A Probability-Based Approach to Solving Minesweeper

For this project I have trained MATLAB to play Minesweeper. This “training” required three steps:

1. Write a Minesweeper program that generates boards and implements the essential rules of gameplay (e.g., calculation of the numbers that tell how many mines are adjacent to a given square, “openings” when a player selects a square adjacent to no mines, and game over when a player selects a square containing a mine).

2. Write code to solve as many situations as possible using systems of linear equations.

3. Write code to determine the probability that each unknown square contains a mine (and then select the square with lowest probability of containing a mine) given a situation in which the code from step 2 gets stuck.

I’ll now elaborate on the methods I used in each of these steps.

Step 1:

The program allows user-specified board length, board width, and number of mines in minesweeper.m:

```
% Specify Board Dimensions, Number of Mines

% Beginner:
width = 8;
height = 8;
um_mines = 10;

% Intermediate:
% width = 16;
% height = 16;
% num_mines = 40;

% Expert:
% width = 30;
% height = 16;
% num_mines = 99;
```

Then, the script generateBoard.m generates a board using these parameters. Mine positions are chosen randomly using the built-in randperm() function:

```
% Randomly Place Mines
num_squares = width*height;
mine_locations_full = randperm(num_squares);
mineLocations = mine_locations_full(1:num_mines);
isMine(mine_locations) = 1;
```
The script `generateBoard.m` then calls the function `calculateNumber.m`, to calculate numbers for all non-mine squares. In order to figure out which squares border each other, the function `calculateNumber.m` calls the function `neighbors2.m`.

The game is then ready to be played. The matrix `knownNumbers` is the “user interface” and can be viewed in MATLAB’s variable editor. (To look at other matrices, such as `isMine`, would be considered cheating!) The syntax for selecting a square in the 3rd row and 2nd column is:

```plaintext
x_to_open = 3; y_to_open = 2; takeAction
```

The script `takeAction.m` calls the function `openSquare2.m`, which performs the three essential processes associated with “uncovering” a square:
1. ending the game if the square is either a mine (in which case we’ve lost) or the last safe square that needed to be opened (in which case we’ve won),
2. updating `knownNumbers` with the value of the square calculated earlier by `calculateNumber.m`, and
3. creating an “opening” via recursion if the square contains a zero.

Below is a typical beginner board mid-game (perhaps the result of uncovering the square in the 3rd row and 2nd column, which would have created an opening) alongside the Windows Minesweeper equivalent:

![Variable Editor - knownNumbers](image1)
![Windows Minesweeper](image2)

When the game ends, `openSquare2.m` displays the time elapsed since starting the game, along with a message (either “Game Over” or “You Win!”).
Step 2:

The script `locateNumbers.m` aims to solve boards to the greatest extent possible using “logic” alone. In all but a few situations, `locateNumbers.m` succeeds in solving boards to the greatest extent possible using “logic” alone, and a gameplay regime that combines `locateNumbers.m` with arbitrary guessing when `locateNumbers.m` gets stuck is sufficient to solve an admirable number of boards.

The “logic” employed by `locateNumbers.m` involves little more than the use of row reduction to solve systems of linear equations that satisfy the constraints of the visible numbers. For example, for the situation introduced above (and reproduced below), `locateNumbers.m` would solve the following system of equations:

\[
\begin{align*}
A &= 1 \\
A &= 1 \\
A + B &= 2 \\
A + B + C &= 2 \\
B + C &= 1 \\
D + E &= 2 \\
D + E &= 2 \\
E &= 1 \\
E + F + G + H &= 1 \\
G + H + I &= 1 \\
H + I &= 1 \\
I &= 1 \\
I + J &= 1 \\
I + J + K + L + M &= 3 \\
L + M + N &= 1 \\
M + N &= 1
\end{align*}
\]

In matrix form:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}
\]
The script `locateNumbers.m` stores this matrix under the name “solving_matrix” and then puts the matrix into reduced-row echelon form (under the name “solved_matrix”) using MATLAB’s built-in `rref()` function.

The script `locateNumbers.m` then looks for rows in `solved_matrix` that contain only one nonzero element in the first (number of columns – 1) columns and then checks the number contained within the last column of these rows. When the number in the last column equals one, the square corresponding to the row’s nonzero element must be a mine, and `locateNumbers.m` “flags” the square as a mine. When the number in the last column equals zero, the square corresponding to the row’s nonzero element must be safe, and `locateNumbers.m` opens the square.

Similarly, `locateNumbers.m` looks for rows in `solved_matrix` whose nonzero elements in the first (number of columns – 1) columns equal one and whose last element equals zero. In these cases, the number of mines in multiple squares sums to zero, and `locateNumbers.m` opens all of these squares.

The script `locateNumbers.m` also looks for rows in `solving_matrix` for which the number of nonzero elements in the first (number of columns – 1) columns equals the number in the last column of the row. In these cases, all squares corresponding to these nonzero elements must be mines, and `locateNumbers.m` flags them as such.

Squares that have been flagged are not included in systems of linear equations during subsequent runs of `locateNumbers.m`, and for solving purposes the numbers indicating the number of mines in adjacent squares are all reduced by the number of adjacent mines that have already been flagged.

As indicated earlier, the sum of these efforts is sufficient to identify all possible logically sound moves in all but a few situations. One situation in which `locateNumbers.m` fails is shown below, with flagged squares highlighted in red:

The 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 4\textsuperscript{th} squares in row 9 must contain exactly one mine, but the 1\textsuperscript{st}, 2\textsuperscript{nd}, and 3\textsuperscript{rd} squares must contain exactly two mines. Thus, the first square must be a mine, and the 4\textsuperscript{th} square must be safe. Although `locateNumbers.m` fails in this situation, a probability-based approach that identifies all combinations of mines and non-mines in these squares that satisfy the constraints should reach the correct conclusion.

This approach is discussed in the next section.
Step 3:

The script `calculateProbabilities.m` takes the following approach:

1. For the \( n \) unknown squares adjacent to known numbers, identify all acceptable combinations of mines and non-mines using the function `generateCombos()`.
   a. The function `generateCombos()` uses a pruning algorithm that “builds” all combinations of mines and non-mines while checking along the way to make sure each combination it is generating is not inconsistent with the known numbers. This pruning typically reduces the number of combinations being considered from \( 2^n \) to a much more tractable number.
   b. It accomplishes this by multiplying the first \( n \) columns of `solving_matrix` by the column matrix of 1s and 0s corresponding to the combination of mines and non-mines in question and comparing to the last column of `solving_matrix`.
      i. While combinations are being built, their lengths \( l \) are less than \( n \). In order to multiply the first \( n \) columns of `solving_matrix` by this combination, we must increase the combination’s length to \( n \). In doing so, we consider two scenarios: one in which all \( (n - l) \) remaining unknown squares contain mines and one in which all \( (n - l) \) remaining unknown squares contain no mine. A contradiction is found when either:
         1. Any entry of the column vector generated by the former scenario is less than the corresponding entry in the last column of `solving_matrix`, indicating that this “worst-case scenario” contains too few mines, or
         2. Any entry of the column vector generated by the latter scenario is greater than the corresponding entry in the last column of `solving_matrix`, indicating that this “best-case scenario” contains too many mines.
      (This method was proposed by Prof. Peter Doyle.)
   c. Through recursion, `generateCombos()` builds all reasonable combinations for the first unknown square, then for the first two unknown squares, then for the first three unknown squares, and so forth, until it has build all acceptable combinations for the \( n \) unknown squares in question.
   d. In the interest of time, if at any point the number of combinations being considered exceeds 1999, `generateCombos()` will suspend combination-building efforts and return the combinations it has already built, first checking to see that they are not inconsistent with the known numbers. Although these incomplete combinations may contain some combinations that would have later been discovered to be invalid, the set of incomplete combinations is still useful.
2. Calculate the relative probabilities of the possible combinations by looking at how many total mines each combination contains and finding the relative probabilities of various total quantities of mines using the known density of mines in squares that remain to be solved and a binomial distribution.
3. Split the probability of containing each number of mines equally among acceptable combinations containing that number of mines.

For example, if the only acceptable combinations have 2, 3, or 4 total mines out of eight unknown squares, and if there remain 36 mines to be identified in the 241
uncovered, unflagged squares, the binomial distribution gives a 23.7% chance that these eight squares will contain 2 mines, an 8.3% chance that these eight squares will contain 3 mines, and a 1.8% chance that these eight squares will contain 4 mines. We scale these percentages so that they sum to 100%; thus, we have a 70.0% chance of 2 mines, a 24.6% chance of 3 mines, and a 5.4% chance of 4 mines.

Let’s say our set of acceptable combinations contains one combination with 2 mines, five combinations with 3 mines, and three combinations with 4 mines. Then, the 24.6% chance of 3 mines is split equally among the five combinations with 3 mines, resulting in a 4.9% probability that each individual combination is correct. Similarly, the 5.8% chance of 4 mines is split equally among the three combinations with 4 mines. Since only one acceptable combination contains 2 mines, this combination has a 70.0% probability of being the correct one.

An illustration of the situation described here can be found on the next page.

4. Use these relative probabilities to compute a weighted average of the acceptable combinations. This weighted average gives the probability that each square in the batch of unknown squares being considered contains a mine.

5. Find the lowest and greatest probabilities among these. Any probability equal to 1 (plus or minus some small amount, due to round-off error) corresponds to a known mine (which is then flagged), and any probability equal to 0 corresponds to a square that must be safe (which is then opened). Because any incomplete combinations of mines and non-mines will err on the side of too many possibilities for a given square, there is no danger of accidentally stepping on a mine even if generateCombos() had to stop early.
   a. If we have managed to identify and open any safe squares, we have managed to avoid guessing! In this case, we return to normal game-play and run locateNumbers.m.
   b. If we have managed to identify and flag any mines, we still have not gained any information and need to continue to step 6 below (in which we decide which guess to make).

6. Use the result of step 3 to compute an expected value of number of mines in the set of unknown squares (or the subset of unknown squares analyzed by generateCombos(), as appropriate). In the example above in which there may be 2, 3, or 4 mines in the 8 unknown squares in question, the expected number of mines here is 0.700(2) + 0.246(3) + 0.054(4) = 2.354. Note that the density of this region—at least 2/8, or 0.25, is necessarily higher than the density of the unknown squares as a whole, 36/241, or approximately 0.149. Thus the density of the unknown region outside the set of eight squares in question must be lower than the board density as a whole! We can compute this expected density: (36 mines left – 2.354 mines in the unknown squares in question)/(241 unknown squares – 8 unknown squares in question) ≈ 0.144.

7. If the lowest probability computed in step 4 is less than or equal to the “generic” density calculated in step 6, select the square with this lower probability. (If multiple squares share this lowest probability, choose arbitrarily.) Otherwise, if the “generic” mine density is lower than the probability computed for any of these squares, arbitrarily select a square for which no information is known.
Summary of Solver:

Overall, the solver works as follows, starting with the upper-left square by default:

```matlab
% SOLVER: Comment these lines out if you wish to play yourself.
x_to_open = 1;
y_to_open = 1;
takeAction;
while game_in_progress == 1;
    locateNumbers;
    if game_in_progress == 1
        calculateProbabilities;
    end
end
```

The script `locateNumbers.m` calls itself repeatedly so long as it is making progress. Then, if it gets stuck, `calculateProbabilities.m` decides where to guess. A message is displayed each time a guess is made.
Verification of Probabilities:

In order to verify the calculated probabilities, we can compare observed guessing successes to expected guessing successes. There exist a number of ways to do this. I choose to compare plots of *accumulated* guessing successes, expected and observed, as a function of calculated probability (to see whether a bias exists preferentially over particular probability ranges). For a large number of guesses, if each correct guess is recorded as ‘1’ and each incorrect guess is recorded as ‘0’, then if the calculated probabilities are accurate, the sum of the 1s and 0s should closely approximate the sum of the calculated probabilities. The plot below shows results for a set of guesses recorded over a run of 10,000 games of expert (16x30 grid with 99 mines):

![Graph showing expected vs. observed accumulated correct guesses](image)

Indeed, we see that the expected and observed accumulated correct guesses agree nicely, with no clear trend. Note that these accumulated successes do not include probabilities calculated when `generateCombos()` failed to produce a full set of combinations. In this case, the observed accumulated successes systematically trail the “expected” accumulated successes (see figure on next page), as might be expected for a guessing method that sacrifices accuracy for efficiency.
The Essential Question: How well does it solve Minesweeper?

In his report “Minesweeper as a Constraint Satisfaction Problem,” Chris Studholme develops a minesweeper solver capable of solving the expert level of minesweeper at a rate of approximately 25.7% (based on a sample of 10,000 games) for “hard” rules (can lose on the first click). My literature search (which was by no means exhaustive) did not reveal any solvers with a higher success rate than this.

Based on a sample of 10,000 games, the solver I developed is capable of solving the expert level at a rate of approximately 28.6% (under the same rules). This difference is significant:

\[ H_0 : P_{\text{Drager}} - P_{\text{Studholme}} \leq 0 \]

\[ z = \frac{P_{\text{Drager}} - P_{\text{Studholme}}}{\sqrt{P_{\text{Averaged}}(1 - P_{\text{Averaged}})(1/N_{\text{Drager}} + 1/N_{\text{Studholme}})}} = \frac{0.0280}{\sqrt{(0.2715)(0.7285)(1/10000 + 1/10000)}} \approx 4.45 \]

\[ p \approx 0.0000043 \]

The reason for this difference is likely that Studholme’s algorithm does not calculate probabilities properly: rather than using the binomial distribution to determine the relative probabilities of various acceptable combinations, Studholme assumes all acceptable combinations have equal probability of being valid. (To his credit, Studholme acknowledges in his report that there is room for improvement in calculating probabilities.)

Of course, in practice minesweeper is a game in which speed is just as (or more) important than accuracy. World rankings (such as http://www.minesweeper.info/worldranking.html) are based entirely on speed, and many world-ranked minesweeper players (such as myself) have absolutely
dreadful completion rates (in my case approximately two orders of magnitude below that achieved by my solver). In this respect Studholme’s solver clearly outshines mine, finishing games in far less than one second (with Y2K technology, no less!) compared to my solver’s average 4.3 seconds per game. Nevertheless, my solver is about one order of magnitude faster than the world’s best human players, with a “high-score” (among completed games) of 2.33 seconds during its round of 10,000 games (the world record for human players is 31.13 seconds).

Future Efforts:

I remain curious as to whether it is best to take “random” guesses—when the overall board density is lower than the probability calculated for any particular mine—according to any scheme in particular. For instance, it might be best to preferentially guess corner squares over edge squares, and edge squares over inner squares, as these squares have the highest probability of revealing an “opening” (no mines in the surrounding spaces).

Additionally, there is room for progress in the pruning algorithm I have implemented here. Clearly there are quicker methods than the one I have implemented here (although this one is much better than the previous method I used, which pruned nothing, leaving us with 2ⁿ combinations to sift through). Quicker methods would also allow for higher accuracy, for as we have seen, the guesses based on probabilities calculated for an incomplete subset of unknown squares are successful less often than are guesses based on probabilities calculated for the entire set!

References:

Pruning algorithm developed with substantial assistance from Prof. Peter Doyle, Dartmouth College.

Although I developed my method of calculating probabilities independently, this honors thesis also describes a minesweeper-playing algorithm that uses a binomial distribution to calculate probabilities: http://www.minesweeper.info/articles/MinesweeperStatisticalComputationalAnalysis.pdf

(Unfortunately, no results are presented for expert-level gameplay, as the thesis instead develops a “perfect method” for solving minesweeper tractable only for tiny (4x4 and smaller) boards.)


Discussed ideas with minesweeper players at http://www.minesweeper.info/ as well as Dartmouth Computer Science majors such as Andrew Bloomgarden, Daniel Mott, and Richard Lange.

Computed p-values and z-scores and such with help from Statistics for Scientists and Engineers by William Navidi.