A Functional Programming Approach to Distance-based Machine Learning

Darko Aleksovski[1], *Martin Erwig*[2], *Sašo Džeroski*[1]
[1]Department of Knowledge Technologies, Jožef Stefan Institute, Ljubljana, Slovenia
[2]School of EE and CS, Oregon State University, Corvallis, Oregon, USA
darko.aleksovski@gmail.com, Saso.Dzeroski@ijs.si

ABSTRACT

Distance-based algorithms for both clustering and prediction are popular within the machine learning community. These algorithms typically deal with attribute-value (single-table) data. The distance functions used are typically hard-coded.

We are concerned here with generic distance-based learning algorithms that work on arbitrary types of structured data. In our approach, distance functions are not hard-coded, but are rather first-class citizens that can be stored, retrieved and manipulated. In particular, we can assemble, on-the-fly, distance functions for complex structured data types from pre-existing components.

To implement the proposed approach, we use the strongly typed functional language Haskell. Haskell allows us to explicitly manipulate distance functions. We have produced a SW library/application with structured data types and distance functions and used it to evaluate the potential of Haskell as a basis for future work in the field of distance-based machine learning.

1. General Framework for Data Mining

A general framework for data mining should elegantly handle different types of data, different data mining tasks, and different types of patterns/models. Dzeroski (2007) proposes such a framework, which explicitly considers different types of structured data and so-called generic learning algorithms that work on arbitrary types of structured data. The basic components of different types of such algorithms (such as distance or kernel-based ones) are discussed. Taking the inductive database (Imielinski and Mannila 1996) philosophy that proposes that patterns/models are first-class citizens that can be stored and manipulated, Dzeroski proposes to store and manipulate basic components of data mining algorithms, such as distance functions.

Structured data

Complex data types are built from simpler types by using type constructors. To be more precise, we start with primitive data types, such as Boolean, Discrete(S) and Real. These serve as the basic building blocks for structured data types, composed by using type constructors. A minimal set of type constructors might be Set() , Tuple() and Sequence(): These take as arguments a data type: Set(T) is the type of sets of elements of type T.

Generic distance-based machine learning algorithms

Distance-based algorithms are popular within the machine learning community. They can be used for both clustering and prediction. Examples of such algorithms are hierarchical agglomerative clustering (HAC) and the (k)nearest neighbor algorithm (kNN) for prediction.

The above mentioned algorithms (HAC and kNN) are generic in the sense that they can work for arbitrary types of data, be it attribute-value (tuples of primitive data types) or structured data. We only need a distance function to be provided on the underlying data type. The distance function and the underlying data type are then parameters to the generic algorithm.

A distance function on type T is a function from pairs of objects of the type T to the set of non-negative reals d :: T x T -> R^{0+}. The three important properties this function has to satisfy are:

1) $d(x,y) \geq 0$
2) $d(x,y)=0$ iff $x=y$
3) $d(x,y)=d(y,x)$

A distance functions that besides these three satisfies the triangle property

4) $d(x,z) \leq d(x,y)+d(y,z)$

is called a metric.

In this work, we propose to use generic distance-based learning algorithms (GDBLA). These would be used in conjunction with a number of data types and corresponding distance functions from the domain of use, which can be passed as parameters to the GDBLAs. We propose to explicitly store and manipulate data types and distance functions for these. In particular, we propose to assemble distance functions for complex structured data types from pre-existing components.

2. Functional programming in Haskell

Since we are interested in storing, retrieving and manipulating distance functions, we consider the use of functional programming, i.e., Haskell (Thompson 1999).

Basics

There are a many features of functional programming and especially the language chosen (Haskell) that help users create succinct and easily understandable code. The code (as stated by people without extensive programming experience), is easily understandable, or at least the concepts, since they resemble the mathematical ones, are easy to grasp. Another desirable property of Haskell is its expressiveness, which allows the programmer/user to spend more time on thinking and reasoning about the application domain in question, rather than trying to conform to the language's style of programming.

The key feature of functional programming languages, including Haskell, is the way of using functions and function compositions. Functions are first-class citizens and as such can be manipulated, passed as parameters, used as return values. Such functions are called higher-order functions.

In the context of our work, higher-order functions are clearly needed. We want to assemble a distance function for a complex data type (output), using distance functions on component simpler types (input). Here, functions are clearly present both as input and as output, and a higher-order function is needed to perform the assembly.

Haskell uses pure functions and nothing else. This means that a Haskell function resembles a mathematical function in the way that for every execution the same result is returned, that is no side-effects are allowed. The interpreters can (because of this lack of side-effects) more efficiently reorder executions. Moreover, some functional languages, such as Haskell, have adopted a lazy evaluation strategy, which supports infinite data structures and which can avoid unnecessary evaluations. This is desirable, since the user can define, for example, a sequence of infinite length and not worry about evaluation of unnecessary elements of the sequence, until they are needed in the program.

Strong Typing

Haskell is strongly typed. This means, e.g., that you can't freely use an Int instead of a Float, but rather have to explicitly convert the Int to a Float. Strong typing helps to find many programming errors. In particular, when combined with static typing, many programming errors can be caught before the program is run.

The type system of Haskell is polymorphic, allowing values of different data types to be handled using a uniform interface. A function that can be applied to values of different types is known as a polymorphic function. An example of a polymorphic data type is List (with elements of arbitrary type).

In our work, we make use of Haskell's fine grained set of types, both in terms of strong typing and polymorphism. These are very powerful features of Haskell. Types are automatically inferred wherever possible, which can help avoiding mistakes in code, and can help inferring the most general type for some polymorphic function.

3. The anatomy of distances for structured data

Distances on primitive data types

The currently considered list of primitive data types is Boolean, Discrete(S), and Real. We use the delta distance function on the first two, and absolute difference for Real. Delta yields zero given two identical inputs, one otherwise.

Distances on complex/structured types

Structured/complex data types are obtained through the (recursive) application of type constructors to simpler/primitive data types, with primitive types as base cases. The type constructors used here are Set() , Tuple() and Sequence(). Given distances on simpler (primitive) data types, we can compose distances for more complex structured types.

Distances on complex objects can be calculated through recursively inspecting the structure of the type. For this, we need (a) a function to generate pairs of objects of the simpler constitutive types, (b) distance functions on (objects of) the simpler types and (c) an aggregation function that we apply to the distance values obtained by applying (b) to the pairs produced by (a) to obtain a single (non-negative real) value of the distance between the complex objects.

In essence, the tree structure of the complex data type is inspected and for that type tree the following holds:

- every internal node represents a type constructor
- every leaf node is a primitive data type

Every internal node of this tree gets a pairing function and an aggregation function attached to it and every leaf node gets a distance function. The way of applying these functions to get a distance value (non-negative real) as a result is discussed next through an example.

For instance, given the Set(Char) type, and a distance function d() over the simple type Char, the distance of two sets of this type could be calculated as follows. If A and B are sets of this type, A={a_i | i=1..n} and B={b_j | j=1..m}, a choice can be made whether AxB={(a_i,b_j)|i=1..n,j=1..m} or just a subset thereof will be taken into account when determining the distance between the sets.

A function of the form

```
p :: [T]->[T]->[(T,T)]
```

can be used to determine the so-called important pairs of elements of the two complex objects, which will have the distance function d() applied to them. The function with this signature will be called a pairing function.

The second choice to be made is about the function that takes the computed distances between the pairs and produces a non-negative real, which is to be the distance between sets A and B. So, an additional function, called the aggregation function is to be defined, with the signature

```
agg :: [Float] -> Float
```

The third and last choice to make concerning the distance calculation on this complex type is which distance measure on our simple type Char to use. If we consider Char as a discrete type, the delta() function is the obvious choice. If we consider Char as an ordinal type (which we haven't discussed here), an absolute difference function which compares the two Chars after converting them to numbers (according to some character conversion table) may be used.

Pairing functions

The pairing functions are of more importance to the complex types using the Set() and Sequence() constructors. For the Tuple() constructor, given that it's heterogeneous, the pairing functions given below are most often in use.

```
p2 (Tuple2 a1 b1, Tuple2 a2 b2) =
[(a1,a2),(b1,b2)]
```

```
p3 (Tuple3 a1 b1 c1, Tuple3 a2 b2 c2) =
[(a1,a2),(b1,b2),(c1,c2)]
```

```
p4 (Tuple4 a1 b1 c1 d1, Tuple4 a2 b2 c2
d2) = [(a1,a2),(b1,b2),(c1,c2),(d1,d2)]
```

In the case of the Set(T) type constructor, a number of pairing functions can be used (Kalousis et al. 2006):

- all-to-all - every element from the first set is paired with every element from the second one
- minimum distance - an element from one set is paired with the closest element of the other set
- surjection pairing - considering all the surjections that map the larger set to the smaller, the "minimal" (with the distance Δ on T) such surjection is used
  $$d_S(S_1, S_2) = \min_{\eta} \sum_{(e_1,e_2)\in\eta} \Delta(e_1, e_2)$$
- linking - a mapping of one set to the other, all elements of each set participate in at least one pair
- matching - each element of the two sets associated with at most one element of the other set

Aggregation functions

An aggregation function has the signature:
```
type Agg = [Float] -> Float
```
Examples of functions that can be used are:

- square-root of the sum of squares (Euclidian distance)
  ```
  sqrt (sum [x*x|x<-xl])
  ```
- plain sum
  ```
  sum [x|x<-xl]
  ```
- minimum (or maximum)
- median

The first three functions give equal weight to all the distances that they aggregate, while the last three only take into account one (or sometimes two, in the case of median) values. All of the above are special cases of the so-called ordered weighted aggregation functions (OWA, Yager and Kacprzyk 1997), which first sort the values to be aggregated, then apply a set of weights before aggregating. Assuming the list is sorted in ascending order, minimum gives a weight of one to the first element, maximum to the last, and median to the middle element (or weights of ½ to the two middle elements if the number of elements is even).

Haskell makes available an interesting and powerful feature when implementing the above. If the following piece of Haskell code is explored, taking into account the definition of the Agg type signature:
```
wSum:: [Float] -> Agg
wSum weights elems =
sum [a*b | (a,b) <- zip weights elems ]
```
it can be concluded that this aggregation function, weighted sum aggregation, has an additional parameter - an array of real values, that is weights. Haskell in this case allows the user to evaluate and use throughout the code the construct `wSum weights`, which is a specific aggregation function obtained through partial evaluation of the `wSum` function: the evaluation is partial as not all parameters are provided, in particular the values to be aggregated. For this function definition to work properly it is required that the list `weights` is at least as long as the list `elems`.

4. A small database of distance function components

Populating the individual aspects

We have implemented a small database DDTD (database of data types and distances) where the definitions of data types and their corresponding distance functions are stored. We start with the primitive data types mentioned above and the basic distance functions on these. We also store additional distance functions on the primitive data types, as well as aggregation and pairing functions.

We have also implemented a generic version of the kNN algorithm, for demonstration purposes as well as for testing the Haskell implementation of the concepts discussed above (structured data and distances thereon). Datasets conforming to type definitions stored in the DDTD can be loaded from a database or from an XML file. This allows us to experiment with machine learning algorithms that work with structured data.

The process of populating DDTD with data type definitions, distance definitions, additional aggregation or pairing functions can be carried out either from the command line, or from a graphical interface (currently supporting a subset of the actions listed below). We can

- create definitions of new data types (composing complex data types out of simpler ones)
- create a definition of a distance over some data type (either using built-in functions or additional custom aggregation, pairing and distance functions)
- add a new distance function on a primitive type
- add a new aggregation function
- add a new pairing function

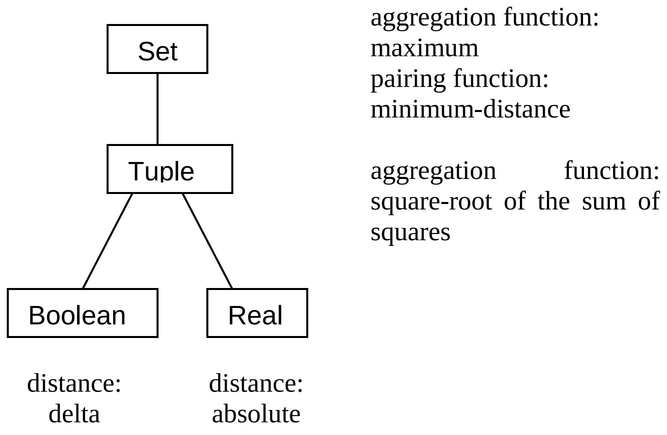
The data types can be described using XML or Haskell code. The additional functions have to be in Haskell syntax. The reason behind using the Haskell syntax is that it provides extensive support for mathematical functions (mostly defined in its Prelude), as well as support for processing lists, which can be easily learned, grasped and reused.

For every function to be added into the system, its signature (expected input data) has to be defined first, since some functions could use additional parameters (as was the case with the `wSum` aggregation function described in section 3 of this text). The definitions of the new functions have to be first checked for errors and then, if they produce the expected results, will be imported into the database, for further use. The possibility for additional functions and custom data types greatly increases the potential for use of DDTD.

DDTD usage scenario

Let us take for example the data type
```
t :: Set (Tuple2 Bool Float)
```
Note that this is the true Haskell definition of the type: Here we use Tuple2 instead of Tuple, as Tuple is really

a class of type constructors of varying arity (Tuple1, Tuple2, …), rather than a type constructor. Two type constructors are used (Set and Tuple) and two primitive types (Bool and Real) in the above definition.


Set
Tuple
Boolean
Real
aggregation function: maximum
pairing function: minimum-distance
aggregation function: square-root of the sum of squares
distance: delta
distance: absolute

Picture 1. A custom data type with a distance defined for it

Using DDTD this custom data type is first declared. Then using a plain XML editor, the dataset that is to consist of this kind of objects is defined or imported from a database (another possibility could be to load it directly from a file). Once we have defined a data type, we can define a distance function on this data type (covered in the next section): As soon as this is done, distances on selected pairs of objects are calculated, for the purpose of checking if the results are as expected (in terms of types). Finally, a machine learning algorithm (like the implementation of k-NN mentioned above) can be invoked on the dataset.

Custom creating distance functions

For the data type in our example, a distance function could be defined in the following way (see Figure 1):

- For the Set type constructor, the aggregation function maximum and the pairing function minimum-distance are used
- For the Tuple type constructor, the aggregation function square-root of the sum of squares and the default pairing function are used
- For the Bool primitive type the distance-delta is used
- For the Real primitive type the distance-absolute is used

This distance function definition is converted into an appropriate XML code and stored in database, for later use.

If the functions supported by DDTD by default are not suitable for our complex distance definition, additional functions can be added. For instance, if an additional aggregation function is needed, for, say the median of a list of real values, the following should be carried out. Since the new function is going to be an aggregation function, the signature for aggregation function should be:

```
type Agg = [Float] -> Float
```

Finally, the body of the function should be written:

```
median xl = s!!(length `div` 2)
     where s = Data.List.sort xl
```

This definition will be checked for errors and then imported into the database and can later be used accordingly.

5. Conclusions and related work

We have been here concerned with distance-based machine learning, and in particular in such approaches that can handle arbitrary types of structured data. We have followed the approach proposed by Dzeroski (2007) to develop generic algorithms (in our case kNN), complemented with a database of definitions of data types and distance functions on these types. Moreover, the database contains basic building blocks for constructing distance functions on structured data and allows the user to custom create new ones, as well as choose from existing distance functions. To implement this, we have chosen a functional programming approach, which supports the higher-order nature of the operations that manipulate functions necessary for this.

Our work is related to inductive databases (IDBs, Imielinski and Mannila 1996): IDBs store patterns (and models) in addition to data. Most of the work in this area has focused on storing (and querying) local (frequent) patterns expressed in logical form. Our DDTD can be viewed as an inductive database storing global predictive models: the combination of a dataset, a distance function and a generic algorithm (such as kNN) yields a predictive model.

Allison (2004) also considers a functional programming approach to machine learning. He uses functional programming to define data types and type classes for models (where models include probability distributions, mixture models and decision trees) that allow for models to be manipulated in a precise and flexible way. However, he does not consider distance-based learning.

Finally, we consider the work on modular domain-specific languages and tools (Hudak 1998) relevant to our approach, especially for further work. Namely, we believe our approach can be extended to arrive at domain-specific languages for data mining. These might be coupled with domain-specific languages in a specific area of interest, e.g., a multi-media language. We believe that this would greatly facilitate the development of domain-specific data mining approached and their practical applications.

Acknowledgement. This work was supported by the EU funded project IQ (Inductive Queries for Mining Patterns and Models).

References

[1] Allison, L.: Models for machine learning and data mining in functional programming. *Journal of Functional Programming* 15: 15–32, 2004.
[2] Džeroski, S: Towards a General Framework for Data Mining. In S. Džeroski, J. Struyf (eds.) *Knowledge Discovery in Inductive Databases, 5th International Workshop, KDID 2006, Revised Selected and Invited Papers*, pp. 259-300. Springer, Berlin, 2007.
[3] Hudak, P: Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse,* pp. 134-142. IEEE Computer Society Press, 1998.
[4] Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. *Comm. of the ACM* 39: 58–64, 1996.
[5] Kalousis, A., Woznica, A., Hilario, M.: A unifying framework for relational distance-based learning founded on relational algebra. Technical Report, Computer Science Dept. Univ. of Geneva (2006)
[6] Thompson, S.: Haskell: *The Craft of Functional Programming.* Add. Wesley (1999)
[7] Yager, R, Kacprzyk, J: *The Ordered Weighted Averaging Operators: Theory and Applications.* Springer, Berlin, 1997.