Predictive Toxicology using a Decision-tree Learner

K.S. Ng, J.W. Lloyd and A.W. Slater

Computer Sciences Laboratory,
Research School of Information Sciences and Engineering,
Australian National University

kee@csl.anu.edu.au, jwl@csl.anu.edu.au, Andrew.Slater@csl.anu.edu.au

1 Introduction

This extended abstract outlines a submission to The Predictive Toxicology Challenge for 2000-2001 [PTC]. The Challenge is to obtain models that predict the outcome of biological tests for the toxicity of chemicals using information related to chemical structure only.

The models reported here are based on the approach outlined in [BGCL00]. (Much more detail is given in [BGCL01].) In essence, the learning system is a decision-tree system. However, it is rather more general than conventional decision-tree learners in that it allows the individuals that are to be classified to be represented as certain terms in a higher-order logic rather than as simple feature vectors. The logic allows the use of sets, multisets, lists, graphs and so on, to represent individuals, thereby capturing complex structural information about the individuals [Llo01]. This information is highly relevant to the main aim of the Challenge, which is to predict toxicity from the chemical structure alone. The approach adopted here provides an alternative to the more usual approach of using first-order logic to capture the structural information.

The next section outlines the approach to knowledge representation. The third section outlines how predicates are constructed. The last section gives some information about the experiments performed and contains ROC diagrams for the models for each of the four datasets.

2 Knowledge Representation

We adopt a standard approach to knowledge representation. The basic principle is that *an individual should be represented by a (closed) term.* For a complex individual, the term will be correspondingly complex. Nevertheless, this approach has significant advantages: the representation is compact, all information about an individual is contained in one place, and the structure of the term provides strong guidance on the search for a suitable induced definition.

What types are needed to represent individuals? Typically, one needs the following: integers, floats, characters, strings, and booleans; data constructors; tuples; sets; multisets; lists; trees; and graphs. The first group are the basic building blocks of terms representing individuals. The type of integers is denoted by `Int`, the type of non-negative integers by `Nat`, the type of booleans by `Bool`, and the type of floating-point numbers by `Float`. Also needed are data constructors for user-defined types. Tuples are essentially the basis of the attribute-value representation of individuals, so their utility is clear. Less commonly used elsewhere for representing individuals are sets and multisets. However, sets, especially, and multisets are basic and extremely useful data types. Other constructs needed for representing individuals include the standard data types, lists, trees, and graphs.

We model the chemical molecules as graphs, in which the vertices are atoms and the edges are bonds. In addition, depending on the experiment, we also include various features provided from the data engineering phase. These features were taken, after a feature selection phase, from the set of the Leuven functional groups, the DRAGON features, and the Treymer features. Thus the representation of a molecule is a tuple most of whose arguments are features from (one or more of) the above feature sets with another argument containing the graph representation of the molecule. We now give the types in the representation and the representation of a typical molecule.

The type of a molecule is `Molecule`, which is defined by the following type synonym.

```
Molecule = (Structure, Aro2n, FG1, ... , FG27, Lumo, Homo, Dipole,
                                                   Dragon1, ... , Dragon839);
```

This declaration just states that `Molecule` is a synonym for the product type on the right hand side. The types appearing in the right hand side are defined as follows.

```
Structure = Graph Atom Bond;
Aro2n = Bool;
FG1 = Nat;
  ...
FG27 = Nat;
Lumo = Float;
Homo = Float;
Dipole = Float;
```

`Structure` is the type of the graph representation of the molecule, where `Atom` is the type of an atom and `Bond` is the type of a bond. In more detail, the type `Graph Atom Bond` is defined as follows.

```
Label = Nat;
Graph Atom Bond = ({(Label, Atom)}, {(Label -> Nat, Bond)});
```

The type `Label -> Nat` is the type of multisets. Here the multisets are intended to contain just two items and thus represent unordered pairs (of the atoms joined by a bond). We use `<.,.>` to denote an unordered pair. `Atom` is the type of the elements `C`, `H`, `Cl`, and so on. `Bond` is the type of bonds, where `-` is the constant for a single bond, `=` for a double bond, `#` for a triple bond, and `~` for a resonant bond. `Aro2n` refers to the boolean feature aro2n in the Leuven set. For `i` = `1`, . . . , `27`, `FGi` is the type of the `i`th functional group feature also in the Leuven set. Each `FGi` is the type of an integer recording how often a particular functional group occurs in a molecule. `Lumo`, `Homo`, and `Dipole` are self-explanatory. For `i` = `1`, . . . , `839`, `Dragoni` is the `i`th DRAGON feature. The values for each DRAGON feature were discretised into 10 values by ordering the set of values over all the molecules for that feature and then putting them into 10 equal-sized, ordered buckets labelled 1 to 10. Thus the constants of type `Dragoni`, for `i` = `1`, . . . , `839`, are `1`, . . . , `10`. Note that this representation already reflects some choices about which features may be useful. We used only lumo, homo, and dipole from the Treymer set of features since, from our experiments, the others seemed not to be useful. Later we shall see how the number of DRAGON features was effectively reduced by the choice of rewrites that give rise to predicates.

As an example, here is the representation of the molecule TR002 (Trichloroethylene), which is also pictured below in Figure 1.

```
(({(1, C), (2, Cl), (3, C), (4, Cl), (5, Cl), (6, H)},
  {(<1,2>,-), (<3,4>,-), (<3,5>,-), (<1,6>,-), (<1,3>,=)}),
 False,
 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -0.0779128, -0.238038, 0.83703,
 10, 3, 1, ..., 1, 4, 3)
```

3 Predicate Construction

We now turn to the issue of constructing predicates in order to perform splits during the building of a decision tree. The problem we face is to induce a suitable definition for the desired target

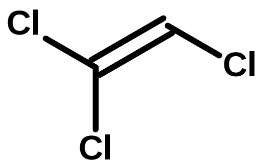

Figure 1: TR002 (Trichloroethylene)

function, assuming that the individuals (the chemical molecules) in the domain of this function are represented by terms as described above. For this, we need to investigate the detailed structure of these terms to construct predicates that the molecules may or may not satisfy and also exploit the type of the terms representing the molecules and the types of the subterms of these terms. However, even with the strong hints provided by these types, finding a suitable predicate is a search problem – one must search in a systematic way through a space of potential predicates. This search space is made up by composing more basic functions into predicates in all possible ways. So we need to say something about what these more basic functions are and how they are composed.

First, we deal with composition. This is simply handled by the function '`.`' having the signature `(R -> S) -> (S -> T) -> (R -> T)` and defined by `(f.g) x = g (f x)`, where `R`, `S` and `T` are types.

Next we turn to the appropriate set of functions. A *transformation* is a function `f` having the signature `(R1 -> Bool) -> ... -> (Rk -> Bool) -> S -> T`, where `R1`, ..., `Rk`, `S` and `T` are types and `k` ≥ 0. The idea is that, if `ri : Ri -> Bool`, `i` $= 1, \ldots,$ `k`, are predicates, then `(f r1 ... rk)` is a function of type `S -> T` and functions obtained like this can be composed to obtain a predicate. A special case is when `k` $= 0$, in which case the transformation is itself such a suitable function.

We now introduce some transformations which will be used in the models we construct. Given several predicates, it is common to want to construct the predicate obtained by forming the "conjunction" of these predicates. This is achieved via the transformation `andN` (where `N` ≥ 2) having the signature `(T -> Bool) ->...-> (T -> Bool) -> T -> Bool` and defined by `andN p1 ... pN x = (p1 x) && ... && (pN x)`. If `N` is an integer, then `(==N) : Int -> Bool` is a transformation, in fact a predicate, defined by `(==N) m = m == N`. Similarly, one can define transformations `(<N)` corresponding to `<`, `(>N)` corresponding to `>`, `(/=N)` corresponding to `/=`, and so on. More generally, if `T` is a type and `C` a constant of type `T`, then we can make a predicate `(==C)` on `T` by defining `(==C) x = x == C`. Similarly, the predicate `(/=C)` on `T` is defined by `(/=C) x = x /= C`. For any type `T`, there is a transformation `top : T -> Bool` defined by `top x = True`.

We now discuss three relevant types more systematically, giving various transformations and predicates defined on them.

The transformations for tuples are the projections `projTi : (T1, ..., Tn) -> Ti` defined by `projTi (t1, ..., tn) = ti`, for `i = 1, ..., n`.

As an example of predicate construction, suppose `C` is a constant of type `T1`. Then, by composition, we can construct the predicate `projT1.(==C)` on individuals of type `(T1, ... , Tn)`. This predicate is true on an individual of this type iff its first component is `C`.

In a higher-order logic, a set is identified with a predicate, its characteristic function. Thus a set whose elements have type `T` is a function of type `T -> Bool`. When we wish to informally make a distinction between sets and predicates (even though mathematically they are identical), we use the synonym `{T}` for `T -> Bool`. Analogously, a multiset whose elements have type `T` is a function of type `T -> Nat`, where the value of the function at some element is the multiplicity of the element, that is, the number of times it appears in the multiset.

For sets of type `{T}`, consider the transformation `setExists1 : (T -> Bool) -> {T} -> Bool` defined by `setExists1 b t = exists \x -> (b x) && x 'in' t`. Then, if `b` is a predicate on the type `T`, `(setExists1 b)` is a predicate of type `{T} -> Bool` that is true of a set iff it has an element which satisfies the predicate `b`. One can also define `setExists2` which looks for two elements satisfying given conditions, and so on. Analogously, for multisets, one has `msetExists2`.

Now consider the transformation

```
domCard : (T -> Bool) -> {T} -> Nat;
domCard b t = card {x | (b x) && x ‘in‘ t};
```

where `card` computes the cardinality of a set. Thus, for each predicate `b`, the function `(domCard b) : {T} -> Nat` computes the cardinality of the subset of elements of the argument that satisfy the predicate `b`.

For (undirected) graphs, there is a type constructor `Graph` such that the type of a graph is `(Graph V E)`, where `V` is the type of information in the vertices and `E` is the type of information in the edges. To avoid the need for explicit references to the label type, we introduce a type constructor `Vertex` such that the type of a vertex is `(Vertex V E)` and a type constructor `Edge` such that the type of an edge is `(Edge V E)`. Here are some transformations on graphs.

```
vertices : Graph V E -> {Vertex V E};
edges : Graph V E -> {Edge V E};
vertex : Vertex V E -> V;
edge : Edge V E -> E;
connects : Edge V E -> (Vertex V E -> Nat);
```

The transformation `vertices` returns the set of vertices of a graph, `edges` returns the set of edges of a graph, `vertex` returns the information at a vertex, `edge` returns the information on an edge, and `connects` returns the (unordered) pair of vertices joined by an edge.

For a particular experiment, the hypothesis language is determined by a corresponding set of rewrites. The predicates in the heads and bodies of these rewrites are made by composing transformations from the above collection. This method of using rewrites to generate predicates is described in detail in [BGCL01].

We now indicate the rewrites for the experiments in this paper. First, here are some relevant transformations.

```
projStructure : Molecule -> Structure;
projAro2n : Molecule -> Aro2n;
projFG1 : Molecule -> FG1;
   ...
projFG27 : Molecule -> FG27;
projLumo : Molecule -> Lumo;
projHomo : Molecule -> Homo;
projDipole : Molecule -> Dipole;
projDragon1 : Molecule -> Dragon1;
   ...
projDragon839 : Molecule -> Dragon839;
```

The top-level rewrites for the male rat experiments were as follows.

```
top >-> projStructure.edges.(and2 top top)
top >-> projStructure.vertices.(domCard top).(<20)
top >-> projStructure.vertices.(domCard top).(>15)

top >-> projAro2n.(==False)
top >-> projAro2n.(==True)

top >-> projFG1.top
  ...
```

```
top >-> projFG27.top

top >-> projLumo.(and2 top top)
top >-> projHomo.(and2 top top)
top >-> projDipole.(and2 top top)

top >-> and2 (projDragon18.top) (projDragon277.top)
  ...
top >-> and2 (projDragon710.top) (projDragon785.top)
```

The rewrite process for generating predicates (to find a suitable predicate to split a node in the decision tree) starts with the weakest predicate `top` (of type `Molecule -> Bool`). The learner looks through the set of rewrites to find those whose head matches this `top`. These are listed above. The first three project into the graph representation of the molecules and set up conditions on the atoms and bonds. The next two check whether the aro2n property holds or not. Then follow the projections into the functional groups. After that there are the projections into the features lumo, homo, and dipole. The last collection of rewrites set up conjunction conditions on the DRAGON features. Our experiments indicated that DRAGON features 18, 277, 288, 710, and 785 were the most useful for the male rat dataset, so we restricted the DRAGON projections to just these features. These were identified by building small decision trees in cross-validations using just DRAGON features and selecting those that occurred most often near the roots of these trees (on the assumption that such features should be good discriminators). In general, a rewrite step involves the replacement of a redex in a predicate by the body of a rewrite whose head matches the redex to produce a new predicate.

By this process, a large number of predicates was generated. Some of these appear in the model for male rats given below. This method of generating predicates is flexible and convenient, and provides precise control over the hypothesis language. Other applications using this approach are in discussed in [BGCL01].

4 Experimental Results

The learning system used in the experiments is described in [BGCL00] and [BGCL01]. However, while functionally similar to the system described in [BGCL00], it was completely re-implemented by one of us (Ng) at the end of 2000 and is now substantially more efficient than the original system.

We carried out a large number of experiments on the four datasets. The models we submitted for the Challenge are given in Figures 2, 3, 4 and 5 below. Each point in the figures corresponds to an average (FP, TP) point obtained from a 10-fold cross validation on the training set. Each diagram has three models that attempt to cover the cost-neutral region in the centre, the "synthetic-chemist friendly" region in the bottom left-hand corner and the "toxicologist friendly" region in the top right-hand corner. The cost-sensitive models were obtained by introducing relative weights for the positive and negative examples in the accuracy heuristic used by the learning system – points in the "synthetic-chemist friendly" region were obtained by a relatively heavier weighting for negative examples, whereas points in the "toxicologist friendly" region were obtained by a relatively heavier weighting for positive examples. Models labelled as "Graph" in the legends used explicit graph-theoretic structural information from the `Structure` component of the representation, as well as various features from the set of functional groups, the DRAGON features, and the Treymer features lumo, homo, and dipole. The models labelled as "A-V" (attribute-value) did not use this explicit structural information (that is, the `Structure` component of the representation was dropped).

To give a feeling for the kind of models produced by the learner, the following is the definition for the function `carcinogenic` that corresponds to the point labelled $\diamond$ in Figure 2.

```
carcinogenic m =
  if and2 (projDragon18.(<7)) (projDragon277.(>1)) m
  then if projLumo.(and2 (<=1.617905) (>=-0.004372)) m
       then if and2 (projDragon277.(<7)) (projDragon710.(>2)) m
            then Negative
            else Positive
       else if projStructure.edges.(and2
                (domCard (and2 (edge.(==~)) connects.(msetExists2
                                             (vertex.(==C)) (vertex.(==C))))).(>9)
                (domCard (and2 (edge.(==-)) connects.(msetExists2
                                          (vertex.(==C)) (vertex.(==N))))).(>1)) m
            then if projDipole.(and2 (>=2.080337) (<=3.463803)) m
                 then Negative
                 else Positive
            else if and2 (projDragon288.(<5)) (projDragon785.(<3)) m
                 then if and2 (projDragon277.(>5)) (projDragon710.(<4) m
                      then Negative
                      else Positive
                 else Negative
  else if projDipole.(and2 (>=0.009589) (<=1.012996)) m
       then Negative
       else if and2 (projDragon18.(>7)) (projDragon288.(<7)) m
            then Negative
            else Positive;
```

Translated to structured English, this model is as follows.

```
if dragon18 < 7 and dragon277 > 1
then if -0.004372 <= lumo <= 1.617905
     then if dragon277 < 7 and dragon710 > 2
          then negative
          else positive
     else if the molecule has > 9 resonant bonds each of which connects two carbon atoms
             and > 1 single bonds each of which connects a carbon and a nitrogen atom
          then if 2.080337 <= dipole <= 3.463803
               then negative
               else positive
          else if dragon288 < 5 and dragon785 < 3
               then if dragon277 > 5 and dragon710 < 4
                    then negative
                    else positive
               else negative
else if 0.009589 <= dipole <= 1.012996
     then negative
     else if dragon18 > 7 and dragon288 < 7
          then negative
          else positive
```

Here are the definitions of the DRAGON descriptors that appear in this definition.

18 nAB number of aromatic bonds constitutional descriptors

277 MATS2e Moran autocorrelation - lag 2 / weighted by atomic Sanderson electronegativities 2D autocorrelations

288 MATS5p Moran autocorrelation - lag 5 / weighted by atomic polarizabilities 2D autocorrelations

710 H3e H autocorrelation of lag 3 / weighted by atomic Sanderson electronegativities GETAWAY descriptors

785 R3v R autocorrelation of lag 3 / weighted by atomic van der Waals volumes GETAWAY descriptors

A condition such as '`dragon18 < 7`' means that the value of DRAGON descriptor 18 must be in the bottom 60% of the set of all values for this descriptor over all molecules in the dataset.

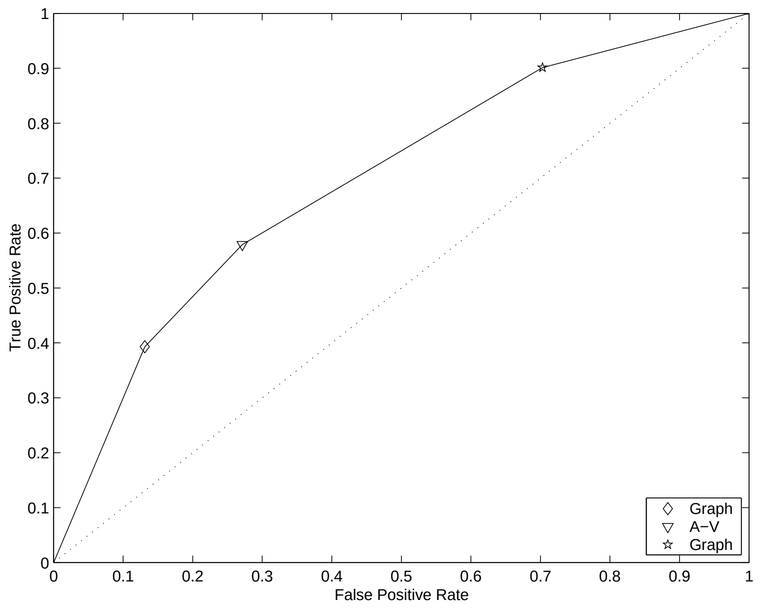

Figure 2: ROC diagram for male rats experiments

5 Conclusion

Our overall conclusion from the experience of trying to find models for predictive toxicology is unsurprising: it is a very difficult problem and we did poorly. We put this down partly to difficulties with the dataset, but more with our failure to identify appropriate hypothesis languages. On the other hand, we are confident that our approach to knowledge representation and predicate construction is highly suitable for this kind of problem and that, at some time in the future, with better data and a much better understanding of the concepts required to capture toxicity, we will succeed in finding good models. We look forward to the next predictive toxicology challenge.

Acknowledgement

We are grateful for expert advice from Dr. Nick Dixon and Professor Rod Rickards of the Research School of Chemistry, The Australian National University.

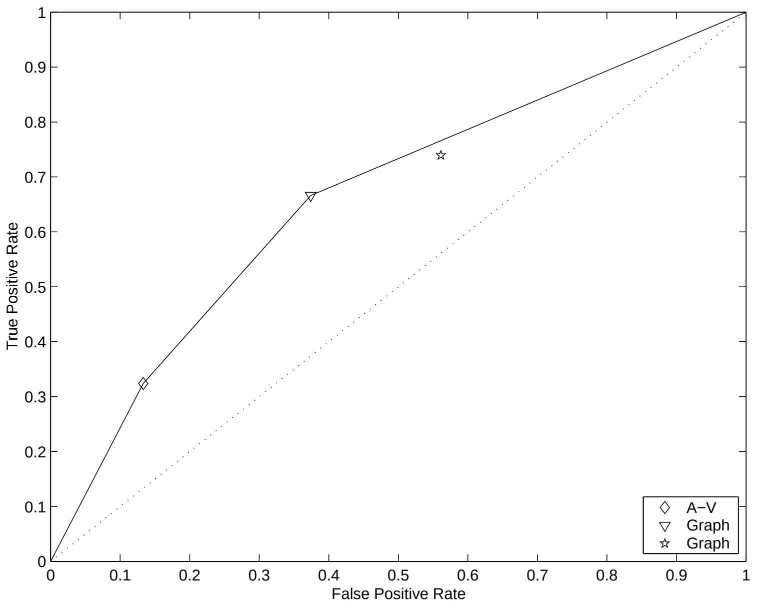

Figure 3: ROC diagram for female rats experiments

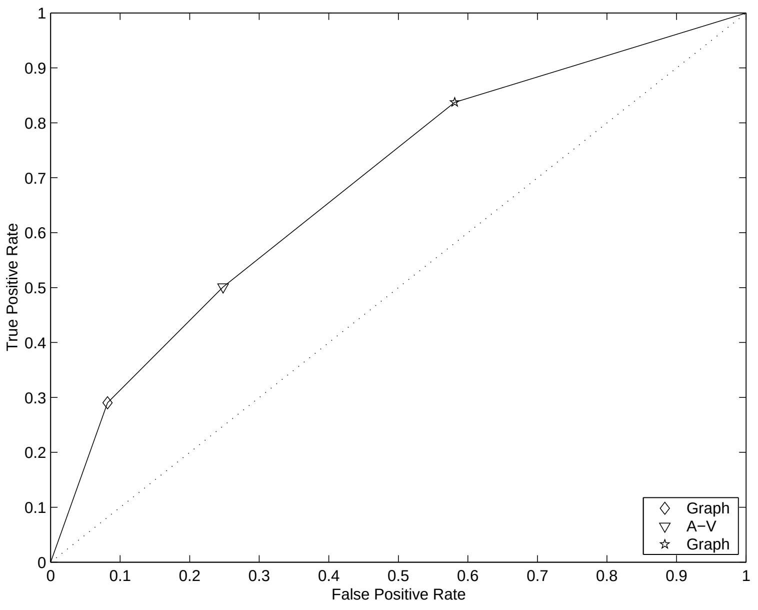

Figure 4: ROC diagram for male mice experiments

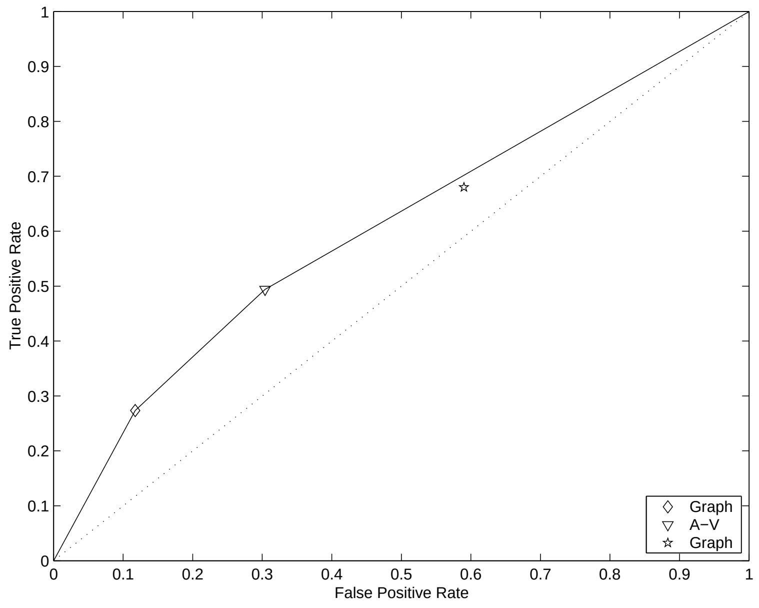

Figure 5: ROC diagram for female mice experiments

References

[BGCL00] A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Machine Learning: Proceedings of the Seventeenth International Conference (ICML2000)*, pages 81–88. Morgan Kaufmann, 2000.

[BGCL01] A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. A knowledge representation framework for inductive learning. Available at `http://csl.anu.edu.au/~jwl`, 2001.

[Llo01] J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://csl.anu.edu.au/~jwl`, 2001.

[PTC] Home page of The Predictive Toxicology Challenge for 2000-2001. `http://www.informatik.uni-freiburg.de/~ml/ptc/`.