IDA Group, CTU in Prague

TreeLiker Tutorial

(Command-Line Interface)

Contents

1 Introduction

TreeLiker is a collection of fast algorithms for working with complex structured data in relational form. The data can, for example, describe large organic molecules such as proteins or groups of individuals such as social networks or predator-prey networks etc. The algorithms included in TreeLiker are unique in that, in principle, they are able to search given sets of relational patterns exhaustively thus guaranteeing that if some good pattern capturing an important feature of the problem exists, it will be found. In experiments with real-life data, the algorithms were shown to be able to construct complete non-redundant sets of patterns for chemical datasets involving several thousands of molecules as well as for comparably large datasets from genomics or proteomics.

The included relational learning algorithms are tailored towards so-called tree-like features for which some otherwise very hard sub-problems (NP-hard) become tractable. The problem of finding a complete set of informative features remains hard also for tree-like features, however, we were able to develop algorithms for tree-like features which scale well for problems of real-life scale. Currently, the machine learning algorithms integrated in TreeLiker include implementations of relational learning algorithms HiFi and RelF and Poly in an intuitive GUI. The three algorithms were described in the following papers:

- **RelF:** Ondřej Kuželka and Filip Železný. Block-Wise Construction of Tree-like Relational Features with Monotone Reducibility and Redundancy. *Machine Learning, 83, 2011*
- **Poly:** Ondřej Kuželka, Andrea Szabóová, Matěj Holec and Filip Železný. Gaussian Logic for Predictive Classification. *ECML/PKDD 2011: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (this paper described a restricted version of Poly)
- **HiFi:** Ondřej Kuželka and Filip Železný. HiFi: Tractable Propositionalization through Hierarchical Feature Construction. *Late Breaking Papers, the 18th International Conference on Inductive Logic Programming, 2008*

There is a graphical user interface for TreeLiker called TreeLiker-GUI which provides access to basic functionality of the algorithms. A good strategy for analysing relational data is to use TreeLiker-GUI with a subset of the data to see what can be mined using relational learning from the data and, only after that, to start making a more complex experiment using either the scripting interface of Tree-Liker or directly the Java API.

2 Learning Examples

Learning examples are first-order-logic interpretations. An example is therefore simply a set of facts which are true in it. TreeLiker currently accepts two formats of learning examples: pseudoprolog and prolog.

A. **The pseudoprolog syntax** is exemplified below:

```
east hasCar(c), hasLoad(c,l), box(l)

west hasCar(c), hasLoad(c,l), triangle(l), hasLoad(c,l2), box(l2)
```

Here, the first *word* on each line denotes class of the example. The rest of the line is then the set of true facts - the description of the example. In this simple case, the first learning example is a *train which goes east* (that is the class of this example) and the train has a car which has a load which is box-shaped. The second learning example is a *train which goes west* and has a car which has a load which is triangle-shaped and another load which is box-shaped.

Note that no facts are shared among the individual learning examples. This means that even if the examples share some *background knowledge*, it is necessary to list it separately for each example. (There is an experimental version of the feature-construction algorithms which can work with inter-connected learning examples. This version can be obtained from the authors by request. It will be integrated into TreeLiker soon.)

B. **The Prolog syntax** is exemplified below. It describes the same learning examples as the pseudoprolog example above.

```
begin(example_1, east).

hasCar(c).

hasLoad(c,l).

box(l).

end(example_1).

begin(example_2, west).

hasCar(c).

hasLoad(c,l).

triangle(l).

hasLoad(c,l2).

triangle(l2).

end(example_2).
```

The number of examples as well as number of literals in them can be arbitrary.

3 Relational Features

The algorithms RelF, HiFi and Poly are intended for construction of relational features. Relational features are conjunctions of literals. For example,

$$\varphi = hasCar(X), hasLoad(X, Y), box(L)$$

is a feature (we use commas instead of $\wedge$'s). We say that a feature φ covers an example Ex if it is true in that example Ex when all free variables in the feature are assumed to be existentially quantified. It can be shown that this is equivalent to the following: A feature φ covers an example if and only if there is a substitution θ to the variables of φ such that $\varphi\theta \subseteq Ex$. So for example, our feature φ covers both of the learning examples from the previous section. This can be also seen as checking whether a conjunctive database query (= the feature) succeeds for a given relational database (= the learning example).

There are several settings in which the feature construction algorithms can work. The first is the **existential setting** in which we are really only interested in whether a given feature covers an example. The second setting is **grounding-counting setting**. In this setting, we are not only interested in whether a feature φ covers an example Ex but also in how many substitutions θ there are such that $\varphi\theta \subseteq Ex$.

Counting true groundings makes sense, for example, when we analyse chemical structures and we want to count the number of occurrences of some patterns (features). See, for example the next paper, in which RelF was used for analysis of DNA-binding proteins:

- **Proteins:** Andrea Szabóová, Ondřej Kuželka, Filip Železný and Jakub Tolar. Prediction of DNA-Binding Proteins from Structural Features. *Proceedings of the Fourth International Workshop on Machine Learning in Systems Biology, 2010.*

Finally, there is also a third setting which is multivariate relational aggregation. This is described in detail in the paper about **Poly**. Briefly, in this setting, a feature may contain several distinguished variables which are used as *extractors* of numerical information. Every substitution θ such that $\varphi\theta \subseteq Ex$ gives us one sample of the numerical variables which is a vector of real numbers. This vector can be used as input to a multivariate function (which may compute e.g. correlations of the variables etc.). The result is then computed by averaging outputs of the multivariate function over all samples for the given example Ex. Poly is able to construct not only features but also these multivariate functions which are multivariate polynomials in the case of Poly.

4 Language Bias - Templates

Templates provide means for syntactically constraining the sets of possible features. In essence, templates are very simple. They are similar in spirit to *mode-declarations* from systems Aleph or Progol, however, they differ from them in some crucial aspects. In the following text, we describe templates in detail.

A. **Templates are sets of literals.** For example, the following expression is a template:

$$\tau_1 = hasCar(-c), hasLoad(+c, -l), box(+l), triangle(+l).$$

The literals in the above template τ_1 are the expressions $hasCar(-c)$, $hasLoad(+c, -l)$, $box(+l)$ and $triangle(+l)$.

B. **The literals in templates have typed arguments.** In the template τ_1 above, the types are: c and l. Types are used to define which arguments may share a variable - these are those with the same types. For example, the next *feature* complies with the *typing* from template τ_1:

$$\varphi_1 = hasCar(X), hasLoad(X, Y), box(Y), hasLoad(X, Z), triangle(Z),$$

which can be checked easily: the variable X appears only in arguments marked by type c, the variable Y appears only in arguments marked by type l and the variable Z also appears only in arguments marked by type l. On the other hand, the next expression is not a valid feature according to the typing in template τ:

$$\varphi_2 = hasCar(X), hasLoad(Y, X)$$

because X appears in arguments marked by two different types: c and l.

Now, consider the next template:

$$\tau_2 = carbon(-a), bond(+a, -b), bond(-b, +a), hydrogen(+b).$$

There is one important difference to the first template and that is that there are two different typings of the same predicate: $bond(+a, -b)$ and $bond(-b, +a)$. Now, both next expressions are valid features:

$$\varphi_3 = carbon(X), bond(X, Y), hydrogen(Y),$$

$$\varphi_4 = carbon(X), bond(Y, X), hydrogen(Y),$$

because, for each literal in them, we can select the right literal from the template for which typing of the features φ_3 and φ_4 is valid.

C. **The arguments of the literals in templates have *modes*.** Modes can be the following (the signs in parentheses are the signs used in templates to denote the modes): *input* (+), *output* (-), *constant* (#), *ignored* (!), *aggregation* (*) and *global constant* (@).

C.1 **Input (+) and output (-) arguments.** We start by explaining the two most important types of modes: *input* and *output* modes. Any variable in a valid feature must appear exactly once as an *output* (i.e. in an argument marked by mode -) and at least once as an *input* (i.e. in an argument marked by mode +). For example, the expressions φ_1, φ_3 and φ_4 comply with this condition w.r.t. the respective templates. On the other hand, for example, the next expressions φ_5 and φ_6 do not comply with modes specified in template τ_1:

$$\varphi_5 = hasCar(C), hasLoad(C, L)$$

$$\varphi_6 = hasLoad(C, L), box(L).$$

The expression φ_5 is not valid because the variable L appears as an *output* but not as an *input*. The expression φ_6 is not valid because the variable C appears as an *input* but not as an output.

A simple way to get familiar with templates is to imagine them procedurally as defining a process for constructing features (although the process that we will show is exactly opposite to what the algorithms RelF, HiFi and Poly actually do - it is much less efficient approach which cannot solve many problems that RelF, HiFi and Poly can solve routinely).

The process can be visualized as follows: We find a literal in the given template (e.g. τ_1) which does not contain any input-argument (i.e. none of its arguments is marked by +) and create the first literal of the feature from it, e.g.

$$hasCar(X) \tag{1}$$

from the template-literal $hasCar(-c)$. Then we search for literals in the template which have an input-argument with such type that can be connected to $hasCar(X)$. Such a literal is the template-literal $hasLoad(+c, -l)$, so we create a literal $hasLoad(X, Y)$ from it and connect it to $hasCar(X)$ which gives us

$$hasCar(X), hasLoad(X, L). \tag{2}$$

Now, we have two options how to extend the partially constructed feature 2. One possibility is to connect another literal to the variable X (because there may be multiple input-occurrences of one variable). The other possibility is to connect a new literal to variable Y. Let us assume that we decided to use the second option. Then we have again two possibilities: to add a literal based on the template-literal $box(+l)$ or to add a literal based on the template-literal $triangle(+l)$. If we select the first possibility then we get the expression:

$$hasCar(X), hasLoad(X, L), box(L). \tag{3}$$

We could continue in this process indefinitely and create larger and larger expressions. However, as has been shown in the paper **RelF 1**, there is only a finite number of non-reducible tree-like features. Moreover, there is usually even less non-reducible and non-redundant features. RelF, HiFi and Poly are able to *search* through all these possible features exhaustively (and efficiently).

C.2 **Ignored arguments (!).** Every variable must appear both as an input and as an output argument. This is not true for variables in so-called *ignored* arguments (marked by mode !). For example when we have the next template

$$\tau_3 = hasCar(-c), hasLoad(+c, !l)$$

then the expression

$$\varphi_7 = hasCar(X), hasLoad(X, Y)$$

is a valid feature despite the fact that the variable Y does appears neither in an input argument nor in an output argument.

C.3 **Constant arguments (#).** Very often, we need not only variables but also constants. Templates can be used to denote which arguments may contain only constants. For example when we have the next template

$$\tau_4 = hasCar(-c), hasLoad(+c, \#const)$$

then one of the possible valid features could be

$$\varphi_8 = hasCar(X), hasLoad(X, load_1).$$

where $load_1$ is a constant. A more meaningful example of a template using constants is shown next:

$$\tau_5 = atom(-a, \#atomType), bond(+a, -b), atom(+b, \#atomType)$$

which specifies features such as:

$$\varphi_9 = atom(X, carbon), bond(X, Y), atom(Y, carbon), bond(X, Z), atom(Z, hydrogen).$$

C.4 **Aggregation modes (*).** The feature-construction algorithm Poly is able to construct multi-variate polynomial relational features. These are polynomial aggregation features which generalize μ-vectors and σ-matrices from Gaussian logic (see **Poly 1**). We need to be able to select which arguments can contain variables that should be used to *extract* the numerical values from the learning examples. We use so-called *aggregation* modes for this. For example the next template:

$$\tau_6 = charge(-a, *chrg), bond(+a, -b), charge(+b, *chrg)$$

defines features which are able to construct multivariate polynomial features involving charges of atoms in molecules such as:

$$\varphi_{10} = charge(X, CH1), bond(X, Y), charge(Y, CH2), bond(X, Z), charge(Z, CH3)$$

which can in turn be used to construct the polynomial aggregation features such as $AVG(CH1 \cdot CH2 \cdot CH3)$, $AVG(CH1^2)$, ...

C.5 **Global constant modes.** Often, relational data is mixed with some attribute-valued data. It is useful to be able to include such attributes to the feature set. For this, it is possible to use so-called *global constants*. For example, parameters LUMO and logP in the Mutagensis dataset are such attribute-valued data. For each molecule, there is precisely one value of LUMO and one value of logP. If you want to add these parameters to the propositionalized data, just add lumo(@lumo) and logp(@logp) to the template.

D. **Any template-literal can contain at most one input-argument.** For example, the next template is not valid

$$\tau_7 = atom(-a, \#atomType), bond(+a, +a)$$

because the literal $bond(+a, +a)$ has two input arguments.

E. **Mode and type declarations must not contain cycles.** There is additional technical requirement on valid templates. Let us define an auxiliary graph. In this graph, we have one vertex for each type (of arguments) contained in the template. There is an edge from a vertex V to a vertex W if and only if there is a literal which contains the type associated to the vertex V in an input argument and the type associated to the vertex W in an output argument. This graph must not contain oriented cycles.

This means that the next template

$$\tau_8 = atom(+a), bond(+a, -a)$$

is not valid because there is a cycle (loop in this case) from a to a. Similarly, the template

$$\tau_9 = atom(-a), bond(+a, -b), bond(+b, -a)$$

is also not valid because there is a cycle $a - b - a$.

F. **Further limiting the features (optional).** It is possible to further limit the set of valid templates.

F.1 **Limiting the number of literals connected to an output argument.** It is possible to limit the number of literals which are connected (through their input-argument) to a given variable. For example, in the next template,

$$\tau_{10} = atom(-a[\mathbf{1}], \#atomType), bond(+a, -b), atom(+b, \#atomType)$$

the expression $-a[\mathbf{1}]$ means that this particular argument can be connected to only one other literal (by sharing the variable in the argument), so

$$\varphi_{11} = atom(X, carbon), bond(X, Y), atom(Y, carbon)$$

is a valid feature because X is contained only in $atom(X, carbon)$ in an output argument and in $bond(X, Y)$ in an input argument. On the other hand, the expression

$$\varphi_{12} = atom(X, carbon), bond(X, Y), atom(Y, carbon), bond(X, Z), atom(Z, hydrogen)$$

is not a valid feature because the variable X is contained in two input arguments (i.e. is connected to more than one literal which contradicts the $-a[\mathbf{1}]$ specification.

F.2 **Limiting the number of literals connected to an input argument.** Analogically, it is possible to limit the number of literals connected to an input-argument, for example, as follows

$$\tau_{11} = atom(-a, \#atomType), bond(+a[\mathbf{1}], -b), atom(+b, \#atomType)$$

.

F.3 **Limiting the number of literals corresponding to the same template-literal connected to an input argument.** Similarly, it is possible to constrain not only the number of literals connected to an input-argument but also the number of literals of given type connected to the given input argument. For example, the next template

$$\begin{aligned}\tau_{12} = \ & hasCar(-c), hasRedLoad[\mathbf{1}](+c, -l), hasBlueLoad(+c, -l),\\ & box(+l), triangle(+l)\end{aligned}$$

allows us to construct only features which contain at most one *hasRedLoad* literal connected to the variable C in a literal $hasCar(C)$, but which may contain any number of *hasBlueLoad* literals connected to C.

5 Command-line Interface with Batch Files

The command line interface of TreeLiker provides full access to all features of TreeLiker (it provides access to some advanced functionalities which are not accessible from TreeLiker-GUI). It allows to set-up more complicated experiments through TreeLiker-batch files.

5.1 Running TreeLiker from Command Line

TreeLiker can be run from command line once we have a TreeLiker-batch file with settings of the experiment. TreeLiker-batch files are described in Section 5.2 in detail. Here, we assume that we already have a TreeLiker-batch file called `experiment.treeliker`. Then, TreeLiker can be run using the following command:

```
java -Xmx1G -cp TreeLiker.jar
   ida.ilp.treeLiker.TreeLikerMain -batch experiment.treeliker
```

5.2 TreeLiker-Batch Files

TreeLiker-batch files specify the data to be processed, algorithms with which they should be processed and the detailed settings of the algorithms. Content of a sample TreeLiker-batch file is shown below:

```
set(algorithm, relf_grounding_counting) % the algorithm
set(output_type, single) % type of output (single = one file)
set(output, 'trains.arff') % where to save the results
set(examples, 'trains.txt') % the learning examples
% the template
set(template, [hasCar(-c), hasLoad(+c,-l), box(+l), triangle(+l)])
work(yes) % tells TreeLiker to run the selected algorithm
   % with the selected parameters
```

The first line `set(algorithm, relf_grounding_counting)` sets the algorithm to be used by TreeLiker. In this case, it is RelF which works in *grounding-counting mode* (as opposed to the *existential mode*).

The second line `set(output_type, single)` sets the type of output. There are three types: 1. `single` which constructs one file using all the examples given in the training data, 2. `cv` which creates the given number (10 by default) of pairs of training and testing `.arff` (WEKA) files which can be used to perform cross-validation, 3. `train_test` which creates two files from the given training and testing data. These options are discussed in detail later in this document.

The third line `set(output, 'trains.arff')` sets the output file in which the constructed relational features and the propositionalized table should be stored. TreeLiker uses `.arff` file format which is used in WEKA.

The fourth line `set(examples, 'trains.txt')` sets path to the training examples which should be used. The training examples should be given in pseudoprolog file format (described in Section 2).

The fifth line `set(template, [hasCar(-c), hasLoad(+c,-l), box(+l), triangle(+l)])` specifies the template which should be used to constrain the space of possible features.

Finally, the sixth line `work(yes)` tells TreeLiker to run the selected feature construction algorithm.

If the above TreeLiker-batch file is run on the following file of training examples (`trains.txt`):

```
east hasCar(c), hasLoad(c,l), box(l)
west hasCar(c), hasLoad(c,l), triangle(l), hasLoad(c,l2), box(l2)
```

then it outputs the following `.arff` file:

```
@relation propositionalization
@attribute 'hasCar(A), hasLoad(A, B), box(B)' NUMERIC
@attribute 'hasCar(A), hasLoad(A, B), triangle(B)' NUMERIC
@attribute 'classification' 'east','west'

@data
'1', '1', 'west'
'1', '0', 'east'
```

It contains just two generated features because the other possible features were discarded using the redundancy-pruning mechanisms. However, we can see that already these features are enough to split the examples to the respective classes.

It is possible to perform more individual *experiments* using just one batch file, e.g. with different templates:

```
set(algorithm, relf_grounding_counting) % the algorithm
set(output_type, single) % type of output (single = one file)
set(output, 'trains.arff') % where to save the results
set(examples, 'trains.txt') % the learning examples
% the template
set(template, [hasCar(-c), hasLoad(+c,-l), box(+l), triangle(+l)])
work(yes) % tells TreeLiker to run the selected algorithm
% Another template
set(template, [hasCar(-c[1]), hasLoad(+c,-l[1]), box(+l), triangle(+l)])
set(output, 'trains.arff') % where to save the second set of results
work(yes) % tells TreeLiker to run the selected algorithm
```

5.3 TreeLiker-Batch File Commands

In this section, the commands which can be set in TreeLiker are described.

A. **Command** `set(key, value)`. This command is used to set many parameters of TreeLiker. These are described in a separate section.

B. **Command** `print(message)`. This is a simple command which just prints some text to console. It can be useful when you want to keep track of progress of some more complex experiment (e.g. involving generation of multiple sets of features).

C. **Command** `work(yes)`. This command starts executing the currently set algorithm with the current settings.

5.4 Parameters which Can Be Set

In this section, values which can be set using the command `set(key, value)` are described.

A. **Training examples** - e.g. `set(examples, 'trains.txt')`. The examples should be in the psudoprolog format.

B. **Output type**. The output type can be one of the following: `single`, `cv` and `train_test`. It can be set e.g. as `set(output_type, single)` or `set(output_type, cv)` or `set(output_type, train_test)`.

   B.1 Output type `single`. Only one `.arff` file is created. The features are constructed using all examples given as training data. The resulting `.arff` file contains as many examples as the input dataset. **Warning:** *RelF uses aggressive filtering of features which are redundant taking into account class-labels of the examples. So, performing cross-validation using the file generated by RelF with output-type* `single` *could give overoptimistic results. It is better to use files generated with* `output_type` *set to* `cv` *for such purposes.*

   B.2 Output type `cv`. The given number of pairs of train and test files (for stratified cross-validation) is constructed (this can be set using the parameter `num_folds`). The names of the generated files are `train_i.arff` and `test_i.arff` where `i` goes from 1 to `num_folds`. The train and test files are always created in the following way which guarantees that there will be no leakage of information from the testing set to the training set: the features are built using the selected algorithm on the training data, then they are post-filtered also using only the training data and then the respective `train_i.arff` file is created. The respective `test_i.arff` file is then constructed by evaluating the features constructed on the training data on the testing data which gives rise to the propositionalized table stored in the file `test_i.arff`.

   B.3 Output type `train_test`. Two files are constructed: `train.arff` and `test.arff`. It is necessary to set which examples should be in the train-set and which examples should be in the test-set. This can be done in two different ways: 1. It is possible to have all examples in one dataset which is set using the parameter `examples` and to indicate which examples should go to the training set and which examples should go to the test-set using: `set(train_set, [0,1,2,5])` and `set(test_set, [3,4,6])` which sets the indices of the examples from the given dataset which should go to the train-set and test-set, respectively (in this case the examples 0, 1, 2, 5 should go to train-set and the examples 3, 4, 6 to the test-set). 2. An alternative way is to have the examples in separate files and set them as follows: `set(train_set, 'trains_train_set.txt')` and `set(test_set, 'trains_test_set.txt')`.

   The train and test `.arff` files are always created in the following way which guarantees that there will be no leakage of information from the testing set to the training set: the features are built using the selected algorithm on the training data, then they are post-filtered also using only the training data and then the respective `train.arff` file is created. The respective `test.arff` file is then constructed by evaluating the features constructed on the training data on the testing data.

C. **Output.** Where the results should be stored is set using `set(output, 'my_file.arff')`. When output-type is `single` then the selected output should be a file. When output-type is `cv` or `train_test` then the selected output should be a directory (in which the train and test files should be stored).

D. **Algorithms.** The algorithm which should be used can be set as follows (this is a complete list of algorithms in this version of TreeLiker):

```
set(algorithm, hifi)
set(algorithm, relf)
set(algorithm, poly)
set(algorithm, hifi_grounding_counting)
set(algorithm, relf_grounding_counting)
set(algorithm, poly_grounding_counting)
```

E. **Minimum frequency.** Although the algorithms RelF, HiFi and Poly are able to construct features even without setting the minimum frequency constraint, it may be sometimes useful to set some minimum frequency which can be done using `set(minimum_frequency, 0.2)` (in this case, we are setting the minimum frequency to 20%). For RelF, frequency is applied separately for examples from different classes.

F. **Covered class** (applies to RelF but not to HiFi and Poly). It is possible to set one class which RelF will try to cover - i.e. it will try to search for features which cover examples from this given class but not many examples from the other classes - for proper definition of a covered class and what it means to find class-non-redundant features, see the paper about RelF. Covered class can be set e.g. using `set(covered_class, east)` (where `east` is just a class to be covered from the trains-example that we use in this tutorial).

G. **Maximum size of features** (applies to HiFi and Poly but not to RelF). It is possible to set maximum size of features (i.e. the maximum number of literals in them) which can both speed-up the feature construction process and make the number of generated features smaller. It can be set e.g. using `set(maximum_size, 10)` (which sets the maximum size of features to 10).

H. **Maximum degree of monomials** (applies only to Poly). It is possible (and advisable) to set the maximum allowed degree of multivariate aggregation polynomials. This also implicitly restricts the number of aggregation variables in a feature (those used to construct the multivariate polynomial aggregation features) which follows from decomposability property of polynomial aggregation features. This can be done using `set(max_degree, 3)` (which sets the maximum degree of the polynomials to 3).

J. **Sampling-based variants of the algorithms.** Sometimes, it may be unnecessary to construct sets of features which are non-redundant for the complete dataset, it may be sufficient to construct a non-redundant set of features for a subset of the original dataset. The algorithms RelF, HiFi and Poly can be used also in so-called *sampling mode* in which the following process is repeated for the given number of times: sub-sample the dataset creating a smaller dataset, find a non-redundant set of features for this dataset and store the features. After a sufficient number of repeats of this process, the features are collected and evaluated w.r.t. the complete dataset. The set of features obtained in this way may often have good quality. The *sampling-mode* is turned on by `set(use_sampling, true)`. The number of repeats of the sampling and feature construction is set by `set(num_samples, 10)` (here, we set it to 10). Finally, the number of examples from each class which should be contained in the samples can be set using `set(sample_size, 5)` (here, we set it to 5). Sampling can be combined e.g. with minimum frequency constraint which can further reduce the runtime and number of generated features. Sampling can be used in conjunction with any of the algorithms RelF, HiFi or Poly.