Building a Better Board Game

Ryan Calme

Machine Learning Engineer Nanodegree

6 September, 2016

I. Definition

Project Overview

With so many board games available today, how might one determine which game(s) are the most enjoyable – without first purchasing and playing each of them? Many people might turn to crowd-sourced online reviews to ensure that others also enjoyed a game before making a purchase. But what if the game was recently published and there are not yet any reviews? What if you are developing an entirely new game, and would like to estimate how well-received said game might be once published?

The definitive online source of board game information and reviews is boardgamegeek.com. Members are able to provide ratings for games, as well as their subjective classifications of game genres, mechanics, and recommended number of players. I will utilize data from this site to attempt to create a model that can reliably predict the rating a previously unseen game might garner from members of the site.

Problem Statement

This data set is classified with crowd-sourced ratings. As such, I will use supervised learning techniques to create a predictive model. Each board game has a rating, on a continuous range from 1 to 10. A game with a rating of 9.9 is nearly universally enjoyed, where a game with a rating of 1.0 is universally loathed.

To assist the board game designer, it would be informative to discover which attributes of a board game (and values thereof) contribute most strongly to a game's rating. Is it possible to direct game design through this method such that the result is well received?

Within this project, I perform the following steps:

1. Obtain a large set of board game data, including user ratings
2. Consider which features of a game should be included/excluded
3. Train a model to predict board game ratings
4. Evaluate the *correctness* of said predictions
5. Predict ratings for a large number of randomly generated potential games
6. Propose some ideal characteristics that should result in a well-received game

Metrics

Since the target variable of this model has a non-finite set of possible values, a regression model is most appropriate. With regression models, the common measure of model correctness is mean squared error (MSE). With this metric, the actual value of the target variable and the predicted value are subtracted to find the error in the model. These errors are then squared for each observation, ensuring that they are all positive, and that larger errors account for a larger penalty. The mean of these squared errors is then computed. MSE values would thus have a lower limit of 0 (perfect agreement between actual and predicted values) and no upper limit.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Figure 1 - Formula for computing Mean Squared Error[1]

Though with MSE, the general rule is that “smaller is better” as a measure of model correctness, there is no agreed-upon standard for a “good” model fit. As such, I will also make use of a correlation coefficient – specifically Pearson correlation. Correlation coefficients range from -1 (perfect inverse correlation) through 0 (perfectly random correlation) to 1 (perfect correlation). A Pearson correlation value of 0.8 or greater indicates a reasonably good model fit.

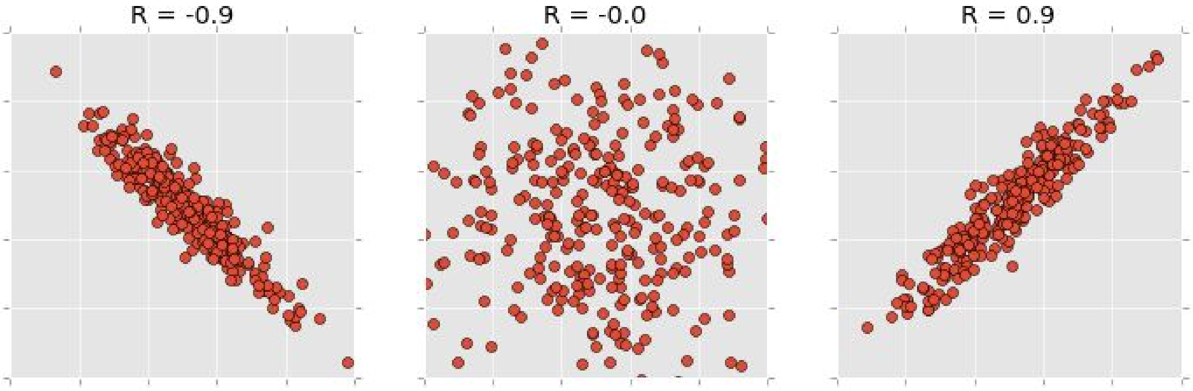

Figure 2 - Examples of Correlation Coefficients

[1] Mean squared error equation image - https://en.wikipedia.org/wiki/Mean_squared_error

II. Analysis

Data Exploration

The data set consists of 84593 rows (games), each with 163 columns of potential data (features). The features are listed below. I have color coded each as follows:

- Green - Candidate features
- Yellow - Special-case features
- Red - Non-predictive metadata
- Blue - Target feature

Column Name	Value Range	Note
id	1 - 202858	Game identifier
name	<String>	Game title
url	<String>	URL for game
ratingScore	1.0 - 10.0	Target feature
ratingCount	0 - 59423	Number of members contributing to ratingScore
ratingStdDev	0 - 4.5	Standard Deviation of ratingScore
year	-3500 - 2016	The year of publication. *There are clearly some outliers.*
priceAverage	$0 - $1300	The average price of sale (in USD) on boardgamegeek.com
priceStdDev	$0 - $581	The standard deviation of the sales prices for said game
playerAgeMin	0+	Stated minimum player age
playersStateMin/Max	1+	Stated minimum and maximum players
playTimeMin/Max	1+	Stated minimum and maximum play time (minutes)
playersBestMin/Max	1+	Member-voted *preferable* min/max player count
weightAvg	1.0 - 5.0	Member-voted measure of difficulty/complexity. 1 is simplest
weight Light / MediumLight / Medium / MediumHeavy / Heavy Pct	0.0 - 100.0	5 Features, the percentage of the vote each of the named categories received from members. These features should sum to 1 for each observation.
category*	0 / 1	84 sparse binary features. 1 indicates members have voted inclusion in this category 0 indicated that members have not
mechanic*	0 / 1	51 sparse binary features. 1 indicates members have voted inclusion in this mechanic

		0 indicated that members have not
subdomain*	0.0 - 100.0	8 features representing the percentage of the vote members have cast that the given game falls under this subdomain The values for these 8 features sum to 100 for each row

Some notes on the features:

1. ***ratingCount*** and ***ratingStdDev*** are likely to have predictive value. Incredibly popular games will receive more votes than others, and are also likely more highly rated than the average. I will be excluding these features from the model. These features may still be used to limit the data considered for training a model.
2. ***year*** presents an interesting choice. I can include it in the model, as it may have predictive power, especially if consumer tastes have changed over time. In an attempt to build a better board game, however, It would be inadvisable to recommend that the ideal game be published in 1982. There are also some games in the data set that claim publication dates of 3500 B.C. This may be historically correct, but these items may not assist in creating an ideal model.
3. ***priceAverage*** and ***priceStdDev*** could be very interesting features to a model, and likely have correlation with the game's rating. However, this data was only available for 23% of the data, and would severely limit the size of the available data for training. I will create models both with and without these features.
4. ***category**** and ***mechanic**** are sparse, binary features. These kinds of features could work better for some modeling techniques than others. They could potentially be excluded or combined via singular value decomposition (SVD) into fewer, more powerful features.

Exploratory Visualization

The first question that arises with this data set might be regarding the distribution of our target variable, ***ratingScore***. Do these ratings fall in a Gaussian distribution?

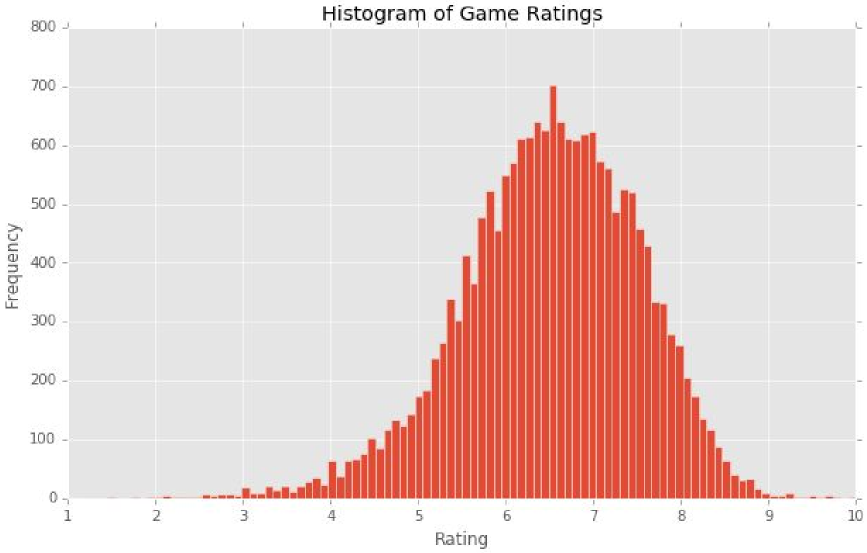

Figure 3 - Frequency of games by ratingScore value

The ratings in the data set do in fact seem to have a relatively normal distribution. The mean is skewed upward within the allowable rating range, to around 6.5.

Algorithms and Techniques

I would like to investigate the comparative performance of multiple supervised learning techniques in fitting a model for this problem. I would also like to provide these child model predictions to a meta-learner to see if I can obtain a final model whose performance surpasses that of the standalone techniques. This technique is at times referred to as *stacking* or *blending*.

This is a supervised learning task, because we have a target variable for all observations. And since the target variable is non-finite set of classes (a real number range), regression techniques are the most appropriate. The following four regression techniques seem to be reasonable candidates in this situation:

Random Forest

The random forest learner creates a *forest* of trees – the count of which is adjustable via the *n_estimators* parameter – in which each considers a different *random* subset of the training data. At each node from the root downward, a *random* subset of the available features are considered, in an attempt to split the observations. The metric computed at each node to measure the quality of the split is referred to as the *gini* coefficient. Once the forest of trees are built, the mean of the predictions from all trees becomes the model's overall prediction. As a tree-based learner, I anticipate that this technique will be more effective in understanding the interrelationships of features.

I will use GridSearchCV to manipulate the following hyperparameters for the random forest learner:

- *max_features* – The maximum number of features to consider at each node, when building trees. Very low values will result in trees with high bias. Very high values (near n_features) ensure that the variability between trees is very small.
  - I will search among the values 10-80 in increments of 10
- *min_samples_split* – At each node, if there are fewer observations in the remaining subset than this value, then there will be no further subdivisions. This node will be a leaf in the tree. This value affects how deeply the trees will grow.
  - I will search among the values 2-6 in increments of 1

K-Nearest Neighbors

The K nearest learner finds the nearest K observations in the feature space, and weights their contribution toward the prediction. This algorithm is fast, but flawed. If training data is evenly distributed, then the feature vector for any observation will likely have K near neighbors that are similar. If the training data is sparse in portions of the target variable range, the near neighbors may be very unlike the feature vector. As a result, KNN predictions for outlier feature vectors will tend to pull toward the average in training.

I will use GridSearchCV to manipulate the following hyperparameter for the K nearest neighbors learner:

- *n_neighbors* – The K in K-nearest. How many neighbors should be considered when producing a prediction. A low value of 1 effectively provides a prediction equal to the target value of the single nearest observation in feature space.
  - I will search among the values 2-32 in increments of 2

Support Vector Machine

The support vector machine algorithm (support vector regressor, in this case) attempts to find a hyperplane in the feature space that follows the area of maximum margin between classes. The technique is intended to find the best dividing boundary between observations in the kernel-transformed feature space.

I will use GridSearchCV to manipulate the following hyperparameters for the support vector regression learner:

- *C* – The penalty parameter. Smaller values of C create models that prefer larger hyperplane margins between classes, at the expense of misclassification. Larger values of C will allow smaller margins, if able to avoid misclassifications.
  - I will search among the values 10e0 - 10e2 in log space

- *gamma* – Large values for gamma will result in models that overfit. Small values will result in models too generalized.
    - I will search among the values 10e-4 - 10e-2 in log space

Gradient Boosting

The gradient boosting regression algorithm is also an ensemble of weak learners, similar to random forest. Instead of averaging the predictions from all trees in the forest however, GBMs consecutively adds models to the ensemble to follow the negative gradient of the loss function.

I will use GridSearchCV to manipulate the following hyperparameters for the support vector regression learner:

- *max_depth* – How deeply to grow the trees. Gradient Boosting Regression is a time-consuming process, so I will only manipulate one hyperparameter.
    - I will search among the values 2-20 in increments of 2

Meta-estimator

The predictions resulting from the models produced by the above four techniques will become the input features to a *stacker*. Stacking techniques can be as simple as averaging the predictions of the child learners (like in random forest), or as complex as any of the child models themselves. For this purpose, I will make use of LassoCV, a linear regressor.

Benchmark

Each child model produced will be evaluated on its performance in correctly predicting the ***ratingScore*** for observations in the held out portion of the data. I will compute the mean squared error and Pearson correlation for these models over this set. These metrics will allow us to perform direct numerical comparisons between the performance of the individual modeling techniques on this task, as well as the performance of the stacked learner.

Using a data set that has not been previously explored ensures that I will not have an established baseline for values of these metrics. The correlation coefficient value of 0.8 will be my target. Upon further evaluation, this threshold may prove either too hopeful, or too conservative. It is not yet clear the predictive power of the feature data collected.

II. Methodology

Data Preprocessing

I anticipate two methods of pre-processing this data, and will attempt both.

First, considering all available features, I will select only the observations that contain numeric values for every feature. As noted above, the ***priceAverage*** feature is unavailable for all but 22% of the observations. In this method, a far smaller subset of observations will be available to subdivide among training and holdout sets.

Second, considering all features *except* ***priceAverage***, and ***priceStdDev***, I will select the subset of observations with numeric values for all remaining features. This method will result in a larger set of data which may allow for the creation of a stronger model.

With both methods above, I will perform some data cleanup.

1. Remove the games whose dates of publication were prior to 1950.
    a. Some games in the data set claim publication dates multiple thousand years B.C. These are valid games, many of which are still very popular today. I have chosen to view these data as outliers, and exclude these observations from use.
2. Remove the games that have been rated by fewer than five members.
    a. Crowd-sourced ratings tend to have nice distributions. However, when a game has been rated by only a few individuals, the confidence in that average rating falls. I have chosen to exclude observations rated by fewer than five individuals, such that no individual rater can have more than a 20% influence on the average rating any game receives.

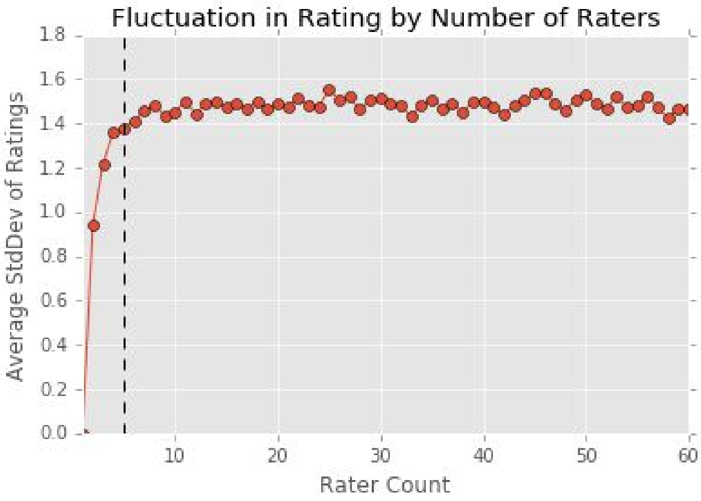

Figure 4 - Justification for the exclusion threshold of 5 raters

Note: For two of the child learners (K-Nearest Neighbors, and Support Vector Regression), I will normalize the feature vector using the StandardScaler.

Implementation

After pre-processing the data as described above, I will partition the remainder into two portions, with an 80% / 20% random split. The 80% portion to be used for training models (referred to hereafter as the *training set*), and the 20% (referred to hereafter as the *holdout set*) for computing performance metrics.

Ideally, the best model is created by using all of the training data, and proving the model's performance by predicting the holdout set. As I intend to investigate the impact of stacking multiple models[2], I will have to train the stacking learner on the predictions of the child learners. If I were to train the child learners on all of the training data, and collect their predictions on the holdout set, I could then train the stacking learner on those holdout predictions. However, I then have no remaining unseen data on which to prove the final model.

Instead I obtained predictions on the training set by using cross-validation. The collected predictions on the held-out data *from each fold* – for each child learner – became the input features for the stacking learner.

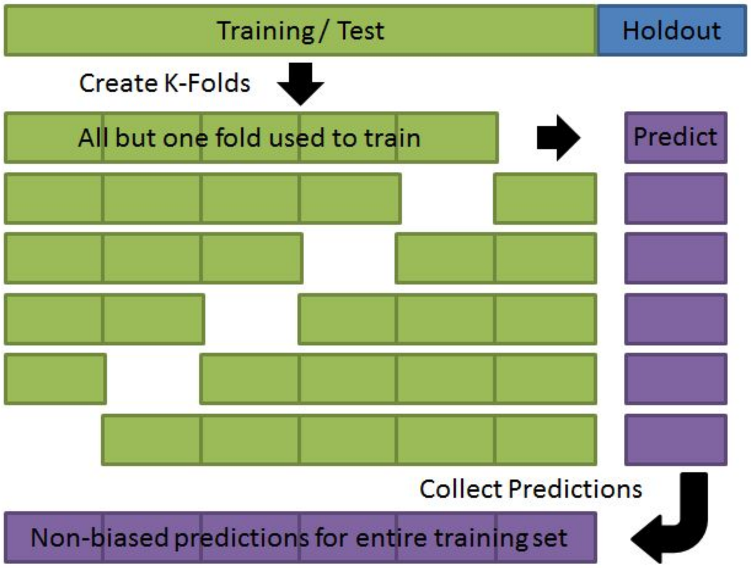
Training / Test
Holdout
Create K-Folds
All but one fold used to train
Predict
Collect Predictions
Non-biased predictions for entire training set

Figure 5 - Method for collecting cross-validated predictions

These cross-validated predictions for each child model become the input features to the stacking learner. An [n x m] matrix, where n is the number of observations in the training set, and m is the number of child learners being stacked to produce the final model. Each position in the matrix is the prediction for the observation that comprises the row, for the child learner that comprises the column. The stacker's target variable remains the same as that provided to the child learners.

For each child learner, I repeat the following process:

[2] http://www.stat.berkeley.edu/~ledell/docs/dlab_ensembles.pdf

1. Perform a cross-validated grid search with the training data on one or more learner-specific parameters
2. Graph the mean squared error of the models resulting from these varying parameter values, determine which parameter value(s) produced the best model
3. Apply these ideal parameter values for the learner
4. Collect cross-validated predictions on the entire training set – to be used as input features for the stacking learner
5. Re-train the learner – with ideal parameter values – on **all** of the training data to produce a final child model for use in predicting the holdout set

I will use a linear regression model as the stacking technique, and fit a model to the input features produced by the child learners. I do *not* need to collect cross-validated predictions in this case, as this step produces the final model to be used for holdout prediction.

The process of predicting the holdout set is then straightforward. The feature vector for a single observation (game) is fed to each child learner in turn, and those predictions become the [1 x n] feature vector for the stacker, which in turn produces the final prediction of the ensemble. I will create a pipeline to perform these chained actions to simplify this step.

I will compute the chosen metrics of mean squared error and Pearson correlation *on the held out set* separately for each child model, and the ensemble as a whole, to see what (if any) boost can be attained by stacking these learners.

Refinement

I made a few attempts at model improvement during my investigation:

1. Grid search vs default parameters
    a. For the random forest learner, I first modeled with the default parameter values. I then implemented the grid search described above. Here is a comparison:

Technique	Mean-Squared Error	Pearson Correlation
Random Forest - Default Params	0.4116	0.7677
Random Forest - Grid Search	0.3699	0.7943

2. Stacking child models, as opposed to using them directly as the final model.
    a. I show the performance statistics of each of the child models individually, in scoring the held out set, as well as the performance of the stacked model.
3. Dropping features to obtain a larger training set size
    a. By removing the *priceAverage* and *priceStdDev* features, for which there are many observations with undefined values, there are more games to include in the

training set. I check whether more features, or a larger training set creates a better model in this case.

IV. Results

Model Evaluation and Validation

After a GridSearch of one or more parameters, the following best_params_ were chosen:

Random Forest regressor:	*max_features* - 40, *min_samples_split* - 3
K-Nearest Neighbors:	*n_neighbors* - 10
Support Vector regression:	*C* - 10, *gamma* - 0.001
Gradient Boosting regression:	*max_depth* - 4

The four child models achieved the following performance on the held out set:

Child Estimator	**Mean-Squared Error**	**Pearson Correlation**
Random Forest Regression	0.3699	0.7943
K-Nearest Neighbors	0.5354	0.6831
Support Vector Regression	0.4086	0.7693
Gradient Boosting Regression	0.3582	0.8005

Two of these learners (Random Forest, and Gradient Boosting) came in very close to the correlation benchmark value of 0.8 on their own. The LassoCV stacked learner narrowly beat the performance of any of the child models alone.

Meta Estimator	**Mean-Squared Error**	**Pearson Correlation**
LassoCV	0.3448	0.8088

The LassoCV linear learner that was used for stacking reports the learned coefficients of its inputs. Since the inputs in this case were the predictions of the child learners, we can infer the comparative importance of the child learners to the final stacked model.

Child Estimator	**Stacker Coefficient (Importance)**
Random Forest Regression	0.4500
K-Nearest Neighbors	0.0826
Support Vector Regression	0.1141
Gradient Boosting Regression	0.4239

As you can see, the bulk of the contribution (87.3%) of the final model comes from the random forest and gradient boosting estimators. However, the stacked model was still able to reach a higher correlation than either of those models alone.

The change in these performance statistics between the training set and the held out set was negligible, which shows that this final model is not overfitting the training data, and generalizes well to previously unseen data.

The final attempt at refinement involved removing two features, and re-modeling with a larger training set. The hope was to increase the training set size from ~17K observations to nearly ~67K (80% of the 84K available data). However, evaluation proved that when using the same data filters, the training set size would only increase to ~30K. It seems that the responses dropped originally due to missing values for *priceAverage* and *priceStdDev*, were also often the same responses later filtered with the *ratingCount* < 5 rule. This implies that games with fewer raters are also those most likely to have no sales histories.

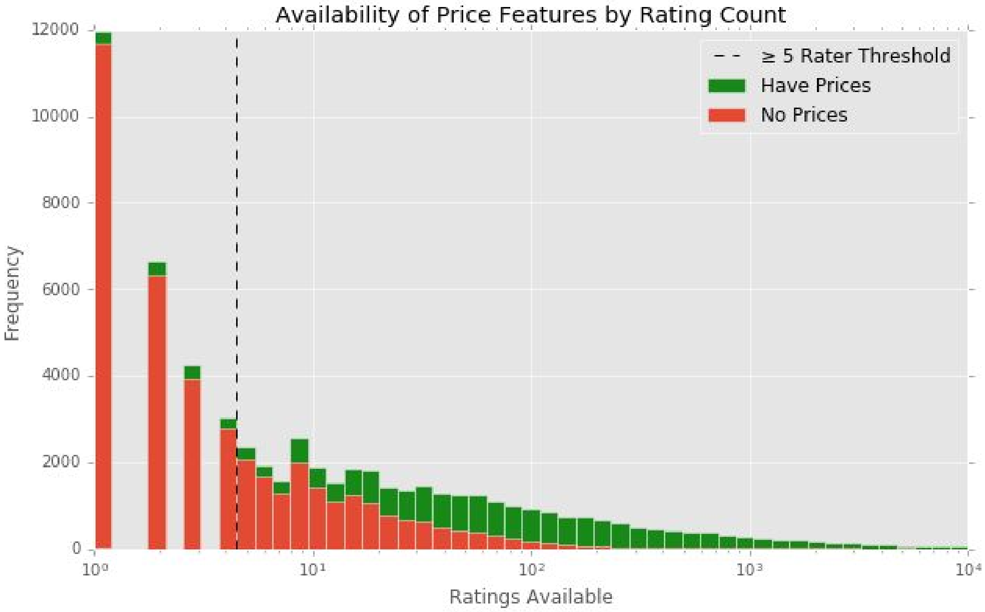

Figure 6 - Comparison of frequency of games by number of raters, with presence of prices

After dropping two price-based features, and growing the size of the training set by ~40%, a second stacked model was produced, with the following performance:

Method	**Mean-Squared Error**	**Pearson Correlation**
Stacker - Fewer Data, More Features	0.3448	0.8088
Stacker - More Data, Fewer Features	0.5572	0.7654

Justification

In this case, with a previously unexplored data set, there was no previous benchmark for performance. The target was a Pearson correlation of 0.8 or higher. One of the child models, and the final stacked model surpassed this threshold.

One of the main investigations in this report was that of stacking child models to obtain better performance than any of those models used as inputs to the stacker. Though the improvement was slight, the stacked model did indeed have the best performance by the correlation metric. The additional time required for this stacking method is slight, which I feel justifies its use in this case.

Application

With this final model in hand, I created random feature vectors for 100,000 potential games, and predicted their potential rating scores. I analyzed a number of both high and low-scoring game designs, and was able to draw the following conclusions:

Common threads

- Card Games are common among all games, and are similarly represented in both best, and worst rated games
- Dice rolling as a mechanic is also similarly represented in both best and worst games

Best performer attributes

- The best performing games are disproportionately expansions for base games - perhaps just because hit games are more likely to spawn expansions in the first place?
- Children's games and war games are frequently categories for well-ranked games that do not often present in poorly ranked games
- 'Hex-and-counter', 'variable player powers', and 'card drafting' are game mechanics found more often in well-ranked games

Worst performer attributes

- Trivia and party games seem to be more likely to fare poorly in ratings
- Games with 'roll/spin and move', and 'acting' mechanics are to be avoided.

Suggestions for hit games

- An expansion for a strategy word game with an American civil war theme, priced at $63
    - (I'd play that)
- A Napoleonic wargame with a hex-and-counter mechanic for $70
- A complex adventure/horror card game aimed at families for $39
- An action/dexterity dice-rolling game with a set-collecting mechanic for $38

- A children's animal-themed simulation party-game for $18

Designs to avoid:

- Don't try to market a customizable children's game in both the 'ancient' and 'fighting' categories, involving a stock-trading mechanic
    - Please don't. At any price.
- A thematic children's trivia game that includes dice, voting, and negotiation - for the suggested price of $5

V. Conclusion

Free-Form Visualization

One of the major techniques explored in this project is that of stacking models in an attempt to produce a stronger model. Here is a comparison of the performance of the child models, and that of the stacked model:

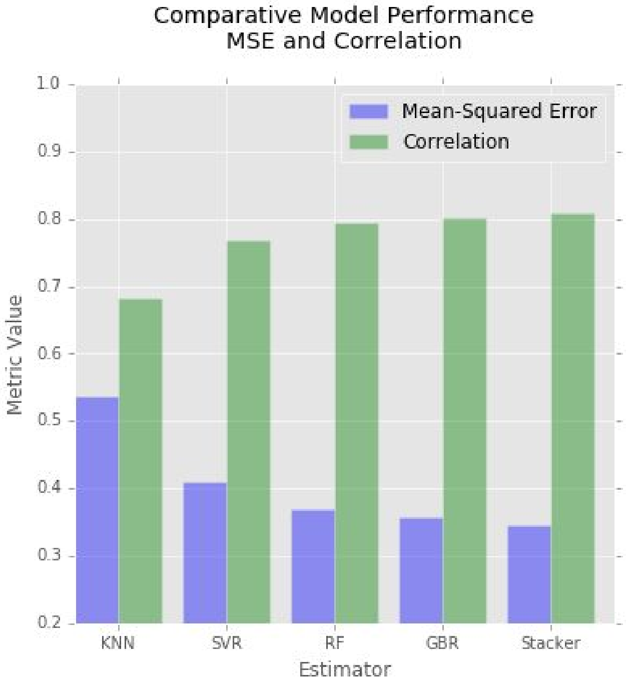

Figure 7 - Comparative performance of models attempted

A lower value for MSE implies a stronger fit, as does a larger value for Correlation.

K-nearest neighbors models are weak predictors, comparatively, and both random forest and gradient boosting are already ensembles of weaker models, resulting in stronger predictive power individually.

The Stacked model is the best performing on both measures, though not by a wide margin.

Reflection

- I have performed a thorough investigation of this board game data set, and found the features to be a good fit for supervised learning techniques
- I utilized multiple learning algorithms, and contrasted their performance
- I implemented a new stacked estimator, compatible with the sklearn tool kit
- I succeeded in surpassing my initial benchmark for model performance
- I generated random feature vectors, and through use of my model inferred some important aspects of a well-ranking future game

Exploring this data set proved interesting. The choice of features to include/exclude from the model could easily be argued. Was including the year of publication the correct choice? How many raters must contribute to a game's rating before outlier averaged ratings are truly eliminated?

In generating feature vectors for potential games, I struggled with the percentage-type features, where a subset of features were expected to sum to one. I first generated random values between 0 and 1 for each, then divided them all by the sum of the set, ensuring a sum of 1. However, this resulted in proportions for each often near 1/n, where n was the number of features comprising the set. It isn't common for each of the eight subdomain features for a given game to receive an equal percentage of the public vote. I needed to generate more realistic data. I discovered and made use of the Dirichlet probability distribution[3], and was able to correct for this, to generate more realistic feature vectors.

I feel that the final model produced in this project does indeed have a strong fit to this data, and could be used to predict the ratings of potential games for the purposes of guiding game design, publisher review of a potential game concept, or end-user purchasing consideration of a newly-released but unreviewed game.

Improvement

Though the final model proved to fit the data well, I was surprised to learn that the support vector regression model was so difficult to tune. I anticipated better performance from this technique with the default parameters. Due to the prohibitive training duration, I was unable to fully explore the parameter grid, as applicable to this problem. Given more time, I would do so.

I would also like to build out the StackedRegressor implementation that I began. I would have expected to find a standard implementation of this within SKLearn. When all of the support functions expected of a fully-fledged estimator are in place, I could consider contributing this to the community for further use.

[3] https://en.wikipedia.org/wiki/Dirichlet_distribution