Natural Language Understanding (NLU) Advanced Training

Anthonio Pettit - June 29, 2017

Useful SQL Calls for identifying Command Thefts

*Theft: A simple ad hoc term for the wrong command is missing. One thing is supposed to be winning, but it's being stolen by another command. Thefts will appear as "**Fail: N-Match**" in the results databases.*

- New views are available SQLite results by **Agent** level and **Command** Level. The results will show the target command and the theft command which ended up winning.
- There are several queries to figure out which commands have changed between two runs so that if a command matched 97% we can find out why.
- There is a SQL Query in SVN which track how many thefts are occurring with a particular criteria handler for example.
- It starts with the previous run's database and then figures out what the change in thefts between runs (the delta)
- The accuracy is then calculated for a particular command based on the frequency of theft occurrence.
- To save on processing resources it might be good to have the query run once and shared among the team as it will be intensive

How it works:

The query is not able to track the priority of the rule; it simply notices that a theft is occurring with a particular command and tracks that.

Priorities must be at 100% for P0; 96% accuracy for P1 & 2

If targeting one particular theft, this can be specified by Agents

If no thefts are occurring the query will return no results; a complementary query is therefore appended to return all commands affected.

When an evaluator result comes back, it will find the appropriate baseline database and exports a CSV of everything that is going wrong. This means the results don't need to be viewed in SQLite. This will be implemented eventually, but not yet ready.

These queries are command based; there is no way to see what has changed based on concepts however.

There is a way to query a specific criteria to find out which commands are making calls to it.

Change "*List*", "*of*", "*Agents*" to "*List*" "*of*" "*Commands*" and then specify criteria in ("*your*", "*handlers*") – this will be processing intensive.

Another strategy is that the Thefts column can be queried by an agent to see which of our domains are being the thieves. Sometimes our agents are the winning agents which can be causing problems somewhere else.

Follow-up RECO States

Reco States are built to specify the state of the machine/device prior to running validation testing in both the Evaluator and Console App.

If a rule is conditional, and will only match if the device is in a state, Reco states specify the necessary environment to validate those commands.

Sometimes it is necessary to check whether a given Reco State exists.

A Reco State node lives In the ACE project tree beneath all of the agents, criteria handlers and tables.

Daughter nodes list all of the available Reco states including Device Reco State, VBTest RecoState etc

These Reco states are used by the Evaluator app and the Console app, and both tols are now polling the Device Reco State node and thus the tools will have access to all available contexts which are listed in these nodes.

All of the Agent Contexts can generally be found under a given Reco state, and should include all of the the new CP commands as well.

The Device Reco State will have all the contexts available to the Evaluator and Console App.

Note that If a given Reco state is disabled, it will not be possible to test results in the Console or Evaluator

The Agent Team will need to enable any agents in the Device Recostate node in order to include these in verification and testing .

It is not necessary to reference a specific context in a Pseudo CARD; simply stating the Agent in the string per usual will automatically pull the context for an agent if it is enabled.