# Handout 2
# More Similarity Searching; Multidimensional Scaling

36-350: Data Mining

August 27, 2008

READING: *Principles of Data Mining*, sections 14.1–14.4 (skiping 14.3.3 for now) and 3.7.

Let's recap similarity searching for documents. We represent each document as a "bag of words", i.e., a vector giving the number of times each word occurred in the document. This abstracts away all the grammatical structure, context, etc., leaving us with a matrix whose rows are feature vectors, a "data frame". To find documents which are similar to a given document $Q$, we calculate the distance between $Q$ and all the other documents, i.e., the distance between their feature vectors. We then return the $k$ closest documents.

Today we're going to look at some wrinkles and extensions.

**Stemming** It is a lot easier to decide what counts as "a word" in English than in some other languages.[1] Even so, we need to decide whether "car" and "cars" are the same word, for our purposes, or not. **Stemming** takes derived forms of words (like "cars", "flying") and reduces them to their stem ("car", "fly"). Doing this well requires linguistic knowledge (so the system doesn't think the stem of "potatoes" is "potatoe", or that "gravity" is the same as "grave"), and it can even be harmful (if the document has "Saturns", plural, it's most likely about the cars).

**Multidimensional Scaling** The bag-of-words vectors representing our documents generally live in spaces with lots of dimensions, certainly more than three,

[1] For example, Turkish is what is known as an "aggulutinative" language, in which grammatical units are "glued together" to form compound words whose meaning would be a whole phrase or sentence in English, e.g., *gelemiyebelirim*, "I may be unable to come", *yapabilecekdiyseniz*, "if you were going to be able to do", or *calistirilmamaliymis*, "supposedly he ought not to be made to work". (German does this too, but not so much.) This causes problems with Turkish-language applications, because many sequences-of-letters-separated-by-punctuation are effectively unique. See, for example, L. Özgür, T. Güngör and F. Gürgen, "Adaptive anti-spam filtering for agglutinative languages: a special case for Turkish", *Pattern Recognition Letters* **25** (2004): 1819–1831, available from `http://www.cmpe.boun.edu.tr/~gungort/`.

which are hard for ordinary humans to visualize. However, we can compute the distance between any two vectors, so we know how far apart they are. **Multidimensional scaling** (MDS) is the general name for a family of algorithms which take high-dimensional vectors and map them down to two- or three-dimensional vectors, trying to preserve all the relevant distances.

Abstractly, the idea is that we start with vectors $v_1, v_2, \ldots v_n$ in a $p$-dimensional space, where $p$ is large, and we want to find new vectors $x_1, x_2, \ldots x_n$ in $\mathbb{R}^2$ or $\mathbb{R}^3$ such that

$$\sum_{i=1}^{n} \sum_{j \neq i} \left(\delta(v_1, v_2) - d(x_1, x_2)\right)^2$$

is as small as possible, where $\delta$ is distance in the original space and $d$ is Euclidean distance in the new space. Note that the new or **image** points $x_i$ are *representations* of the $v_i$, i.e., representations of representations.

There is some trickiness to properly minimizing this **objective function** — for instance, if we rotate all the $x_i$ through a common angle, their distances are unchanged, but it's not really a new solution — and it's not usually possible to make it exactly zero (See Sec. 3.7 in the textbook for details.) We will see a lot of multidimensional scaling plots, because they are nice visualization tools, but we will also see a lot of other **data reduction** or **dimensionality reduction** methods, because sometimes it's more important to preserve *other* properties than distances.

**Classification** One very important data-mining task is **classifying** new pieces of data, that is, assigning them to one of a fixed number of **classes**. Last time, our two classes were "about automobiles" and "about motorcycles". Usually, new data doesn't come with a class label, so we have to somehow guess the class from the features.[2] With a **nearest neighbor** strategy, we guess that the new object is in the same class as the closest already-classified object. (We saw this at the end of the last lecture.) With a **prototype** strategy, we pick out the "most representative" member of each class, or perhaps the average of each class, as its prototype, and guess that new objects belong to the class with the closer prototype. We will see many other **classifier rules**, in addition to these two, but these are ones we can apply as soon as we know how to calculate distance.

**Queries Are Documents** I promised that we could avoid having to come up with an initial document. The trick to this is to realize that a query, whether an actual sentence ("What are the common problems of the 2001 model year Saturn?") or just a list of key words ("problems 2001 model Saturn") *is* a small document. If we represent user queries as bags of words, we can use our similarity searching procedure on them. If this works, we have a search technique which find mostly-relevant things (the **precision** is high), and most relevant items are found (the **recall** is high).

[2] If it does come with a label, we read the label.

| Normalization | Equal weight | IDF weight |
|---|---|---|
| None | 83 | 79 |
| Document length | 63 | 60 |
| Euclidean length | 59 | 21 |

Table 1: Number of mis-classifications in a larger (199 document) collection of posts from `rec.auto` and `rec.motorcycles`, for different normalizations of Euclidean distance, with and without IDF weighting. (Classification is by the nearest neighbor method.)

**Inverse Document Frequency (IDF) Weighting** We are using features (word counts) to identify documents which are **relevant** to our query. Not all features are going to be equally useful. Some words are so common that they give us almost no ability at all to discriminate between relevant and irrelevant documents. In (most) collections of English documents, looking at "the", "of", "a", etc., is a waste of time. We could handle this by a fixed list of **stop words**, which we just don't count, but this at once too crude (all or nothing) and too much work (we need to think up the list).

**Inverse document frequency** (IDF) is a more adaptive approach. The **document frequency** of a $w$ is the number of documents it appears in, $n_w$. The IDF weight of $w$ is

$$IDF(w) \equiv \log \frac{N}{n_w}$$

where $N$ is the total size of our collection. Now when we make our bag-of-words vector for the document $Q$, the number of times $w$ appears in $Q$, $Q_w$, is multiplied by $IDF(w)$. Notice that if $w$ appears in every document, $n_w = N$ and it gets an IDF weight of zero; we won't use it to calculate distances. This takes care of most of the things we'd use a list of stop-words for, but it also takes into account, implicitly, the kind of documents we're using. (In a data base of papers on genetics, "gene" and "DNA" are going to have IDF weights of near zero too.) On the other hand, if $w$ appears in only a few documents, it will get a weight of about $\log N$, and all documents containing $w$ will tend to be close to each other.

Table 1 shows how including IDF weighting improves our ability to classify posts as either about cars or about motorcycles.

You could tell a similar story about any increasing function, not just log, but log happens to work very well in practice, in part because it's not very sensitive to the exact number of documents. So this is not the same log we will see in information theory, or the log in psychophysics. Notice also that this is *not* guaranteed to work. Even if $w$ appears in every document, so $IDF(w) = 0$, it might be common in some of them and rare in others, so we'll ignore what might have been useful information. (Maybe genetics papers about laboratory procedures use "DNA" more often, and papers about hereditary diseases use "gene" more often.)

— This is our first look at the problem of **feature selection**: how do we pick out good, useful features from the very large, perhaps infinite, collection of possible features? We will come back to this in various ways throughout the course. Right now, concentrate on the fact that in search, and other classification problems, we are looking for features that let us **discriminate** between the classes.

**Feedback** People are much better at telling whether you've found what they're looking for than explaining what it is that they're looking for. Queries are users trying to explain what they're looking for (to a computer, no less), so they're often pretty bad. An important idea in data mining is that people should do things at which they are better than computers and vice versa: here they should be deciders, not explainers. **Rocchio's algorithm** takes feedback from the user, about which documents were relevant, and then refines the search, giving more weight to what they like, and less to what they don't like.

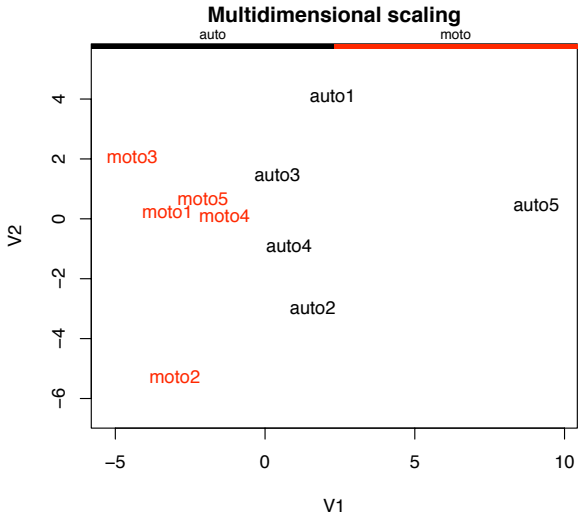
The user gives the system some query, whose bag-of-words vector is $Q_t$. The system responses with various documents, some of which the user marks as relevant ($R$) and others as not-relevant ($NR$). The system then modifies the query vector:

$$Q_{t+1} = \alpha Q_t + \frac{\beta}{|R|} \sum_{\text{doc} \in R} \text{doc} - \frac{\gamma}{|NR|} \sum_{\text{doc} \in NR} \text{doc}$$

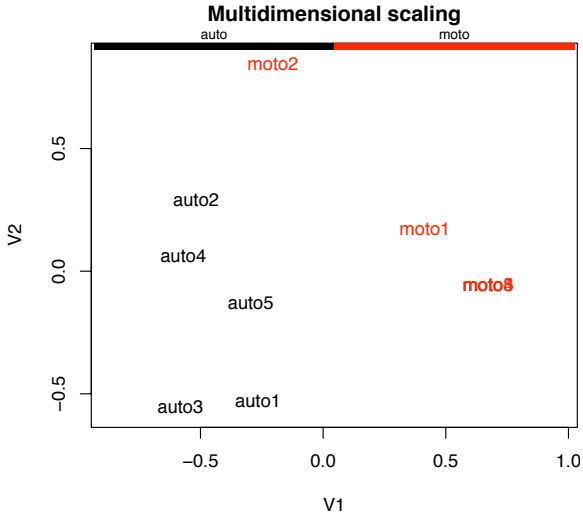
where $|R|$ and $|NR|$ are the number of relevant and non-relevant documents, and $\alpha$, $\beta$ and $\gamma$ are positive constants. $\alpha$ says how much continuity there is between the old search and the new one; $\beta$ and $\gamma$ gauge our preference for recall (we find more relevant items) versus precision (more of what we find is relevant). The system then runs another search with $Q_{t+1}$, and cycle starts over. As this repeats, $Q_t$ gets closer to the bag-of-words vector which best represents what the user has in mind, assuming they have something definite and consistent in mind.

N.B.: A word can't appear in a document a negative number of times, so ordinarily bag-of-words vectors have non-negative components. $Q_t$, however, can easily come to have negative components, indicating the words whose *presence* is evidence that the document isn't relevant. Recalling the example of problems with used 2001 Saturns, we probably don't want anything which contains "Titan" or "Rhea", since it's either about mythology or astronomy, and giving our query negative components for those words suppresses those documents.

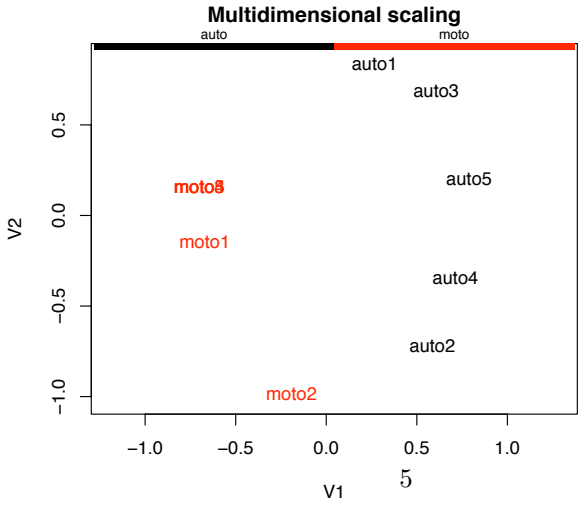
Rocchio's algorithm works with any kind of similarity-based search, not just text. It's related to many machine-learning procedures which incrementally adjust in the direction of what has worked and away from what has not — the **stochastic approximation** algorithm for estimating functions and curves, **reinforcement learning** for making decisions, **Bayesian learning** for updating conditional probabilities, and **multiplicative weight training** for combining predictors (which we'll look at later in the course). This is no accident; they are all special cases of adaptive evolution by means of natural selection.



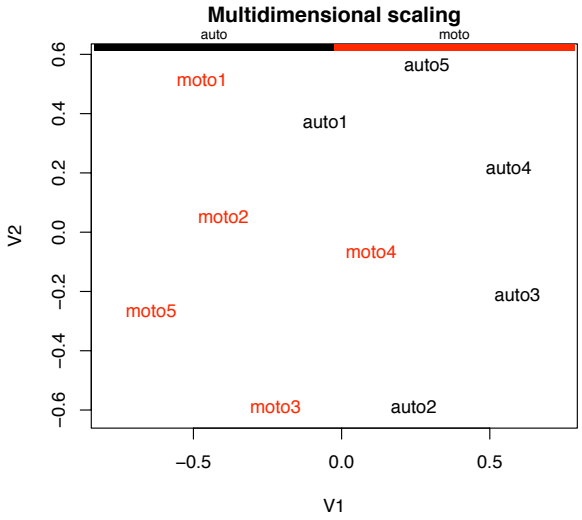

10 best words,
Un-normalized counts,
1 error (picks `moto4` for `auto3`)



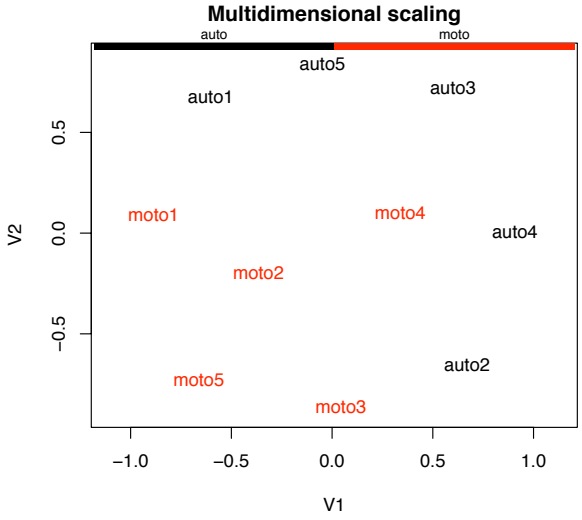

Normalized by document length,
1 error (picks `auto5` for `moto2`)



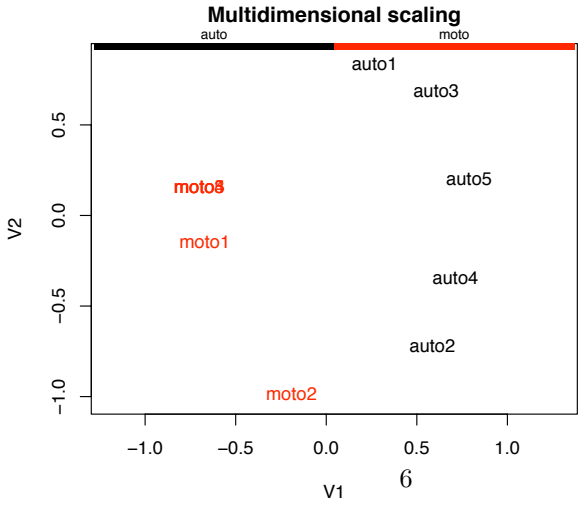

Normalized by Euclidean length,
No errors



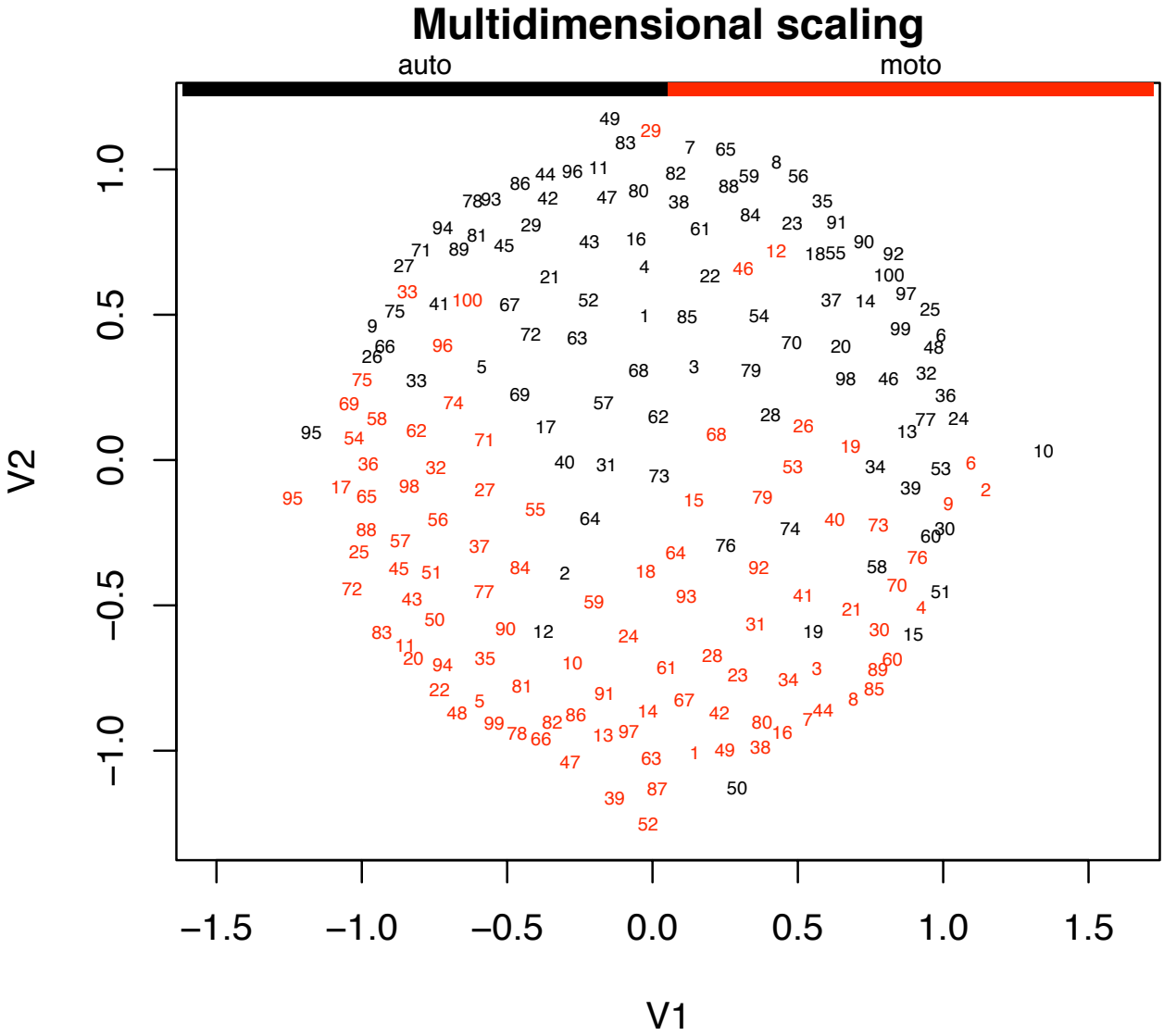

182 words, equal weighting
5 errors (`auto1,2,4`, `moto2,4`)
(as bad as guessing)



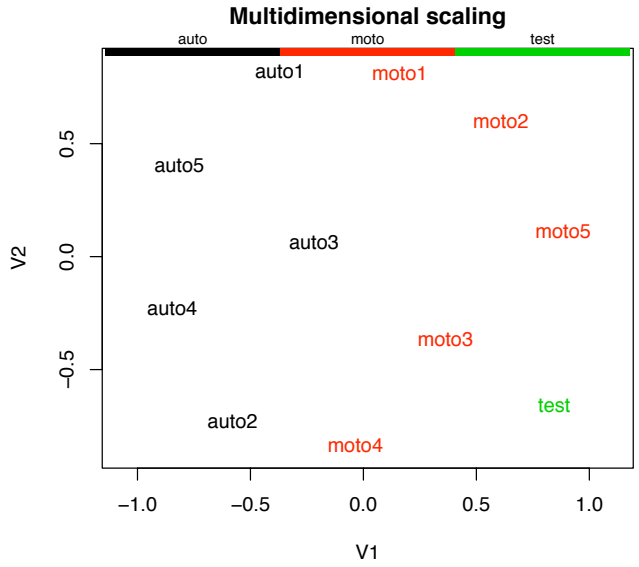

182 words, IDF weighting
3 errors (`auto4`, `moto1,4`)




10 best words (from last time)


Multidimensional scaling
auto
moto
V2
V1
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

**Nearest-neighbor method**




**Prototype method**— here prototype is the average of already-labeled documents