Solving the TTC Movie Database Case with FunnyQT

Tassilo Horn
horn@uni-koblenz.de

July 22, 2014

Generating Synthetic Models

Implementation: create-positive! function

```
 1 (defn ^:private create-positive! [model i]
 2   (let [m1 (ecreate! model 'Movie :rating (+ 0.0 (* 10 i)))
 3         m2 (ecreate! model 'Movie :rating (+ 1.0 (* 10 i)))
 4         m3 (ecreate! model 'Movie :rating (+ 2.0 (* 10 i)))
 5         m4 (ecreate! model 'Movie :rating (+ 3.0 (* 10 i)))
 6         m5 (ecreate! model 'Movie :rating (+ 4.0 (* 10 i)))]
 7     (ecreate! model 'Actor   :name (str "a" (* 10 i))       :movies [m1 m2 m3 m4])
 8     (ecreate! model 'Actor   :name (str "a" (+ 1 (* 10 i))) :movies [m1 m2 m3 m4])
 9     (ecreate! model 'Actor   :name (str "a" (+ 2 (* 10 i))) :movies [m2 m3 m4])
10     (ecreate! model 'Actress :name (str "a" (+ 3 (* 10 i))) :movies [m2 m3 m4 m5])
11     (ecreate! model 'Actress :name (str "a" (+ 4 (* 10 i))) :movies [m2 m3 m4 m5])))
```

Implementation: create-negative! function

```
 1 (defn ^:private create-negative! [model i]
 2   (let [m1 (ecreate! model 'Movie :rating (+ 5.0 (* 10 i)))
 3         m2 (ecreate! model 'Movie :rating (+ 6.0 (* 10 i)))
 4         m3 (ecreate! model 'Movie :rating (+ 7.0 (* 10 i)))
 5         m4 (ecreate! model 'Movie :rating (+ 8.0 (* 10 i)))
 6         m5 (ecreate! model 'Movie :rating (+ 9.0 (* 10 i)))]
 7     (ecreate! model 'Actor   :name (str "a" (+ 5 (* 10 i))) :movies [m1 m2])
 8     (ecreate! model 'Actor   :name (str "a" (+ 6 (* 10 i))) :movies [m1 m2 m3])
 9     (ecreate! model 'Actress :name (str "a" (+ 7 (* 10 i))) :movies [m2 m3 m4])
10     (ecreate! model 'Actress :name (str "a" (+ 8 (* 10 i))) :movies [m3 m4 m5])
11     (ecreate! model 'Actress :name (str "a" (+ 9 (* 10 i))) :movies [m4 m5])))
```

Implementation: Calling the functions

```
1 (defn create-example [n]
2   (let [model (new-resource)]
3     (dotimes [i n]
4       (create-positive! model i)
5       (create-negative! model i))
6     model))
```

(Extension) Tasks 2 and 3: Finding Couples/Cliques

Task 2/3: Finding Couples

```
1 (defrule make-groups-of-2!
2   {:forall true, :no-result-vec true}
3   [model c]
4   [m<Movie>           :when (>= (person-count m) 2)
5    m -<persons>-> p0 :when (>= (movie-count p0) c)
6    m -<persons>-> p1 :when (>= (movie-count p1) c)
7    :when (neg? (compare (eget p0 :name)
8                         (eget p1 :name)))
9    :when-let [cms (n-common-movies? c p0 p1)]
10   :as [cms p0 p1]
11   :distinct]
12  (ecreate! model 'Couple :p1 p0 :p2 p1
13            :commonMovies cms :avgRating (avg-rating cms)))
```

Extension Task 2/3: Finding 3-Cliques

```
1 (defrule make-groups-of-3!
2   {:forall true, :no-result-vec true}
3   [model c]
4   [m<Movie>           :when (>= (person-count m) 3)
5    m -<persons>-> p0 :when (>= (movie-count p0) c)
6    m -<persons>-> p1 :when (>= (movie-count p1) c)
7    :when (neg? (compare (eget p0 :name)
8                         (eget p1 :name)))
9    :when (n-common-movies? c p0 p1)
10   m -<persons>-> p2 :when (>= (movie-count p2) c)
11   :when (neg? (compare (eget p1 :name)
12                        (eget p2 :name)))
13   :when-let [cms (n-common-movies? c p0 p1 p2)]
14   :as [cms p0 p1 p2]
15   :distinct]
16  (ecreate! model 'Clique :persons [p0 p1 p2]
17            :commonMovies cms :avgRating (avg-rating cms)))
```

Generating the Rules as HOT

```
1  (defmacro define-group-rule [n]
2    (let [psyms (map #(symbol (str "p" %)) (range n))]
3      `(defrule ~(symbol (str "make-groups-of-" n "!"))
4         {:forall true :no-result-vec true}
5         [~'model ~'c]
6         [~'m<Movie> :when (>= (person-count ~'m) ~n)
7          ~@(mapcat (fn [i]
8                      (let [ps (nth psyms i)]
9                        `[~'m -<persons>-> ~ps
10                         :when (>= (movie-count ~ps) ~'c)
11                         ~@(when-not (zero? i)
12                             `[:when (neg? (compare (eget ~(nth psyms (dec i)) :name)
13                                                    (eget ~ps :name)))])
14                         ~@(when-not (or (zero? i) (= i (dec n)))
15                             `[:when (n-common-movies? ~'c ~@(take (inc i) psyms))])]))
16                    (range n))
17          :when-let [~'cms (n-common-movies? ~'c ~@psyms)]
18          :as [~'cms ~@psyms]
19          :distinct]
20         (ecreate! ~'model ~@(if (= n 2)
21                               `['Couple :p1 ~(first psyms) :p2 ~(second psyms)]
22                               `['Clique :persons [~@psyms]])
23                   :commonMovies ~'cms :avgRating (avg-rating ~'cms)))))
24
25 (define-group-rule 2) ;; Generate the Couples rule
26 (define-group-rule 3) ;; Generate the 3-Cliques rule
27 (define-group-rule 4) ;; Generate the 4-Cliques rule
28 (define-group-rule 5) ;; Generate the 5-Cliques rule
```

Extension Tasks 1/4: Compute Top-15 Couples/Cliques

Comparators

1. Average Rating
2. Number of Common Movies
3. Lexicograpic Order of Actor Names

Implementation of Comparators

```
1 (defn rating-comparator [a b]
2   (compare (eget b :avgRating)
3            (eget a :avgRating)))
4
5 (defn common-movies-comparator [a b]
6   (compare (.size ^java.util.Collection (eget b :commonMovies))
7            (.size ^java.util.Collection (eget a :commonMovies))))
```

Implementation of Comparators

```
1 (declare-polyfn actors [group])
2
3 (defpolyfn actors movies.Couple [group]
4   [(eget group :p1) (eget group :p2)])
5
6 (defpolyfn actors movies.Clique [group]
7   (emf/eget-raw group :persons))
8
9 (defn names-comparator [a b]
10  (compare (join ";" (map #(eget % :name) (actors a)))
11           (join ";" (map #(eget % :name) (actors b)))))
```

Comparator Combinator

```clojure
(defn comparator-combinator [& comparators]
  (fn [a b]
    (loop [cs comparators]
      (if (seq cs)
        (let [r ((first cs) a b)]
          (if (zero? r)
            (recur (rest cs))
            r))
        (u/errorf "%s and %s are incomparable!" a b)))))
```

The Top-15 Metrics

```
1 (defn groups-by-avg-rating [groups]
2   (sort (comparator-combinator rating-comparator common-movies-comparator
3                                names-comparator)
4         groups))
5
6 (defn groups-by-common-movies [groups]
7   (sort (comparator-combinator common-movies-comparator rating-comparator
8                                names-comparator)
9         groups))
```

Performance

Performance: Synthetic Models

Model (N)	Gen	Couples	3-Cliques	4-Cliques	5-Cliques
1000	0.12	0.274602528	0.349061152	0.483500704	0.422209712
2000	0.23	0.547466160	0.722435888	0.724224848	0.628254896
3000	0.29	0.867141568	1.069310832	1.028428128	0.949939408
4000	0.39	1.119036992	1.437665888	1.375804384	1.216168544
5000	0.49	1.406045888	1.798415536	1.723140656	1.502462976
10000	0.96	2.830416128	3.607990864	3.482801824	3.098102832
50000	4.82	14.324653712	17.972272736	17.346692096	15.327367392
100000	9.62	28.300907200	36.352164048	35.396692384	31.440208448
200000	20.35	57.159804192	72.209621472	69.233912672	60.430165808

Performance: IMDb Models

Model	Couples	3-Cliques
imdb-0005000-49930.movies.bin	1.15	7.473
imdb-0010000-98168.movies.bin	1.81	12.584
imdb-0030000-207420.movies.bin	2.86	13.959
imdb-0045000-299504.movies.bin	4.11	16.917
imdb-0065000-404920.movies.bin	6.67	24.248
imdb-0085000-499995.movies.bin	10.47	30.936
imdb-0130000-709551.movies.bin	21.16	63.162
imdb-0200000-1004463.movies.bin	40.06	126.58
imdb-0340000-1505143.movies.bin	96.63	385.14
imdb-0495000-2000900.movies.bin	173.75	770.857
imdb-0660000-2501893.movies.bin	283.38	1524.507
imdb-all-3257145.movies.bin	576.75	4675.072