Opponent Modelling in No-Limit Texas Hold'em

Machine Learning Engineer Nanodegree Capstone Project

Nash Taylor, September 2016

Definition

Project Overview

The problem of creating an artificially-intelligent poker-playing agent is well studied. In fact, some variants of the game such as Limit Hold'em are already considered solved[1]. However, in more complex variants such as No-Limit Hold'em, there are many factors that limit the effectiveness of such game theoretic models as have been proposed thus far. One of the leading contributors to the difficulty of the game of poker from an AI perspective is its imperfect information; because the hole cards of one's opponents are not known, predicting their response to different possible actions can prove difficult for even the most experienced of human players. This is the specific sub-problem I will attempt to solve.

Predicting a player's action in a given situation is a perfect example of supervised learning. Given features of the state of the game, the player's tendencies, the player's view of his opponents, and the characteristics of the community cards, a supervised model should be able to guess what the player will do next. Note that these features leave out one critical component, which was mentioned earlier: the player's hole cards. However, as we will see, the combination of a well-engineered feature set and a well-chosen function approximator can make up for this imperfect information state.

All supervised models constructed as part of this project utilized data from HandHQ.com[2].

Problem Statement

The goal of this project is to produce a set of supervised learning models that take in features of a poker game and predict the action of whichever player is to act next. The decision of whether to treat this as classification or regression was tricky, and I have settled on a somewhat 'middling' approach: I will do classification for the actions, but for bets and raises I will create a separate regression learner to predict the amounts.

Because the rules of the game allow for certain actions in certain situations, and because the state of the board is different after each "street" (dealing of

[1] http://ai.cs.unibas.ch/_files/teaching/fs15/ki/material/ki02-poker.pdf

[2] http://web.archive.org/web/20110205042259/http://www.outflopped.com/questions/286/obfuscated-datamined-hand-histories

community cards), I have decided to break up the model into 7 different models, one applied to each of the following situations:

- Pre-flop, facing a bet
- Post-flop, not facing a bet
- Post-flop, facing a bet
- Post-turn, not facing a bet
- Post-turn, facing a bet
- Post-river, not facing a bet
- Post-river, facing a bet

The reason there is no model for "Pre-flop, not facing a bet" is because there are 'blinds', which are mandatory bets that begin every game; therefore, there is only one relatively rare situation in which a player would not be facing a bet before the flop, which is in the Big Blind if no bets are made leading up to that player. This is not an interesting prediction problem (they typically check), and there is not enough data to learn this over.

The result is 7 models, each taking a slightly different subset of the full feature set and each predicting an action from a subset of the actions described above. In "facing bet" situations, the actions are: fold, call, raise[amount]. In "not facing bet" situations, the actions are: check, bet[amount].

The overall procedure for learning these models is as follows:

1. Parse the raw text of game logs into a feature set providing information on, for each action taken:
    1. the play style of the player (e.g. their preflop raise percentage)
    2. the player's opponents at the table (e.g. the average stack size)
    3. the community cards (e.g. the number of pairs on the board)
    4. the state of the game (e.g. the number of players remaining)
2. Split the feature set into 7 subsets, separated by the Round and FacingBet fields. For each resulting dataset, take only the corresponding subset of columns (see Data Preprocessing for full lists)
3. For each dataset, train the following:
    1. a classification model to predict the action selected by the player (e.g. bet)
    2. a regression model to predict the amount of the bet or raise (e.g. 0.55 of pot)
4. Evaluate and tune each model using a validation set
5. Obtain an overall score over all models to test against the benchmark

The result is a set of 14 networks: one for actions, and one for amounts, for each of the 7 situations.

Metrics

The variety in the structures of these 14 learning problems necessitates 3 different scoring metrics: for the binary classification of non-bet-facing actions, F1 score; for the multiclass classification of bet-facing actions, multiclass-adjusted F1 score; and for the regression of bet and raise amounts, Mean-Absolute-Error.

Calculation

The F1 computation in code looks like this, where `confusion` is the 2x2 confusion matrix for the binary classification task (whether truly binary or a one-vs-all setting):

```
precision = confusion[0,0] / confusion[:,0].sum()
recall = confusion[0,0] / confusion[0,:].sum()
f1 = 2 * (precision*recall) / (precision+recall)
```

To calculate the F1 for the multi-class case, I issued a custom implementation

The mean absolute error is easier represented mathematically, and it is calculated as:

$MAE = \frac{1}{N}\sum_{i=1}^{N}|p_i - o_i|$, where p_i is the i'th prediction and o_i is the i'th true observation

Justification

The choice of F1 score for binary classification was due to the natural interpretation of the problem as an identification of bets. If the problem is stated as, "will this person make a bet?", it becomes obvious what the positive and negative labels are. The classes aren't horribly unbalanced, but they aren't 50/50, so for these reasons F1 score is a good fit.

For multiclass, the typical metric is accuracy score. However, because these labels are particularly weighted towards folds (especially in the Preflop case), accuracy score is inadequate for measuring the true predictive value; it considers all labels equal, when really, a model that only predicted folds would do reasonably well, better than random guessing. The best response to this situation is to adapt the F1 score, which is designed for imbalanced classes, to the multiclass setting. This is done by taking the F1 scores of each one-vs-all model (of which we have 3) and taking a weighted average according to the distribution of labels.

The main decision for the choice of regression metric is between RMSE and MAE (mean absolute error). When the desired effect is for large errors to be punished linearly proportional to their size, MAE is appropriate. When large errors should be punished with greater effect, RMSE is better. For this problem, making the mistake of assuming a very large raise or bet when it was simply a large raise or bet is not necessarily terrible; the response from the majority of opponents will be fairly robust to this, and so the agent's view of the game and ability to roll forward and plan would not be greatly effected by this mistake. For this reason, I will use MAE instead of RMSE.

Analysis

Data Exploration

The original data collected for this project was a corpus of text files containing logs of online games from 5 different online poker sites. After parsing relevant information from this text, 3 tables were created and a relational database was formed. From these, a large feature set was constructed.

Boards

Field	Data Type	Description
GameNum	String	Primary key; identifies which game the board is associated with
LenBoard	Int	"Number of cards on the board (represents round of the game; e.g. Flop)"
Board1	Int	"Integer representation of one of the 52 cards in the deck; e.g. 2c = 1"
Board2	Int	"Integer representation of one of the 52 cards in the deck; e.g. 2c = 1"
Board3	Int	"Integer representation of one of the 52 cards in the deck; e.g. 2c = 1"
Board4	Int	"Integer representation of one of the 52 cards in the deck; e.g. 2c = 1"
Board5	Int	"Integer representation of one of the 52 cards in the deck; e.g. 2c = 1"

Actions

Field	Data Type	Description
GameNum	String	Primary key; identifies which game the board is associated with
Player	String	Obfuscated name of the player
Action	String	Action without amount
SeatNum	Int	Seat number starting from the top right of the table

Field	Data Type	Description
RelSeatNum	Int	Seat number starting from the dealer button
Round	String	Round of the game; e.g. Pre-flop
RoundActionNum	Int	“Numbered actions; reset at the start of each new round (e.g. Flop)”
StartStack	Float	Amount of chips for Player at the start of the game
CurrentStack	Float	Amount of chips for Player at current moment (before action)
Amount	Float	Amount of chips associated with action
CurrentBet	Float	The amount of the bet that Player must respond to
CurrentPot	Float	The amount of chips currently at stake
InvestedThisRound	Float	The amount of chips Player has invested thus far in the round
NumPlayersLeft	Int	The number of players remaining in the hand
Winnings	Float	The amount that Player received at the end of the hand
HoleCard1	Int	Integer representation of Player’s first hole card
HoleCard2	Int	Integer representation of Player’s second hole card
SeatRelDealer	Int	Player’s seat number relative to the dealer button
isFold	Boolean	Dummy representation of Action
isCheck	Boolean	Dummy representation of Action
isCall	Boolean	Dummy representation of Action
isBet	Boolean	Dummy representation of Action
isRaise	Boolean	Dummy representation of Action

Games

Field	Data Type	Description
GameNum	String	Primary key; identifies which game the board is associated with
Source	String	The online poker site from which the game was scraped
Date	DateObj	The date the game was played
Time	DateObj	The time the game was played
SmallBlind	Float	The size of the small blind for that game
BigBlind	Float	The size of the big blind for that game (should be 2*SmallBlind)

Field	Data Type	Description
TableName	String	Obfuscated name of the table at which the game was played
Dealer	Int	Number representing seat number of the dealer button
NumPlayers	Int	Number of players active at the beginning of the hand

Features

From these 3 tables, roughly 120 features were produced. To save space, I won't list them here, but they can be found in the data sample. These features can approximately be broken up into 4 categories:

- Features of the play style of Player, e.g. Preflop Raise %
- Features of the player's opponents, e.g. Average Table Stack
- Features of the community cards, e.g. Number Of Pairs
- Features of the state of the game, e.g. Is Last To Act

The majority of these features are numeric, and the breakdown of datatypes is as follows:

Data Type	Count
Categorical	1
Boolean	22
Numeric	86

The data being learned over is only a subset of the full data available, due to my own computational limits. This is discussed further in Data Preprocessing. The final shape[3] of each dataset is:

Filename	NumRows	NumCols
Preflop-False	77289	79
Preflop-True	14249257	85
Flop-False	2445347	88
Flop-True	1354565	94
Turn-False	1318627	98
Turn-True	626877	104
River-False	822986	109
River-True	349217	115

[3]The actual shapes are slightly larger, as one variable was later converted to a dummy. This would add between 4-6 columns to each dataset.

There is a clear decay in the size of the data as we approach later rounds, which makes logical sense; fewer hands get all the way to the river than start at all. The Preflop section is by far the largest and should theoretically be the most effectively trained model.

The breakdown of labels for each dataset is:

Table	fold	check	call	bet	raise
Preflop-True	0.669	0.027	0.136	0.0	0.168
Flop-False	0.0	0.621	0.0	0.379	0.0
Flop-True	0.546	0.0	0.334	0.0	0.12
Turn-False	0.0	0.634	0.0	0.366	0.0
Turn-True	0.497	0.0	0.405	0.0	0.097
River-False	0.0	0.633	0.0	0.366	0.0
River-True	0.555	0.0	0.362	0.0	0.083

Exploratory Visualization

My exploratory visualizations will examine the distributions of actions and amounts, as this is the most important information in the dataset (the label). For the amounts, I will then split the data over rounds. This information was previously presented in table form, but it is always a good idea to view information of this structure in a visual format. Figures 1-3 are, in order: the distribution of actions, the distribution of amounts for bets, and the distribution of amounts for raises. I will discuss these in order.

Preflop, it is clear that the majority of actions will be folds. Across future rounds, the distribution between check/bet and fold/call/raise is remarkably similar, with checks consistently making up roughly 60% of actions in non-bet-facing situations, and folds, calls, and raises taking a 50/40/10 split in bet-facing situations. A model that just predicted folds would get about 55% accuracy overall, which is a good benchmark to be aware of.

Bet amounts peaked for the most part just over 1/2 of the pot size, which follows from standard poker theory. The distributions are roughly (very roughly) normal, except for the River where there is a secondary mode directly at pot-size. Flop bets had a sharper peak than Turn and River, which tended to be more spread out around the mean.

Raise amounts had a distnictly positive skew, as they peaked around 3/4 pot size and decayed from there. The exception was preflop raises. An incredibly distinct pattern emerges in preflop raises, which is that the great majority of players would make 2-pot raises, with strong secondary amounts of 4/3- and 7/3-pot. I would anticipate that this pattern should ease the learning process for this particular network.

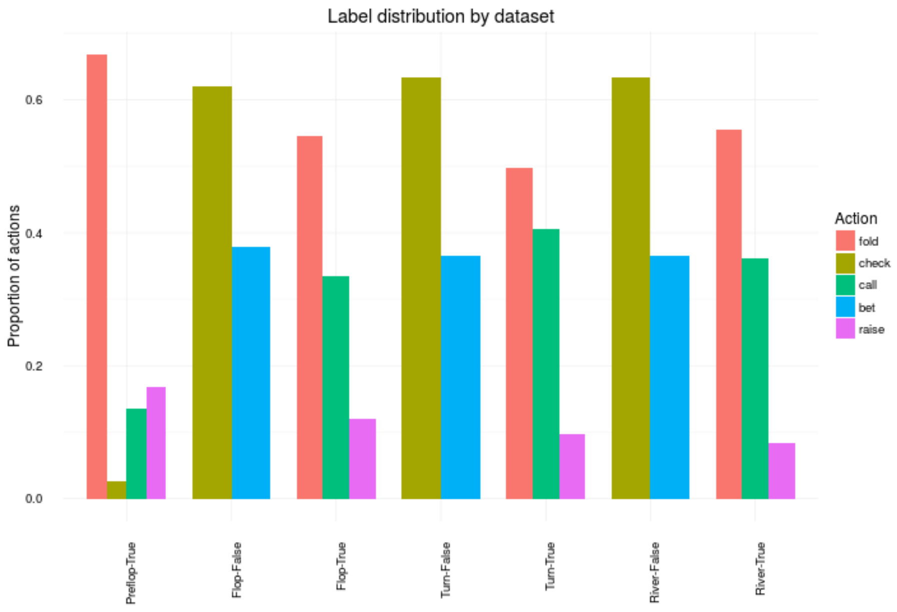

Figure 1: Action distribution

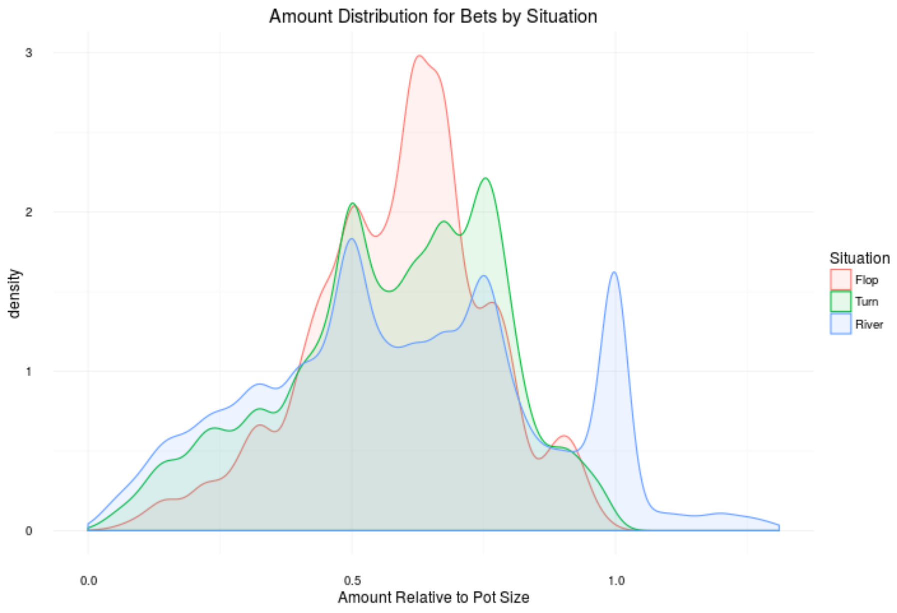

Figure 2: Bet Amount distribution

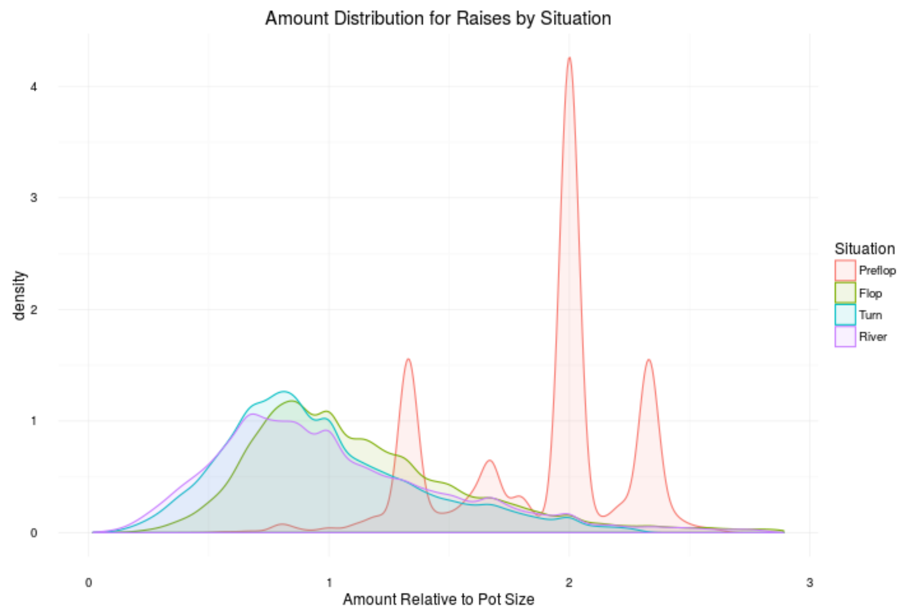

Figure 3: Raise Amount distribution

Algorithms and Techniques

Rather than apply a large and complex function approximator like a deep neural network to this problem, I will attempt to find an optimal solution among the simpler and less computationally expensive models. I will conduct this search by thoroughly testing the wide range of classifiers and regressors available in the Python library `sklearn`. The performance of these learners will be measured using cross-validation, to prevent information from the test set from leaking into the training process. Among my primary concerns with the classifiers are:

- Training F1 score; does it represent the true function reasonably well?
- Testing F1 score; does it generalize?
- Prediction time; would a reinforcement learning agent be able to use this to do efficient planning and lookahead?

In view of the last point, models with expensive predictions, like the computationally intensive SVM and instance-based (computation deferring) kNN, will not be considered. These are the algorithms I will consider:

Classifiers

Decision Tree

Decision trees power countless supervised learning algorithms, most of which deploy sets of many trees to generate predictions. The reason decision trees are so commonly used in this way is because they are amazingly fast to train and predict with. A decision tree essentially breaks down the problem of prediction into a series of splits on the features of the training data. At the top of the tree is the feature that splits the data the best according to maximum information gain. At each successive layer of the tree is another set of features, that further splits those groups according to their information gain. This recursive process is applied until the tree reaches some stopping criterion: in the base case, it is when every leaf node contains exactly one kind of label. Because this is a textbook case of overfitting, various hyperparameters can be tuned to reduce the variance of the learner and increase its bias. Such hyperparameters include the maximum depth of the tree, the minimum samples required to split, and the maximum features allowed as splitting criteria. These hyperparameters all control essentially the same characteristic of the tree, its depth, and therefore typically only one is required to be tuned for a given tree.

AdaBoost

AdaBoost builds a collection of simple learners (in my case, decision trees), called an ensemble, and fits an additive model with these trees in an effort to get "the best of all worlds". Intuitively, the model is constructed by beginning with a very simple tree, typically with a max depth of 1 or 2, and then successively adding new trees to the model, all of which are trained on the same data. The difference in their training processes is how the data is weighted; if previous learners have been getting examples wrong consistently, then the focus of new learners is shifted towards these examples by increasing their relative weights. This process continues for N steps, fitting N simple (or "weak") learners, where N is a hyperparameter to be tuned. The advantage of ensemble methods is that they are unlikely to overfit due to the weak nature of the base classifiers, but they're also unlikely to underfit due to the sample re-weighting and the amount of classifiers present.

Gradient Boosting

Gradient boosting is an adaptation of the AdaBoost algorithm that makes one simple change: where AdaBoost fits successive learners to re-weighted versions of the original data, Gradient Boosting fits to the residuals between the labels and the current predictions. AdaBoost is actually a special case of Gradient Boosting.

Random Forest

Random forests are somewhat the opposite of boosting methods. Where boosting works by fitting a series of *weak* learners to different versions of *all* labels, random forests work by fitting a series of *strong* learners to different *subsets* of the labels. This technique is called "bagging", short for bootstrap aggregation. Bootstrapping is the technique whereby one takes random samples of the data with replacement as training sets for each learner; when these strong learners fit to subsets are aggregated, their high variance is mitigated and bias is introduced, thus combating the bias-variance tradeoff in a very effective way. To boost is to start with a high bias learner and add variance; to bag is to start with a high variance learner and add bias.

Logistic Regression

Logistic regression is an extension of linear regression for classification purposes. A linear model is fit using iterative methods (closed form methods such as least squares are not available in this setting due to the residuals not being normally distributed), and that is passed through the logistic function:

$\sigma(t) = \frac{1}{1+e^t}$

which produces a probability that the result is a positive label. Typically, for multi-class, the solver will invoke a one-vs-all method that produces one classifier per label. Logistic regression works great when there is a roughly linear relationship to be exploited, and it is very fast to predict with (plug the values into the equation), although training can take some time.

Naive Bayes

Naive Bayes is a simple application of the basic rules of conditional probability. Using the training data, it computes conditional probabilities for each label given a feature value. The product of all of the conditional probabilities for an example's features for a label is its probability of having that label, and the most likely label is predicted. This model makes a strong assumption about the conditional independence of the features, since their probabilties are simply mulitplied together. However, it has been shown empirically to be successful even when applied to data that does not satisfy this property. This algorithm is fast to train and fast to predict with, and is among the simplest of machine learning classifiers.

Regressors

Decision Tree

Like a lot of algorithms, decision trees can be extended from classification to regression with a surprisingly simple tweak. Whereas classification trees use their discrete labels to calculate metrics like entropy for measuring a split's effectiveness, regression trees use their continuous labels to calculate metrics like mean squared error. This change in the cost function is really the only thing necessary to extend decision trees from classification to regression; it then continues to split as described before.

Extra Trees

"Extra" trees has a somewhat confusing name, as the word "Extra" does not mean there are more trees; rather, it is a portmanteau of "Extremely" and "Random"; it is essentially a random forest, with extra "random". This extra randomness comes from the way in which splitting is conducted with respect to the features: the point at which data is split is entirely random. This means that for e.g. a continuous feature with range 0-100, the split could come at any point between 0 and 100. ExtraTrees can be faster to train, but often grow much bigger than the trees in random forests, due to the drop in performance in the splits (random instead of optimal).

AdaBoost

See Regressors –> Decision Tree. The trees in this ensemble are regression trees now, rather than classification trees.

Random Forest

See Regressors –> Decision Tree. The trees in this ensemble are regression trees now, rather than classification trees.

Linear Regression

A classical statistical method that can be derived from many mathematical angles. From the perspective of linear algebra, linear regression is a least squares approximation of the solution to $Ax = b$, where m > n and there is no exact solution; the solution comes from projecting b into the column space of A, which leads to the revised equation $A^T Ax = A^T b$, which has one solution x' and is solvable, since A and b are known. What's interesting about this derivation is that it makes it clear why the least squares solution is optimal: the error vector, $e = b - prj_{C(A)}b$, is by definition orthogonal to every vector in C(A), which includes every column of A. When fitting with an intercept, one column of A is the vector of all 1's, which means $e \cdot [1...1] = \sum_i e_i = 0$. Therefore, the total error of the fit is guaranteed to be 0. Linear regression, as the name suggests, is best used when the relationship is assumed to be linear, as it cannot capture non-linear interactions between features. Linear regression has some other fundamental assumptions, including normally distributed errors, homoscedasticity (similar variance between features and target), and the absence of multicollinearity between features. Linear regression can be trained in closed form if the data matrix is small enough (as it simply involves solving the equation previously mentioned, which can be done without inversion). As the size of the data grows into the millions of records, however, gradient descent may be preferred.

Ridge Regression

Ridge regression adds a term to the sum-of-squared-errors cost function that is equivalent to an L2-regularization on the parameters of the model. The regularization term is added to combat multicollinearity in the data (which, as mentioned, is an assumption of linear regression). There is a Bayesian interpretation for this model, which has its own implementation in sklearn, and both will be explored.

Lasso Regression

While ridge regression is essentially linear regression with L2 regularization, Lasso regression can be thought of as linear regression with L1 regularization.

Unsurprisingly, nearly every (non-linear) algorithm is based on decision trees, as they are among the fastest classifiers to train and test with, and speed is a concern. Once a classifier and regressor have been selected, they will be tuned using a grid search approach over their candidate hyperparameters. This grid search optimization will be conducted separately over each of the 7 datasets; the selection of each algorithm, however, will be universal.

No feature selection or significant transformation will be performed prior to training, as these features should in theory be designed optimally for the task. Because all of this optimization, from algorithm selection to hyperparameter tuning, will be done using cross-validation, the testing set will be reserved entirely for a final evaluation of the project as a whole. The metrics calculated as part of that evaluation will be over the aggregation of all testing sets across situations; for example, the metric to be compared to the benchmark will be the overall F1 score calculated by summing all of the confusion matrices to provide a single number.

Benchmark

For the classification task, the most basic benchmark is the proportion of the most common label, either fold or check. If folds make up, e.g., 60% of labels in one dataset, any classifier should be able to get greater than 60%, otherwise it is no better than a naive model that just predicts folds. For the non-bet-facing datasets, this will be the proportion of checks; for the bet-facing datasets, it will be the proportion of folds. I would anticipate the accuracy of each model to be at least 10% greater than its greatest proportion. As a secondary benchmark, there have been reported accuracy scores using deep networks of 90%[4], so I

[4]https://www.aaai.org/ocs/index.php/WS/AAAIW13/paper/download/7132/6470

would love to be able to beat that (though my metric is F1, not accuracy score, so 90% is a soft value).

For the regression case, my focus was centered around a decision I made early in the project design. Originally, my architecture was similar to a paper[5] I discovered, where the author proposed a single multi-class classification with labels that were essentially binning the bet/raise values at 1/4 pots. To improve on this, I split the problem into classification and regression. If my modification is truly an improvement, I shouldn't have a mean error greater than 1/4 pot, otherwise the classification bins are likely to perform better. For this reason, a mean absolute error of 0.25 is my primary benchmark for regression.

Methodology

Data Pre-Processing

The original data for this project was a corpus of game logs from various online poker sites, such as PokerStars and PartyPoker. These game logs had errors, missing information, names were obfuscated, and because it was from many sources, it had many different formats. The first and most time consuming task of this project was to parse this text into a database of structured data, and then build features from that data.

The parsing from text to tables was done in `data/lib/fileReaders.py`, where I wrote 5 separate parsing functions for the 5 data sources, then using Bash and MySQL stored them in a relational database. The database had 3 tables: `actions`, `games`, and `boards`. `games` contained general information for each hand that was played, boards contained the community cards (mapped to ints) for each hand, and actions contained all remaining information for every action in the dataset. This process, for all ~33,000 files, took approximately 5 hours on my local machine. Once the data was parsed, the next step was to construct the feature set, which was done in `data/lib/buildDataset.py`. The result was 114 features covering various aspects of the environment (discussed above in Problem Statement), most of which were specific to certain subsets of the data. Because of the specific nature of these features (which fell out of the inherently different underlying structure of each situation), 7 subsets of the data needed to be separated. This brought the problem from a task of learning one model to learning many, though their structure would remain the same.

In terms of feature transformation, only one change was needed, which was to convert LastAction (a string variable) to numeric. This was converted to a series of dummy variables.

There were few abnormalities in the data in the sense of illegal or improbable actions. The sites are run with restrictions in place to prevent actions that would

[5] http://www.ai.rug.nl/~mwiering/Tom_van_der_Kleij_Thesis.pdf

result in bad data, and because the stakes of these games were quite high in some cases (as these were games played with real money), the decision-making of the players can be trusted somewhat more than a fake-money site. That said, there were numerous instances in the raw text where information had been overwritten and lost, or errors had been made in recording that resulted in strange values, but for the most part these were cleaned in the orginal processing. Thankfully, the obfuscating of names was consistent, so there was no confusion as to which player was which, even though their real names had been removed.

As a final step of pre-processing on the features, I removed all players with less than 50 actions, as their play-style summary statistics would not be accurate and would not be good data to train with. To avoid computational challenges, particularly with fitting the data in-memory, I limited training to the first 1 million rows of each dataset (if it was less than 1 million rows, I used it all).

For the labels, the main concern was around all-in plays. These bets and raises were of course associated with abnormally large values, and the question I was faced with was whether to include them in the prediction task. My final decision was ultimately to remove them from the dataset; all-in plays are very rare compared to other actions, and for an agent to predict such a move (which would ultimately end that imagined trajectory and not lead to any more useful data) was something I deemed unnecessary.

Overall, the hand-crafting of this feature set straight from raw text was the most intensive of all tasks involved in this project.

Implementation

Because the data did not have a hierarchical structure that could be naturally exploited by deep learning, and the features were already hand-crafted, I opted instead for a more traditional supervised learning approach. And rather than settle on one algorithm, I thought it would be best to test as many as possible and compare their performance. Having selected a set of algorithms to test and a set of metrics to compare with, as discussed in Algorithms and Techniques, I first loaded in the dataset I would use to choose the algorithm. Because testing every algorithm on every dataset with cross validation would be too time consuming of a task, I decided to select just one dataset of the 7 and choose an algorithm based on that. I selected the Preflop-True dataset, as it was the largest (and therefore most reliable), but also contained the minimum number of features; if an algorithm can learn effectively with few features, it should be able to learn with many.

Before reporting the results, I should note that a special implementation was required in order to obtain an overall F1 score for all classifiers. This was due to their distinct label sets.

```
import numpy as np
```

```
fullConfusion = np.zeros((5,5))

for df in ['Preflop-True','Flop-False','Flop-True','Turn-False','Turn-True',
           'River-False','River-True']:
    isFB = df[-4:]=='True'

    #####
    ## do grid search and fit optimal learner, get predictions and store in finalPreds
    #####

    finalConfusion = confusion_matrix(y_test, finalPreds)
    inds = [0,2,4] if isFB else [1,3]
    for i,row in zip(inds, finalConfusion): fullConfusion[i,inds] = row

def F1fromConfusion(confusion):
    allF1scores = []
    allProportions = []
    for i in range(confusion.shape[0]):
        # get 2x2 confusion matrix
        newConfusion = np.zeros((2,2))
        newConfusion[0,0] = confusion[i,i] # TP
        newConfusion[1,0] = confusion[:,i].sum() - confusion[i,i] # FP
        newConfusion[0,1] = confusion[i,:].sum() - confusion[i,i] # FN
        newConfusion[1,1] = confusion.sum() - newConfusion.sum() # TN
        # get F1 from newConfusion
        precision = newConfusion[0,0] / newConfusion[:,0].sum()
        recall = newConfusion[0,0] / newConfusion[0,:].sum()
        f1 = 2 * (precision * recall) / (precision + recall)
        allF1scores.append(f1)
        # get proportion
        prop = confusion[i,:].sum() / confusion.sum()
        allProportions.append(prop)

    return np.dot(allF1scores, allProportions)

print "Final F1 score is:", F1fromConfusion(fullConfusion)
```

The idea was to build up a full 5x5 multi-class confusion matrix by updating it with each learner, then manually convert that confusion matrix into a multi-class F1 score. This is how all classifier metrics to follow were obtained. The results were as follows:

Classifier	TrainF1	ValidF1	AvgPredictTime
DecisionTreeClassifier	1.0	0.619761128146	0.2604629
AdaBoostClassifier	0.634530995129	0.633517587204	2.3664752
GradientBoostingClassifier	0.673310587923	0.67026916632	2.3892453

Classifier	TrainF1	ValidF1	AvgPredictTime
RandomForestClassifier	0.988985311689	0.625657948956	1.1078702
LogisticRegression	0.612976360242	0.612559856093	0.1191354
GaussianNB	0.558917368124	0.558547428167	0.4723037

Some thoughts on each:

- The decision tree overfit the training data maximally, as decision trees do, with an F1 of 1.0; it was among the fastest predictors, though
- AdaBoost and Gradient Boosting took the longest to predict by quite a margin (both 3 times that of the 3rd slowest), but their validation scores were the best of the bunch. They are certainly candidates, but the prediction time is a concern that would have to be tuned for
- Random forest performed comparably to AdaBoost and Gradient Boosting, but it did so with predictions that took almost 1/3 of the time; like the decision tree, it overfit, but this is tunable
- Logistic regression seemed to have trouble approximating this function, as its training and validation score were both worse than each of the tree-based methods. Its predictions were the fastest of all of them, but its performance removes it from the discussion
- Naive Bayes was by far the worst of the bunch and cannot be considered

With these results, the decision is actually fairly easy: the random forest struck the nicest balance between validation set performance and prediction time. While the boosting methods did outperform it, the random forest would allow for twice as much planning and lookahead for a reinforcement learner applying it, which is worth far more than an extra percentage point or two of accuracy.

Selection: **Random Forest**

Next I compared the regression methods, with these results:

Classifier	TrainMAE	ValidMAE	AvgPredictTime
DecisionTreeRegressor	8.46964077974e-16	0.137425022878	0.1861917
ExtraTreesRegressor	3.34280761525e-16	0.118336210259	1.2406995
AdaBoostRegressor	0.286253607113	0.286290802092	0.6138471
RandomForestRegressor	0.0485754317762	0.126793287737	1.1153137
Ridge	0.234291926431	0.234371875722	0.0618735
BayesianRidge	0.233553628069	0.233641849719	0.0614729
Lasso	0.283842175409	0.283865354025	0.0676938
LinearRegression	0.233477795526	0.233564820964	0.0620243

Breaking down each result:

- The decision tree once again fit perfectly to the training data, but it actually came out with an admirable validation score. I would generally prefer to use an ensemble method, but the performance of the single decision tree is encouraging

- Extra Trees overfit just like the decision tree, but it out-performed the decision tree on the validation data; its prediction time was the slowest, however, which makes it somewhat unusable
- AdaBoost was incredibly disappointing, with even worse validation error than the linear methods. This might have been due to the max depth of the base trees, but I can't say for sure. It was faster than the other ensembles, but is clearly not a good candidate
- AdaBoost was not far behind Extra Trees in validation accuracy, but where it really suffered was prediction time; it took 3 times longer than any other algorithm to calculate its predictions, which makes it unusable
- Random forest was similar to Extra Trees, though it overfit less and predicted faster. Its predictions were still slow compared to the other methods however, and it would be difficult to justify using it
- All of the linear models, with their various regularization techniques, performed horribly on the validation data, in particular base linear regression, which informs me that this is a non-linear relationship and these should not be considered

At first glance there seems to be an obvious first choice. The decision tree had solid performance on the validation data, and its predictions were 5 times faster than the random forest. However, a single decision tree always overfits, and this one was no exception; my concern is that if I rely on hyperparameter tuning to add bias to the model, its validation score may drift away. In general, ensemble methods are almost always preferred to single trees. By choosing an ensemble method, I would be increasing prediction time by 500%, but this is a sacrifice I'm willing to make; this model will only be invoked on bet and raise predictions, which will likely be around 20% of the time based on label distributions. For this reason, speed is not as much of a concern as it was with the classifier, which will be required at every step.

Selection: **Random Forest Regressor**

With the classifier and regressor chosen, I could move on to hyperparameter tuning, which will be discussed in the following section.

Refinement

The classifier was tuned with the following dictionary of possible hyperparameter values:

max_features	max_depth
10	5
30	10

max_features	max_depth
50	20
70	40
None	60

Of note is the absence of n_estimators in the grid search space. It has been my experience that random forests with more trees always perform at least as well as those with fewer trees; for this reason, I chose to set n_estimators to a large value by default (200), speeding up the grid search while in all likelihood maintaining the same performance. I am essentially operating with the assumption that grid search would choose the largest n_estimators anyway. In addition, rather than perform grid-search (whose purpose is to find values for the other hyperparameters) using `n_estimators=200`, I will drop it to 50. The assumption here is that values of max_depth/bootstrap/max_features that work well with 50 estimators, which is already a lot, should also work with 200, but with 200 the performance should be greater.

The best set of hyperparameters found by GridSearchCV, for each dataset, was:

Dataset	max_features	max_depth
Preflop-True	None	40
Flop-False	None	10
Flop-True	None	60
Turn-False	None	40
Turn-True	None	20
River-False	None	40
River-True	None	40

Meanwhile, the regressor was tuned with the same grid search values, and its best hyperparameters by dataset were:

Dataset	max_features	max_depth
Preflop-True	10	5
Flop-False	10	5
Flop-True	10	5
Turn-False	10	5
Turn-True	30	5
River-False	10	5
River-True	10	5

Beyond hyperparameter tuning, not a lot of refinement could be made to these learners. The final results will be discussed next.

Results

Model Evaluation and Validation

The final set of models produced the following F1 scores for each dataset, tested on a held-out testing set that has not been encountered thus far:

Subset	F1
Preflop-True	0.864
Flop-False	0.680
Flop-True	0.678
Turn-False	0.440
Turn-True	0.703
River-False	0.418
River-True	0.665

And the overall F1 score was 0.670.

The mean absolute error for each test set was:

Subset	MAE
Preflop-True	0.233
Flop-False	0.131
Flop-True	0.256
Turn-False	0.159
Turn-True	0.238
River-False	0.202
River-True	0.282

And the mean of these is 0.215.

By holding out this data until this point, I have given this data the ability to represent a sample of the real world, which means the results here should accurately describe the model's generalizing ability, and its results can be trusted.

Having said that, these classification results are shockingly terrible. It seems that the Preflop dataset was the only one with any real ability to be predicted, as the F1 scores of every other dataset failed to reach even 0.7. The regression didn't drop off as steeply, but still performed quite a bit worse. In an effort to explain this astounding decline in performance, I checked the top 5 features of the Preflop classifier by feature importance (given by a decision tree fit):

Feature	Importance
ESvsAgg	0.145

Feature	Importance
AggStack	0.116
StackToPot	0.079
PreflopCallPct	0.059
AggInPosVSMe	0.046

3 of these features are based on information about the aggressor, or the player that made the bet that the agent is facing. That means that the most useful information to predict a player's action is about the bet that they face, which could explain why non-bet-facing situations are so difficult to predict. What it doesn't explain is why successive bet-facing situations are also so difficult.

The regressors performed quite a bit better, and most managed to break the benchmark I set. Ironically, the non-bet-facing situation was easier for this task, where it was predicting bet sizes instead of raise sizes. The grid search results, creating as simple of models as they did, lead me to believe this prediction task may have actually been much easier than I anticipated, and that really the action selection is the complex problem for a player.

A point to make about the feature set is that some of them, the ones that describe a player's style (e.g. PreflopCallPct) were calculated using the entire dataset, which could be seen as a form of cheating (using future information). I would justify this choice by saying, to calculate these features incrementally would be only creating worse estimates of the true values (a player's true style), and would only make the performance worse. In a real setting, these features could still be constructed, they would just be viewed as approximations that would be improving over time. As an agent gets deeper into a game, these features will start to converge on their true values, and the learners will perform as they did in training/testing. An agent could never sit down at a poker table and immediately know everything about all the players, so its predictions at the beginning will be inaccurate anyway; it's how the model performs later in the game that matters, and that's why I decided to use "illegal", but more accurate, estimates of player style.

Justification

The results of the final solution were wildly variant. The first result was a solid F1 score for the the Preflop-True classification model, which outperformed my base benchmark by a healthy margin, and just missed the cut on the secondary benchmark. As will be discussed in Improvement, I think I could find a way to eclipse 90% for this setting. The second result was the rest of the classification, which came nowhere close to 90%, and failed to even break the base benchmark of "largest proportion + 10%". This result was disappointing, and is certainly an area to return to in future work, again possibly with stronger learners. The regression case was generally more successful, as most subsets were able to beat the 0.25 benchmark I set, and on average they were below it. However, the regression model is only as useful as the classification that invokes it, so while this was a victory, it was a small one. I cannot conclude that these models as they stand have solved the problem of opponent modelling in poker, and future work must be done to improve on these results.

Conclusion

Free-Form Visualization

For this visualization, I decided to depict how much computation is required for each stage of this project. From parsing the text files, to building the many kinds of features, to training the classifiers and regressors, each stage is a heavy computational burden, and I think this visualization makes that clear.

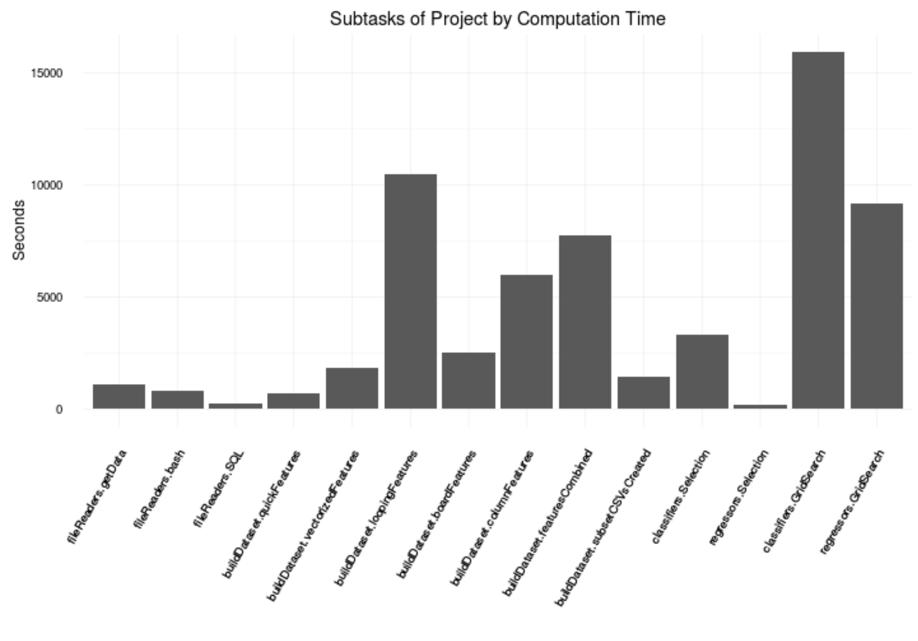

Figure 4: Subtask timings

As you can see, because I limited the amount of text files to about 10%, parsing only took roughly 20 minutes to complete. Had I given it all raw data available, it would have been around 4 hours. Building the dataset is a long, painful process, with many stages, and these are shown here, broken down by the style of computation needed. The features that required me to loop over all rows and hold dictionaries containing certain information were among the slowest, while the simple SQL queries were lightning quick in comparison. Finally, training the classifiers took much longer than training the regressors, due to the amount of data available for regression; only bets and raises were used for those, which was a small subset of the whole dataset.

Reflection

This project started when I found a large corpus of text logs of poker games on an old website. I downloaded the data and examined the text files, and found that they were well structured, and contained enough information for me to extract interesting patterns from. This was in contrast to the other poker databases I had found, which had very specific features and in most cases did not even fully describe each action. I knew it would be a challenge to parse this from raw text to a full feature set, and it was, but I felt it was worth it. After writing the original site-specific parsing functions, I used a few quick bash lines to handle the large resulting CSVs, and then imported all the data into a MySQL database for easy retrieval. This turned out to bring a whole new set of challenges as I navigated the world of relational databases, and in some ways it only made things more difficult.

From there, I got work building the feature set. Originally, I tried to copy the feature set of related paper, but I found these features weren't performing nearly as well as I wanted. So after spending a few days watching and studying the game of poker, I constructed my own list of features. I did this systematically by breaking the game down into 4 categories: player tendencies, opponent information, game state, and community cards. By developing features for each of these categories, I felt confident that I had represented the information that would be relevant to a professional poker player.

Once the data was finally all together, I had the challenge of deciding how to deal with the distinction between action types and action values. Originally, I planned to bin the values of bets and raises and create new labels from these, but I felt that this would lose too much information and would ultimately be less useful for the reinforcement learning agent that these models were designed to be fed to. By considering the situation as two separate problems, I essentially created twice the work for myself in generating models, but my hope was that it would be worth the effort.

Actually finding and tuning the classifier and regressor were, as they so often are, the quickest parts of this project. Because of the ease of using Python libraries like Pandas and sklearn, the fitting and predicting process was a breeze. Where I ran into challenges here, however, was that is caused me to realize how many of my features were actually leaking information that the agent was not supposed to know. For instance, when I fit my first decision tree as a classifier on the River-False dataset, it got a near perfect F1 score of 0.999. I knew this was too good to be true, and as it turned out, I had multiple features indicating in some way the amount associated with the action (which, of course, perfectly distinguishes between checks and bets). After a few more of these discoveries I eventually got my feature space to be “legal”. Ultimately, this project was really a test of my ability to engineer a useful feature set for a complex problem, and the algorithm selection process could be viewed as simply a tool to evaluate that.

The final model was quite a disappointment, and if it were used to inform a poker-playing agent, the simulation and lookahead would be too error prone to provide any useful trajectories. In fact, it would likely lead the agent off course and cause it to miss the optimal policy. If this is as good as opponent modelling gets, the agent is better off going model-free.

Improvement

All that said, I feel that the computational limits that were imposed by my local machine seriously limited my ability to learn the problem, and that there are a few very important improvements that could be made.

1. Using all of the data. I originally parsed about 270 million rows for the feature set, but time and memory limits prohibited me from using it all. If a learner was fed that amount of data, I have no doubts in its ability to surpass the 90% benchmark I set, with the well-behaved dataset.
2. Deep networks. It’s no secret that the combination of deep learning and big data is an effective one, and with the amount of interaction between the many facets of the game, and all that goes into a player’s decision-making, I’m sure the architecture of these networks would fit the problem well.

Applications

As previously mentioned, this project is not meant to be a standalone model. My original project idea was to build an entire poker-playing AI, using model-based reinforcement learning techniques such as policy gradients to learn the optimal action in every situation. Because poker is a game of imperfect information, certains gaps in the agent’s knowledge have to be estimated; these are the play style of the opponents, their hole cards, and the actions they might take in response to its own. By creating a supervised model to predict actions, I attempted to solve the last of these tasks. The problems of guessing hole cards and identifying play style are density estimation and clustering problems respectively, and ones that deserve to be looked at separately. Once these tools have been generated, the agent can begin to learn with its fully constructed MDP, and hopefully converge on a near optimal solution. Of course, it won’t be perfect, and it will only be as good as the models it is given, and this is the problem I was faced with in this project. I plan to continue on in this journey, in the hopes of one day releasing the full agent to compete against others and beat the game of No-Limit Texas Hold’em Poker.