A Linear Kernel on Strongly Typed Terms

Thomas Gärtner[1,2] and Peter A. Flach[1]

[1] Department of Computer Science, University of Bristol,
Woodland Road, Bristol BS8 1UB, U.K.
{gaertner,flach}@cs.bris.ac.uk
[2] Knowledge discovery team, AiS, GMD,
Schloß Birlinghoven, D-53754 Sankt Augustin, Germany
thomas.gaertner@gmd.de

Abstract. Support vector machines and other kernel methods have successfully been applied to various tasks in attribute-value learning. A kernel function is any function in input space that corresponds to an inner product in some feature space. In this discussion paper, we propose a kernel function on strongly typed first-order terms, and we show how this kernel corresponds to the linear inner product in a feature space containing a propositionalised description of the individuals.

1 Introduction

Support vector machines and other kernel methods [1,2] have successfully been applied to various tasks in attribute-value learning. The structure of real-world data, however, is usually too rich to be adequately represented by attribute-value tuples. Using a term-based representation, first-order logic generalises the attribute-value representation by allowing complex types at the top-level and at intermediate levels. An individual, or example, is described by a single structured term. As has been argued in [3], using a strongly typed language has the distinct advantage that the type structure of terms describing individuals can be exploited as a powerful declarative bias.

Kernel methods are a class of learning algorithms that have been formulated such that individuals never occur isolated, but always in the form of inner products on pairs of individuals. A kernel function is any function in input space that corresponds to an inner product (a kind of unnormalised similarity measure) in some feature space. Defining kernel functions on strongly typed terms means crossing the boundary between attribute-value and relational learning – it enables support vector machines and other kernel methods to be applied to much more expressive representation spaces. The (implicit) feature transformation associated with a kernel function establishes a form of propositionalisation.

In this discussion paper, we propose a kernel function that closely corresponds to the dot product, i.e., the linear inner product, in a feature space containing a propositionalised description of the individuals. The outline of the paper is as follows. Section 2 gives an example of a strongly typed term representation. Section 3 gives the definition of the proposed kernel, and indicates how an equivalent feature space can be constructed. Section 4 concludes with some discussion topics.

2 Strongly typed term representations

We start by giving an example of a strongly typed term representations using the syntax of the Escher language [4]. Our example concerns the well-known mutagenicity classification task, involving the prediction of mutagenic activity of molecules [5]. An abstract view of a molecule is that it is a graph with atoms as nodes and bonds as edges. We represent this graph by the set of atoms and the set of bonds. The type signature is as follows:

```
data Element = Br | C | Cl | F | H | I | N | O | S;

type Ind1 = Bool;
type IndA = Bool;
type Lumo = Float;
type LogP = Float;
type Label = Int;
type AtomType = Int;
type Charge = Float;
type BondType = Int;
type Atom = (Label,Element,AtomType,Charge);
type Bond = ({Label},BondType);
type Molecule = (Ind1,IndA,Lumo,LogP,{Atom},{Bond});

mutagenic::Molecule->Bool;
```

A molecule is represented by a sextuple, the first four components of which are simple types such as Booleans and floating point numbers. The fifth component is a set of atoms, where an atom is represented by a four-tuple (labels are needed because molecules are graphs rather than trees). The last component of a molecule sextuple records the bond information.

Here is (part of) an example, classifying one particular molecule as mutagenic.

```
mutagenic(True,False,-1.246,4.23,
  {(1,C,22,-0.117),(2,C,22,-0.117),…,(26,O,40,-0.388)},
  {({1,2},7),…,({24,26},2)}) = True;
```

So, for instance, in this molecule there is a bond between Carbon atom 1 and Carbon atom 2 of type 7. The following is a possible (partial) induced definition based on the theories induced by Progol.

```
mutagenic(m) =
  ind1P(m) == True ||
  lumoP(m) <= -2.072 ||
  (exists \a -> a 'in' atomSetP(m) && elementP(a)==C &&
          atomTypeP(a)==26 && chargeP(a)==0.115) ||
  (exists \b1 b2 -> b1 'in' bondSetP(m) && b2 'in' bondSetP(m)
  &&
          bondTypeP(b1)==1 && bondTypeP(b2)==2 &&
          not disjoint(labelSetP(b1),labelSetP(b2)) ||
  (exists \a -> a 'in' atomSetP(m) &&
          elementP(a)==C && atomTypeP(a)==29 &&
          (exists \b1 b2 ->
                  b1 'in' bondSetP(m) && b2 'in' bondSetP(m) &&
                  bondTypeP(b1)==7 && bondTypeP(b2)==1 &&
                  labelP(a) 'in' labelSetP(b1) &&
                  not disjoint(labelSetP(b1),labelSetP(b2))))
  ||
  …;
```

The projection `atomSetP` is the projection onto the fifth component of a molecule. Similarly, `labelSetP` is the projection onto the first component of a bond. The background predicate `disjoint` returns false if its arguments intersect and true otherwise.

The types involved in this example are simple types ranging over atomic values (technically these are data constructors with zero arity), tuples (or Cartesian products) and sets. Other types are possible, e.g. lists (usually constructed from one binary data constructor – the `cons` functor – and one nullary data constructor – the empty list) and multisets (a generalisation of sets where the characteristic function maps into positive integers rather than Booleans). Below, we restrict attention to tuples, nullary data constructors, and multisets, but we also suggest possible extensions to other datatypes.

3 The Kernel and Corresponding Feature Space

In this section we define a kernel on strongly typed terms (Section 3.1) and construct the corresponding propositionalised feature space (Section 3.2). Section 3.3 presents a simple example, and Section 3.4 suggests various possible extensions of the basic kernel.

3.1 Basic Kernel Definition

The kernel function on individuals x and z is denoted by $k(x; z)$. Our definition is recursive and covers the following generic data types: tuples, data constructors of zero arity (i.e., constants), and multisets. The kernel definitions are subscripted with the type which they can be applied to.

- **Tuples**:

  $$k_T(x;z) = \sum_{i=1}^{n} k_{T_i}(x_i;z_i)$$

  where $x = (x_1, ..., x_n)$ and $z = (z_1, ..., z_n)$ are tuples of type $T = T_1 \times \ldots \times T_n$.
- **Data constructors of zero arity (the "matching" kernel function):**

  $$k_T(x;z) = k_0(x;z) = \begin{cases} 1; \text{ iff } x = z \\ 0; \text{ otherwise} \end{cases}$$

  where x and z are data constructors of type T.
- **Multisets:**

  $$k_{V \to \Re}(x;z) = \sum_{e \in V; f \in V} k_V(e;f) \ * \ x(e) * z(f)$$

  where $x :: V \to \Re$ and $z :: V \to \Re$ are multisets over type V.

For practical reasons it is often required to fix a depth limit. If the depth of a recursion is deeper that this depth limit then the matching kernel function k_0 is applied, no matter what the actual data type is.

3.2 Propositionalised Feature Space

We define a propositionalisation such that any term consisting of tuples, multisets and constants is transformed into a tuple of integers. The significance of this propositionalisation is that the dot product of tuples in this transformed space corresponds to the kernel function defined above. Even though the kernel does not actually perform this propositionalisation, it can be seen as justifying the definition of the kernel by providing its semantics.

- **Tuples:**
  recursively transform each component into a tuple of integers and then join these together into one flat tuple of integers.
- **Data constructors of zero arity:**
  fix an order on the set of data constructors and transform the i-th data constructor into a tuple of integers with the i-th component set to 1 and all others set to 0.
- **Multisets of data constructors of zero arity:**
  fix an order on the set of data constructors and transform the multiset into a tuple of integers with the i-th component set to the number of occurrences of the i-th data constructor in the multiset.
- **Multisets of tuples**
  transform into a tuple of multisets, recursively transform the multisets and finally apply the tuple case.
- **Multisets of multisets**
  merge into a new multiset using multiset sum of all inner multisets and iterate.

It should be noted that this propositionalisation requires knowledge of the domain of each data type. However, the kernel definition in Section 3.1 does not require any knowledge about the domains of each type, which is useful if the domain of a type is unknown or infinite.

3.3 Example

Consider a type definition similar to Michalski's east- and westbound trains learning problem:

A shape is a type consisting of several data constructors of zero arity:

Shape = {rectangle, u_shaped, bucket, hexa}

A load is a type consisting of several data constructors of zero arity:

Load = {circle, hexagon, triangle}

A car is a 2-tuple:

Car = Shape x *Load*

An example of a car is (bucket, triangle). This term is propositionalised into the tuple of integers (0,0,1,0,0,0,1). Similarly, the car (rectangle,triangle) is propositionalised into (1,0,0,0,0,0,1).

We can further define a train as a (multi-)set of cars: $Train = Car \to \Re$. The train {(bucket, triangle), (rectangle, triangle)} is then represented by the vector: *(1,0,1,0,0,0,2)*. The inner product of the trains *x* = {(bucket, triangle)} and *z* = {(bucket, triangle), (rectangle, triangle)} is calculated as the dot product in the propositionalised feature space: *(0,0,1,0,0,0,1) * (1,0,1,0,0,0,2) = 3*.

The linear kernel (as defined above) on the same trains is calculated as follows:

$$
\begin{aligned}
&k_{Car\to\Re}\left(\{(\text{bucket},\text{triangle})\},\{(\text{bucket},\text{triangle}),(\text{rectangle},\text{triangle})\}\right)\\
&= k_{Car}\left((\text{bucket},\text{triangle}),(\text{bucket},\text{triangle})\right)*1*1\\
&\quad + k_{Car}\left((\text{bucket},\text{triangle}),(\text{rectangle},\text{triangle})\right)*1*1\\
&= \left(k_{Shape}(\text{bucket},\text{bucket}) + k_{Load}(\text{triangle},\text{triangle})\right)\\
&\quad + \left(k_{Shape}(\text{bucket},\text{rectangle}) + k_{Load}(\text{triangle},\text{triangle})\right)\\
&= (1+1)+(0+1) = 3
\end{aligned}
$$

3.4 Kernel Extensions

We suggest the following extensions to the linear kernel.

- **Data constructors of arbitrary arity:**

  $$k_T(x;z) = 1 + \sum_i k_{T_i}(x_i;z_i)$$

  where $x = f(x_1 \ldots x_n)$ and $z = f(z_1 \ldots z_n)$ with *f* a data constructor of type $T \to T_1 \times \ldots \times T_n$; if *x* and *z* have non-matching data constructors or different arities $k_T(x;z)=0$.

The above definition of a kernel on data constructors of zero arity is a special case of this kernel.

- **Numbers:**
  $k_{\Re}(x;z) = xz$ where *x* and *z* are numbers of the same type.
- **Functions:**
  $k_{V \to U}(x;z) = \sum_{e \in V; f \in V} k(e;f) \ * \ k\big(x(e);z(f)\big)$
  where $x :: V \to U$ and $x :: V \to U$ are functions

The above definition of a kernel function on multisets is a special case of this kernel function combined with the previous case for numbers.

- **Other kernel functions:**
  $k'_T(x,z)$ is chosen iff *x* and *z* are of type *T*, and $k_T :: T \to T \to \Re$ is known to be an appropriate kernel function on instances of type *T*. Please note, that *T* is not a generic, but a concrete type.
- **Distance functions:**
  $k(x;z) = \frac{1}{2}\left[d^2(0,x) - 2d(x,z) + d^2(0,z)\right]$
  where *x* and *z* are of type T, $d :: T \to T \to \Re$ is known to be an appropriate distance function on instances of type T, and **0** is an appropriate zero element of type *T*.

These kernel definitions have not been described above, as the construction of the corresponding propositionalised feature space is an open problem. Still it can easily be shown that the function they make up is actually a kernel function.

4 Discussion Topics

The following is a list of topics for possible discussion at the workshop. We also hope to be able to present some empirical results.

- Calculating our kernel function is computationally at most as complex as calculating the inner product in the propositionalised feature space. Usually, it is less complex.
- Lists are a special case of data constructors: Applying the data constructor kernel to lists gives a result similar to anti-unification (i.e., lists which share prefixes are much closer than lists which share suffixes).
- The matching distance is frequently replaced by a function estimated on the training data. This is not necessary here, as the support vector machine will anyway (implicitly) reweight the features.
- Logical combinations (monomials) of these features can be learned by using a polynomial kernel function at the top-level $k_{polynomial}(x;z) = \big(k(x;z) + p\big)^n$.

- The strongly typed term language is used, as our kernel definition is easier to understand in this language, however, equivalent kernels can easily be defined in other (multi-relational) languages.

Acknowledgements

Part of this work is supported by the Esprit V project (IST-1999-11495) *Data Mining and Decision Support for Business Competitiveness: Solomon Virtual Enterprise*. We would like to thank the two anonymous reviewers for their comments and suggestions.

References

[1] Boser, B.E., Guyon, I. M., and Vapnik, V.N. A training algorithm for optimal margin classifiers. In D. Haussler (Ed.), *Proceedings of the fifth Annul ACM Workshop on Computational Learning Theory* (pp. 144-152). ACM Press, 1992.
[2] Cristianini, N., and Shawe-Taylor, J. *An introduction to Support Vector Machines and other kernel-based methods*. Cambridge University Press, 2000.
[3] Flach, P.A., Giraud-Carrier, C., and Lloyd, J.W. Strongly typed inductive concept learning. In D. Page (Ed.), *Proceedings of the eighth International Conference on Inductive Logic Programming*, Springer-Verlag, 1998.
[4] Lloyd, J.W. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1998.
[5] Srinivasan, A., Muggleton, S., King, R., and Sternberg, M. Mutagenesis: ILP experiments in a non-determinate biological domain. *Proceedings of the fourth International Workshop on Inductive Logic Programming*, GMD-Studien 237, 1994.