# Homework 2: SOLUTIONS

36-350: Data Mining

1. *Quantize the (R,G,B) cube with 8 prototypical colors:*

| Color | Red | Green | Blue | Color | Red | Green | Blue |
|---|---|---|---|---|---|---|---|
| Black | 0.25 | 0.25 | 0.25 | Red | 0.75 | 0.25 | 0.25 |
| Green | 0.25 | 0.75 | 0.25 | Yellow | 0.75 | 0.75 | 0.25 |
| Blue | 0.25 | 0.25 | 0.75 | Magenta | 0.75 | 0.25 | 0.75 |
| Cyan | 0.25 | 0.75 | 0.75 | White | 0.75 | 0.75 | 0.75 |

   *That is, each pixel's RGB vector gets replaced by that of the closest prototypical color.*

   *Take a six-pixel image, with RGB values (0.22,0.37,0.8), (0.19,0.8,0.19), (0.6,0.1,0.05), (0.8,0.3,0.22), (0.7,0.32,0.8), and (1,0.4,0.34). What is the bag-of-colors representation of the image? What is the representation after norming by Euclidean length?*

   Answer: The color labels of the points are, in order, "blue", "green", "red", "red", "magenta" and "red". The bag-of-colors representation would thus be

```
   blue   green magenta     red
      1       1       1       3
```

   before normalization and

```
   blue   green magenta     red
   0.29    0.29    0.29    0.87
```

   after normalization.

2. *Suppose we search for the image in question 1 via the color (0.7,0.2,0.2). What is the bag-of-colors representation for this query, and what is the query's Euclidean distance from the image, after both have been divided by Euclidean length?*

   Answer: The RGB values quantize to "red". The bag-of-colors representation is thus just

```
red
  1
```

which is unchanged by Euclidean-length normalization. The Euclidean distance is $\sqrt{3(0.29)^2 + (0.87 - 1)^2} = 0.52$.

3. *Describe a potential problem in measuring distance if we use too many prototypical colors in the representation (besides increased computation and storage). Describe a potential problem if we use too few prototypical colors in the representation.*

   ANSWER: Using too few colors will make too many images seem similar to each other. It will reduce the precision of a query, since many irrelevant results will be indistinguishable from relevant ones. Using too many colors will make it seem, in our representation, as though images were very different even when they are similar in every way which matters; it will reduce recall, since many images which are relevant will not be retrieved.

4. *Suppose we have a collection of 50 flower and 50 ocean images, and use just three prototypical colors, red, green and blue. The flower images have red and green but no blue, and the ocean images have green and blue but no red. What are the inverse-picture-frequency (IPF) weights for the three colors?*

   ANSWER: Green appears in every picture, so its IPF weight is $\log 100/100 = 0$. Red and blue both appear in 50 out of 100 pictures, getting IPF weights of $\log 100/50 = \log 2$.

5. *Suppose the query vector is* $(1, 0, 0, 0)$ *and there are two items in the database, with vectors* $(3, 1, 1, 1)$ *and* $(3, 2, 0, 0)$ *(these might be documents or images or something else). Which vector is closer to the query when we normalize by the sums? Which one is closer when we normalize by the Euclidean length? Does it matter whether we are talking about images or documents? Why or why not?*

   ANSWER: The second vector is closer when normalizing by the sum, the first vector when normalizing by the Euclidean length. The meaning of the vectors is completely irrelevant to their distances, which only depend on the numerical values of their components.

6. *In a typical data mining application, a news agency wants to search through video archives and detect all frames depicting an event, e.g. fireworks at night, given some examples. You can regard a video as a sequence of images. How could this be implemented using similarity search? (Don't give code, just a brief description of the method.)*

   ANSWER: Construct an index, which would record the bag-of-colors vector for each frame of each video. Then form the bag-of-colors vector of the example, and start computing the distances between the example vector and the vectors in the index. Take the $k$ closest matches and use the index to return the corresponding images (rather than their bag-of-colors representations).

7. *Recall that in* **nearest-neighbor** *classification, we guess that a new vector belongs to the same class as the closest perviously-seen vector whose class is known. In* **prototype** *classification, we represent each class by the average of the vectors belonging to that class, and assign new vectors to the class whose prototype is closests.*

   (a) *Write an R function to do nearest-neighbor classification. Your function should tkae as inputs the vector to be classified, and a data frame of labeled example vectors, and should give as its output the guessed label of the new vector. You can re-use code from last time and from the solutions.* Remember to comment your code, and explain the reasoning behind it.

   ANSWER: Basically, this is almost the same as the function for similarity searching from last time. We need to find the nearest vector from a set of training vectors, which we can do with the `nearest.points` function, and then look up the label of that closest point in our vector of labels. Like last time, the function ends up being basically a wrapper for `nearest.points`.

   (b) *Test the accuracy of your nearest-neighbor classifier on the newsgroups data from the last assignment. That is, what fraction of documents does it mis-classify? (You should use IDF weighting and Euclidean-length normalization.) Include, with comments and explanations, the code you used to calculate the error rate.*

   ANSWER: There were several appropriate procedures. The code accompanying this uses the leave-one-out method, where we go over the documents and try to classify each one, using the other documents as the labeled examples.

```
> leave.one.out.error(pol.rel,pol.rel.labels,method="nn")
$error.rate
[1] 0.375

$error.indices
[1] 1 2 3 6 7 8
```

   The second output gives the row-numbers of the vectors which were mis-classified.

   The leave-one-out function in the code works for both the nearest-neighbor classifier (when give the argument `method="nn"`) and for the prototype method (`method="prototypes"`), and is easily extended to other classifiers.

   (c) *A simple mistake in the previous part would lead you to conclude that nearest neighbor classification is* always *100% accurate on* any *data set. Describe the mistake, and how to avoid it.*

   ANSWER: The problem is including the document to be classified among the labeled examples. Divide your data into training and test

parts before forming the bag-of-words data frame. Note that the leave-one-out method automatically handles this.

(d) *Write a function to do prototype classification. It should have the same inputs and outputs as your nearest neighbor classifier. (*Hint*: one way to do this is to use that function!)*

ANSWER: Again, there are several ways to do this. Using the hint lets us divide the problem into two parts: first, form prototype vectors for each class. Then, treat those prototyp vectors as labeled examples, and pass them to the nearest-neighbor classifier.

Forming the prototype vectors is just a matter of averaging all the vectors with the same class label. R provides a function (`aggregate`) to do this, but it's also not hard to write one's own, if so inclined. (The solution code shows how.) We should apply the normalization at this stage, otherwise we are just averaging word *counts*, and so we have all the problems of comparing big documents to small documents again. However, this means that when we pass off the actual classification job to the nearest-neighbor function, we have to be able to tell it not to do that pre-processing over again.

This is a trickier step than some of the others, so it's worth checking that this function is working properly. I did this by constructing some cases where I either knew the answer, or could check it working by hand. One of them was just to give th prototype function one example vector for each class (I used $(1, 0)$ and $(-1, 0)$ in two dimensions) and check that it gave the right results. A slightly more ambitious check was to give it a larger number of examples, but keeping the same means for each class as in the previous check, and seeing that it computed the right means (yes) and classified correctly (yes). As a third check, I generated some random points in two dimensions, assigned them classes depending on whether they were to the left or the right of the $y$-axis, and let it classify the points.

The last check is a little subtle and deserves an extra word. The *code* works when it does what the algorithm specifies, which may or may not be to classify correctly. (That is, whether your *implementation* works is a different question from whether your *algorithm* is right.) Running the leave-one-out function on my random data, I saw that one point was mis-classified. To see whether or not this was the correct behavior, I found the two class means (holding that point back), drew the line between them and then draw the perpendicular bisector of that line. All the points with one label were to the right of that line, and all the points with the other label were to its left, *except* for the one mis-classified point. (See figure.) Since the algorithm mis-classified that point, the code was working correctly.

(e) *Calculate the error rate of your prototype classifier.*

ANSWER: Using the leave-one-out method,

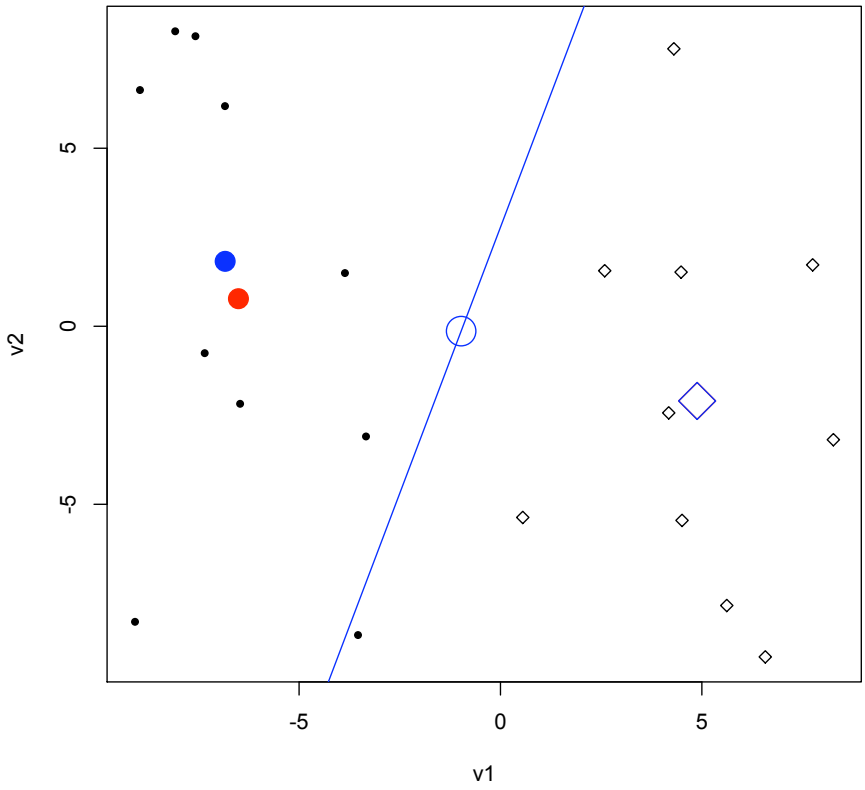


Figure 1: Checking example for the prototype classifier. Twenty random points with $v_1$ and $v_2$ coodinates uniformly distributed between $-10$ and 10. Those to the left of the $v_2$ axis (ten in all) get one class label, the dot, those on the right the other label, the diamond. The sample means of the two classes are marked in red. When the point at $(-3.5, -8.7)$ is held out, the mean of one class shifts, but not the other; the means under hold-out are shown in blue, along with the mid-point between them, and the perpendicular bisector of the line segment joining them (blue line). Since the point in question falls to the right of that line, the prototype method guesses it belongs to the diamond class, rather than the dot class. The `prototype.classifier` function classifies it in this way, so even though it's substantively wrong it's correctly implementing the algorithm.

```
leave.one.out.error(pol.rel,pol.rel.labels,method="prototypes")
$error.rate
[1] 0.3125

$error.indices
[1]  1  2  3  8 13
```

Note that the result here is very sensitive to whether or not words which appeared in only one document were omitted in making the data frame. If they're kept, the error rate of the prototype method goes up to 0.5625. (The words are rare, so they have large IDF weights, but that makes a new document seem further from the average of old ones in the same class than we'd like it to be.)