Optional exercise (2c) (INF4820, Fall 2013)

This is a completely **optional** exercise set, intended for those interested in doing some extra programming.

Summary of goals for this exercise:

- Add more functionality to the vector space model from exercise 2a.
- We will compute a so-called proximity matrix for all the words in the vector space, storing the pairwise similarities between each of them.
- Based on the similarity values in the proximity matrix, we will extract lists of k-nearest-neighbors (kNNs).

Files you'll need

We'll be re-using the data sets from exercise 2a–b, in particular 'brown20000.txt'. We also assume you have a working solution for 2a, like the provided `vs.lsp` (the text for exercise 2b gives the details for how to obtain this).

Computing a proximity matrix and extracting kNN relations

(1) In this exercise you'll implement what is sometimes called a *proximity matrix* (or a *similarity matrix*) for our vector space model. For a given set of vectors $\{\vec{x}_1, \ldots, \vec{x}_n\}$ the proximity matrix, M is a square $n \times n$ matrix where each element M_{ij} gives the proximity of $\vec{x}_i$ and $\vec{x}_j$. We'll assume the similarity measure in our semantic space model is the *dot-product*, as implemented in exercise 2a. In other words we want M_{ij} to store the value of the dot-product computed for the (length normalized) feature vectors $\vec{x}_i$ and $\vec{x}_j$.

The reason why we want to compute and store all the pairwise similarities is that this will make it easier to later extract lists of nearest neighbors in the space (more on that below). But such proximity matrices are also often used as input to clustering algorithms, such as many instances of agglomerative clustering that are based on repeatedly looking up pairwise similarities in every iteration.

An important observation here is that, since most similarity measures are *symmetric*, including the dot-product, the proximity matrix will also be symmetric. In other words, since $\vec{x_i} \cdot \vec{x_j} = \vec{x_j} \cdot \vec{x_i}$, we also have that $M_{ij} = M_{ji}$. This means that we would waste a lot of space if we stored each of these identical values separately. Try to take this into account when you choose a data structure and when you implement functions for accessing and updating the matrix.

Implement a function '`compute-proximities`' that takes a vector space structure ('`vs`') as its single argument and then computes a proximity matrix for all the feature vectors in the space. Add an extra slot '`proximity-matrix`' to the '`vs`' structure to store the result.

For testing purposes it might be helpful to write a function '`get-proximity`' expecting three arguments: a '`vs`' structure and two words. It should then return the dot-product of the two feature vectors that corrspond to the given words. (Of course, it should look up the value from the proximity matrix, not actually compute the function.) For example:

```
CL-USER(55): (compute-proximities space)
#S(VS ...)
CL-USER(56): (get-proximity space "kennedy" "nixon")
0.5411588
CL-USER(57): (get-proximity space "nixon" "kennedy")
0.5411588
```

(NB: Your exact proximity values would likely be different from these!)

(2) Here's an idea for those who feel like trying their hands at implementing a more advanced abstract data type for symmetric matrices, using more advanced Lisp techniques. (But if you want you can also skip directly to the next problem without solving this one.)

We want to define an abstract data type (ADT) '`symat`' for compactly representing square symmetrical matrices like the proximity matrix described above. We can use '`defstruct`' to define a structure of type '`symat`'. Write a function '`make-symat`' that takes a numeric argument n, and creates a '`symat`' for representing an $n \times n$ symmetric matrix. You should also define associated functions for both *referencing* and destructively *modifying* elements in the matrix. (The built-in macro '`defsetf`' will be helpful for doing the latter.) We want to able to do things like the following:

```
CL-USER(66): (setf m (make-symat 10))
#<SYMAT ...>
CL-USER(67): (setf (symat-ref m 0 9) 0.5)
0.5
CL-USER(68): (symat-ref m 9 0)
0.5
```

As we can see, after destructively modifying an element M_{ij} to store some value, referencing M_{ji} should give us that very same value. But here is the fun part: It is possible to implement such a square symmetric matrix using just a one-dimensional / linear array, and without wasting any elements! How? Try to implement one.

(3) Write a function '`find-knn`' that extracts a ranked list of the nearest neighbors for a given word in the space. The function should take an optional argument specifying how many neighbors to return (defaulting to 5). Use the stored values in the proximity-matrix to extract the ranked list. Example calls (your results might differ):

```
CL-USER(70): (find-knn space "egypt")
("italy" "america" "europe" "germany" "government")

CL-USER(71): (find-knn space "salt")
("pepper" "mustard" "sauce" "butter" "water")
```

(4) You're now very close to having everything you need to implement a kNN classifier (as an alternative to Rocchio) for the classification problem described in exercise 2b...