A Functional Programming Approach to Distance-based Machine Learning

Darko Aleksovski, Martin Erwig, Sašo Džeroski
Outline

• General Framework for Data Mining
• Functional Programming in Haskell
• The anatomy of distances for structured data
• A small database of distance function components
• Conclusions
General Framework for Data Mining

- Different types of data
- Different data mining tasks
- Different types of patterns/models
- Generic learning algorithms - algorithms that work on arbitrary structured data types
- Components of distance-based and kernel-based algorithms
Structured Data

• Complex data types built from simpler ones
• Primitive types include:
  – Real, Discrete(S), Boolean
  – Example: Discrete({s,d,t,q}) - a discrete data type, that can take one of four possible values
• Type constructors - Set(), Tuple(), Sequence()
  – Set(T) - set of elements of type T
  – Example: Set( Discrete({s,d,t,q}))
Generic Distance-Based Machine Learning Algorithms

- Examples of distance-based algorithms:
  - Hierarchical agglomerative clustering (HAC)
  - (k)nearest neighbor algorithm (kNN) for prediction
- Work for arbitrary types of (structured) data
- They require a distance function on the underlying data type

- A **distance function** is a function $d : T \times T \rightarrow \mathbb{R}^{0+}$ that satisfies
  1) $d(x,y) \geq 0$
  2) $d(x,y) = 0$ iff $x = y$
  3) $d(x,y) = d(y,x)$
- A **metric** is a distance function which also satisfies
  4) $d(x,z) \leq d(x,y) + d(y,z)$
Functional programming in Haskell

• Need for storing, retrieving and manipulating distance functions
• Use of functional programming
• Use of the language Haskell
  – Produces succinct and understandable code (or at least the concepts)
  – Expressiveness - spend more time thinking and reasoning about the application domain
  – Usage of functions and function compositions
  – Higher-order functions
Functional programming in Haskell

- Use of higher-order functions
  - Assembling a distance function for a complex data type - from distance functions on simpler types and additional helper functions

- Other properties of Haskell
  - Usage of pure functions
  - Lazy evaluation strategy
    - Can aid in some aspects of programming
    - Can create infinite data structures (example: sequence of infinite length)
Strong Typing

- No implicit type conversions
- Static typing
- When strong typing is combined with static typing, many programming errors can be caught before the program is run
- Haskell's type system - a polymorphic type system
- Polymorphic function - can be applied to values of different data types
- Properties helpful for us:
  - Very helpful automatic inference of types - also helps in avoiding mistakes in code
  - Much use of Haskell's fine grained set of types
The anatomy of distances for structured data

- Distances on primitive data types (Boolean, Discrete(S), Real)

  - Distance function used on Boolean and Discrete(S) types
    \[
    \rho(x, y) = \begin{cases} 
    1 & \text{if } x \neq y \\
    0 & \text{if } x = y
    \end{cases}
    \]

  - Distance function used on Real types
    \[
    \text{abs}(x, y) = |x - y|
    \]
Distances on complex/structured types

- Distances on complex/structured types
  - The structure of the complex types is revised
  - Recursive application of type constructors
    (Set(), Tuple() and Sequence())
  - Tree structure of these types
    - Internal node - a type constructor
    - Leaf node - a primitive data type
Distances on complex/structured types

- Finally, the distance calculation on complex/structured types requires three types of functions:
  
  (a) a function to generate pairs of objects of the simpler constitutive types – *pairing function*
  
  (b) distance functions on (objects of) the simpler types (or even primitive types)
  
  (c) an *aggregation function* that is applied to the distance values obtained by applying (b) to the pairs produced by (a)
Pairing functions

- A function of the form:
  \[ p : [T] \to [T] \to [(T, T)] \]
determines the important pairs of elements of the complex object

- Pairing functions for the Tuple() type constructor ("default" tuple-pairing function)
  \[ \text{pairs}((a_1, a_2, ..., a_n), (b_1, b_2, ..., b_n)) = [(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)] \]

- Pairing functions used for Set():
  - all-to-all
  - minimum-distance
  - surjection-pairing
  - linking
  - matching
Aggregation functions

• A function of the form
  \[ \text{agg} :: \text{[Float]} \rightarrow \text{Float} \]
  aggregates the distances between the pairs of simpler objects

• Examples of functions that can be used are:
  – Euclidian aggregation
  – Plain sum aggregation
  – Minimum aggregation
  – Maximum aggregation
  – Median aggregation

• Special cases of ordered weighted aggregation functions (OWA)
A small database of distance function components

- Distance function components stored in a Database of Data Types and Distances (DDTD)
  - Primitive data types and distances on them
  - Definition of complex data types
  - Definition of distances on these complex data types
- Generic distance-based machine learning algorithm
  - Implemented a generic version of the k-Nearest Neighbors algorithm, which uses the distance functions defined in DTD
DDTD usage scenario

**Set**
- aggregation function: maximum
- pairing function: minimum-distance

**Tuple**
- aggregation function: square-root of the sum of squares

**Boolean**
- distance: delta

**Real**
- distance: absolute

**Set**
- aggregation function: maximum
- pairing function: minimum-distance

**Tuple**
- aggregation function: square-root of the sum of squares

**Boolean**
- distance: delta

**Real**
- distance: absolute
Custom creating distance functions

Actions required to define the distance for the scenario:

- **Set** type constructor:
  - The aggregation function **maximum** is selected
  - The pairing function **minimum-distance** is selected

- **Tuple** type constructor
  - The aggregation function **square-root of the sum of squares** (for Euclidian distance) is selected
  - The default pairing function for tuples is selected

- **Bool** primitive type
  - The distance **delta** is selected

- **Real** primitive type
  - The distance **absolute** is selected
Results

- Results obtained applying the kNN algorithm to the Diterpene dataset
- Several distances were constructed and the leave-one-out method was applied

<table>
<thead>
<tr>
<th>Distance definition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set (aggregation:minimum; pairing:all-to-all) ; Pair (aggregation:sum) ; Discrete (distance:delta); Real (dist:absolute)</td>
<td>30.14% or 453 correctly classified data objects, out of 1503 total</td>
</tr>
<tr>
<td>Set (aggregation:sum; pairing: minimum-distance-MD); Pair( aggregation: weighted sum - weights: [50.0,1.0]) Discrete (distance:delta); Real (dist:absolute)</td>
<td>93.28% or 1402 correctly classified data objects, out of 1503 total</td>
</tr>
<tr>
<td>Set (aggregation:center-element; pairing: minimum-distance-MD); Pair( aggregation: weighted sum - weights: [50.0,1.0]) Discrete (distance:delta); Real (dist:absolute)</td>
<td>94.07% or 1414 correctly classified data objects, out of 1503 total</td>
</tr>
</tbody>
</table>
Conclusions

• Our approach could be extended to arrive at domain-specific languages for data mining

• It would greatly facilitate the development of domain-specific data mining approaches and their practical applications
The application
DiterpeneType
PairType

CenterElement-aggregation-with-Char-as-ordered-type
CenterElement-aggregation-with-Weighted-Sum-On-the-Tuple
MinimumAggregation-with-SumAggregation-on-the-Tuple
SumAggregation-with-Weighted-Sum-on-Tuple-elements

ROOT
- Set
  - Tuple
    - Char
    - Double

ROOT
- Distance-on-1-type: (agg:CenterElement; pairing: Md)
  - Distance-on-2-types: (agg:SumAgg; pairing: TuplePairing)
    - Simple-distance: (distance function: Absolute)
    - Simple-distance: (distance function: Absolute)
Questions ?