Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **9** Due: **3 Nov 98** 6.034 Item #

This problem set was handed out on Tuesday, November 3 and will be due on November 10th.

Important Note: At the end of this problem set you will have constructed a system that can predict a person's movie preferences from only a few questions. One important part of this prediction is having some real data on movie preferences. Before you start the problem set please go to `http://www.ai.mit.edu/courses/6.034/Movies/survey98.html` and answer the questions on the web page. This will create a database of preferences on which your predictions can be based.

Goals

- To introduce the notion of memory based reasoning.
- To learn more about nearest neighbor classification.

Background

Up to this point we have explored different techniques for encoding knowledge like rule bases, constraint graphs, and Bayes nets. Each of these techniques can be used to create AI applications. Applications are constructed by a "knowledge engineer" who analyzes the problem, selects a technique, encodes the necessary knowledge, and then selects a reasoning methodology. For example, in order to create a system to diagnose automotive problems the knowledge engineer chooses to setup a rule-base of knowledge, spends months writing down the correct set of rules, and then uses backwards chaining in order to reason from hypothetical problems. All of this requires a lot of expertise, work, and of course money.

Imagine instead a system that had access to thousands of example automotive problems[1]. Rather than building a rule base that captured all of the regularities in this data, we could

[1] This sort of data is almost certainly available. All modern cars contain computers that record many details about the function of the car – like voltage levels, and the presence of fuel. These computers can be read by plugging in a proprietary connector and then downloading. Since, for example, each GM dealer downloads 10 or 20 cars a day and there are 500 dealers, in the last year there have been over one million cars downloaded.

use the data directly to reason by "analogy". So if your car had low fuel pressure, poor voltage regulation, and rough idling we could look into the database to find all cars that had these problems (this is a quick relational database query). With luck we might find that all of these cars had the same problem (perhaps a loose fan belt??). Sounds simple doesn't it? No knowledge engineer is necessary, heck even a professor could set this up. Of course there several necessary tricks that will keep knowledge engineers in business.

In general this type of approach is called "memory-based" reasoning. Rather than carefully writing down all of the necessary knowledge and then performing "knowledge-based" reasoning, these systems simply remember everything that happened in the past and then *remember* the correct conclusion.

Memory-based reasoning is great when there is a lot of past data and when the domain is either very complex or poorly understood. So for example you might try to write a rule-based system that told you how to bet at horse races (by taking information about horses, jockeys, the race course, the time of year, previous horse performance, etc.). Of course it has been tried, and it can work to some extent. But if it were a well understood domain there would be no fun in the game. Plus, if it were that easy to make money at the races, I would have retired long ago.... Since there is plenty of race data, horse racing might be a good domain for a memory-based reasoner. Similar arguments can be made for the stock market.

One of the great commercial successes of memory-based reasoning is in character recognition. The US Post Office handles millions of letters a day. Each letter must be routed to the correct local post office based on the zip code. In the past each letter's address was read and then hand routed by a human "sorter" – first to the correct city and then to the correct office in that city. As the volume of letters went up, machines were purchased that could handle the mail, but they could not read the letter's address. Since a lot of mail has had its address printed by a computer, machines were designed that could read these addresses. This is an easy problem, computers always write neatly and reliably. But there were still many handwritten letters which were addressed by sloppy and unreliable humans (see Figure 1 for examples). One of the most successful approaches for recognizing the digits of the zip code was *nearest neighbor*, a memory based approach. In nearest neighbor, you are given a labeled set of training examples (the characters in the first row of Figure 1 would be labeled "0", in the second row "1", etc.). A new character is recognized by finding the closest labeled example and returning that label.

Tricks of the Trade

- In some cases there will be no item in the database that matches the current situation exactly. Take the characters for example. Since each character is a 16x16 binary image there are $2^{256} \approx 10^{77}$ possible characters. You can never have an exhaustive database. In such a situation you must have a mechanism where you can find the "closest" character in the database.
- In some cases you may have many examples in the database that are equally good

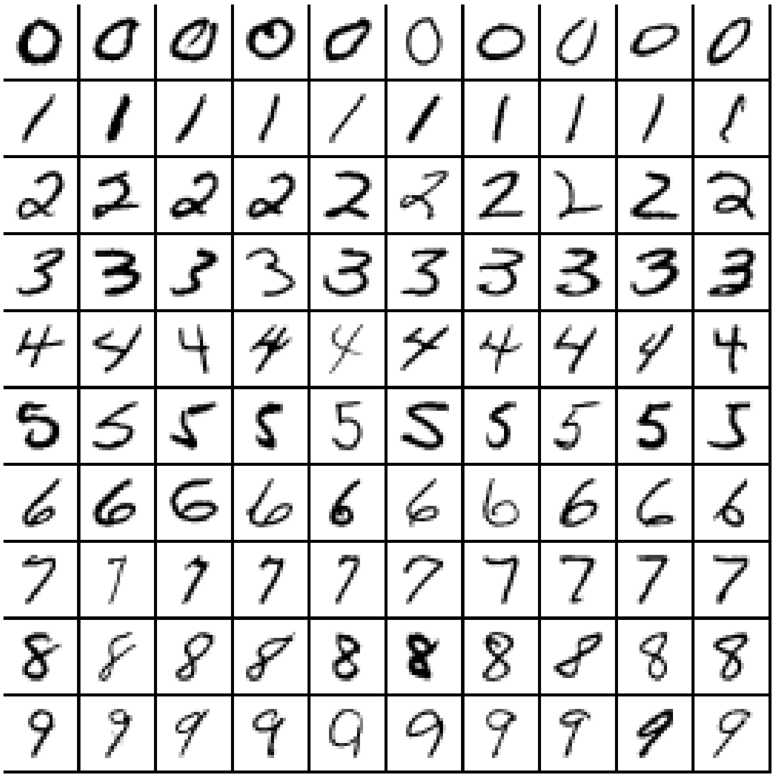

Figure 1: A collection of handwritten digits collected from the addresses of actual letters.

matches to the current situation. If they all agree in their conclusions (that the character is a "5") then there is no real problem. If they disagree you must design a voting scheme. So if 10 characters are near the current character, and 8 are labeled "5" while 2 are labeled "6", then perhaps the correct label is "5".

Problem 1 Chess Warmup

Many people now play chess on the internet with the help of something called the Internet Chess Server (ICS). ICS matches up players from across the world based on their skill levels. Perhaps 1000 games are played each day through the server. Interestingly, you can download every game that has ever been played on the server, including every board position and the move that each player made in that position. Ben Bitdiddle is intrigued, and he proposes to build a memory-based chess program using this data.

During a game, Ben's program will be asked to make moves from particular board positions. Rather than using alpha-beta or anything complex like that, he will simply search the database to find out what move has been made in the past by players in this position. His algorithm is quite simple: find the first example of this board position in the database, and then make the move that the human player made.

Assume Ben has a computer with an plenty of memory (the entire database of ICS chess games, perhaps 1,000,000 games, would fit in it) and that it is fast enough to search it quickly. Briefly give three problems with Ben's approach (no more than one or two sentences each).

Suggesting Movies

Alyssa P. Hacker has a great idea for building a system to suggest movies, "Why don't we build a web page where we ask people how much they like particular movies – like *Independence Day*. Based on a few movie preferences we can then predict what other types of movies they might like." Ben's first suggestion for solving the problem was to build a rule-based system. Of course this is a very difficult problem. In order to make the rule-base work you must somehow encode a theory of why people like particular movies. One big problem is that some people will like a movie, while others will hate it. So your rule-based theory of movie preferences must model different sorts of people.

Alyssa scoffs at the thought that anyone, besides perhaps Wittgenstein, could build such a theory. She suggests a memory-based system. "We can take data from all our friends. I'll bet that people cluster into a few different types. You know people who like action movies, or people who like art films. Some of the classes might overlap, others won't. When a new user logs in we could ask them for their ratings of a few movies they have seen. Using these preferences we could try to guess which of our friends they are most like. Then we could predict that they will like the same films as that person. It is silly to try to build a theory of movie preferences when we can use a memory-based technique instead."

Ben and Alyssa set out to test these ideas on some simulated data (their friends are working too hard on 6.034 to help them out right now...). Alyssa chooses 30 movies as a good starting point: "A Night at the Roxbury", "Antz", "Apt Pupil", "Armageddon", "Beloved", "Bride of Chucky", "Deep Impact", "Holy Man", "I Still Know What You Did Last Summer", "John Carpenter's Vampires", "Life is Beautiful", "Lolita", "Mask of Zorro", "Monument Ave.", "Next Stop Wonderland", "One True Thing", "Orgazmo", "Pi", "Pleasantville", "Practical Magic", "Ronin", "Rush Hour", "Saving Private Ryan", "Scream 2", "Soldier", "The Mighty", "The Siege", "Urban Legends", "Wag the Dog", and "What Dreams May Come".

Users will be allowed to rate these movies between 0 (terrible) and 5 (fantastic). For the simulated data the movies have been grouped into sub-categories, five in total: ACTION/ADVENTURE, DRAMA, SCI-FI, COMEDY, and SUSPENSE/HORROR. When generating test data we will assume that participants like the different movies in a category about equally. So if someone likes "Deep Impact" they will also like "Armageddon". And if they hated "One True Thing" they will also hate "Beloved". This is an assumption that underlies the generation of test data, not necessarily real data.

- ACTION/ADVENTURE: "John Carpenter's Vampires", "Saving Private Ryan", "Mask of Zorro", "Ronin", "The Siege".

- DRAMA: "Pleasantville", "Beloved", "Lolita", "The Mighty", "Monument Ave.", "One True Thing", "Practical Magic", "What Dreams May Come".
- SCI-FI: "Deep Impact", "Armageddon", "Pi", "Soldier".
- COMEDY: "Orgazmo", "Antz", "Holy Man", "A Night at the Roxbury", "Rush Hour", "Wag the Dog", "Next Stop Wonderland", "Life is Beautiful".
- SUSPENSE/HORROR: "Apt Pupil", "Bride of Chucky", "Scream 2", "I Still Know What You Did Last Summer", "Urban Legends".

When generating simulated data we will also assume that the participants fall into 6 categories: ARTY, AVERAGE-JOE, NON-VIOLENT, KIDS, CURMUDGEON, and OPTIMIST. On average ARTY people will not like ACTION/ADVENTUREs (rating them about 1.5), they will like DRAMAs (about 4.5), are not overjoyed with SCI-FI films (2.0), are a little happier with COMEDY (2.5), and they are somewhat non-committal about SUSPENSE/HORROR (3.0).

The different types of simulated people like movies in the following way:

Type	Action/Adventure	Drama	Sci-Fi	Comedy	Suspense/Horror
Arty	1.5	4.5	2.0	2.5	3.0
Average-Joe	4.5	2.0	4.0	3.5	3.0
Non-Violent	1.0	4.0	2.0	4.0	2.5
Kids	4.5	1.0	4.5	3.0	1.0
Curmudgeon	1.0	1.0	1.0	1.0	1.0
Optimist	4.5	4.5	4.5	4.5	4.5

All average-joe's are not the same (nor is any other type of person). There is natural variation in the ratings that particular people will give a movie. We will model this by having 20 examples of each type of person where each simulated person has slightly different preferences. The numbers in the table above are the mean ratings for each category; the actual preferences will have a distribution about this mean with a standard deviation of 0.5.

Code

- The simulated data that Ben and Alyssa will work with is in `sim-movie-data.scm` and `sim-movie-data-missing.scm`.
- The code for this problem set is in `ps9.scm`.

Problem 2

(For this problem you will use the data stored in `sim-movie-data.scm` which should be read into `*movie-data*` by the function `setup-data`.)

As a first test, Ben and Alyssa choose to verify that if a new user of the system is an AVERAGE-JOE then the system will correctly label them as an AVERAGE-JOE. Unfortunately it may be the case that no two AVERAGE-JOEs have exactly the same preferences. Some measure of similarity between preference sets will be necessary.

Ben and Alyssa have written some software to do this test. The only hangup is the similarity measure, which Ben and Alyssa cannot figure out how to write. Write and hand in a function called `compare` that takes two preference lists and returns a number that is small if the two are similar and is large if the two are very different. So for example, the data set contains two CURMUDGEONs:

```
((CURMUDGEON 15) ( 1.05  0.3  1.2  0.5  0.8  0.7 1.33 1.53 1.28 1.54
                   1.0  0.9 1.59  1.0  0.7 1.57  0.9  0.2 1.46  1.1
                   0.0 1.11  0.0  0.9  0.1  0.0 1.55  1.3  0.0 1.13))
((CURMUDGEON 14) (  0.7  0.0  0.7 1.53  0.5  0.8 1.76 1.48  0.7  0.3
                    0.8  1.5 1.28  0.8 1.04 1.91  0.7  0.9 1.19  0.6
                    1.52  0.7  0.7 0.03  0.9 1.43  0.0 1.49 1.15 1.15))
```

This call to `compare` should return a small number:

```
(compare '( 1.05  0.3  1.2  0.5  0.8  0.7 1.33 1.53 1.28 1.54
                   1.0  0.9 1.59  1.0  0.7 1.57  0.9  0.2 1.46  1.1
                   0.0 1.11  0.0  0.9  0.1  0.0 1.55  1.3  0.0 1.13)
         '(  0.7  0.0  0.7 1.53  0.5  0.8 1.76 1.48  0.7  0.3
                    0.8  1.5 1.28  0.8 1.04 1.91  0.7  0.9 1.19  0.6
                    1.52  0.7  0.7 0.03  0.9 1.43  0.0 1.49 1.15 1.15))
```

The comparison between a CURMUDGEON and an OPTIMIST should yield a large number.

Alyssa decides that one good way to test whether her (and your) code work properly is to simulate a set of movie ratings for each of the types of people, and then to determine which element of the large simulated dataset (which is in `*movie-data*` if you have called `(setup-data)`) most closely matches each such rating. Presumably, if the matching idea and code are right, then the closest match to the movie preferences of a curmudgeon should be from a simulated curmudgeon in the database, etc. For interpreting the output below, note that Alyssa's code generates test cases in the order listed in the table above. Each test case is generated by sampling from a normal distribution centered at the average preference of a particular type of person for a particular type of movie, first assuming that the standard deviation of the distribution is 0.5 rating points.

Test your implementation by calling `test-prob-2` and hand in the results.

Alyssa has written a program that is designed to test the limits of the nearest-neighbor algorithm called `test-prob-2-b`. It generates test cases in the same way as `test-prob-2`, but with a much larger standard deviation for the distributions. Run `test-prob-2-b` and hand in the results.

When you call `test-prob-2-b` many times in a row it returns different results, why? Should we be concerned about the fact that `test-prob-2-b` does not always return the correct result? Is it possible for `test-prob-2` on occasion to return different results? Why or why not?

Problem 3

(For this problem you will use the data stored in `sim-movie-data.scm` which can be read into `*movie-data*` by the function `setup-data`.)

Alyssa likes the results so far. In general it should be easy to classify a new participant if that person has seen all of the movies in the list. But she realizes that not every participant will have seen each and every movie on the list. Some may have seen half the movies, and some hardly any. When these people are done ranking movies, many of the preferences will be missing (we choose to specify this with the empty list `#f` or `'()`.) Is it possible to classify these people?

Ben points out that these people are the ones who may like the system best. They login and rate only a few movies. If a user can be classified correctly, then the system can recommend movies they have not yet seen.

In order to deal with missing preferences you must write a new comparison program called `compare-missing`. `compare-missing` is called with two preference lists, but the first may have missing elements. Hand in a definition for `compare-missing`.

Test your implementation by calling `test-prob-3` and hand in the results. Once again Alyssa has written a program that is designed to test the limits of the nearest-neighbor algorithm with missing data, this time called `test-prob-3-b`. Run `test-prob-3-b` and hand in the results.

Problem 4

(For this problem you will use the data stored in `sim-movie-data-missing.scm` which can be read into `*movie-data-missing*` by the function `setup-data`.)

After some serious thought Alyssa realizes that the missing data problem effects both the new participants and the people in the original database. It will be hard to find people for the database that have seen every one of these movies.

Alyssa and Ben need the definition of a new program, `compare-missing-2`, that can accept missing values in either preference list. Be careful with this definition. In some cases the preference lists will have very few preferences in common. Those comparisons should not be "rewarded" over comparisons between preference lists with many preferences in common. Briefly discuss your solution and why you think it might work.

You can test your implementation by calling `test-prob-4`. Don't worry if you don't always get the "right" answer. Hand in the results of calling `test-prob-4`. Once again Alyssa writes a program that is designed to test the limits of the nearest-neighbor algorithm with missing data, this time called `test-prob-4-b`. Run `test-prob-4-b` and show the results.

Problem 5

(Note: This problem requires a solution to Problem 4 in order for the system to work. If you have given serious thought to the problem, and can't solve it, you should feel free to contact a friend to get the solution. Please mention that you have done this in your problem set. Alternatively, you can contact your TA and get a solution to Problem 4.)

Ben and Alyssa are just about ready to try out this code on some real data before Alyssa realizes that some of the assumptions in the simulated data may not be realistic. For example we can never be sure that people will only come in one of 6 different types. Nor will it be easy to classify every type of movie.

In a flash of brilliance, Alyssa realizes that we do not need to classify each of the different types of people, we simply have to keep track of each person's preferences. One approach is to take a partial list of preferences from a new participant and then find the nearest neighbor in the database. The preferences of the nearest neighbor could then be used to suggest (or reject) movies that new participant hasn't seen. Ben thinks that this is a good idea, but it has two problems: i) the nearest neighbor may not have seen all of the movies; ii) the nearest neighbor might be a very unusual person with strange tastes. Ben proposes that the algorithm retrieve K nearest neighbors instead. The preferences of these K people are then averaged together to get the predicted preferences.

Alyssa has written an implementation of this algorithm: `predict-preferences`. She has tested it when it is run on `*movie-data*` and it works, but it doesn't work when run on `*movie-data-missing*`. Fix `predict-preferences` for Alyssa and hand in the new program. Also turn in the results of running `test-prob-5-works` and `test-prob-5-doesnt-work` (it will work after you fix Alyssa's problem).

(Extra credit: Is it possible to improve the performance of the system by weighting the average over neighbors? Perhaps neighbors which are further away should get less weight? What about neighbors which have few preferences in common with the set of query preferences? Credit will be given for a modification of the code that deals with these issues. Please include a short discussion.)

Problem 6

Now that the code works, Ben and Alyssa can test it out on some real data. We have collected data from 6.034 staff and students and placed them in the file `http://www.ai.mit.edu/courses/6.034/Movies/database` (periodically we will try to make copies of this file available in the problem set directory on Athena). They have written a program called `show-predicted-preferences` which displays the movie name next to the predicted preferences.

Try this out by typing in your preferences. Try it on your friends' preferences. Does it work? (You need not hand in any analysis here.)

The Web site also collects various kinds of information about each informant, which is also recorded in the file `database`. Would you think that finding "nearest neighbors" in the space of these personal descriptors would do a better or worse job of predicting movies you would like than when we use nearness of preferences on movie ratings? Justify your argument.

(Extra credit.) Implement a program similar to `sort-database-by-nearness` that in fact sorts by similarity of a particular person to other people who have provided rating, judged by their personal characteristics. Use that program to recommend movies for yourself. What can you conclude about the relationship between these recommended movies and the ones the earlier technique recommended? Was your argument above borne out?