DOTA: In Depth Details of SFI CSSS 2016 Work

Anne L. Sallaska

July 27, 2016

1 Summary of Files

The following describes an overview of R scripts that were written for this project.

1.1 Important Scripts

- RawProcessMulticore.R (and the non-multicore RawProcess.R): converts either raw binary files or converted json files into zipped time-series csv files for each raw game.
- Raw.R: processes the zipped csv files for SFIHMM. Clusters variables to determine states, plots variables, video of evolution of positions.
- DistanceToState.R: either reads in the zipped csv files or simply takes a data frame output of Raw.Process and converts the relative distances into state variables based on a threshold.
- AggregateGameCluster.R: reads in a battery of zipped csv files, aggregates over specified time unit for each game, clusters positions on all given games. Produces sequence output files for SFIHMM as well as a data frame of all games glued together. It also analyzes the output sequences of the model, when comparing to game states, determined from the gameDetails.json, but, more importantly, from the statistics determined directly from the raw data, such as difference in accumulated XP for the winning and losing teams. Correlation plots are also produced. Average positions for each kmeans cluster are plotted.
- seq.R: reads the output of SFIHMM (Viterbi path sequence and two module sequences) and produces a combined data frame along with plots of the observed states, hidden states, and both levels of modules. This also plots distributions of dwell times and a network model of the calculated transition matrix. KL divergence is also calculated between teams of a single game.

2 Data Processing for Raw Game Data: RawProcessMulticore.R

2.1 Run Instructions

Steps to convert files for raw game data into zipped csv files from the terminal. Converts into a time series either 1) each raw binary file in the form `match_id.dem`, or 2) each converted json file in the form `match_id.dem.results`. Final format is a unique time stamp with all events that occur at that step for each row of data. Produces `match_id.dem.results`.csv.zip files (and `match_id.dem.results` json files, if the relevant java convert flag is set to "T"). Details in the following subsection. Statistics for each game (duration, radiant win, league ID, start time, lobby type, game mode, player ID) are also wrapped into a lookup table, `gamestats.csv`.

- `Rscript RawProcessMulticore.R directory_for_general_files root_directory_path n_cores convert_flag version`

- *directory_for_general_files*: directory where extra files are (heroes.json, gameDetails.json, gamestats.csv, full-dota2-exp-1.2.jar).
- *root_directory_path*: root directory where files to be converted are located. All files with the correct file name structure in **each subfolder** will also be converted.
- *n_cores*: number of desired cores to parallel process.
- *convert_flag*: (if applicable) flag to set conversion from binary `match_id.dem` to json `match_id.dem.results`. If "T", `match_id.dem` files will be converted with the version given in the next input argument.
- *version*: (if applicable) java full-dota2-exp-X.jar version X. (e.g., "1.2").

- Note: It is not necessary to put in arguments for the flag and jar version if no binary files will be converted.
- If files are converted from binary (java -jar full-dota2-exp-1.2.jar ../data/raw/662332063.dem > ../data/raw/662332063.dem.results)
  - For each `match_id.dem` file in the directory and subdirectories that does not have an already converted file, the script produces `match_id.dem.results` json files in which each row is a single event in the raw game. Multiple events at the same time produce multiple rows with different tags.
  - We have two versions of the parser. Some games may need additional parsers, as not all elements are returned (e.g., initial TI5 games, pre Main and Playoffs do not have gold parsed out with v1.2). This script will convert files without all the currently required variables into the json format but will not process the time series.
- Pre-processing of each `match_id.dem.results` file includes removing extra junk at the beginning of each file that the java parser adds and adding the match_id to each event in order to provide easier sorting capabilities in mongo or R.
- Statistics for each game (duration, radiant win, league ID, start time, lobby type, game mode, player ID) are also wrapped into a lookup table at `directory_for_general_files/gamestats.csv`.

2.2 Discussion of Main Code

From the input path, this script processes batches of matches for the given input directory and all subdirectories. This produces a data frame for each game. Each row is a unique time step (in game "ticks") and includes the following columns for each hero on each team:

- match_id
- tick
- time
- x (absolute horizontal position)
- y (absolute horizontal position)
- norm.x (positions normalized to center of mass of the team)
- norm.y (positions normalized to center of mass of the team)
- d.hero$_j$_hero$_j$ (relative distance from hero$_i$ to hero$_j$)
- avgDist (for each team)

- KillsHero
- KillsOther
- KillsTower
- Deaths
- DAMAGETakenFromHero
- DAMAGETakenFromOther
- DAMAGEDealtToHero
- DAMAGEDealtToOther
- XP
- GOLD

For the variables that accumulate (most, excluding position), both the net value at the time step and the accumulated value are included. For gold, the accumulated amount may decrease if gold is taken away (e.g., during a fight). Gold used for purchases are not included but can be if deemed relevant. Abilities and items purchased or used were not included in this first pass. These could be easily added.

General steps for a single game:

1. Read in `match_id.dem.results` json file or convert `match_id.dem` binary file, if the flag is set and if the converted file does not already exist.
2. Change names of tags for simplicity (e.g., "DT_DOTA_Unit_Hero_" removed from position tags)
3. Parse out which hero is the main attacker and/or main receiver for each event.
4. Read in aggregated gameDetails.json, heroes.json files in order to figure out which team is the Radiant and which is the Dire, as the boolean win is recorded for the Radiant, and to add statistics to the main data frame.
5. Fill out data frame of values listed above for variables that change throughout the game.
6. Add normalize positions to be relative to the center of mass of the team (while retaining the absolute positions "x", "y").
7. Remove initial rows that have zero positions for heroes. This also removes initial net gold for some heroes, which needs to be added to the first row where all hero positions are non-zero.
8. Add overall game statistics to the data frame.
9. Export the data frame to a csv, zip it, and remove the csv.

Because this script utilizes multicore machines, the above will occur simultaneously for a number of games equal to the total number of cores indicated in the run statement.

3 Preparation for SFIHMM

If deterministic positions, run DistanceToState.R to output the sequences format that SFIHMM requires. If clustering one game, use Raw.R. If clustering multigames and aggregating over time, use AggregateGamesCluster.R.

Running hmm from terminal:

1. HMM: ./parallel_glue.rb cores iterationspercore seqfilename.seq
2. Viterbi path reconstruction: ./hmm -v seqfilename.seq outputfilename > seqfilename_hiddenstates.seq
3. Two module path: ./hmm -m seqfilename.seq outputfilename > seqfilename_M.seq
4. Multimodule path: ./hmm -M seqfilename.seq outputfilename > seqfilename_M_1.seq

4 Results

What the plot shows is an aggregate of 49 raw games:

- Time series of games were concatenated together in a long sequence string with a dummy state character marking the end of each game
- Dashed vertical lines indicate a new game
- Only player locations were taken into account
- States were determined from a kmeans clustering of all 20 player positions (10 heroes, each with x, y)
- 25 clusters were chosen. This may not be an ideal number but was used because that?s what the ideal number of clusters had been in the past, so picked it for consistency. It explains something like 50% of the variance. Clearly, more exploration is necessary here (Simon and I discussed using the Gaussian mixture model to better determine cluster number and membership). See attached plot for example of what the average position is for each hero in each cluster. Simon really is pushing for clustering, as he believes our deterministic states may miss some hidden structure.
- x and y positions were normalized. First the x CM (center of mass) of all 10 players was determined, followed by y CM as well as standard deviations from this mean center of mass. So each $x_{i,norm}$ = (x_i - x_{CM}) / sd_{CM}. This center of mass was determined for each time step.
- Time has been aggregated for both 10s and 60 s time intervals (separate plots) in order to make the code run faster and because the 60 ms time scale of the ticks is just way too short for anything really interesting to happen. For each aggregation, an average position was determined for each hero and each coordinate. These average positions are then renormalized as discussed above.
- For each game, a completely new set of heroes is chosen, potentially. The data structure for each time stamp is: match_id, time, hero1.x.team1, hero2.x.team1, hero5.y.team2. So the question of how you concatenate games when hero1 may not correspond to the same hero, and you need to maintain this structure. I first did this randomly, keeping teams separate. Then I ordered by gold accumulated within a team. Then added in the win (i.e., team that won has the first 1-10 position columns, followed by the losing team, each ordered in descending order of accumulated gold.). There does not seem to be a strong association with any of these in terms of the overall final state structure (i.e., random ordering and gold/win ordering produced same patterns). The attached plots show the gold/win ordering for the 60 s aggregated plot and only gold for the 10 s (takes much longer to run).

I think the 60 s plot is the most interesting. At first, both seem to be a mess, but if you look closer at the patterns, some do emerge: 1) For the two state module, it seems a game is either in one state or another. There is not very much switching between modules at all, appearing to be a constant dependent on the game details. I plan to look into if this is correlated with anything later today. 2) For the multi-state module, there are many games that switch between module 2 and module 18.

I plan to do some more analysis on the state oscillations. Now, for the 10 s aggregated results, this two state oscillation between games does not seem to hold. For this plot, you can see many more module switches within games. For the others, there is so much data displayed that it is hard to pull anything more off the graphs. More analysis is needed.

So, in conclusion, I think the two state module system may be our indicator of underlying strategy and when it changes. I will be curious to see the SFIHMM that find intermediate state changes, and if it reveals different structure.

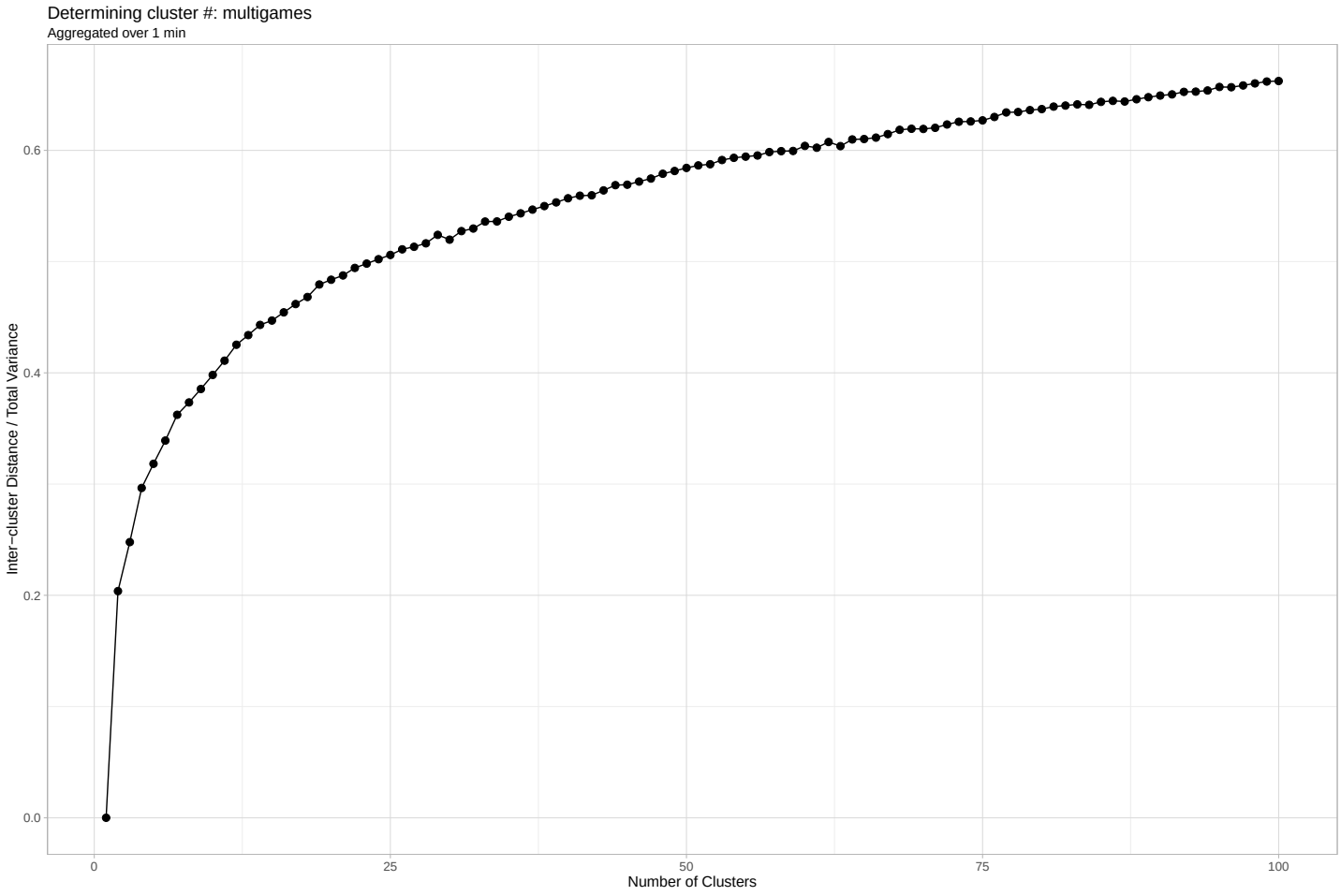

Figure 1: Average positions within clusters for 49 games from TI5.

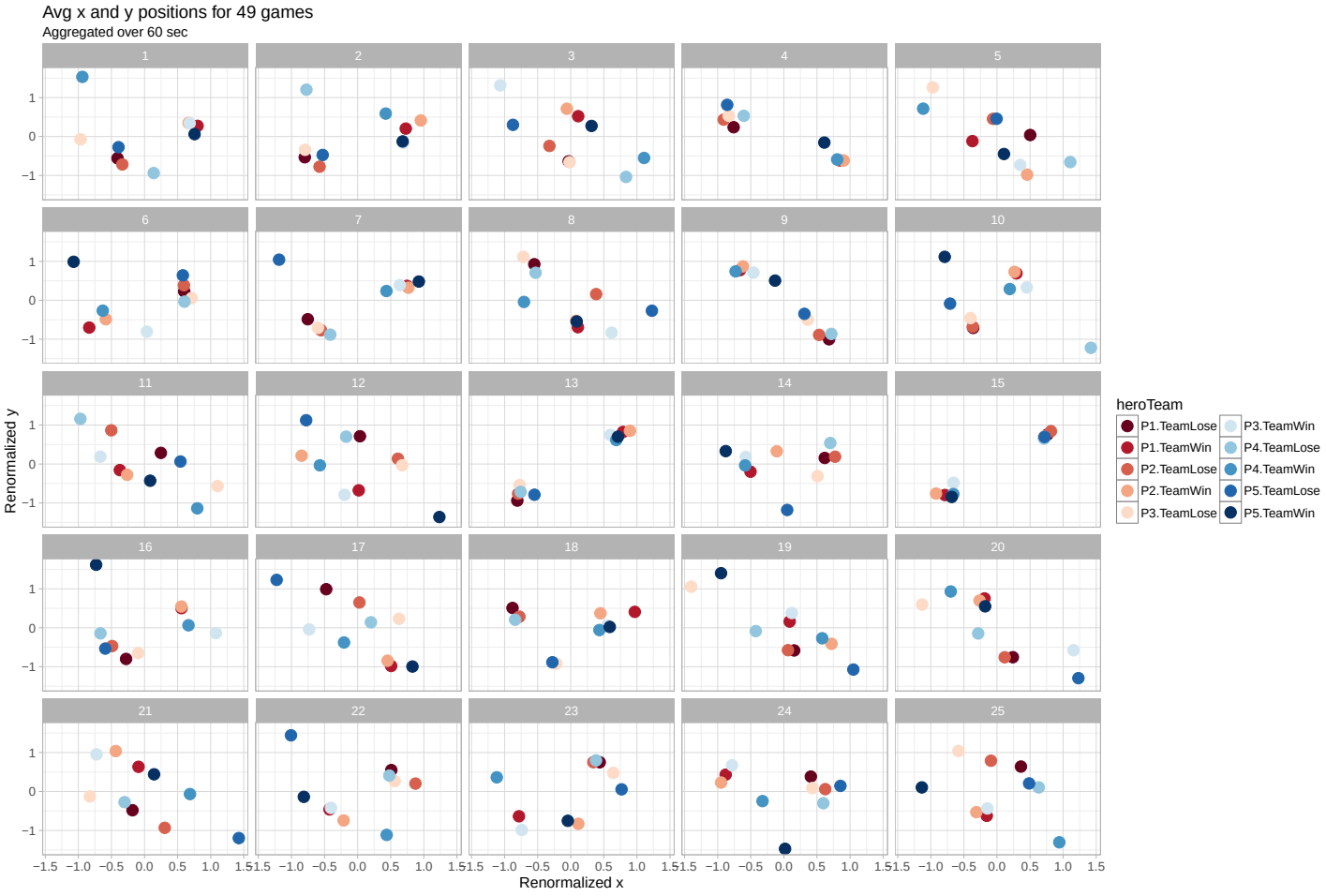

Figure 2: Average positions within clusters for 49 games from TI5.

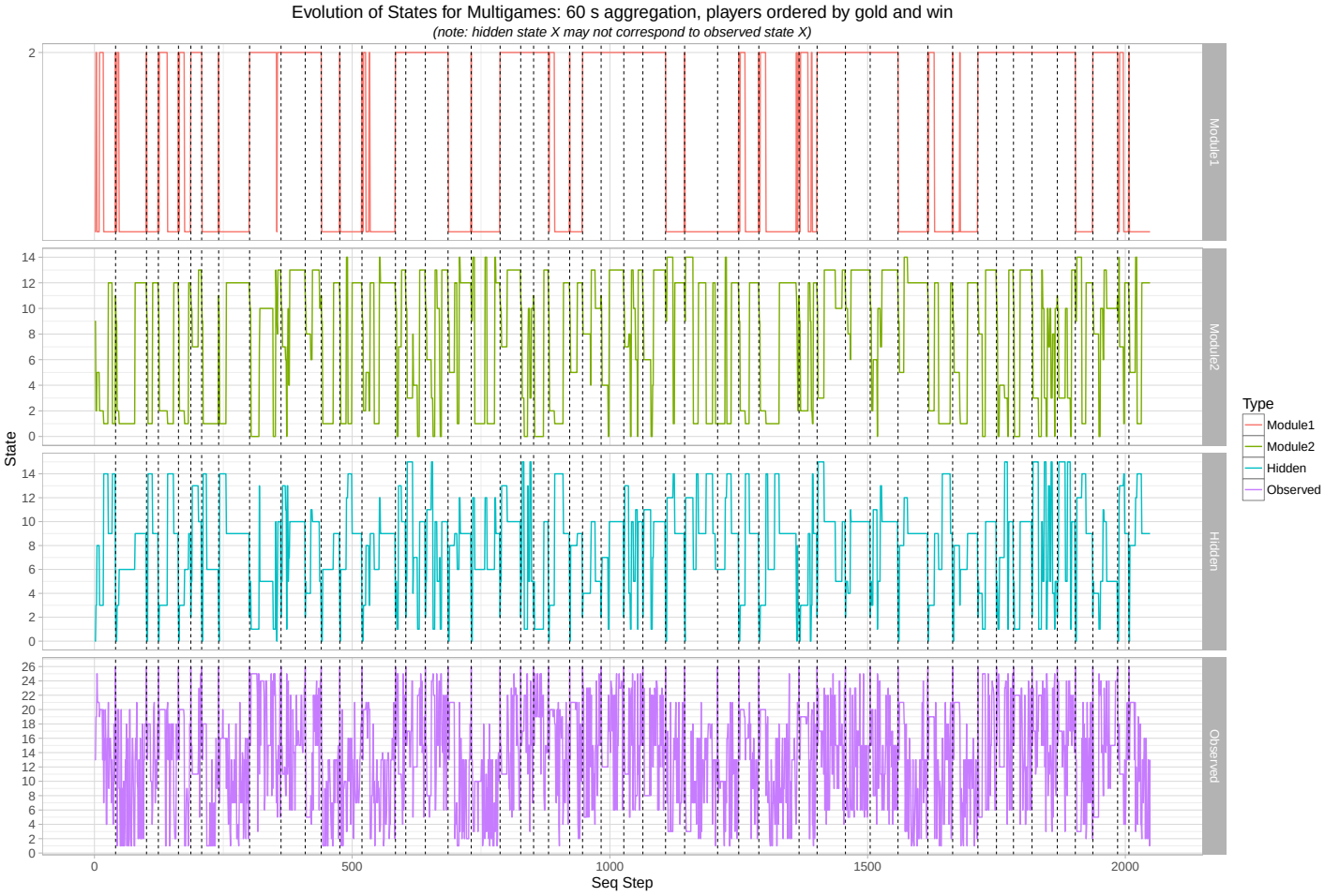

Figure 3: 49 games from TI5.

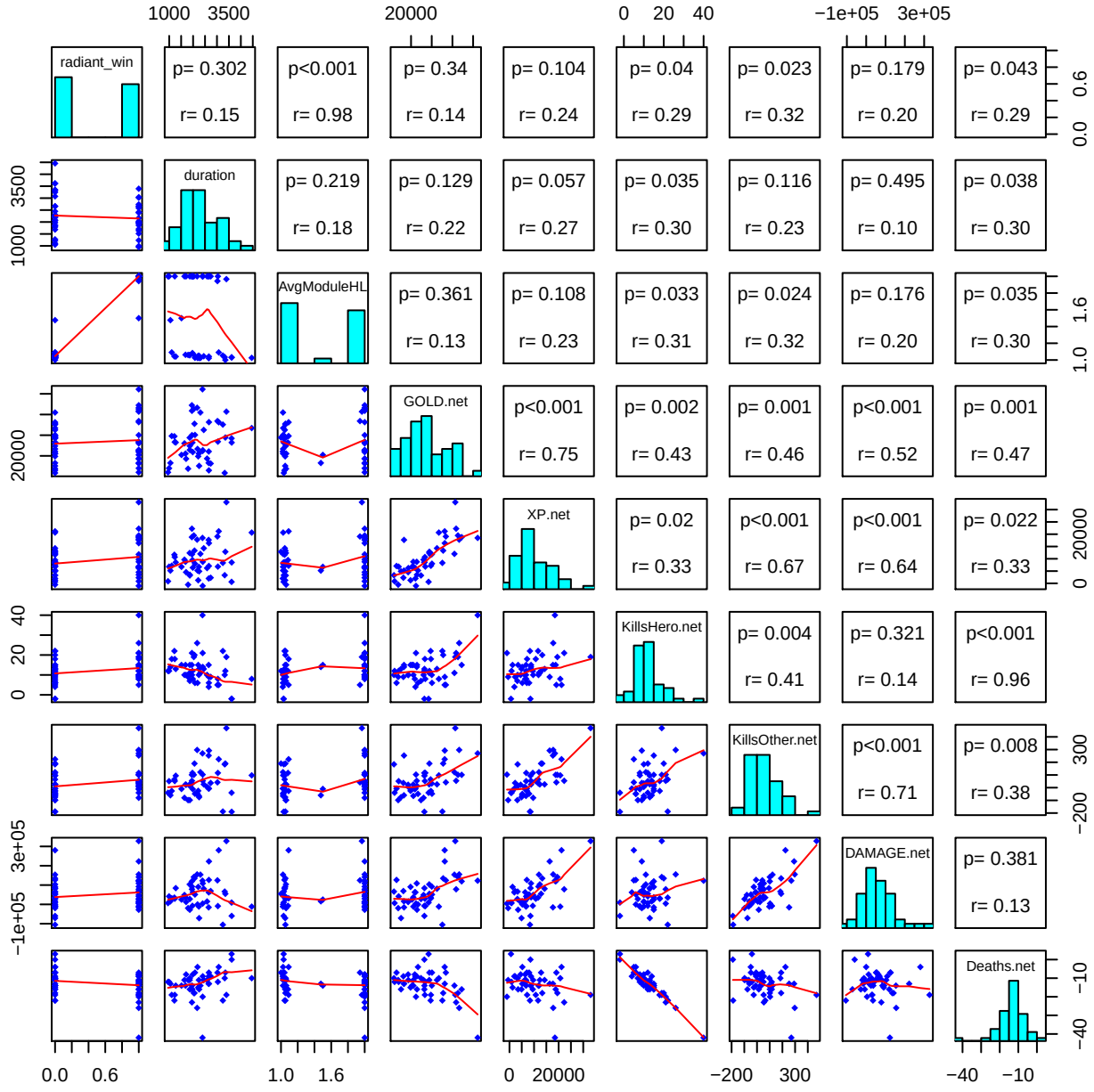

Figure 4: Correlations among variables for multigame clustering.

A Auxiliary Scripts

- DistanceToStateStripped.R: determines the distance between teams in a given game for a battery of games. Reads in zipped csv files, converts relative distances to states for each team, uses an optimal matching algorithm to calculate the similarity between sequences.
- HeroStats.R: reads in hero statistics and plots them.
- Hero_Analysis_Network.R: network analysis of hero drafts. Produces nothing interesting yet.
- Video.R: parred down script to show how to produce a video from a series of ggplot objects.
- sh JarToRaw.sh
  - (java -jar full-dota2-exp-1.2.jar ../data/raw/662332063.dem > ../data/raw/662332063.dem.results)
  - Loop over `match_id.dem` files in a given directory and produces `match_id.dem.results` json files in which each entry is a single event in the raw game. Multiple events at the same time produce multiple rows with different tags.
  - We have two versions of the parser. Some games may need additional parsers, as not all elements are returned (e.g., initial TI5 games, pre Main and Playoffs do not have gold parsed out with v1.2)
- python RawToJSON.py directorypath format
  - Pre-processes each `match_id.dem.results` file in directorypath directory and creates `match_id.dem.result` files. Removes extra junk at the beginning of each file that the java parser adds. Also adds match_id to each event in order to provide easier sorting capabilities in mongo or R.
  - format: "R" for importing json into R, "M" for importing json into mongo. The R flag adds commas and brackets at the end of each entry.