Learning a Double Dummy Bridge Solver

Michael Mernagh
mernagh@stanford.edu

1 Introduction

Contract Bridge is a interesting area for AI research. One of the outstanding areas involves improving the cardplay, which starts with 52 cards uniformly distributed between 4 players. This is an imperfect information scenario where only 26 of the cards' locations are known, with a large branching factor, so the state space of all possible plays to evaluate is about 2.5×10^{21}, which is far too large to cover. Instead, after reaching some depth, a heuristic function must be quickly applied to estimate the value of the the current state. Double Dummy Solvers are a class of such functions that operate on a perfect information scenario.

In this paper we describe efforts to improve Double Dummy Solvers for each possible number of cards in hand. We learned 13 different functions, one for when players are holding the original 13 cards, one for when they are holding 12 cards, etc. Each function for a given *hand size* maps a game state to an integer $\in [0, \textit{hand size}]$. The game state is represented by the cards held by each player. We then used several different learners (softmax classification, random forests, and gradient boosting trees) to create a function that predicts the number of tricks that the North-South team will win.

2 Related Work

Although not much attention has been given to the topic, several researchers have explored different strategies. Mossakowski and Mandziuk [7] tried to create predictors only for the the initial hand, when all players have 13 cards. They used a very simple game state representation (the cards held by each player), and fed this into a neural network. While this approach seems promising, since the neural network may be able to extract meaningful features, in practice they obtained only about 35% accuracy on the more difficult notrump scenario.

On the opposite end of the spectrum, Smith et al. [9] included as much domain expertise as possible in constructing a hierarchical task network that encoded several hundred specific bridge strategies. Despite the considerable effort that this approach required, it performs merely as well as a novice to intermediate human.

Several intermediate approaches have attempted to reduce the search space. Most notably, Matthew Ginsberg describes partition search in Ginsberg [4], which is used in the commercial software GIB. Several other authors have attempted various combinations of search pruning, including notably Chang [1]. Detailed pseudocode is described by Haglund [5]. These approaches are of limited use, however, because even at the fastest speeds on a single machine they still take > 10 seconds, and even with the most extreme pruning (up to $\frac{1}{2}$ of the leaves), a complete search would require 5×10^7 seconds, or more than 19 months.

3 Dataset and Features

3.1 Original Data

We obtained more than 300,000 records of professional tournament games from `http://www.bridgetoernooi.com`, packaged into several hundred archive files. The data for each game was recorded in PBN format, described at `http://www.tistis.nl/pbn/`. Game information includes essentials, such as the cards dealt and the order that cards were played, and supplemental information, such as the names of the players.

3.2 Pre-parsing

For each game, we extracted the score, the cards dealt, the winning bid, the first player to play, and the cards played. Using this information we determined the number of tricks won by the North-South team. We then stepped through the cardplay, updating cards held by each player, and keeping track of the tricks that the North-South team won.

Table 1: Sample set sizes

Hand	1	2	3	4	5	6	7	8	9	10	11	12	13
Training	4812	8949	15256	21426	27021	31700	36041	39969	43233	45366	46574	53606	48864
Test	2062	3835	6538	9182	11581	13586	15446	17129	18529	19442	19960	20942	22974

After each trick, we stored a representation of the current game state, consisting of the number of tricks that North-South won after this state, and the cards held by each player. To facilitate the features we would extract, we normalized the state representation by "shifting up" the cards held by each player. For example, if Player W held the ace and queen of spades, but someone had already played the king of spades, then we represented this as the ace and king of spades. We also shifted suits, by swapping the trump suit holdings with spades.

We stored all the states for trump and no trump games separately, and we also separated the records for each hand size. In the end we wrote 26 files, each of which held records similar to

```
4 6.J6.A82.QT98 AQ.AQ.Q53.J76 K987.97.KT76. JT.KT8.J94.AK
```

The first integer is the number of tricks that NS won after this hand. The next four space-separated sections are a representation of the cards held by north, east, south, west, respectively, where each suit is separated by a period, and the suits in order are spades, hearts, diamonds, and clubs.

For all subsequent training, we used a random 70-30 split of the data. Note that because of the way that PBN stores games, not all hand sizes were represented for each game, so there is more data for larger hand sizes.

3.3 Features

From each of these records, we extracted a feature vector in $\mathbb{Z}^{80}$. For each suit held by each player, we used 5 features:

- The number of cards that the player has of this suit.
- The sum of the ranks of the cards that the player has in this suit, where rank is in $[0, 14]$.
- The largest rank of the cards that the player has in this suit.
- A feature for the the stopper card, if any, that the player has in this suit. The value of this feature is 7 minus the number of cards in this suit that the player would lose before they held the highest card in this suit. For example, an ace has a rank of 7, a king and some other card has a rank of six, and a single king has a rank of zero (no stopper card).
- The number of sure winners that the player can count on in this suit, if the player's strategy is to always play the highest card in this suit.

4 Methods

4.1 Softmax classification

Softmax regression is a generalized linear model that extends the idea of logistic regression to a multi-class prediction in which $y \in [1, K]$, and $p(y = i \mid x; \theta) = \frac{\exp(\theta_i^T x)}{\sum_{j=1}^K \exp(\theta_j^T x)}$. So to fit the model, we want to minimize the cost function:

$$J(\theta) = -\sum_{i=1}^{m} \sum_{k=1}^{K} \mathbb{1}\{y^{(i)} = k\} \log \frac{\exp(\theta_k^T x^{(i)})}{\sum_{j=1}^K \exp(\theta_j^T x)}$$

The gradient of this with respect to θ_k is

$$\nabla_{\theta_k} J(\theta) = -\sum_{i=1}^{m} x^{(i)} \left(\mathbb{1}\{y^{(i)} = k\} - P(y^{(i)} = k \mid x^{(i)}; \theta) \right)$$

Since the cost function is convex, we can solve it using gradient descent.

4.2 Random forest

Random forests[6] are an ensemble method over a multitude of decision trees. Each decision tree is formed by recursively splitting the data into separate branches. To determine what feature should be used to split the data, we used Gini impurity, where the data under consideration has K labels

$$I_G = \sum_{k=1}^{K} \sum_{j=1}^{K} \mathbb{1}\{j \neq k\} \frac{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = j\}}{m} \frac{\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = k\}}{m}$$

After the trees have been constructed, each leaf (which corresponds to a class prediction) is assigned a probability based on the fraction of samples that it represents.

To make a prediction on a sample, a random forest outputs the weighted average of the classes that are predicted for the sample by each tree, where the weights are the probabilities of the individual predictions.

The individual trees are trained on random subsets of the data. Moreover, at each node, a random subset of the total number of available features are selected. This is done to avoid correlation between trees, many of which would be strongly affected by a small set of features.

4.2.1 Extremely randomized trees

Extremely randomized trees, proposed by Geurts et al. [3], differ slightly. Instead of computing the optimal split from a random subset of features, a feature is selected at random, and a split along that feature is also selected at random. The entire data is then used to grow each tree.

4.3 Gradient boosting trees

Gradient boosting trees, as described by Friedman [2] are a stage-wise boosting approach that iteratively adds to the prediction function. At each iteration, a gradient descent step is taken to approximate the best addition to the prediction function.

$$-\arg\min_{a} \sum_{i=1}^{m} L\left(y^{(i)}, F_{t-1}(x^{(i)}) - a\frac{\partial L\left(y^{(i)}, F_{t-1}(x^{(i)})\right)}{\partial F_{t-1}(x^{(i)})}\right)\left(\sum_{i=1}^{m} \nabla_{F_{t-1}} L\left(y^{(i)}, F_{t-1}(x^{(i)})\right)\right)$$

But since determining the actual gradient descent can be hard for arbitrary loss functions, we instead fit a regression tree to the *pseudo-residual* for each possible class, and use a search like Newton's method to find the value of the first argument. The loss function that we are minimizing is the multinomial logistic loss function.

5 Results

For all the models generated, the metric we considered paramount was accuracy: the fraction of predictions that were correct. This is the only metric that matters for a computer bridge player, since it is an unbiased estimator of the accuracy of a new prediction. However, we also collected precision and recall metrics, and for some of the models that we built as regression models, we also collected a modified sum of squared error metric: $R^2 = \frac{\sum_{i=1}^{m}\left(y^{(i)} - f(x^{(i)})\right)^2}{\sum_{i=1}^{m}(y^{(i)} - \hat{y})^2}$, where $\hat{y} = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$.

5.1 Softmax classification

We used stochastic gradient descent to minimize the multinomial logistic loss, with a step size of $\frac{1}{\sum t}$, and tried several different numbers of iterations. We used a batch size of 1, since on a fast machine the runtime was not unbearable. (It took 12 minutes to run 2500 iterations on all 13 hand-sizes).

The results for different numbers of iterations did not vary much. The accuracy was very low–for most of the hand sizes, it was about 20% greater than a random guess. The variance however, was also very low, indicating that the models generalized well.

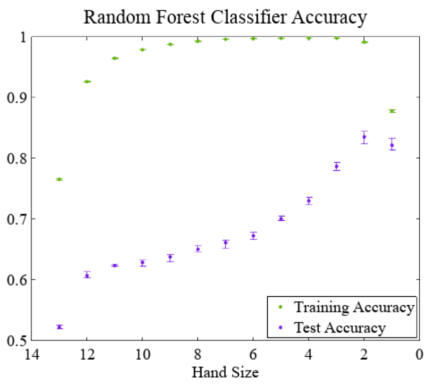
The most likely explanation for the poor results is that the model assumption that the data can be modeled by a linear predictor is incorrect. The next models took this possibility into account.

5.2 Random forest

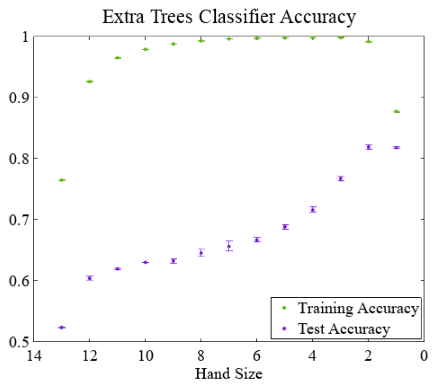
We built both classification and regression trees, and used both regular random forests and extremely randomized trees. We tried different numbers of trees, and at each split considered $\sqrt{f}$ features, where $f = |\text{available features}|$.

Table 2: Softmax classification accuracies

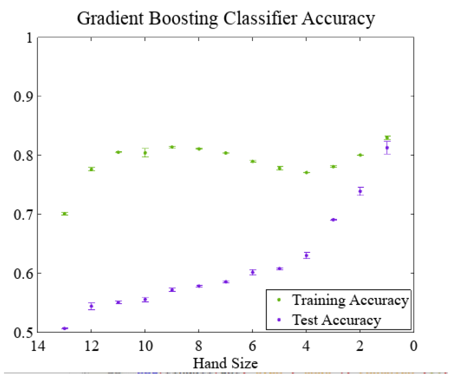
Iterations	100		500		1000		2500	
	Training	Test	Training	Test	Training	Test	Training	Test
13	24.95%	24.13%	26.15%	25.21%	26.60%	25.32%	26.99%	25.51%
12	25.31%	24.28%	26.38%	25.58%	27.03%	26.32%	27.14%	25.72%
11	25.58%	24.82%	27.73%	26.24%	28.00%	26.60%	27.95%	26.47%
10	26.99%	26.37%	29.74%	28.52%	29.69%	28.42%	29.62%	28.89%
9	27.95%	26.89%	30.85%	29.79%	31.26%	30.05%	31.36%	29.68%
8	30.82%	30.29%	32.78%	31.39%	33.72%	31.41%	33.66%	32.03%
7	32.37%	31.52%	34.76%	33.52%	35.21%	33.73%	35.59%	34.56%
6	34.37%	33.70%	36.64%	35.93%	37.55%	36.47%	37.98%	37.44%
5	35.68%	35.41%	39.23%	37.67%	39.36%	39.10%	40.13%	38.31%
4	39.98%	38.85%	42.99%	41.13%	42.60%	41.36%	42.97%	42.87%
3	43.20%	42.41%	46.00%	45.97%	47.95%	45.80%	47.93%	46.80%
2	52.57%	50.91%	56.43%	55.63%	56.15%	56.52%	57.17%	55.92%
1	74.75%	74.84%	76.99%	75.76%	77.45%	75.91%	77.93%	76.93%



(a) Random forest classifiers.



(b) Extremely randomized trees classifiers.



(c) Gradient boosting classifier

Figure 1: Accuracies and error bars for ensemble decision tree classifiers.

As with softmax classification, the results for different numbers of trees did not vary much. However, the accuracy here was much better, and a marked improvement over the neural network results obtained by Mossakowski and Mandziuk [7]. However, the variance was also much higher, which is perhaps somewhat to be expected given the nature of trees to easily overfit. The variance did not disappear when we adjusted the number of trees–we tried sizes of 100, $\frac{\text{number of training samples}}{70}$, $\frac{\text{number of training samples}}{35}$, $\frac{\text{number of training samples}}{10}$, $\frac{\text{number of training samples}}{7}$, 7000, 8500.

As can be seen in Figure 1, the number of trees generated had minimal impact and the two methods had very similar accuracies. When comparing results for random forests created with regression trees (see Figure 1), we again observe that the two methods performed similarly.

5.3 Gradient boosting trees

We used a fixed learning rate of 0.1, and built trees with maximum depths of 3, 4, and 5, all of which returned nearly identical results. We also built classifiers with varying numbers of stages: 100, $\frac{\text{number of training samples}}{70}$, $\frac{\text{number of training samples}}{35}$, 7000.

The test accuracies obtained were similar to, though slightly smaller than, those obtained using random forests, but the variance was much lower. Nevertheless, an irregular hump centered on hands of size 9 seems to indicate that some overfitting was still occurring, though the test accuracies do not show the same irregularity. The confusion matrix shows that the predictions were approximately normally distributed about the true prediction. Similar confusion matrices were generated for each hand size.

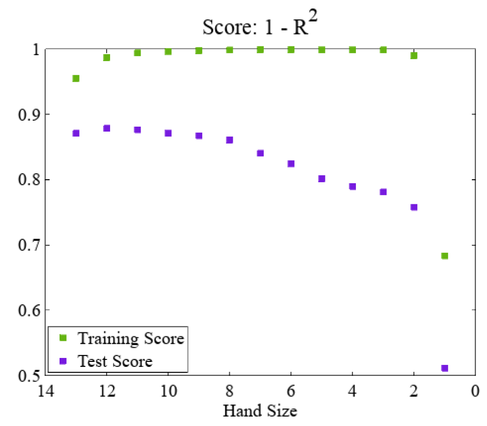


Figure 2: Random forest regressor $1 - R^2$ score.

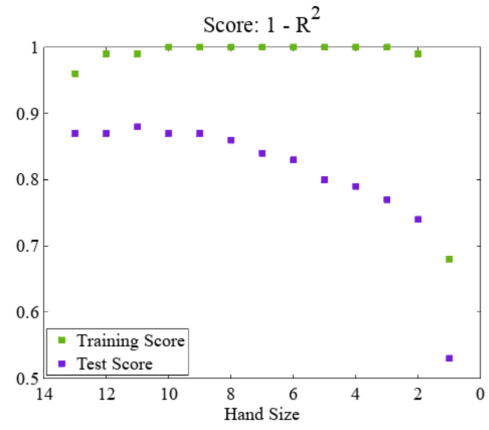


Figure 3: Extremely randomized trees regressor $1 - R^2$ score.

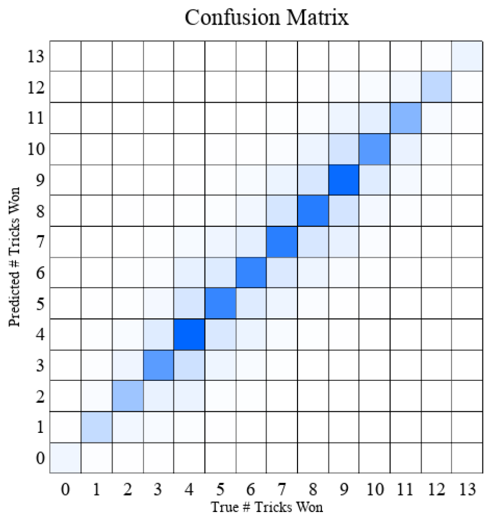


Figure 4: Confusion matrix for gradient boosting.

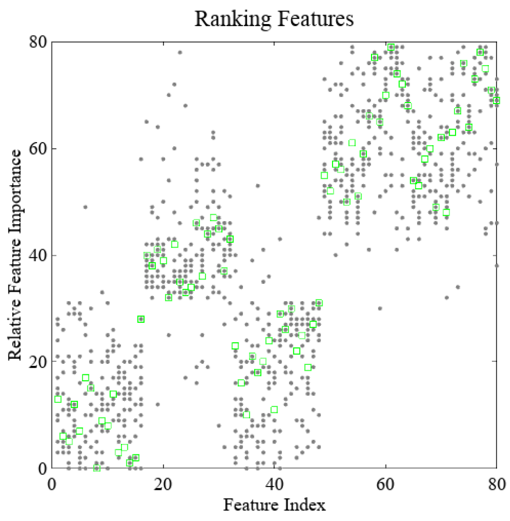


Figure 5: Relative ranking of feature importances

5.4 Feature importances

From Figure 5, it is clear that the most important features are the number of sure winners, the stopper card rank, and to a lesser extent, the rank of the highest card in each suit. These first two features are "contrived" features, which differentiate these models from those trained by the neural networks described in Mossakowski and Mandziuk [7], and explain the large improvements of these models.

6 Conclusion and Future Work

There is much room for improvement on the accuracies. A good start would be to identify even more features that could be relevant. These could include cross-partner features, such as how teammates can swap control. Training on a neural network may also create prediction functions that use relationships between features in a more nuanced way.

We were able to achieve results that were 35% better than the best previously-reported accuracies for Double Dummy Solvers, by using improved features trained on extremely randomized trees. We also achieved similar results using gradient boosting. Softmax classification provided inferior results, because of the likely erroneous linearity assumption. Importantly, such a drastic improvement in results suggests that further improvements are also possible by using even better features and more refined learners.

References

[1] Ming-Sheng Chang. Building a fast double-dummy bridge solver. 1996.

[2] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[3] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[4] Matthew L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

[5] Bo Haglund. Search algorithms for a bridge double dummy solver, 2010.

[6] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995.

[7] Krzysztof Mossakowski and Jacek Mandziuk. Learning without human expertise: a case study of the double dummy bridge problem. *IEEE Transactions on Neural Networks*, 20(2):278–299, 2009.

[8] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[9] Stephen JJ Smith, Dana S Nau, and Thomas A Throop. Total-order multi-agent task-network planning for contract bridge. In *AAAI/IAAI, Vol. 1*, pages 108–113. Citeseer, 1996.

[10] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.