Predicting Win Percentage in Hearthstone

Alec Howard



Motivation and Applications

- Hearthstone is currently one of the most popular games in the world, with over
 40 million registered players per month
- Having the ability to potentially help 40 million players get better and win more often is an extremely powerful and valuable asset

o Idea behind project stems from helping both players and spectators understand Hearthstone

on a more analytical level.

- Inspired by win percentages in poker Pro players can only guess win percentage of certain matchups of different archetypes
 - For example, "Freeze Mage"
 vs "Control Warrior" is 10/90
- I'm a nerd



Problem Statement and How to Solve it

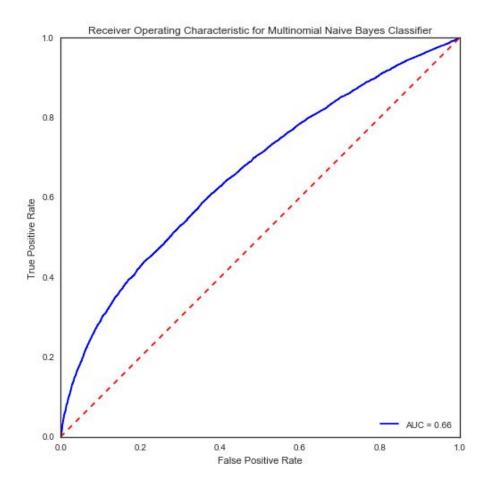
- Despite the massive amount of random chance in Hearthstone, a player's chance of winning can be accurately predicted each turn based on a number of measureable features, such as mana efficiency, cards mulliganed, and board state at each turn.
- Questions and potential roadblocks:
 - How to gather/collect data? How much data is needed? What will the data look like? Will there
 be any computational limitations? What model to use based on the type of data?
- Outline:
 - Research best Hearthstone simulator, run it, build logger to gather data from it
 - More games = better model, but also requires more time and computational power
 - Thus, run simulations/logger on powerful AWS EC2 instance
 - Clean and munge data, then fit model onto it and predict future games
- In retrospect, very ambitious project

Finding a Simulator

- After some research, found online open source Hearthstone simulator called "Fireplace" made by jleclanche
 - Source code consisted of several scripts run in tandem, took weeks to truly figure it out
 - Edited the code to fit project's needs, made each simulation completely random
- Built logger around Fireplace in order to gather data from these simulations
 - Difficulties:
 - Initially wasn't logging games correctly, had to rebuild logger from the ground up
 - Logger ran slowly on micro EC2 instance, had to run two larger instances simultaneously to generate enough games in time
- Ran logger through a threaded process, drastically increasing speed at which games were generated
 - After two days of simulations, gathered approximately 35,000 games resulting in over 6 million rows of data

Data Cleaning and Munging

- Stored data in a local MySQL database where some initial EDA was done
 - Exported multiple databases per instance to csv files
- Given sheer amount of data, cleaning/munging in pandas not optimal
 - Csv files imported in R, where majority of cleaning/munging was done by apprentice Kiefer
- Goal was to munge the data into a wide format for the model, where each event key was transformed into a binary column
 - Resulted in over 9,000 columns and several GBs of data
- Due to the sheer size of the data, wasn't computationally feasible to run a model across all of the data
 - Therefore, 115 "batches" of data (~300 games per batch) were imported for the model to be tested and trained on



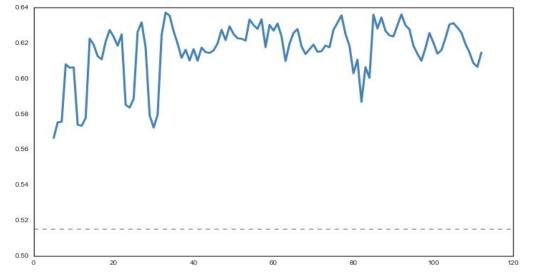
Modeling

- Problem: how do you fit a model on multiple batches instead of one huge batch?
- Discovered method to fit model on multiple datasets: scikit learn's "partial_fit"
 - Only a few models have the partial_fit function
 - Of these models, Multinomial Naive Bayes algorithm seemed most appropriate as it's a classifier that can handle data with discrete features
- Used MNB to partial fit on first 112 batches, then tested it on the final 3
 - Resulting ROC Curve shown to the left
 - Given the amount of data as well as the randomness of each simulation, an AUC score of 66% is actually quite reasonable
 - Shows that given more data, model could perform significantly better

Interpreting the Results

- During the iterative partial fitting, tested the model on next 3 datasets and stored the resulting accuracy score
 - Graph below shows accuracy of model's predictions at each iteration
 - o Baseline for all batches of data, denoted by dashed grey line, approximately 0.515
- Confusion matrix shows a mix of predictions, which is good





Confusion Matrix

Predicted	0	1	A11
True			
0	5803	3591	9394
1	3861	6083	9944
A11	9664	9674	19338

Classification Report

	precision	recall	f1-score	support
0	0.60	0.62	0.61	9394
1	0.63	0.61	0.62	9944
al	0.62	0.61	0.61	19338

Interpreting the Results (continued)

- Interpretation of coefficients are inverse to the norm
 - Indicates the probability that that card was played/chosen/targeted given the player won
- While coefficients may seem small, keep in mind there are over 9,500 of them
 - Even seemingly small coefficients are significant
 - Due to an error during the data munging phase,
 some coefficients are inflated, suppressing other coefficients

0.004282	mulligan_choices[og_340]
0.004281	mulligan_choices[tt_010]
0.004104	target[skele21]
0.004022	target[pro_001]
0.003412	first_player



Future Steps and Goals

- Fix some bugs in code (such as coefficient inflation issue)
- Run many more simulations to train the model on
 - Run these on multiple AWS EC2 instances most likely
 - Would ideally like millions upon millions of games
- Have different models "play" against each other and subsequently learn from the decisions they make
- Build a logger that can take data from real Hearthstone games in real time and feed it to the model for it to predict players' chances of winning each turn
- Build an application that integrates model and its predictions into Hearthstone client