Concretely Annotated Corpora API, Utilities, and Data Format

Schema

The latest Concrete schema is available at http://hltcoe.github.io/concrete/schema/. The tutorial referenced below (http://hltcoe.github.io/concrete/) provides in-depth information about the schema.

Quick-Starts

Docker

A docker image containing the latest concrete, and Java and Python libraries can be found on Dockerhub. Run

```
$ docker pull hltcoe/concrete
$ docker run -i -t hltcoe/concrete:latest /bin/bash
#
```

Tutorial

Both basic and in-depth information regarding Concrete can be found at http://hltcoe.github.io/concrete/. This includes getting-started guides for both Python and Java.

Command-line Utilities

concrete-python contains a number of useful command line tools for viewing and working with Communcations and archives of Communications. These tools are included in the above docker image.

The main concrete-python page details the utilities, but here is a quick summary of the most relevant ones:

- concrete-inspect.py: This script lets you quickly inspect a Communication's contents from the command line. For instance, this can produce CoNLL-style output for named entities, part of speech tags, and parses.
- concrete2json.py: This pretty-prints a JSON version of the provided Communication to stdout.
- create-comm.py, create-comm-tarball.py: These provide a very quick and easy way to create Communications. They read in text files and whitespace segment them into Sections and Sentences.
- validate-communication.py: This script prints out information about any invalid fields within a provided Concrete Communication. The core functionality is contained in the `concrete.validate` library, and can be used programmatically.

Browser Visualization

Quicklime is a utility for viewing a Communication within your local web browser.

It is Python-based: either install with pip:

```
pip install quicklime
```

or run the latest docker image: https://hub.docker.com/r/hltcoe/quicklime/.

Once installed, given a path to Communication(s), run

```
qlook.py /path/to/communications
```

and point your browser to `http://localhost:8080`. The path can be either a single Communication file, a tar.gz archive, a zip archive, or a directory of Communication files (with extensions `.comm` or `.concrete`).

Data Format

Capturing Document Structure

A Communication is the primary document model. A Communication's full document text is stored in the `Communication.text` string field; a Communication's `id` may be a headline, URL, or some other identifying/characterizing feature. Communications have Sections, which themselves have Sentences. A Sentence has a Tokenization, which is where DependencyParses, Constituent Parses, and other sentence-level (syntactic) structures are stored. Token-level annotations, like part of speech and named entity labels, are stored as TokenTaggings within a Tokenization. All of these structures and annotation objects have a unique identifier (UUID). UUIDs act as pointers: they allow annotations to be cross-referenced with others.

Global Annotations

Semantic, discourse and coreference annotations can cut across different sentences. Therefore, they are stored at the Communication level. Semantic and discourse annotations, like frame semantic parses, are stored as SituationMentions within SituationMentionSets, while individual *mentions* of entities are stored as EntityMentions within EntityMentionSets. While EntityMentions and SituationMentions both can ground out in specific tokens (using UUIDs to cross-reference), SituationMentions can ground out in EntityMentions or, recursively, other SituationMentions. If coreference decisions are made, then individual mentions can be clustered together into either SituationSets or EntitySets.

UUID Format

UUIDs, or universally unique identifiers, are 128-bit numbers represented as 32 hexadecimal numbers split in five chunks:

```
xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.
```

Though UUIDs have an official specification regarding how certain hex digits are regenerated, the UUIDs contained in these corpora have used a specification different from the official one. This was done for space considerations. Specifically, the UUIDs here follow the pattern

```
xxxxxxxx-xxxx-yyyy-yyyy-zzzzzzzzzzzz
```

where the first 12 numbers (the "x"s) are randomly generated per Communication; the next 8 numbers ("y"s), given a Communication, are randomly generated for each invocation of an analytic; and the final 12 (the "z"s) are used to distinguish each UUID an analytic produces.

The above scheme decreased compressed storage by roughly 1/3 and it did not introduce duplicate UUIDs.