User Manual of WILL 0.1

Shavlik Group

University of Wisconsin-Madison
http://www.cs.wisc.edu/machine-learning/shavlik-group/

September 14, 2010

Contents

1 Overview **1**

2 The WILL algorithm **1**

3 Getting started with WILL **2**
3.1 Data files naming convention and format 2
3.1.1 Background knowledge file 2
3.1.2 Positive example file *prefix*_pos.txt 3
3.1.3 Negative example file 3
3.1.4 Additional fact file 3
3.1.5 Advice file 4
3.2 Running WILL 4

4 Interpreting the results **5**

5 WILL Features **6**

6 Command line options **7**

7 Accessing advanced options **9**

8 Bibliography **10**

1 Overview

WILL (Wisconsin Inductive Logic Learner) is an ILP engine. Inductive Logic Programming (ILP) induces general rules from specific facts and suggested rule structures.

WILL is based heavily on the Aleph implementation of ILP. [4] The Aleph implementation works by starting with the most specific clause (called the bottom clause) and iteratively evaluating more inclusive versions of this clause, selecting the best version at each step. The best version is the one which most accurately covers the positive examples provided while not covering the provided negative examples.

2 The WILL algorithm

The WILL ILP algorithm implementation is similar to that of Aleph, with some variations. The basic algorithm is illustrated in Algorithm 1.

Algorithm 1 BASIC WILL ALGORITHM

```
1: while not stoppingCondition do // This is the "outer loop"
2:    Select an example e from exampleSet
3:    Find best covering clause based on e // This operation is the "inner loop"
4:    Add best covering clause to the final theory
5:    Remove covered examples from exampleSet
6: Return the final theory
```

The stoppingCondition might be that a certain number of examples are covered by the theory, or that the theory has reached a maximum number of clauses.

There are some possible variations on what constitutes the "best covering clause", but it is generally the most general clause that covers some minimum percent of positive examples while excluding some percent of negative examples.

3 Getting started with WILL

Before you begin, you will need:

- The WILL jar file
- Data files

3.1 Data files naming convention and format

The data files have a very specific naming convention. They must exist in a directory with some name *prefix*, and the files must be named:

- *prefix*_bk.txt, which contains background knowledge
- *prefix*_pos.txt, which contains positive examples
- *prefix*_neg.txt, which contains negative examples
- *prefix*_facts.txt, which contains additional facts
- *prefix*_bkRel.txt which contains additional advice (optional file)

Example data files are provided in the `SampleTestbeds` directory. This document uses the "trains" testbed.

Java-style `//` comments can be used to add comments to the ends of lines in the data files files.

3.1.1 Background knowledge file (*prefix*_bk.txt)

The background knowledge file is structured similarly to that used in Aleph. It consists mostly of mode declarations and "known" rules we give to the system.

The following examples are based on a simple data set, where the goal is to devise a theory that can be used to identify eastbound trains. [2] Mode declarations specify legal formats for predicate heads generated by WILL. A mode declaration might look like:

```
mode: eastbound(+train).
```

Such a rule tells WILL that it is legal to produce a rule such as:

```
eastbound(A) :- has_car(A,B), wheels(B,3), has_car(A,C), load(C,triangle,1).
```

Additional syntax can be used to limit the variable bindings. The following rule states that the second variable in the wheels predicate must be bound to a constant of type `int`.

```
mode: wheels( +car, #int) maxPerInputVars=1.
```

I'm not entirely clear on the meaning of maxPerInputVars, but here is what was in the FileParser.java comments:

- **Optionally [not yet implemented] can say that the above mode only applies when learning this target. A sample is 'parentOf/2' (the literal whose predicate name is 'parentOf' and which has two arguments).**
- **Optionally say that typed literal can appear in a learned clauses at most (some integer) times.**
- **Optionally indicate that PER SETTING to the 'input' (i.e. '+') variables, can occur at most this many times (an idea taken from Aleph).**

Are all of these/one of these true?

In the following rule the minus sign designates car as in input variable (that must be bound to a value before the rule is called) and train as an output variable.

```
mode: has_car(+train, -car).
```

Additional background knowledge usually describes facts that would be known by anyone familiar with that domain. For example, in the NFL domain, background knowledge might specify that the team with the higher score in a game is the same as the game winner.

More detailed information about the background knowledge file can be found in the Aleph manual, "Background Knowledge File" section. [5]

3.1.2 Positive example file (*prefix*_pos.txt)

The positive example file gives a list of positive examples. These predicates are true. For example, The following statement means "Train east5 is an eastbound train". A number of these statements would appear in the positive example file.

```
eastbound(east5).
```

3.1.3 Negative example file (*prefix*_neg.txt)

The negative example file gives a list of negative examples. These predicates are false. For example, the following statement means "Train west7 is an eastbound train".

```
eastbound(west7).
```

3.1.4 Additional fact file (*prefix*_facts.txt)

The fact file contains facts about the data set. These facts generally would not appear in the background knowledge file because they are specific to the data set and not generalizable to the entired data domain. The fact file may also contain typing predicates which define the types of constants in the data set. The following statements mean that east1 is of type train, $\mathrm{car}_1 1 is of type train, and rectangle is of type shape.$

```
train(east1).
car(car_11).
shape(rectangle).
```

The fact file also contains statements that give details about the data set that will be provided by WILL. WILL will use these details when constructing a theory. The following statements mean that:

- Car car_11 is long.
- Car car_11 is open.
- Car car_11 is shaped like a rectangle.
- Car car_11 is loaded with 3 rectangles.
- Car car_11 has 3 wheels.
- Train train east1 has car_11 as one of its cars.

```
long(car_11).
open_car(car_11).
shape(car_11,rectangle).
load(car_11,rectangle,3).
wheels(car_11,2).
has_car(east1,car_11).
```

3.1.5 Advice file (*prefix*_bkRel.txt)

This file is optional. If the -relevance switch is used while running WILL, WILL will attempt to read relevance advice from this file.

For more detail see publication?

3.2 Running WILL

Note to UW-Madison developers: WILL.jar is built by an ant build.xml file in the bootstrap project root directory. The resulting WILL.jar file will be created in the dist directory and copied to the project root directory.

WILL requires one argument: the directory containing the data files.

You may wish to add additional arguments to govern the memory used by the Java Virtual Machine. Memory consumed by the testbeds varies according to the size of the data. If running ILPMain results in an out-of-memory error, arguments -Xms and -Xmx can be used to allocate a minimum and maximum amount of memory to the virtual machine, respectively.

The following example runs WILL on a `trains` testbed located in the SampleTestbeds/trains directory.

```
user@ machine:~/Workspaces/MachineReading/bootstrap$ java -jar WILL.jar \
SampleTestbeds/trains
```

The following example requires that WILL consumes between 2 GB and 6 GB of memory.

```
user@ machine:~/Workspaces/MachineReading/bootstrap$ java -Xms2G -Xms6G \
-jar WILL.jar SampleTestbeds/trains
```

4 Interpreting the results

The results of the run of WILL will be written to the screen and to the "dribble" file. The dribble file is a detailed log of the run, and as WILL is a research tool under development, much of this file may not be relevant to all WILL users. For most users, the most interesting part of the dribble file will appear at the end.

The following is an abbreviated sample of a dribble file, showing only the end of the file.

```
% ------------------------------------------------


% Best Theory Chosen by the Onion:

%     Clauses:

    eastbound(A) :-
       has_car(A, B),
       load(B, circle, 2). // Clause #1.

    eastbound(A) :-
       has_car(A, B),
       closed(B),
       short(B). // Clause #2.

    eastbound(A) :-
       has_car(A, B),
       load(B, rectangle, 3). // Clause #3.



%              Actual
%             Pos   Neg Total
% Model Pos     5     0     5
%       Neg     0     5     5
%     Total     5     5

% False Pos mEst  = 0.0100
% False Neg mEst  = 0.0100

% Accuracy  = 0.9980
% Precision = 0.9980
% Recall    = 0.9980
% F(1)      = 0.9980


% Chosen Parameter Settings:
%   maxNumberOfCycles  = 6
%   maxNumberOfClauses = 3
%   maxBodyLength      = 3
%   maxNodesToCreate   = 105
%   maxNodesToConsider = 10
%   minNumberOfNegExamples = 1
```

```
%   minPosCoverage     = 0.1350
%   maxNegCoverage     = -1.0000
%   minPrecision       = 0.9450
%   mEstimatePos       = 0.0100
%   mEstimateNeg       = 0.0100
%   minimum strength   = null
%   map mode '*' to '-'
%   modes in use: [eastbound, short, closed, long, open_car, double,
jagged, shape, load, wheels, has_car, addList, multiplyList, abs,
minus, plus, mult, div, allNumbers, positiveNumber, negativeNumber,
in0toDot001, in0toDot01, in0toDot1, in0to1, in0to10, in0to100,
in0to1000, equalWithTolerance, greaterOrEqualDifference,
smallerOrEqualDifference, isaEqualTolerance, lessThan, greaterThan,
lessThanOrEqual, greaterThanOrEqual, inBetweenOO, inBetweenCO,
inBetweenOC, inBetweenCC, memberOfList, firstInList, restOfList,
positionInList, nthInList, lengthOfList
%   all modes:    [eastbound, short, closed, long, open_car, double,
jagged, shape, load, wheels, has_car, addList, multiplyList, abs,
minus, plus, mult, div, allNumbers, positiveNumber, negativeNumber,
in0toDot001, in0toDot01, in0toDot1, in0to1, in0to10, in0to100,
in0to1000, equalWithTolerance, greaterOrEqualDifference,
smallerOrEqualDifference, isaEqualTolerance, lessThan, greaterThan,
lessThanOrEqual, greaterThanOrEqual, inBetweenOO, inBetweenCO,
inBetweenOC, inBetweenCC, memberOfList, firstInList, restOfList,
positionInList, nthInList, lengthOfList]

% ------------------------------------------------

% Took 4.150 seconds.
% Executed 46,807 proofs in 0.67 seconds (69444.31 proofs/sec).
% Performed 212,868 unifications while proving Horn clauses.
```

The first part of this excerpt shows the "best theory" chosen by WILL. The theory consists of three clauses, and in this simple example, they cover 100% of the examples and achieve 100% accuracy.

- **How did the results arrive at .998 accuracy when we know accuracy is 100%?**
- **What are False Pos mEst and False Neg mEst? stopping conditions?**
- **The other output seems self-explanatory to me. Is that reasonable?**
- **Lots of the "modes in use" don't appear in the theories. Should we explain this, i.e. these are just predicate heads that exist somewhere in the data files?**

5 WILL Features

This section is lifted from ILPouterLooper.java header comments.

WILL includes a number of features specific to the WILL ILP implementation which may be unfamiliar or slightly different from what appears in the Aleph ILP implementation. [3]

1. Positive and negative examples can be weighted.
2. Both standard ILP heuristic search and Rapid Random Restart (RRR) can be performed.

3. A GLEANER is used to keep not only the best rule per iteration, but also the best rule per interval of recall (i.e. the best rule whose recall is between 0.50 and 0.55.) [1]

4. On each cycle, multiple seed examples can be specified. A node (i.e. a possible clause) needs to satisfy some fraction of the positive seeds and no more than some fraction of the negative seeds.

5. During successive iterations of the outer ILP loop,the positive examples can be down-weighted and the covered negative examples can be up-weighted. This is similar to boosting, although the current version does not compute boosting algorithm weights.

6. In the inner loop, each node records the examples rejected at this node. This means that on any path to a node, examples are stored at more than one node.

7. Java HashMaps are used for efficiency, as opposed to list structures which require linear time lookups.

8. WILL does not construct a "bottom clause" like Aleph. Instead, WILL uses multiple seeds to guide the search of candidate clauses.

9. No Prolog is required by WILL. Everything needed is provided in the Java code in WILL.jar.

10. As in Aleph and other ILP systems, arguments are typed to help control search. i.e. 'human' and 'dog' would be different types, and would not be interchangable arguments in a clause. In this code, the typing is hierarchical.

11. As in Aleph, users are able to define prune(node) by which the user can cut off search trees. Related to this is the built-in capability to process "intervals" such as isInThisInterval(1,value,5). If the previous literal is already in a clause, there is no need to add isInThisInterval(2, value, 4), since the latter will always be true given the former. Similarly, there is no need to include isInThisInterval(7, value, 9) since it will always be false.

6 Command line options

The following options can be added to the Run Configuration command line options in Eclipse, or at the command line. Note that command line options begin with a dash and use an equals sign to specify parameters, unless stated otherwise.

-rrr
RRR stands for random rapid restart. This causes the solution search to take random jumps through the search space and restart the search.

The -rrr option enables random rapid restart. Default is disabled.

-std
Disables random rapid restart.

-yes
Enables random rapid restart. (Same as -rrr.)

-no
Disables random rapid restart. (same as -std.)

-flip
This causes ILPMain to treat the negative examples as positive examples and vice versa. Default is not flipped.

-prefix=*prefix*
Use this option to force ILPMain to look for data files with a different filename prefix from the directory name. For example: `java -jar WILL.jar SampleTestbeds/chess -prefix=muggleChess` causes ILPMain to use the data files muggleChess_pos.txt, muggleChess_neg.txt, muggleChess_bk.txt, and muggleChess_facts.txt instead of chess_pos.txt, chess_neg.txt, chess_bk.txt and chess_facts.txt.

n
Number of folds used by ILPMain. Default is 1.

-folds=*n*
same as *n*

-fold=*n*
Specifies what fold number to start with. Default is 0, which is the first fold. Example: `java -jar WILL.jar SampleTestbeds/chess -folds=5 -fold=0` Causes ILPMain to use 5 folds starting with fold 0. (Folds are numbered from 0, not 1.)

-checkpoint
Enables checkpointing. If enabled, this option forces ILPMain to write to a gleaner file (summary result file) periodically. Default is disabled.

-relevance
Enables relevance. If enabled, ILPMain will attempt to find an additional data file, of the form *prefix*_bkRel.txt in the data file directory. This file will contain "advice" about relevant and irrelevant examples. These examples add predicates and/or modes to the ILP search space. Relevance advice can include ground advice (i.e., this example is true because x y), feature advice (i.e., predicate q is relevant to the concept), and statement advice (i.e., literal q(x,y) should be included in the search space). Default is disabled.

-norelevance
Disables relevance

-maxTime=*n*
The program will be terminated after *n* seconds. If this argument is omitted, the maxTime argument defaults to three days.

-useOnion
Use the *Onion* algorithm. The Onion is basically a parameter tuning mechanism. It is similar to a grid search in which parameter settings are varied over multiple runs to find the best parameter setting. The difference between the Onion and a standard grid search is that a grid search would evaluate all parameter settings to find the best ones where as the Onion will stop early if a set of parameters results in a valid solution according to a set of criteria. It uses a cross-validation loop to evaluate the parameters.

The evaluated parameters of the Onion are ordered such that each layer of the Onion results in a larger search space or relaxed stopping criteria. Thus the Onion can also be seen as a type of iterative-depth search. However, in addition to increasing the maximum depth at each layer, it also changes other parameters.

Default is don't use.

-onion
Same as -useOnion.

-noOnion
Don't use the *Onion* algortihm.

***extension* (with no dash)**
Specifies a different file extension for data files. Default is .txt.

7 Accessing advanced options

Some additional options not available from the command line may be changed directly in the source code within `src/edu/wisc/cs/ILP/ILPouterLoop.java`.

Should this section be expanded or deleted?

8 Bibliography

References

[1] M. Goadrich, L. Oliphant, and J. Shavlik. Gleaner: Creating ensembles of first-order clauses to improve recall-precision curves. *Machine Learning*, 64(1/2/3):231–262, 2006.

[2] Ryszard S. Michalski. Pattern recognition as rule-guided inductive inference. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-2(4):349 –361, jul. 1980. The origin of the trains example.

[3] J. Shavlik. Ilpouterloop.java. These comments appear in the ILPouterLoop source code.

[4] A. Srinivasan. The Aleph manual. `http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html`.

[5] A. Srinivasan. Background knowledge file. `http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html#SEC7`. The Aleph Manual.