Algorithms for AI and NLP, Fall 2009, Exercise 4:

— Part 2:2 of Obligatory Assignment 2 —

Summary of goals for this exercise:

- Construct a vector space model for a given list of words, extracting features from the full Brown corpus.
- Define an abstract data type for compactly representing high-dimensional square symmetric matrices.
- Using this data type, we will then compute a proximity matrix for all the objects in the vector space.
- Based on the similarity values in the proximity matrix, we will extract lists of k-nearest-neighbors.
- 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.

1 Data Sets and Theoretical Background

Our data set for this session we will be the full Brown corpus, comprising more than a million words, distributed over more than 50.000 sentences. As the data files are licensed, please do not re-distribute. Copy the file '`browncorpus.txt`' to your IFI home directory:

```
cp ~erikve/inf4820/browncorpus.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 3) can be copied from the same directory:

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

The theoretical background for this exercise set is mainly what was covered in the lectures of Nov. 3 and Nov. 10. See the course web-page for lecture slides and relevant sections from Manning, Raghavan & Schütze (2008): `http://www.uio.no/studier/emner/matnat/ifi/INF4820/h09/undervisningsplan.xml`

2 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. In other words, we take the same sentence-level *bag-of-words* approach to context. However, unlike the set-up for the previous exercise, we now only want to 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 (142 all together). The corpus file, '`browncorpus.txt`', which we will use for extracting the features, has the same format as '`brown5000.txt`' in exercise 3; one sentence per line.

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 function). 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(7): (setq space (normalize-vs
                         (compute-associations
                          (read-corpus-to-vs "browncorpus.txt" "words"))))
#S(VS :STRING-MAP ...
      :ID-MAP ...
      :MATRIX ...
      :COUNTS ...
      :ASSOCIATION-FN ...
      :SIMILARITY-FN ...
      :PROXIMITY-MATRIX ...
      :CLASSES ...)
```

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 3 and 4 do not depend on each-other. (An exeption is perhaps the theoretical question in Problem 4 *e.*) All relevant functionality for each problem can be implemented independently of the other.

3 Computing the Proximity Matrix and Extracting kNN Relations

a) In this exercise you'll implement 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 $\vec{x}_j$.

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.

Your first task, therefore, is to define an abstract data type (ADT) '`symat`' for compactly representing such square symmetrical matrices. 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 also need to 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(6): (setf m (make-symat 10))
#<SYMAT ...>
CL-USER(7): (setf (symat-ref m 0 9) 0.5)
0.5
CL-USER(8): (symat-ref m 9 0)
0.5
CL-USER(9):
```

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.

(For the brave: It is possible to implement a square symmetric matrix using just a one-dimensional / linear array, and without wasting any elements. How?)

b) Using our new data type '`symat`', 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 resulting '`symat`' structure in the slot '`proximity-matrix`' in the given '`vs`'.

For testing purposes it will 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.)

```
CL-USER(11): (compute-proximities space)
#S(VS ...)
CL-USER(12): (get-proximity space "kennedy" "nixon")
0.1225828663153755d0
CL-USER(13): (get-proximity space "nixon" "kennedy")
0.1225828663153755d0
```

c) 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.

```
CL-USER(18): (find-knn space "egypt")
("london" "italy" "congo" "england" "france")
CL-USER(19): (find-knn space "rice" 10)
("grains" "salt" "sauce" "eggs" "supper" "food" "wine" "mustard" "peas" "coffee")
```

d) In the previous assignment (exercise set 3) we created a vector space model where the set of words and the set of features were identical. This meant that the set of feature vectors could be thought of as an $n \times n$ co-occurrence matrix. Would it have made sense to implement this as a structure of type '`symat`'? State the reasons for your position.

4 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.)

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(30): (read-classes space "classes")
#S(VS ...)
CL-USER(31): (compute-class-centroids space)
#S(VS ...)
CL-USER(32): (rocchio-classify space)
(("fruit" :FOODSTUFF 0.1864267655995263d0)
 ("california" :PLACES 0.13484172448163065d0)
 ("peter" :TITLES 0.1101483876316607d0)
 ("egypt" :PLACES 0.12747253814510132d0)
 ("minister" :TITLES 0.13187990491115562d0)
 ("department" :INSTITUTIONS 0.21868344824708214d0)
 ("hiroshima" :PLACES 0.09141659518788094d0)
 ("robert" :NAMES 0.18579606370217458d0))
```

Among other things, this would mean that we found the centroid of the class '`:places`' to be the one closest to the feature vector representing '`egypt`', and that the dot-product of these two vectors is approximately 0.128.

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 3), 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.

5 Submitting Your Results

Please submit your results (code+answers) in email to Erik ('`erikve@ifi.uio.no`') before the end of Sunday 22. November. It is possible to obtain a maximum of 100 points for this exercise set.

Good luck and happy coding!