Massachusetts Institute of Technology

6.034 Artificial Intelligence

Solutions # **8** 6034 Item # **22**

Problem 1 Chess Warmup

There are several possible problems with this approach:

1. The match that the search returns – the *first* match – may be from a game played by a novice player, or may simply be a bad move. Better moves may occur further in the search. So, the search returns the first move, irrespective of the quality of the move.
2. There may not be any match to the current board arrangement in the database, since the current board configuration may never have been seen before – the number of possible board positions is huge. The algorithm would simply fail in this case.
3. The search is based only on the board position, not on which player is to move. For example, in board position X the white player may want to move her King. In that same position the black player may want to move her Rook. So if *white* is to move, you must search your memory for a matching board position from which white has moved.

Problem 2

```
(define (compare pref1 pref2)
    "Compare two sets of preferences.  Return the squared difference."
  (do ((p1 pref1 (cdr p1))
       (p2 pref2 (cdr p2))
       (sum 0.0))
      ((null? p1) sum)
    (set! sum (+ sum (square (- (car p1) (car p2)))))))
```

```
(test-prob-2)

=> ((average-joe 16) (arty 17) (non-violent 3) (kids 16) (curmudgeon
7) (optimist 11))
```

```
(test-prob-2)

=> ((average-joe 6) (arty 19) (non-violent 15) (kids 20) (curmudgeon
9) (optimist 12))

(test-prob-2)

=> ((average-joe 14) (arty 9) (non-violent 1) (kids 12) (curmudgeon
19) (optimist 11))
```

```
(test-prob-2-b)

=> ((kids 15) (arty 19) (non-violent 15) (average-joe 20) (curmudgeon
3) (arty 4))

(test-prob-2-b)

=> ((average-joe 8) (arty 17) (arty 3) (kids 9) (curmudgeon 14)
(optimist 6))

(test-prob-2-b)

((average-joe 4) (non-violent 7) (arty 8) (kids 16) (curmudgeon 14)
(optimist 1))
```

When `test-prob-2-b` is called many times in a row, it returns different results because the distributions from which it is generating types of people have large variances. This means that data points could be relatively far away from the prototypical examples, and may fall closer to another class. If the variance is very small, the data points will fall around the prototypical example.

If we think that our "real" data will fall close to the prototypes, ie. there is not much variation or noise in the real world, then we do not need to be concerned. If, however, we do think that the data for a single class could be pretty widely distributed, then we may need to change our similarity measure to better deal with this.

Problem 3

```
(define (compare-missing vec1 vec2)
  (do ((v1 vec1 (cdr v1))
       (v2 vec2 (cdr v2))
       (sum 0.0))
      ((null? v1) sum)
    (if (not (null? (car v1)))
(set! sum (+ sum (square (- (car v1) (car v2))))))))
```

```
(test-prob-3)

=> ((average-joe 16) (arty 15) (non-violent 3) (kids 20) (curmudgeon
20) (optimist 3))

(test-prob-3)

=> ((average-joe 14) (arty 17) (non-violent 6) (kids 20) (curmudgeon
16) (optimist 17))

(test-prob-3-b)

=> ((optimist 2) (non-violent 12) (non-violent 13) (kids 20)
(curmudgeon 4) (average-joe 5))

(test-prob-3-b)

=> ((average-joe 11) (non-violent 20) (average-joe 5) (average-joe 18)
(curmudgeon 14) (optimist 2))

(test-prob-3-b)

=> ((average-joe 2) (arty 8) (non-violent 11) (kids 13) (curmudgeon
14) (optimist 15))
```

Problem 4

The following piece of code computes the distance for only those features that are present in both and then normalizes this distance by the number of non-null features that were present. This way, a distance generated between two people that only had a couple of common features can be compared to a distance generated between two people that had many features present.

```
(define (compare-missing-2 vec1 vec2)
  (do ((v1 vec1 (cdr v1))
       (v2 vec2 (cdr v2))
       (tot 0)
       (sum 0.0))
      ((null? v1) (if (eq? tot 0)
      100000000
      (/ sum tot)))
    (if (and (not (null? (car v1)))
     (not (null? (car v2))))
(begin
 (set! tot (+ tot 1))
 (set! sum (+ sum (square (- (car v1) (car v2)))))))))

(test-prob-4)

=> ((average-joe 9) (arty 15) (non-violent 15) (kids 19) (curmudgeon
10) (optimist 3))

(test-prob-4)

=> ((average-joe 13) (non-violent 4) (non-violent 20) (kids 2)
(curmudgeon 12) (optimist 9))

(test-prob-4-b)

=> ((kids 14) (kids 16) (non-violent 4) (kids 15) (kids 16)
(average-joe 3))

(test-prob-4-b)

=> ((optimist 5) (average-joe 17) (arty 9) (kids 19) (curmudgeon 6)
(optimist 16))

(test-prob-4-b)

=> ((non-violent 13) (kids 16) (non-violent 16) (arty 19) (curmudgeon
6) (average-joe 19))
```

Problem 5

```
(define (predict-preferences database pref k)
  "Returns the average of the preferences of the K nearest neighbors."
  (define (average . vals)
    (do ((v vals (cdr v))
 (sum 0.0)
 (num 0 ))
((null? v) (if (= 0 num)
       '()
       (/ sum num)))
      (if (not (null? (car v)))
  (begin (set! num (+ num 1))
 (set! sum (+ sum (car v)))))))
  (apply map (cons average
   (map data-point-preferences
(k-nearest-neighbors database pref k)))))
```

```
(test-prob-5-works)

=> ((4.391 4.208 4.578 4.611 4.518 4.639 2.032 2.006 2.078 2.189
1.9829999999999999 1.9639999999999997 3.8610000000000007
3.8819999999999992 3.9859999999999998 3.8770000000000002 3.521
3.7779999999999996 3.562 3.1399999999999997 3.043 3.062
3.0409999999999995 2.905 2.066 2.1580000000000004 2.109
2.0300000000000002 2.2929999999999997 2.0919999999999996)
(1.3210000000000002 1.25 1.591 1.5779999999999998 1.434 1.45 4.511
4.508 4.425 4.458 4.15 4.5680000000000005 1.829 2.103
1.8989999999999998 1.875 2.508 2.588 2.529 3.918 4.314 3.964
3.8810000000000002 4.055 2.9280000000000004 2.7529999999999997
2.9210000000000003 2.789 2.8289999999999997 3.074) (1.4360000000000004
1.456 1.6589999999999996 1.6420000000000001 1.185 1.3319999999999999
4.414 4.499 4.1899999999999995 4.398000000000001 4.239
4.518000000000001 1.991 1.9919999999999998 1.9829999999999999
1.9639999999999997 2.7390000000000003 2.762 2.9080000000000004
3.9870000000000005 4.215999999999999 3.873 3.9 4.023
3.2990000000000004 3.2909999999999995 2.956 2.8080000000000007 2.935
3.2990000000000004) (4.473 3.9889999999999994 4.517000000000001 4.461
4.514000000000001 4.1930000000000005 1.2500000000000002 1.057 1.083
1.2469999999999999 .9939999999999998 1.0390000000000001
4.3069999999999995 4.2669999999999995 4.294 4.48 2.979 2.886 2.94
.8859999999999999 1.041 .943 1.034 1.179 .958 1.0909999999999997 1.024
.9139999999999999 .825 1.043) (.866 .9189999999999998
```

```
1.0789999999999997 1.1260000000000001 .741 1.121 1.017 1.008
.7770000000000001 .7050000000000001 .817 1.067 1.264 1.145 .826 .933
.646 .891 1.131 1.0110000000000001 1.048 1.127 1.272
1.1800000000000002 .8879999999999999 .723 1.0790000000000002
.9059999999999999 1.1179999999999999 .9339999999999999)
(4.569999999999999 4.5409999999999995 4.79 4.6579999999999995
4.462999999999999 4.349 4.248 4.406000000000001 4.4479999999999995
4.356 4.292 4.348999999999999 4.464 4.676 4.396 4.489 4.423
4.726000000000001 4.526000000000001 4.364 4.557 4.676
4.535000000000001 4.555000000000001 4.331999999999999 4.311 4.413
4.424000000000001 4.403 4.634))
```

```
(test-prob-5-doesnt-work)
```

```
=> ((4.4075 4.2775 4.401666666666666 4.4174999999999995
4.316666666666667 4.786666666666666 1.9633333333333332
2.0733333333333333 2.0660000000000003 3.15 1.9260000000000002 1.76
3.701250000000001 3.893333333333333 4.1659999999999995 3.8325
3.7466666666666666 3.4524999999999997 3.9560000000000004 3.0975
3.1585714285714284 3.0433333333333333 3.2449999999999997
2.856666666666667 1.93 2.181666666666667 2.4499999999999997 1.918
2.5366666666666666 2.198) (1.318 1.234 1.7533333333333332 1.886 1.386
1.2514285714285713 4.079999999999999 4.7375 4.5840000000000005
4.466666666666666 4.1433333333333335 4.522 1.475 1.825
1.7333333333333334 2.4899999999999998 2.356666666666667 2.19 2.2725
3.7950000000000004 3.516666666666667 3.89 4.0280000000000005 4.0375
2.9214285714285713 3.005555555555555 2.6566666666666667 2.90875 2.895
2.9225) (1.008 1.386 1.375 .99 1.3 .65 4.362 4.112 3.8475
4.178333333333334 4.182 4.5225 2.116 1.7850000000000001 2.1425
2.1216666666666666 3.71 4.252000000000001 3.517142857142857 4.4325
4.476666666666667 4.358 4.4333333333333345 4.31375 3.395 3.265
3.3428571428571425 3.0425000000000004 3.6885714285714286 3.465)
(4.458571428571429 4.34 4.6025 4.787999999999999 4.1925
4.515000000000001 1.2914285714285714 .9949999999999999 .8725 .56
1.0916666666666666 .9450000000000002 4.5649999999999995 4.215 4.7
4.452857142857144 2.961428571428571 3.2659999999999996 2.935 1.44
.5625 1.0171428571428571 .7916666666666666 1.1375 1.21 .775
.9600000000000001 1.1283333333333332 1.3960000000000001 .99)
(.8320000000000001 1.01 .8959999999999999 .9583333333333334 1.245
.9524999999999999 1.03 .98 .8183333333333334 .9360000000000002
.9400000000000001 1.2200000000000002 .9333333333333332
1.2274999999999998 .9460000000000001 .9533333333333333
.9833333333333334 .8133333333333334 .9516666666666665
.8183333333333332 1.19 .818 .9400000000000001 1.1175000000000002
```

.8683333333333335 .6900000000000001 1.0033333333333332
1.3900000000000001 .628 1.3571428571428572) (4.562 4.433333333333333
4.305 4.265000000000001 4.3725000000000005 4.414 4.452
4.458749999999999 4.4425 4.478888888888889 4.654999999999999 3.705
4.716 4.128 4.555000000000001 4.45 4.0125 4.634285714285714 4.246
4.29125 4.52 4.494999999999999 4.282 4.3425 4.208 4.486 4.734 3.995
4.7975 4.7700000000000005))