Machine Learning for a Computationally Complex Problem:

Reversing Conway's Game of Life

David Augustine

Contents

Foreword

This report has been created as part of an R project requirement for the completion of HarvardX's Data Science Professional Certificate program. HarvardX is a University-wide strategic initiative at Harvard University, designed to facilitate online education by faculty of Harvard for students everywhere.

Chapter 1

Introduction

Of the many reasons to which machine learning's increasing importance has been attributed, one more likely to be overlooked than the others is the fact that there exist *computationally complex* problems that, although solvable and understood, cannot be addressed in any practical or meaningful way using today's technology.

In this report, we present, analyze and model a well-known example of one such problem, known as *Conway's Game of Life*. This game is a specific type of model belonging to the more general class of models referred to as *cellular automata*, and is characterized and distinguished from the other models in this class by the particular set of rules that define it.

Any instance of Conway's GoL can be defined as a sequence of square grids of at least 9 cells, such that each cell is either living (has a value of 1) or dead (has a value of 0), and with each grid in the sequence being completely determined by the grid preceding it using the rule that if a cell is living and has 3 or 4 living cells in its neighborhood (defined as the square consisting of the 9 cells surrounding and including the cell), or is dead and has exactly 3 living cells in its neighborhood, then the corresponding cell in the same position in the next grid in the sequence will be living; otherwise, that corresponding cell will be dead.

Despite this simple rule, Conway's GoL cannot be reversed easily or quickly. Reversing the GoL by determining the grids preceding a given grid is therefore the problem we concern ourselves with in this report. Some properties of GoL grids that we should point out before proceeding are that there exist grids for which no preceding grids exist, that there are certain sequences of grids that actually repeat themselves in an infinite loop, and that there can be several preceding grids that evolve into any given subsequent grid.

It's worth mentioning that, as a class of objects widely studied in theoretical computer science, the importance of being able to quickly and efficiently solve problems involving cellular automata can be suggested by noting that many scholars have raised the question of whether or not our own universe is a cellular automaton.

Our work will be done entirely in R. We'll first attempt to solve a couple small examples of the Game of Life (GoL) using a brute-force approach, before deducing it'd be impractical to solve a collection of 50,000 larger examples of the GoL using this approach due to runtime constraints and instead resorting to two of the most basic machine learning methods known to accomplish this task. To construct our machine learning models, we'll be training them on a dataset of 50,000 25x25 GoL configurations before evaluating their performance on a second test dataset consisting of 50,000 more 25x25 GoL grids. For all examples in both datasets, the number of steps between starting and ending grids can be anywhere from 1 to 5.

Chapter 2

Methods and Analysis

2.1 Initialization

Before working with any data, we must first initialize our work environment within R by loading several different packages and defining multiple objects to be used throughout the subsequent steps. We do this by executing the following code (note that not all packages loaded are used for this project; those listed are simply the ones I generally load before doing any work in R).

```r
# Define names of packages to load for scripts. Not all are actually
# necessary

Packages <- c("bigmemory", "broom", "caret", "compiler", "caTools",
  "data.table", "doParallel", "doSNOW", "dplyr", "dslabs", "e1071",
  "fastAdaboost", "foreach", "formatR", "gam", "genefilter", "ggplot2",
  "ggrepel", "gridExtra", "HistData", "kernlab", "knitr", "Lahman",
  "lpSolve", "lubridate", "MASS", "matrixStats", "mvtnorm", "naivebayes",
  "parallel", "pdftools", "purrr", "randomForest", "ranger", "Rborist",
  "RColorBrewer", "recommenderlab", "recosystem", "reshape2", "ROSE",
  "rpart", "rpart.plot", "rtweet", "rvest", "scales", "snow", "stringr",
  "svMisc", "textdata", "tibble", "tidyr", "tidytext", "tidyverse",
  "tree", "zoo")

# Download and install any packages not already installed and then load
# them

for(p in Packages){
  if(!require(p, character.only = TRUE)){install.packages(p,
    character.only = TRUE, repos = "http://cran.us.r-project.org")}
    library(p, character.only = TRUE)
}
```

```r
# Define a function that takes a game grid in vector form and creates a
# matrix consisting of 0s and 1s, the locations of which being based on
# both the relationship between the indices of cells when game grids are
# represented as vectors and the indices of cells when game grids are
# represented as matrices as well as the relationship between the indices
# of cells when game grids are represented as matrices and the indices of
# their neighboring cells when game grids are represented as matrices

create_aux_matrix <- function(S.p1){
  S.length <- length(S.p1); S.mtrx.dim <- sqrt(S.length)
  if(S.mtrx.dim %% 1 != 0){
    print("Provided vector can't be expressed as a square matrix!")
  }else{
    M_aux <- matrix(nrow = S.length, ncol = S.length)
    for(k in 1:S.length){
      # Define index pair (i,j) of grid in matrix form corresponding to
      # index k of grid in vector form
      j <- ifelse(k %% S.mtrx.dim == 0, S.mtrx.dim, k %% S.mtrx.dim)
      i <- ((k - j) + S.mtrx.dim)/S.mtrx.dim
      # Define indices for 9 cells in neighborhood of each cell, including
      # the cell itself, in grid in matrix form
      for(J in ifelse((j - 1):(j + 1) %% S.mtrx.dim == 0, S.mtrx.dim,
        (j - 1):(j + 1) %% S.mtrx.dim)){
        for(I in ifelse((i - 1):(i + 1) %% S.mtrx.dim == 0, S.mtrx.dim,
          (i - 1):(i + 1) %% S.mtrx.dim)){
          # Define indices K of grid in vector form corresponding to index
          # pairs (I,J) of grid in matrix form
          K <- (I - 1)*S.mtrx.dim + J
          M_aux[k, K] <- 1
        }
      }
    }
    # Define a value of 0 to elements not assigned a value of 1
    M_aux[is.na(M_aux)] <- 0
    return(M_aux)
  }
}

# Define a function that takes a game grid in vector form, evolves the grid
# forward one time step, and returns the resulting grid in vector form

evolve_S <- function(S.p2){
  N <- create_aux_matrix(S.p2) %*% S.p2
  ifelse(S.p2 == 1, ifelse(N %in% c(3, 4), 1, 0), ifelse(N == 3, 1, 0))
```

```r
}

# Define a function that takes a game grid in vector form, evolves the game
# grid forward the specified number of time steps, and returns the resulting
# grid in vector form

generate_S_stop <- function(S_start.p, delta.p1){
  S_temp <- S_start.p
  for(j in 1:delta.p1){
    S_temp <- evolve_S(S_temp)
  }
  return(S_temp)
}

# Define a function that takes a game grid in vector form, that game grid's
# corresponding neighborhood sum matrix in vector form and a second game grid
# in vector form and checks the relationships between the three objects,
# returning TRUE if there are no inconsistencies between any of the objects'
# elements (assuming the objects follow the rules of Conway's Game of Life)

check_S_parent <- function(S_parent.p, S_child.p, N.p){
  all((S_child.p == 0 | (N.p %in% c(3, 4) & (N.p != 4 | S_parent.p == 1))) &
    (S_child.p == 1 | (N.p != 3 & (N.p != 4 | S_parent.p == 0))) &
    ((N.p %in% 1:9 | S_parent.p == 0) & (N.p %in% 0:8 | S_parent.p == 1) &
    S_parent.p %in% c(0, 1)))
}

# Define a function that takes a game grid in vector form, recursively solves
# it backwards the specified number of times, and then returns the first
# resulting grid ancestor that it generates as a solution in vector form

solve_S_ancestor <- function(S_descendent.p, t.p){
  # Define row indices of auxiliary matrix M that correspond to those cells
  # in S_stop that are living
  indices <- M[which(S_stop == 1),] %>%
    apply(1, function(x){which(x == 1)}) %>% as.vector() %>% unique() %>%
    sort()
  # Define the set of candidate values for each cell in parent grid, based on
  # whether corresponding cells in S_stop are living or dead
  S.range <- lapply(as.list(1:length(S_descendent.p)), function(s){
    if(s %in% indices){
      c(0, 1)
    }else{
      0
    }
  })
  # Initialize index to 1 for all cells in parent grid, so that, to start,
  # the first element of each set in S.range is assigned to the value of its
  # corresponding cell in parent grid
  S.index <- rep(1, times = length(S_descendent.p))
  # Rather than using for() loop, use repeat() loop, as number of iterations
  # of loop is unknown at execution
  repeat{
    iter_ct <<- iter_ct + 1
    # Assign values to parent grid for current iteration
    S_parent <- mapply(function(x, y){x[[y]]}, S.range, S.index)
    # Define vector of neighborhood sums corresponding to resulting
    # parent grid
    N <- M %*% S_parent
    if(check_S_parent(S_parent, S_descendent.p, N) == TRUE){
      # Record ancestor grid in log of reversal process, then either assign
      # indicator variable a value of "Y" so that function is exited or move
      # on to solving previous ancestor grid
      game_log[[t.p]] <<- list(name = paste("S_", t.p - 1, sep = ""),
        value = S_parent)
      if(t.p == 1){
        solution_found <<- "Y"
      }else{
        solve_S_ancestor(S_parent, t.p - 1)
      }
    }
    m <- match(TRUE, S.index[indices] < 2)
    # If either a solution is found or there are no more possible values to
    # try for parent grid, exit the current repeat() loop, otherwise define
    # the indices to be used to determine next candidate parent grid
    # and continue
    if(solution_found == "Y" | is.na(m)){
      break
    }else{
      S.index[indices][index(S.index[indices]) < m] <- 1
      S.index[indices][m] <- S.index[indices][m] + 1
    }
  }
}

# Define a function that takes a game grid in vector form then first
# initializes a few objects before calling on solve_S_ancestor() to either
# ultimately return the first resulting grid ancestor generated in vector
```

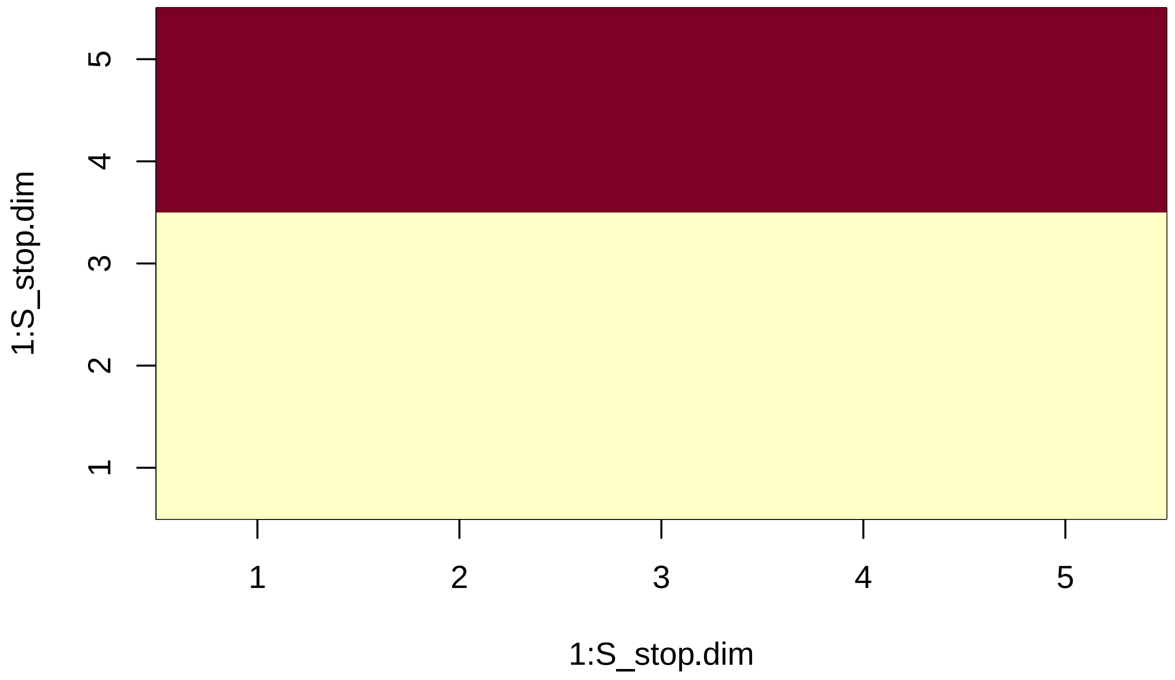
```r
# form or a message stating no grid ancestor exists for the input game grid
# and number of generations

solve_S_start <- function(S_stop.p, delta.p2){
  stopifnot(sqrt(length(S_stop.p)) %% 1 == 0)
  iter_ct <<- 1
  game_log <<- list(); solution_found <<- "N"
  M <<- create_aux_matrix(S_stop.p)
  solve_S_ancestor(S_stop.p, delta.p2)
  if(solution_found == "N"){print("No solution exists!")}
}

# Import train and test datasets into R

raw_data.dir.path <- paste(getwd(), "/Data", sep = "")

train_data.filename <- "train.csv"
train_data.path <- file.path(raw_data.dir.path, train_data.filename)
train_data <- read_csv(train_data.path)

test_data.filename <- "test.csv"
test_data.path <- file.path(raw_data.dir.path, test_data.filename)
test_data <- read_csv(test_data.path)

rm(Packages, p)
```

2.2 Brute-Force Approach

The solve_S_start() function previously defined represents our method to solve Conway's GoL using brute-force. Although the "true" brute-force approach of trial and error would involve trying every possible combination of 0s and 1s for the starting grid and then evolving the grid forward the specified number of generations to check if the resulting grid coincides with the target ending grid, our approach offers a slight improvement in two ways: it makes use of the GoL's rules to check each candidate parent grid instead of evolving each candidate parent grid forward, which saves time, and it also eliminates the need to guess the values of certain cells, which further saves time by reducing the total number of possible iterations our function must loop through.

The following code provides us with two examples of the GoL and shows the runtimes of our brute-force approach. Performing some simple arithmetic using the produced runtimes or trying out larger examples will quickly reveal that attempting to solve 50,000 25x25 GoL grids using this function would require too much time to be of any significant value in practice.

```
# Example 1:

S_stop <- c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0)

S_stop.dim <- sqrt(length(S_stop))

S_stop.mtrx <- matrix(S_stop, c(S_stop.dim, S_stop.dim), byrow = TRUE)

S_stop.image <- image(1:S_stop.dim, 1:S_stop.dim,
  t(S_stop.mtrx[S_stop.dim:1, ]))
```



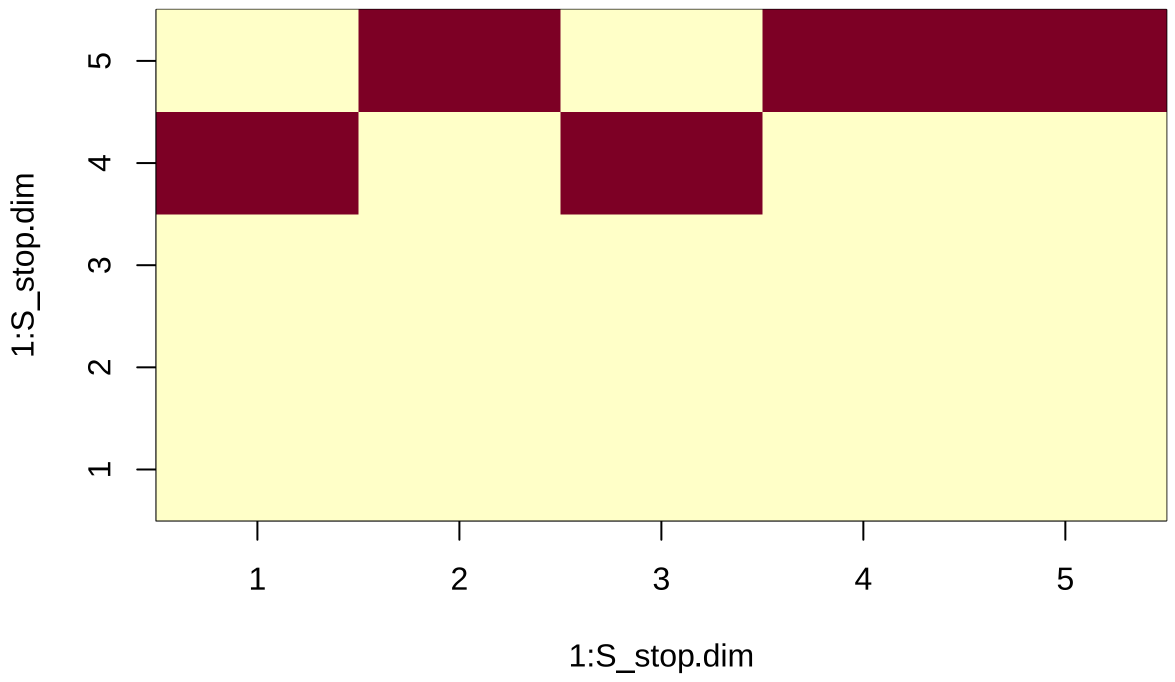
```
delta <- 1

system.time(solve_S_start(S_stop, delta)); print(iter_ct)
```

```
##    user  system elapsed
##    0.07    0.00    0.06
```

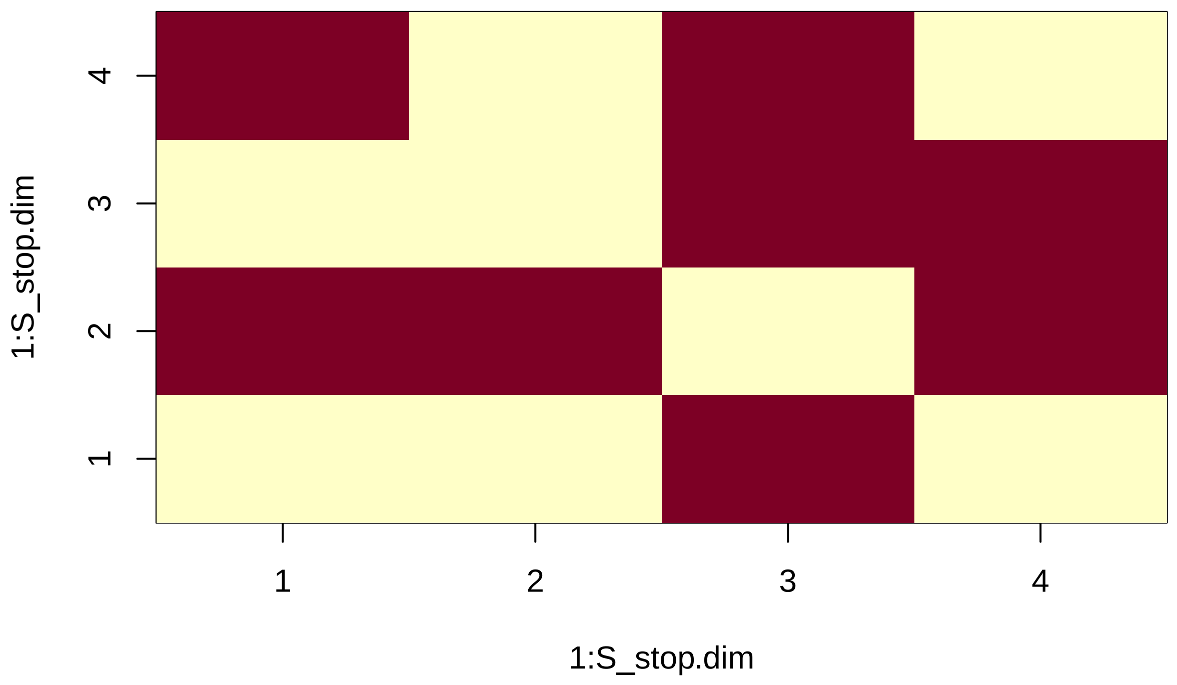
```
## [1] 188
```

```
S_start.mtrx <- matrix(game_log[[1]][["value"]],
  c(S_stop.dim, S_stop.dim), byrow = TRUE)
```

```
S_start.image <- image(1:S_stop.dim, 1:S_stop.dim,
  t(S_start.mtrx[S_stop.dim:1, ]))
```



```
# Example 2:

S_stop <- c(1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0)

S_stop.dim <- sqrt(length(S_stop))

S_stop.mtrx <- matrix(S_stop, c(S_stop.dim, S_stop.dim),
  byrow = TRUE)

S_stop.image <- image(1:S_stop.dim, 1:S_stop.dim,
  t(S_stop.mtrx[S_stop.dim:1, ]))
```

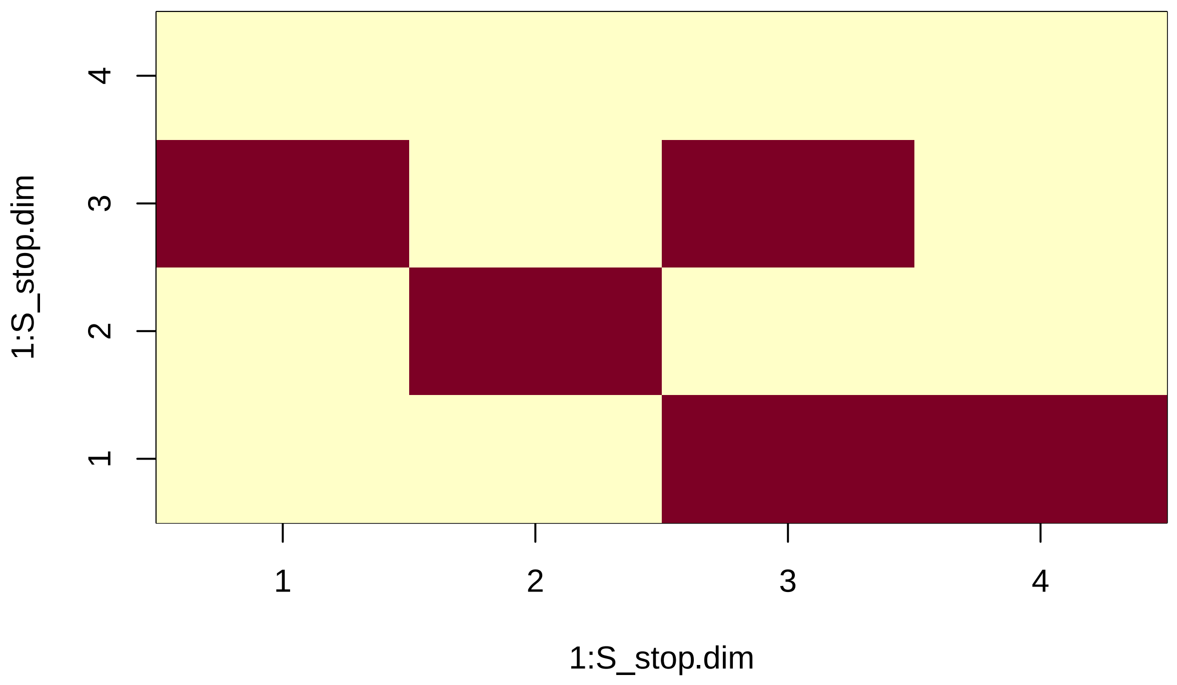


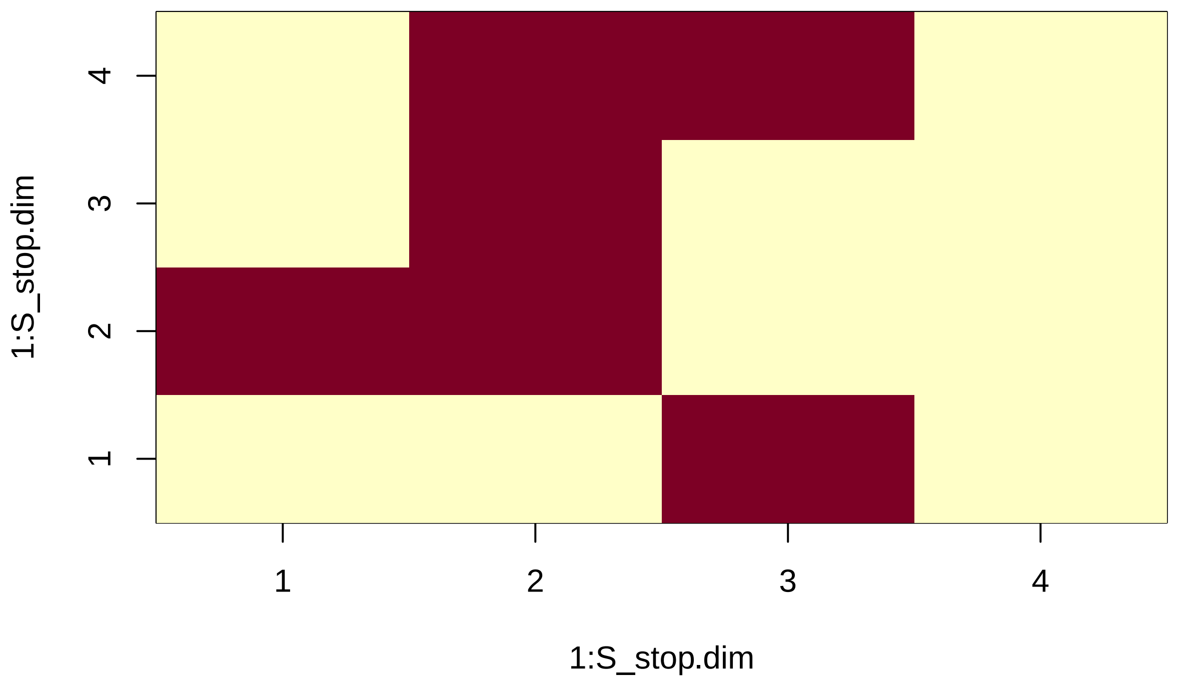
```r
delta <- 5

system.time(solve_S_start(S_stop, delta)); print(iter_ct)
```

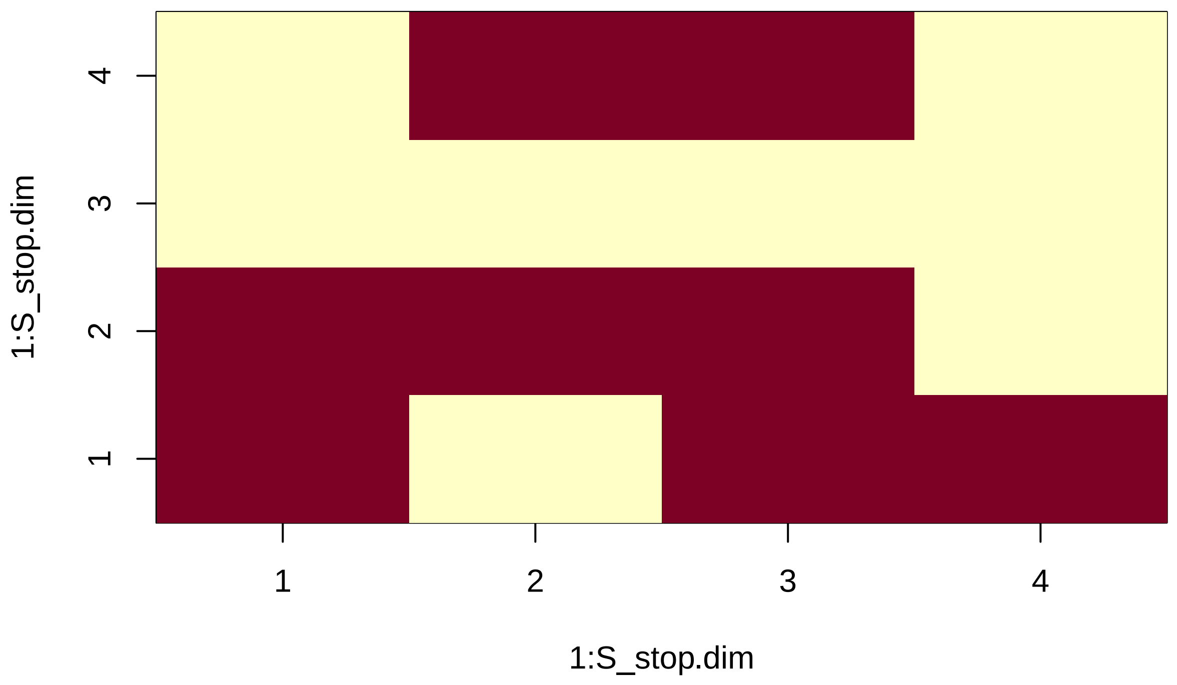
```
##    user  system elapsed
##    7.31    0.19    7.50
```

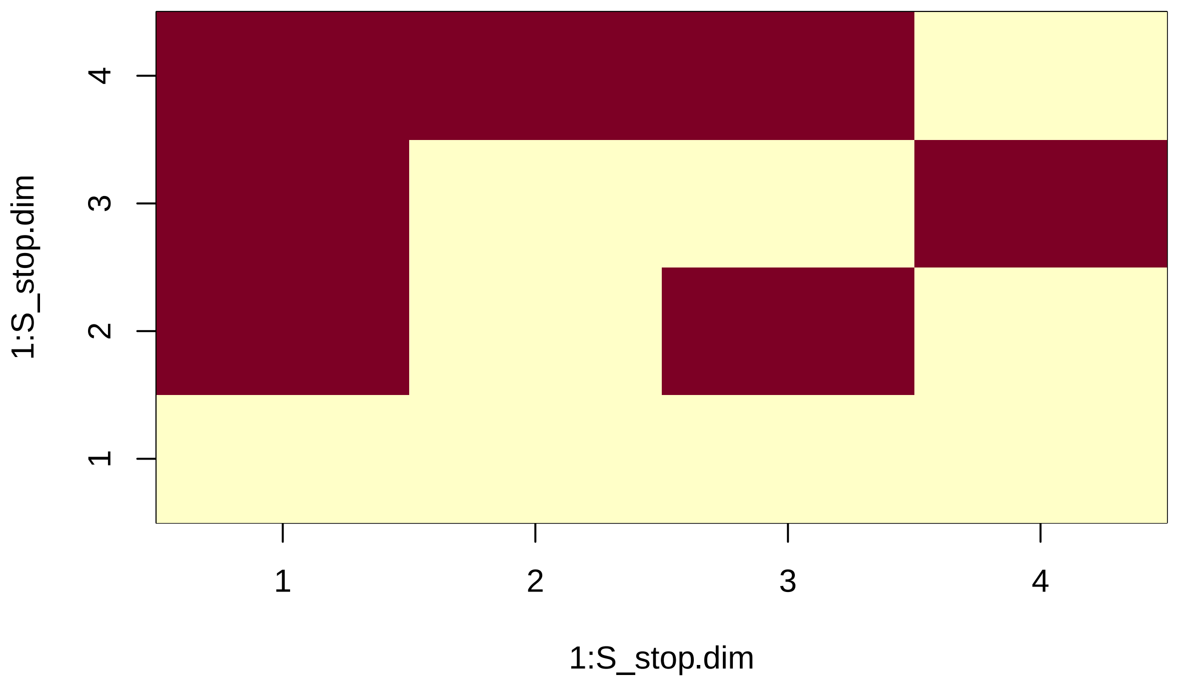
```
## [1] 144672
```

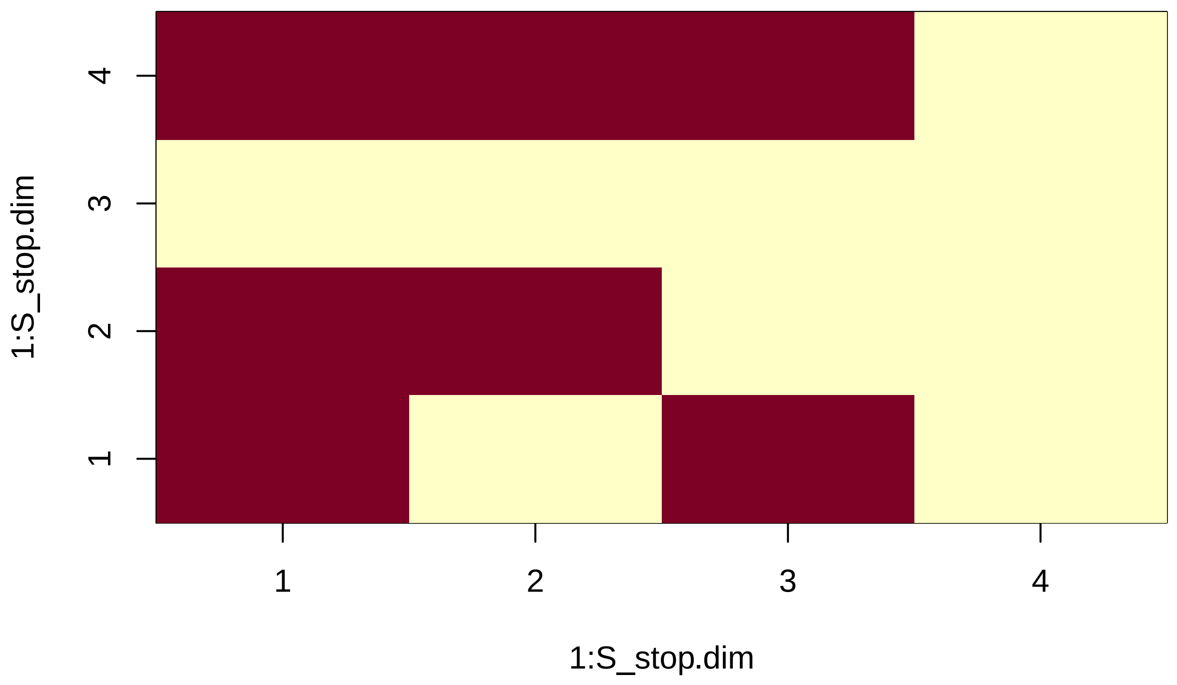
```r
for(j in 1:delta){
  S_start.mtrx <- matrix(game_log[[j]][["value"]],
    c(S_stop.dim, S_stop.dim), byrow = TRUE)
  S_start.image <- image(1:S_stop.dim, 1:S_stop.dim,
    t(S_start.mtrx[S_stop.dim:1, ]))
}
```


1:S_stop.dim
4
3
2
1
1
2
3
4
1:S_stop.dim


1:S_stop.dim
1
2
3
4
1:S_stop.dim
1
2
3
4


1:S_stop.dim
1
2
3
4
1:S_stop.dim
1
2
3
4


1:S_stop.dim
1
2
3
4
1
2
3
4
1:S_stop.dim



2.3 Machine Learning Methods

The following code trains and generates our two machine learning models and then evaluates them on our test dataset.

```r
# Train Naive Bayes model

p_y_hat <- train_data %>% dplyr::summarize(across(start_0:start_624, mean))

S_stop.N.indices <- list(); p_x.y_hat <- list(); data.stratified <- list()

for(k in 1:625){
  # As was done in function definition for define_aux_matrix(), determine
  # indices (i,j) and (I,J) relating index for each cell and indices for
  # cells in its neighborhood

  j <- ifelse(k %% 25 == 0, 25, k %% 25); i <- ((k - j) + 25)/25
  J <- ifelse((j - 1):(j + 1) %% 25 == 0, 25, (j - 1):(j + 1) %% 25)
  I <- ifelse((i - 1):(i + 1) %% 25 == 0, 25, (i - 1):(i + 1) %% 25)

  # Define vector of indices for neighboring cells for kth cell
  S_stop.N.indices[[k]] <- plyr::mdply(expand.grid(x = I, y = J),
    function(x, y){(x - 1)*25 + y}) %>% .[3] %>% pull()

  # Define subset of data corresponding to kth cell that includes the
  # observations of the kth starting grid cell value, delta, the kth
  # stopping grid cell value, and the corresponding neighborhood sum of
  # cells in the stopping grid

  data.stratified[[k]] <- train_data %>% dplyr::select(c(delta, 2 + k,
    627 + S_stop.N.indices[[k]]))
  data.stratified[[k]] <- data.stratified[[k]] %>%
    mutate(X_3 = data.stratified[[k]] %>% as.matrix() %>% .[, 3:11] %>%
    rowSums()) %>% dplyr::select(c(2, 1, 7, 12)) %>%
    setNames(c("Y", "X_1", "X_2", names(.)[4]))

  p_x.y_hat[[k]] <- list()

  # Given above defined subset of data pertaining to kth cell, define
  # PMFs for X|Y = 0 and X|Y = 1

  for(m in 0:1){
    p_x.y_hat[[k]][[paste(m)]] <- data.stratified[[k]] %>%
      filter(Y == m) %>% dplyr::select(-Y) %>% group_by(X_1, X_2, X_3) %>%
      summarize(p = n()/nrow(.), .groups = "drop")
      %>% arrange(X_1, X_2, X_3)
  }
}

# Define predictions for test data

X.test <- list(); p_y.x_hat.test <- list()

for(k in 1:625){

  # Define subset of predictors pertaining to kth starting grid cell

  X.test[[k]] <- test_data %>%
    dplyr::select(c(delta, 2 + S_stop.N.indices[[k]]))
  X.test[[k]] <- X.test[[k]] %>% mutate(X_3 = X.test[[k]]
    %>% as.matrix() %>% .[, 2:10] %>% rowSums()) %>%
    dplyr::select(c(1, 6, 11)) %>%
    setNames(c("X_1", "X_2", names(.)[3]))

  # Define PMF values for Y|X = x for starting grid cells in test dataset
  p_y.x_hat.test[[k]] <- p_y_hat[[k]]*(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p)/
    (p_y_hat[[k]]*(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["1"]], by = c("X_1", "X_2", "X_3")) %>% .$p) +
    (1 - p_y_hat[[k]])*(X.test[[k]] %>%
    left_join(p_x.y_hat[[k]][["0"]], by = c("X_1", "X_2", "X_3")) %>% .$p))
}

p_y.x_hat.test <- bind_cols(p_y.x_hat.test) %>% as.matrix()
p_y.x_hat.test[is.na(p_y.x_hat.test)] <- 0
colnames(p_y.x_hat.test) <- paste("p_y_", 1:625, ".x_hat", sep = "")

Y_hat.Bayes.test <- ifelse(p_y.x_hat.test >= 0.5, 1, 0)
colnames(Y_hat.Bayes.test) <- paste("Y", 1:625, "hat", sep = "_")

# Evolve predicted starting grids using Naive Bayes model the appropriate
# number of time steps to compare resulting grids to provided stopping grids

Y_hat.Bayes.test.evolved <- bind_cols(test_data[2], Y_hat.Bayes.test) %>%
  apply(1, function(x){
    generate_S_stop(x[-1], x[1])
})

# Evaluate Bayes model on test data

Bayes_model_MAE <- mean(abs((test_data %>% dplyr::select(-c(1:2)) %>%
  as.matrix()) - t(Y_hat.Bayes.test.evolved)))

# Train logistic regression models

Y_hat.logistic.test <- list()

# Define logistic regression model for each unique starting grid cell and
# predictions for test data

for(k in 1:625){
  glm_temp <- data.stratified[[k]] %>%
    glm(Y ~ X_1 + X_2 + X_3, data = ., family = "binomial")
  p_hat.logistic_temp <- predict(glm_temp, X.test[[k]], type = "response")
  Y_hat.logistic.test[[k]] <- ifelse(p_hat.logistic_temp >= 0.5, 1, 0)
}

Y_hat.logistic.test <- bind_cols(Y_hat.logistic.test) %>% as.matrix()
colnames(Y_hat.logistic.test) <- paste("Y_", 1:625, "_hat.logit", sep = "")
```

```r
# Evolve predicted starting grids using collection of logistic regression
# models the appropriate number of time steps to compare resulting grids
# to provided stopping grids

Y_hat.logistic.test.evolved <- bind_cols(test_data[2],
  Y_hat.logistic.test) %>% apply(1, function(x){
    generate_S_stop(x[-1], x[1])
})

# Evaluate logistic regression models on test data

logistic_model_MAE <- mean(abs((test_data %>% dplyr::select(-c(1:2)) %>%
  as.matrix()) - t(Y_hat.logistic.test.evolved)))
```

Chapter 3

Results

Running the above code, we can see that our Naive Bayes model results in a *mean absolute error* of roughly 0.14206, whereas our logistic regression models result in an MAE of roughly 0.1396.

If we had randomly guessed at each cell's value, we would have expected an MAE of around 0.5. Although there is some degree of error in our predictions for the test dataset, the fact that we were able to estimate 50,000 25x25 GoL grids using two different machine learning methods in less time than it would take us to solve a single 25x25 grid using the previously defined brute-force approach function illustrates how even the most basic machine learning methods can prove to be of great value when dealing with problems that are inherently computationally complex.

Chapter 4

Conclusion

4.1 Summary

This report illustrates one of the more likely to be overlooked reasons why machine learning is of great importance: some problems are just too computationally demanding to address otherwise.

4.2 Limitations

With that being said, it's worth noting that we could have improved both our brute-force approach as well as our machine learning models.

For our brute-force approach, we could have utilized parallel processing methods to solve each grid simultaneously from a different starting point and by going in a different direction, stopping our function once the first solution is found.

For our machine learning models, we could have chosen more efficient algorithms for this type of problem. For instance, we could have used *k-nearest neighbors*. We also could have improved both models by expanding the predictive cells pertaining to each starting grid's cell for observations with delta > 1 by recursively including the cells surrounding each preceding grid's cells.

4.3 Future Work

Cellular automata theory is currently an active area of study among computer scientists and will continue to be at the forefront of theoretical computer science for the foreseeable future.