Score

Overview

Depending on your data the next step may be to match incoming data with data already in VIVO. For example, if you have just pulled in some publication information from Pubmed, you might want to compare the author names with people in your VIVO, so that you can link the publications with the authors. This comparison is done via the Score tool, which compares any values you want between VIVO and the input data, and assigns a number to the comparison.Score.java provides a method used to score incoming data. Data is assumed to be in a VIVO-like ontology and stored in a Jena model. Method can call any combination of scoring algorithms. Scoring function will attempt to match data to individuals in VIVO. Several algorithms can be utilized to determine when the match will be inserted into VIVO. Data produced by this method is stored in a separate scoring model which then is required by the Match to do the data changes.

A Score run involves several concepts:

- The **namespace** in the input model of data to score. This allows different Score runs to be performed for different types of data, for example to score authors, publications, and journals separately.
- The **URI** on which to compare an individual in the input model to an individual in VIVO. For example,

  ```
  http://xmlns.com/foaf/0.1/firstName
  ```

  to compare authors by their first names.
- The **algorithm** with which to run the comparison. An algorithm takes two strings and returns a floating-point number between 0.0 and 1.0. A 0.0 indicates complete rejection, while a 1.0 indicates a complete match. For example, the *equality test* algorithm takes the two strings and determines whether they are precisely the same string. If so, it returns 1.0; if not, it returns 0.0. Other algorithms, such as *Levenshtein distance*, perform a more thorough comparison of the strings and can return values in-between one and zero inclusively.
- The **weight** of the particular comparison. This is typically a number between 0.0 and 1.0 and is multiplied by the output of the algorithm to get the score value for that pair of items and that URI. A lower weight means that this particular comparison is less important than others for this run.

A Score run can contain multiple sets of URI, algorithm, and weight (linked together by a common, arbitrary parameter suffix). The total Score value of the individual is the sum of the products of the algorithm output and weight for each set. For example a Score run that is intended to exactly match the full name of a person might be passed in a URI of first name, algorithm of equality test, and weight of 0.3, plus a URI of last name, algorithm of equality test, and weight of 0.5. If both first and last name match, the total Score value will be (1.0 * 0.3) + (1.0 * 0.5) = 0.7. If only last name matched, it would be (0.0 * 0.3) + (1.0 * 0.5) = 5.0.

At this point Score is finished. All it does is generate these values. It is Match that determines what to do with them.

Arguments

Short Option	Long Option	Parameter Value Map	Description	Required
i	inputJena-config	CONFIG_FILE	inputJena JENA configuration filename	true
I	inputOverride	override the JENA_PARAM of inputJena jena model config using VALUE	false	
v	vivoJena-config	CONFIG_FILE	vivoJena JENA configuration filename	true
V	vivoOverride	override the JENA_PARAM of vivoJena jena model config using VALUE	false	
s	score-config	CONFIG_FILE	score data JENA configuration filename	true
S	scoreOverride	override the JENA_PARAM of score jena model config using VALUE	false	
t	tempJenaDir	DIRECTORY_PATH	directory to store temp jena model	false
A	algorithms	for RUN_NAME, use this CLASS_NAME (must implement Algorithm) to evaluate matches	true	
W	weights	for RUN_NAME, assign this weight (0,1) to the scores	true	
F	inputJena-predicates	for RUN_NAME,match	true	
P	vivoJena-predicates	for RUN_NAME, assign this weight (0,1) to the scores	true	
n	namespace	SCORE_NAMESPACE	limit match Algorithm to only match rdf nodes in inputJena whose URI begin with SCORE_NAMESPACE	false

Usage

Explanation

```
# Execute Score for Departments
$Score $SCOREMODELS -n ${BASEURI}org/ -AdeptId=$EQTEST -WdeptId=1.0 -FdeptId=$UFDEPTID -PdeptId=$UFDEPTID
```

Here $SCOREMODELS refers to the models being scored between.

```
SCOREINPUT="-i $H2MODEL -ImodelName=$MODELNAME -IdbUrl=$MODELDBURL -IcheckEmpty=$CHECKEMPTY"
SCOREDATA="-s $H2MODEL -SmodelName=$SCOREDATANAME -SdbUrl=$SCOREDATADBURL -ScheckEmpty=$CHECKEMPTY"
SCOREMODELS="$SCOREINPUT -v $VIVOCONFIG -VcheckEmpty=$CHECKEMPTY $SCOREDATA -t $TEMPCOPYDIR -b $SCOREBATCHSIZE"
```

$VIVOCONFIG refers to the Configuration within vivo.xml

$SCOREINPUT is the current harvested data model

$SCOREDATA is a model containing the data generated from the scoring process and is used by Match to make the changes needed.

The $UFDEPTID contains the predicate being scored on.

```
UFDEPTID="http://vivo.ufl.edu/ontology/vivo-ufl/deptID"
```

The -n ${BASEURI}org/ filter the changes to the specific namespace,

Your desired namespace will be something like

```
http://vivo.myDomain.edu/category
```

As long as all your department URIs begin with that string. This is important if your predicate seems to be part of many different resources.

The EQTEST is making sure that the match is 100% equal (A stands for algorithm)

The *F* and *P* flags are determining the predicates that matched on within the input and VIVO models respectively.

The *deptId*= part needs to be consistent since one score statement can score in multiple ways. you may want to choose label=

Methods

init

Initializes the variables

verifyRunNames

Verify that each map contains the same keys

loadRdfToScoreData

Load a batch of data into the score model

1. build SPARQL insert statement
2. execute statement

buildSelectQuery

Builds the select query.

1. create a StringBuilder with the initial part of the SELECT statement
2. append the rest of the statement with the predicates related to the RUN_NAME
3. return statement as a string

execute

1. Create a vivoClone
2. Create an inputClone
3. Place both vivo and score into same dataset.
4. Build the Query using buildSelectQuery
5. Apply the query to the dataset.
6. For every result:
    a. Build a score record.
    b. Build a fragment of the SPARQL statement
    c. Send SPARQL fragments to loadRdfToScoreData