DATA FORMAT

andreacondorelli edited this page on 7 Sep 2015 · 10 revisions

Data model

The data container contains all datasets available in the reference framework. A dataset will be represented by a multi-graph a set of nodes "*entities*" and edges "*relations* The multi-graph structure allows the efficient representation of any relation between entities. The data model supports complex annotations for nodes and edges stored as JSON objects. In order to support temporal changes (e.g, in stream-based scenarios), every node and edges is annotated with a timestamp.

Entities

An *entity* will have 5 attributes:

- a `type` (e.g., movie, book, person), stored as a `string`.
- an `ID`, stored as a `string`.
-

Pages 11

- **Idomaar**
  - Architecture
  - Evaluation process
  - Data format
  - HTTP Interface
- **Evaluation**
  - [How to use eval.py] (https://github.com/crowdrec/idomaar/wiki/How-to-use-eval.py)

a timestamp, stored as a long value based on the Unix epoch time. The timestamp defines when the entity has been created.

- a set of `properties` (e.g., the `genre` of an movie: `comedy`), stored as a JSON-formatted string
- a set of `linked-entities` (e.g., the actor of the movie: the `person` entity "Tom Cruise"), stored as a JSON-formatted string. The linked-entities typically consist of a list of entities defined by the entity type and the entity ID.

Relations

A *relation* will have 5 attributes:

- a `type` of the relation (e.g., has-rated), stored as a `string`.
- an `ID`, stored as a `string`.
- a timestamp, stored as a long value based on the Unix epoch time. The timestamp defines when the relation has been created.
- a set of `properties` (e.g, the rating values the user has assigned to an item), stored as a JSON-formatted string. JSON is to be formatted according to the Standard JSON rules, using double quotes for strings
- a set of `linked-entities` (e.g., the user and the movie that are connected by the rating edge), stored as a JSON-formatted string. The linked-entities typically consist of a list of entities defined by the entity type and the entity ID.

Remarks

Every `entity` and every `relation` must have a `type` and an `ID`. `Timestamp`, `properties`, and `linked-entities` are optional attributes. Missing values are indicates by empty fields (empty strings).

Data file structure

The entities and relations are stored in a 5 column tab separated value file (TSV). `entityType <TAB> entityID <TAB> timestamp <TAB> properties <TAB> linked-entities`
`relationType <TAB> relationID <TAB> timestamp <TAB> properties <TAB> linked-entities`

The end of file is marked with a line containing at least the first column set with "EOF".
`EOF <TAB> 0 <TAB> 0 <TAB> {} <TAB> {}`

example data file

We define three entities. The entities have the type `person`. The properties `name` and `gender` are defined in the 4th column. The column `linked-entities` is empty. `person <TAB> 3001<TAB> <TAB> {gender:"male",name:"Travolta, John"} <TAB>  person <TAB> 3004<TAB> <TAB> {gender:"male",name:"Jackson, Samuel"} <TAB>  person <TAB> 3003<TAB> <TAB> {gender:"male",name="Tarantino, Quentin"}`

We define a movie related to the previously defined persons.
`movie<TAB>2202<TAB>129121892189<TAB>{title:"Pulp Fiction",year:"1994"}<TAB>{actors:["person:3001","person:3004"],director:"person:3003"}`

We define a user. `user<TAB>1002<TAB>129121892189<TAB> {twitterId:"177651718",gender:"female",city:"Barcelona"} <TAB>`

We define that the user has explicitly rated the movie. `rating.explicit <TAB> 1001 <TAB> 129121892189 <TAB> {rating:5} <TAB> {subject:"user:1002",object:"movie:2202"}`

Output format

The recommender algorithms should provide the recommendations in the TSV/JSON format that is similar to the format used for the input data. The recommendations for each request are stored in one line. Each line consists of 6 columns, separated by a character.

The columns in output files

- *subject_etype*, a `string` , e.g., `user`
- *subject_eid*, a `string` , e.g., `1001`
- *request_timestamp*, a `long` value based on the Unix epoch time format, e.g., `1404910899`
- *request_properties*, a JSON-formatted `string` , e.g., `{"device":["smartphone", "android"], "location":"home"}`
- *recomm_properties*, a JSON-formatted `string` , e.g., `{ "explanation":"suggested by your close friends"}`
- *response_time*, a `long` value
- *linked_entities*, the predicted recommendations might annotated with a score, a JSON-formatted `string` , e.g., `[{"id":"movie:2001","rating":3.8,"rank":3}, {"id":"movie:2002","rating":4.3,"rank":1}, {"id":"movie:2003","rating":4,"rank":2,"explanation":{"reason":"you like","entity":"movie:2004"}}]`

Output for evaluation

The groundtruth is a list of evidences with this format:

```
 "evidences":[{"evidence":{"type":"rating","value":3}, "subject": {"type": "user",
"id": 27},...]
```

The eval.py script needs as input something like this:

```
 recommendation recID ts {"reclen":5, "expected": {"evidences":[{"evidence":
{"type":"rating","value":3}, "subject": {"type": "user", "id": 27}, "object": {"type":
"movie", "id": "1024648"}},{"evidence":{"type":"purchase","value":1}, "subject":
{"type": "user", "id": 27}, "object": {"type": "movie", "id": "1623205"}}]}}
reccomendation_Time {} [{"id":"1702439","rating":"7.5554733","rank":"1"},
{"id":"1623205","rating":"7.3246436","rank":"2"},
{"id":"1024648","rating":"7.1299243","rank":"3"},
{"id":"1351685","rating":"6.6666665","rank":"4"},
{"id":"1371111","rating":"6.464102","rank":"5"}]
```

In order to minimize the overhead when benchmarking recommender algorithms, the output format might simplified to match the interface of the evaluator (e.g. RiVal project)

Typical scenarios

Static datasets

If the data model is used for representing a static dataset do not providing information about timestamps, the 3rd column in the TSV file will be empty.

Streams

In stream-based scenarios the timestamp is important for re-playing recorded streams. Entities or relations do not have a timestamp are known at any time. That means that these entities have been created before the start of the stream.

Importing the data files

Since the file structure relies on tab-separated values (TSV/CSV) and JSON standard parsers can be used for reading the data files. In order to import the data with `JAVA` the Apache Commons CSV project can be used.

Limitations:

- In order to ensure that the columns in the file can be properly separated, tab-characters in text fields must be protected/escaped. Most csv parsers support `quoting` and `escaping`.
- LF and CR characters cannot be used, since a line-based data-format is used