Obligatory assignment 3 (INF4820, Fall 2012)

This is the third out of five obligatory assignments in INF4820 for the fall 2012. The assignments won't count in the final grade, but all have to be passed in order to qualify for the final exam. If you have any questions, send an email to `murhaff@ifi.uio.no` and/or `erik.velldal@ifi.uio.no`.

As mentioned before: More important than getting everything 100% right is to make sure you show that you've tried your best; If there is a problem you find you can't solve, write down your thoughts on where/why you got stuck. If you find you cannot provide working Lisp code, try to provide pseudo-code or explain in words how you believe the problem could be solved. You should aim to have an answer for all questions.

Submitting: Please provide your code and answers (in the form of Lisp comments) in a *single* '.lisp' file. Submit the file via Devilry by the end of the day (23:59) on Monday, October 8th: `https://devilry.ifi.uio.no/`.

Summary of goals for this exercise:

- Add more functionality for the vector space model from the previous assignment:
- While we will be using a larger corpus, we will only extract feature vectors for a given list of words.
- We will then compute a so-called proximity matrix for all the words in the vector space, representing the 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).
- Implement a Rocchio classifier. This means we need to define functions for
  - reading in information about predefined classes (the training data);
  - computing centroid representations for the classes; and
  - classifying unknown words as a function of centroid distance.
- **NB**: The programming we need to do for this exercise naturally extends on what we did for the previous one. To ensure that everyone is starting out from a working (and sufficiently efficient) implementation of that functionality, we provide a solution (i.e., source code) that you're free to use as you want.

Files you'll need

Our data set will be the first 20.000 sentences of the Brown corpus, comprising almost half a million words. As the data files are licensed, please do not re-distribute. Copy the file '`brown20000.txt`' to your home directory:

```
cp ~erikve/inf4820/brown20000.txt ~/
```

In addition you'll need the file '`words`' (containing a list of words) and the file '`classes`' (containing a list of predefined classes and their members):

```
cp ~erikve/inf4820/{words,classes} ~/
```

An implementation of the previous assignment (exercise 2) can be copied from the same directory:

```
cp ~erikve/inf4820/vs.lsp ~/
```

1 Reading in the corpus data

Just like for the previous assignment, our first goal is to construct a vector space model for words, with contextual features corresponding to other words co-occurring within the same sentence (*bag-of-words*). However, unlike the set-up for the previous exercise, we'll only construct vectors *for a specified list of words*. Given a list of n words, we will construct a space of n feature vectors. The dimensions of the space (i.e. the elements of the vectors) will still encode whatever m contextual features we end up extracting from the corpus. (So, while we for the previous exercise constructed an $n \times n$ co-occurrence matrix, it will now be $n \times m$.)

The list of words is given as the file '`words`', with one word per line (122 all together). The corpus which we will use for extracting features for these words contains one sentence per line ('`brown20000.txt`').

The provided source file '`vs.lsp`' includes an implementation of the function '`read-corpus-to-vs`' which optionally takes such a word list (viz. a file) as a second argument. The function then returns a vector space model, defined in terms of the Lisp-structure '`vs`'. Feel free to modify it or use it as is (of course, you're also free to extend/write your own code for this). Make sure to always use *compiled* (rather than just interpreted) code when dealing with large data sets like we'll do here, as this makes the code run faster. As described in your Emacs cheat sheet, this can be achieved by loading your code using `:cl file` (rather than `:ld file`) at the Lisp prompt.

Note that the implementation in '`vs.lsp`' includes a global variable '`*stemming-p*`' that determines whether or not we'll apply the '`stem`' function (as defined in '`stemmer.lisp`' from the previous assignment). For all examples in this assignment text, '`*stemming-p*`' is set to `nil` for better readability of the examples, but you should experiment with various settings (`t` would mean that all words and features are stemmed).

The following sequence of calls will create a vector space model for the words in the file '`words`', using length-normalized vectors of association weights:

```
CL-USER(34): (setq *stemming-p* nil)
CL-USER(35): (setq space
               (length-normalize-vs
                 (read-corpus-to-vs "brown20000.txt" "words")))

#S(VS :STRING-MAP (:FEATURE ...
                   :WORD ... )
      :ID-MAP (:FEATURE ...
               :WORD ...)
      :MATRIX ...
      :SIMILARITY-FN ...
      :PROXIMITY-MATRIX NIL
      :CLASSES NIL)
```

As you'll see, the structure definition of '`vs`' in '`vs.lsp`' has been extended with a few more slots to accommodate the extensions to the vector space model that we will be implementing here. Please take some time to familiarize yourself with the code in the provided source file, and try to convince yourself that it does what it should.

NB: The solutions to problems 2 and 3 do not depend on each-other. All relevant functionality for each problem can be implemented independently of the other.

2 Computing a proximity matrix and extracting kNN relations

(a) 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 $\vec{x}_j$. As implemented in the previous exercise set, the similarity measure in our semantic space model is the *dot-product*. So 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.

A key 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 acessing 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. Store the result in the slot '`proximity-matrix`' in the given '`vs`'.

<**Optional**> NB: The following describes an *optional* "bonus exercise" for those who feel like trying their hands at implementing a more advanced abstract data type for symmetric matrices, using more advanced Lisp techniques.

Define an abstract data type (ADT) '`symat`' for compactly representing square symmetrical matrices like 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. </**Optional**>

(b) 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. (Recall that kNNs also form the basis for the kNN classifier as described in class. We won't be implementing kNN classification here, however.) Example calls:

```
CL-USER(70): (find-knn space "egypt")
("italy" "america" "europe" "germany" "government")
CL-USER(71): (find-knn space "salt")
("pepper" "mustard" "sauce" "butter" "water")
```

(c) In the previous assignment (exercise set 2) we created a vector space model where the set of words and the set of features were identical. This meant that the feature vectors could be thought of as an $n \times n$ co-occurrence matrix. Could this be regarded as a symmetric matrix like the proximity matrix described above? Explain your position.

3 Implementing a Rocchio classifier

(a) In this particular exercise we'll implement a *Rocchio classifier*. The first thing we need to do is read in information about which words are associated with which classes. Have a look at the file '`classes`'. This file contains lists specifying the class membership of the different words in our model. The first element in each list specifies the class name (given as a Lisp keyword, e.g. '`:food`'. The rest of the list specifies the words associated with the given class, e.g. '`(potato food bread fish ...)`'. Some of the words are *unclassified*, however, and these are listed with the dummy class '`:unknown`'. The unknown words make up our *test data*; the words we will want to classify. The other words defines our *training data*. (Note that, the words found in the file '`classes`' are the same as those in the file '`words`'. This means that all the relevant feature vectors, both for the training items and the test items, are already included in our model.)

Write a function '`read-classes`' that reads the lists from the file '`classes`' and stores the information about class-membership in the slot '`classes`' in our vector space structure. Exactly how to store and organize that information is up to you. (But you'll want to make it easy to retrieve information about the members of each class, and perhaps also make it possible to add more information about classes later, such as the corresponding centroid representation. You probably also want to take care to convert all the words to lowercase strings.) Remember that the function '`read`' is handy for reading s-exps (like lists).

(b) The Rocchio classifier represents classes by their *centroids* $\vec{\mu}$. For a given class c_i, the centroid vector $\vec{\mu}_i$ is simply the average of the vectors of the class members:

$$\vec{\mu}_i = \frac{1}{|c_i|} \sum_{\vec{x}_j \in c_i} \vec{x}_j$$

The class centroids are often not normalized for length, but in order to avoid bias effects for classes with different sizes (classes with many members will typically have less sparse centroids and a larger norm), we will here define our centroids to have unit length. By this we mean that their Euclidean length should be one; $\|\vec{\mu}_i\| = 1$. Luckily we implemented functionality for normalizing vectors in the previous exercise set.

Write a function '`compute-class-centroids`', expecting only a '`vs`' structure as its argument. The function should compute the length normalized centroids for each class. Store the centroids within the vector space structure (for example adding it to the already existing '`classes`' slot).

(c) We now have all the pieces we need in order to write a Rocchio classifier and classify the unknown words. Write a function '`rocchio-classify`' that for each unclassified word in our model (i.e. the words that were listed after '`:unknown`' in the file '`classes`') tells us which class is associated with its nearest centroid.

Recall that, the Rocchio classifier assigns each word to the class which has the nearest centroid. When measuring how close a given feature vector is to a given centroid, we continue to use the dot-product, just as when measuring the distance between pairs of feature vectors. As an example, the output of '`rocchio-classify`' could look something like this:

```
CL-USER(73): (read-classes space "classes")
#S(VS ...)
CL-USER(74): (compute-class-centroids space)
#S(VS ...)
CL-USER(75): (rocchio-classify space)
(("fruit" :FOODSTUFF 0.45438093) ("california" :PERSON_NAME 0.4153575)
 ("peter" :PERSON_NAME 0.3715801) ("egypt" :PLACE_NAME 0.42086726)
 ("department" :INSTITUTION 0.6266466) ("hiroshima" :PLACE_NAME 0.28371271)
 ("robert" :PERSON_NAME 0.5719534) ("butter" :FOODSTUFF 0.4608509)
 ("pepper" :FOODSTUFF 0.3369783) ("asia" :PLACE_NAME 0.45727462)
 ("roosevelt" :PLACE_NAME 0.3946177) ("moscow" :PLACE_NAME 0.5722181)
 ("senator" :TITLE 0.464052) ("university" :INSTITUTION 0.6812986)
 ("sheriff" :TITLE 0.30681545))
```

Among other things, this would mean that we found the centroid of the class '`:place_name`' to be the one closest to the feature vector representing '`egypt`', and that the dot-product of these two vectors is approximately 0.42. (Your results might differ.)

(d) So far we haven't said anything about the particular *data type* used for implementing the *centroid vectors*. We have silently assumed that the data type is the same as what we've used for the feature vectors of individual words. In a few sentences, discuss whether or not you believe this is a wise choice.

(e) With the functionality that we now have in place (also considering Problem 2), we're only a few steps away from implementing a *kNN-classifier*. In a few sentences, sketch what remains to be done in order to assign class memberships to the '`:unkown`' words using the kNN classification method instead of Rocchio.