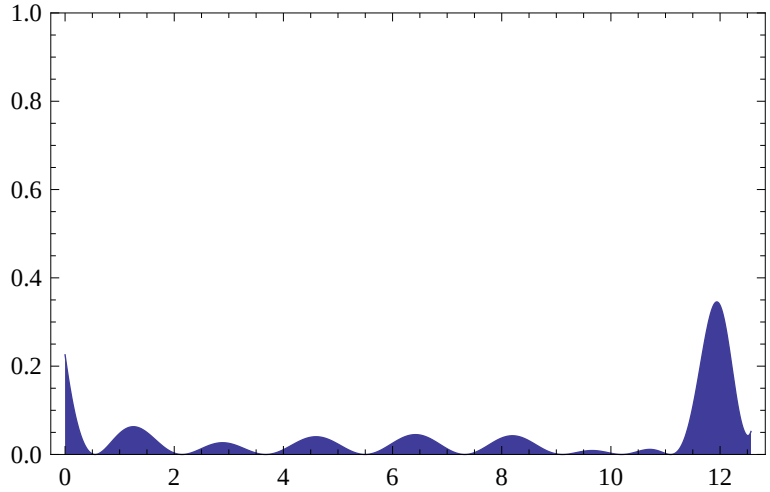




$$\int_0^{4\pi} \left(f_{14,0}(x) - \sin(x)\right)^2 dx \approx 15.35$$

$m = 14, \lambda = 10, \sum_{k=1}^{m} w_k{}^2 \approx 0.0096$





$$\int_0^{4\pi} \left(f_{14,1}(x) - \sin(x)\right)^2 dx \approx 0.47$$

Zadanie dla studentów

1. Zaimplementować *Polynomial Ridge Regression* jako rozszerzenie klasy `weka.classifiers.Classifier`. W szczególności:
   - Ustawić odpowiednie *capabilities*.
   - Nadać możliwość podawania dwóch parametrów (*options*): stopień wielomianu m i wartość współczynnika *ridge* λ.
   - Przeciążyć metodę `toString()`, tak aby wypisywała aproksymującą funkcję jako sumę składników w postaci: waga · baza, gdzie baza będzie zapisana jako iloczyn czynników wraz z potęgami. Dodatkowo umieścić w napisie informację o sumie kwadratów wag.
2. Podpiąć klasę, tak aby była widziana z poziomu WEKA Explorer w drzewku algorytmów.
3. Przetestować działanie algorytmu na zbiorach: `sin` oraz `cpu`.
4. Dla stopnia $m = 1$ porównać wartości otrzymywanych współczynników w_k z istniejącym algorytmem `LinearRegression`.
5. Dla zbioru `sin` sporządzić wykresy otrzymanych funkcji (korzystając z dowolnego zewnętrznego programu).

Dokumentacja

- Dokumentacja tekstowa: `/doc/WekaManual.pdf`.
- Dokumentacji Java API: `/doc/index.html`.
- Dokumentacja implementacji klasyfikatorów: *http://weka.wikispaces.com/file/view/Build_classifier_353.pdf*.
- Zbiory danych (w tym do regresji): *http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html*.