INF4820; Fall 2016: Obligatory Exercise (2b)

Background

Exercise (2b) is part two (of two) of the *second* obligatory exercise in INF4820. You can obtain up to 10 points for this problem set. For (2a) and (2b) in total you need a minimum of 12 points (i.e. 60,% of the maximum). Please make sure you read through the entire problem set before you start. If you have any questions, please post them on our Piazza discussion board or email `inf4820-help@ifi.uio.no`, and make sure to take advantage of the laboratory sessions.

Submitting Solutions must be submitted through Devilry by *noon* (12:00) on Sunday, October 16: `https://devilry.ifi.uio.no/`. Please provide your solution as a single '`.lisp`' file, including your code and answers (in the form of Lisp comments). Please also generously document your code with comments (the Lisp reader will ignore everything following a semicolon '`;`').

Required reading

The theoretical background for this problem set is covered in the following selection of sections from Manning, Raghavan, & Schütze: *Introduction to Information Retrieval*, 2008:

- Classification: Sec. 14–14.4
- Clustering: Sec. 16–16.4.1 (you can skip sec. 16.3)

Important note on the relation to exercise (2a)

The programming we need to do for this assignment naturally extends on what we did for the previous one. To ensure that everyone is starting out from a working and sufficiently efficient implementation, we will provide a solution for 2a that you are free to use if you want. If you prefer to instead continue building on your own implementation from 2a, then that is also absolutely fine. Note that our source code for (2a) will not be distributed until Monday, October 3, since some students have been granted a three-day extension to the (2a) deadline due to illness. However, you can still start working on the assignment before Saturday as the questions in section 3 are all theoretical and do not require working code from (2a).

Files you'll need

We will be re-using the data sets from exercise (2a); '*brown2.txt*' and '*words.txt*'. In addition, you'll need the file '*classes.txt*', containing lists of predefined classes and their members. If you '`svn update`' your INF4820 repo you will find this file added to a new directory '*2b*'. The sample solution for (2a) will also be added to this directory.

Reading in the corpus data

The provided source file '*solution2a.lisp*' includes an implementation of the function '`read-corpus-to-vs`' taking two arguments; a corpus file and a file specifying the words to model. The function returns a vector space model, defined in terms of the Lisp structure '`vs`'. Feel free to modify it or use it as is. Make sure to use *compiled* (rather than just interpreted) code when dealing with large data sets, as this makes the code run faster.

The following call will create a length-normalized vector space model for the words in the file '*words.txt*':

```
CL-USER(35): (defparameter vspace
               (length-normalize-vs
                 (read-corpus-to-vs "brown2.txt" "words.txt")))

#S(VS :MATRIX ...
      :SIMILARITY-FN ...
      :CLASSES NIL
      :PROXIMITY-MATRIX NIL)
```

As you can see, the structure definition of '`vs`' in '*solution2a.lisp*' 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.

1 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 conceptually a square $n \times n$ matrix where each element M_{ij} gives the proximity of $\vec{x}_i$ and $\vec{x}_j$.

We'll here assume that the similarity measure in our semantic space model is the *dot-product*, as implemented in assignment (2a) (and stored in the slot named '`similarity-fn`' of our abstract data type '`vs`'). 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 (we'll return to this later in lecture 8).

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/effort if we stored/computed each of these identical values separately. Try to take this into account when you choose a data structure and when you implement your 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. Unless you're re-using '*solution2a.lisp*', 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 correspond to the given words. (Of course, it should look up the value from the proximity matrix, not actually compute the function.)

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

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

(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. 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" 1)
("pepper")
```

2 Implementing a Rocchio classifier

(a) In this 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.txt*'. 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. '`:foodstuff`'. The second element is a list specifying the words associated with the given class, e.g. '`(potato food bread fish ...)`'. Some of the words are *unlabeled*, however, and these are listed as '`:unknown`'. The unknown words are the words we want to classify. The other words define our *training data*. Note: The words found in the file '*classes.txt*' are the same as those in the file '*words.txt*'. This means that all the relevant feature vectors, both for the training items and the test items, are already available in our model.

Write a function '`read-classes`' that reads the lists from the file '*classes.txt*' and stores the information about class-membership in the slot '`classes`' in our '`vs`' 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 very 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,

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

where $|c_i|$ denotes the cardinality of the class (i.e., the number of class members as observed in the training data). 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 assignment (2a).

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 somewhere 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 implement a Rocchio classifier and predict labels for all the '`:unknown`' words. Write a function '`rocchio-classify`' that for each unlabeled word in our model assigns it the label of the class with the nearest centroid (using the dot-product as our similarity function).

As an example, the output of '`rocchio-classify`' could look something like this:

```
CL-USER(73): (read-classes vspace "classes.txt")
NIL

CL-USER(74): (compute-class-centroids vspace)
```

```
NIL

CL-USER(75): (rocchio-classify vspace)
(("fruit" :FOODSTUFF 0.36678165) ("california" :PERSON_NAME 0.29967937)
 ("peter" :PERSON_NAME 0.3088614) ("egypt" :PLACE_NAME 0.30741462)
 ("department" :INSTITUTION 0.54971284) ("hiroshima" :PLACE_NAME 0.23045596)
 ("robert" :PERSON_NAME 0.59566414) ("butter" :FOODSTUFF 0.384543)
 ("pepper" :FOODSTUFF 0.36385757) ("asia" :PLACE_NAME 0.3986899)
 ("roosevelt" :TITLE 0.2597395) ("moscow" :PLACE_NAME 0.48412856)
 ("senator" :TITLE 0.3563019) ("university" :INSTITUTION 0.5805098)
 ("sheriff" :TITLE 0.22804888))
```

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 0.31. (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.

3 Classification theory

(a) Give an outline of the main differences between Rocchio classification and kNN classification. Limit your discussion to no more than half a page.

(b) In just a couple of sentences, compare and contrast Rocchio classification and k-means clustering. What sets these algorithms apart, and in what ways can they be considered similar? (We're interested in rather general and basic traits here.)

(c) One missing piece in the implementation above is *evaluation*. In a few sentences, outline what would need to be done in order to evaluate our classifier here. Include some comments on the different strategies we could use for computing our evaluation scores.

Good luck and happy coding!