An Empirical Evaluation of Machine Learning Approaches for Angry Birds

Anjali Narayan-Chen, Liqi Xu, and Jude Shavlik
University of Wisconsin-Madison
narayanchen@wisc.edu, lxu36@wisc.edu, shavlik@cs.wisc.edu

Abstract

Angry Birds is a popular video game in which players shoot birds at pigs and other objects. Because of complexities in Angry Birds, such as continuously-valued features, sequential decision making, and the inherent randomness of the physics engine, learning to play Angry Birds intelligently presents a difficult challenge for machine learning. We describe how we used the Weighted Majority Algorithm and Naïve Bayesian Networks to learn how to judge possible shots. A major goal of ours is to design an approach that learns the general task of playing Angry Birds rather than learning how to play specific levels. A key aspect of our design is that the features provided to the learning algorithms are a function of the local neighborhood of a shot's expected impact point. To judge generality we evaluate the learning algorithms on game levels not seen during training. Our empirical study shows our learning approaches can play statistically significantly better than a baseline system provided by the organizers of the Angry Birds competition.

1 Introduction

Angry Birds is one of the most popular video games around the world. Many millions of players use a slingshot to shoot a variety of birds at structures to destroy targets, mainly pigs. Human players, who may easily understand how to play the game, are limited to their visual reasoning and intuition when choosing shots. Conversely, an artificially intelligent agent lacks human intuition and relies on computer-vision techniques to identify objects on the playing field. Our goal is to build, via machine learning, a computer program that plays Angry Birds well.

Playing Angry Birds requires making four choices:

Shot angle: the release angle of the bird being shot. Our algorithms focused on evaluating candidate shot angles; that is, they learn the predicate *goodShot* (for simplicity, we use the term *badShot* to indicate ¬*goodShot*, but note we only predict one variable).

Shot strength: how far the bird is pulled back in the slingshot. For ease, we always execute shots with maximum shot strength to ensure full energy upon impact.

Tap time: when the game's screen is "tapped," the behavior of the flying bird changes. Tap times are learned using a simple statistical analysis of training data tap times; this is elaborated in Section 3.4.

Delay after shot: it takes a while for the game state to settle after a shot. Following prior work, we wait 10 seconds between shots.

Players must interpret different outcomes due to simulated physics of gravity and collisions resulting from the above choices. The problem of designing an intelligent player thus is challenging. We choose to investigate creating an intelligent artificial player of Angry Birds. In doing so, we may learn more about effective machine learning techniques to deal with other dynamic, real-world problems involving large and continuous-valued feature spaces.

The Angry Birds testbed we use is the "Poached Eggs" game set provided by the Angry Birds Chrome plugin (chrome.angrybirds.com). Each level consists of a playing field, containing structures of wood, ice, stones, terrain, and TNT, as well as one or more pigs (if the game is unfamiliar, see Fig. 3). To complete a level, one is required to destroy all the pigs by shooting a limited number of birds. Birds are shot via a slingshot and come in different colors and sizes, indicating special bird features. In this testbed, we are limited to *red* birds, which have no special features; *blue* birds, which divide into a set of three birds when the player taps the screen; *yellow* birds, which accelerate upon taps; *white* birds, which drop bombs upon taps; and *black* birds, which explode themselves upon taps or after a set time upon impact. Our experiments involve Levels 1-21, which do not have any TNT nor white or black birds.

Our paper is structured as follows. We first introduce in Section 2 the supervised machine learning algorithms we use: the Weighted Majority Algorithm [Littlestone, 1988; we follow Table 7.1 of Mitchell, 1997] and Naïve Bayesian Networks [Russell and Norvig, 2010]. We choose these two because they provide a rank-ordering of examples, rather than just a binary decision; additionally, they are easy to

implement and can process large numbers of training examples in a short amount of time. We then explain in Section 3 how we collect data to learn from by using an agent provided by *aibirds.org* and several agents we wrote ourselves. Our goal is to learn a general-purpose Angry Birds player, rather than one tailored to the specific levels from which we get training data. To address that goal, we have developed a data structure we call "the grid" with which we represent the local environment surrounding the impact point of a shot. We explain this localized representation and our approach for both training and using models in Section 3. Section 4 presents our experiments where we show that our machine learning approaches can produce statistically significantly better Angry Birds players than the provided agent from *aibirds.org*.

2 Background and Related Work

Before presenting our approach, we provide some background, including a pre-existing AngryBirds player.

2.1 Experimental Control

The agent we use to help collect training examples and also as our experimental control is the *NaiveAgent* provided by the IJCAI 2013 Angry Birds AI competition (aibirds.org/angry-birds-ai-competition.html; for some prior results, see aibirds.org/benchmarks.html). This agent detects the slingshot and other objects, then shoots at randomly chosen pigs using either high or low trajectories. These trajectories are randomly chosen between high-arching shots (release angle ≥ 45°) and direct shots (angle < 45°), with a preference towards direct shots. This agent uses predetermined tap-time intervals for both types of shots. The sample agent provides a baseline for agent performance on the Angry Birds testbed, and our goal is to clear levels faster and with higher scores than this agent. We use the provided trajectory-generating code in all of the players we have implemented.

2.2 Weighted Majority Algorithm

The Weighted Majority Algorithm (WMA) is a machine learning technique that uses a pool of prediction algorithms to build an efficient compound algorithm. The algorithm evaluates examples by taking a weighted vote among the members of this pool. The algorithm learns by altering the hypotheses' associated weights when it makes mistakes.

An advantage of this algorithm is that it operates without any prior knowledge about the quality of the prediction algorithms in the pool. The Weighted Majority Algorithm is able to handle inconsistent training data, is simple to implement, and runs very quickly.

Below is the pseudocode for WMA. Our implementation uses β = 0.95 (we did not yet experiment with any other values). In our usage for Angry Birds, each "prediction algorithm" is simply a Boolean feature or its negation, describing a property of our localized representation of the Angry Birds game state (more details appear later). Basically, each feature has a weighted vote *for* and *against* a shot, and the difference between the sum across all the features of weighted votes *for* a shot and the weighted votes *against* a shot is the net score for that shot. Votes that lead to incorrect decisions have their weights reduced. In our experiments, we process each training example a few times (more precisely, we execute the WMA algorithm's outer loop one million times, which takes about one minute).

```
Given a pool A of algorithms, where a_i is the i^th prediction algorithm; w_i, where w_i ≥ 0, is the associated weight for a_i; and β is a scalar < 1:
    Initialize all weights to 1
    For each example in the training set {x, f(x)}
        Initialize y_1 and y_2 to 0
        For each prediction algorithm a_i,
            If a_i(x) = 0 then y_1 = y_1 + w_i
            Else if a_i(x) = 1 then y_2 = y_2 + w_i
        If  y1 > y2  then g(x) = 1
        Else if y1 < y then g(x) = 0
        Else g(x) is assigned to 0 or 1 randomly.
        If g(x) ≠ f(x) then for each prediction algorithm a_i
            If a_i(x) ≠ f(x) then update w_i with βw_i.
```

2.3 Naïve Bayesian Networks

The Naïve Bayes (NB) classifier encodes a directed, acyclic graph with conditional independence relations among variables. In Naïve Bayes, a dependent class variable Y is the root and feature variables X_1 through X_n are conditioned by Y's value. The naïve (yet effective) assumption is that each feature X_i is conditionally independent of every other feature X_j for $j \neq i$ given Y.

We wish to estimate the probability $p(Y \mid X_1, \dots, X_n)$. For Angry Birds, the Y is *goodShot* and the X's are the features used to describe the game's state and the shot angle. We use the same features for NB as we used for WMA. Using Bayes' Theorem, we can rewrite this probability as

$$p(Y \mid X_1, \dots, X_n) = \frac{p(Y)\, p(X_1, \dots X_n \mid Y)}{p(X_1, \dots, X_n)}$$

Because the denominator of the above equation does not depend on the class variable Y and the values of features X_1 through X_n are given, we can treat it as a constant and only need estimate the numerator.

Using the conditional independence assumptions utilized by NB, we can simplify the above expression:

$$p(Y \mid X_1, \dots, X_n) = \frac{1}{Z}\; p(Y) \prod_{i=1}^{n} p(X_i \mid Y)$$

where Z represents the constant term of the denominator. Learning in NB simply involves counting the examples' features to estimate the simple probabilities in the above expression's right-hand side.

Finally, to eliminate the term Z, we take the ratio

$$\frac{p(Y \mid X_1, \dots, X_n)}{p(\neg Y \mid X_1, \dots, X_n)}$$

which represents the odds of a favorable outcome given the features of the current state.

Our implementation of Naïve Bayes estimates the odds of the class variable *goodShot* given the same Boolean features used in WMA. For each feature, we compute the probability that it is present for both *goodShot* and *badShot*; the ratio of these probabilities is the score of a candidate shot. NB processes each training example only once (for 400,000 training examples, training takes about 3 seconds).

3 Learning to Play Angry Birds

We train our learning algorithms on a set of training examples generated by non-learning agents with varying degrees of intelligence. This data is then categorized into positively and negatively labeled datasets. To gather enough positively categorized data (from winning games), we engineer agents that learn how to play each specific level. To help reduce noise within our datasets, we employ a number of filtering criteria, weeding out ambiguously categorized shots, shots with bad tap times, and duplicate examples. Finally, we discard excess data in order to achieve approximately equal numbers of positive and negative examples per level.

We define a standard example representation, the *AngryBirdsGridExample*, and separately determine tap time using a simple estimator explained below. After training and tuning on the examples, we use our algorithms to play Angry Birds by scoring a set of a few dozen candidate shot angles, choosing one of the highest-scoring candidates, where the probability of selecting a shot is proportional to its score. Coupled with the tap time recommended by our estimator, we then execute these shots in games.

3.1 Data Collection

We use a number of different game-playing agents to facilitate gathering enough data from which our learning algorithms can learn to play Angry Birds. Such agents include the provided NaiveAgent and our "RandomAngleAgent" that randomly chooses shot angles, preferring less extreme angles. Each agent runs on a number of machines, either directly or through remote connection (via Putty) and we collect all the data in a central location.

Due to the inability of the NaiveAgent and the RandomAngleAgent to win games consistently, however, we engineered another agent, called the "TweakMacro" agent, to help gather more winning games. This agent takes a list of saved shot sequences that resulted in one of the three highest scores for each level and replays the exact shots with some small random variation in the shot angles. In doing so, we explore the neighborhood of successful games. However, since our goal is to build agents that can solve *unknown* levels, we use the TweakMacro agent strictly for data collection, as it depends on knowing the specific level being played to re-execute successful shots.

3.2 Data Categorization and Filtering

Fig. 1 illustrates how we convert data saved from games into training examples. We first label examples as either *positive* or *negative*. Positive examples include shots in winning games, while negative examples include shots in losing games, with one exception.

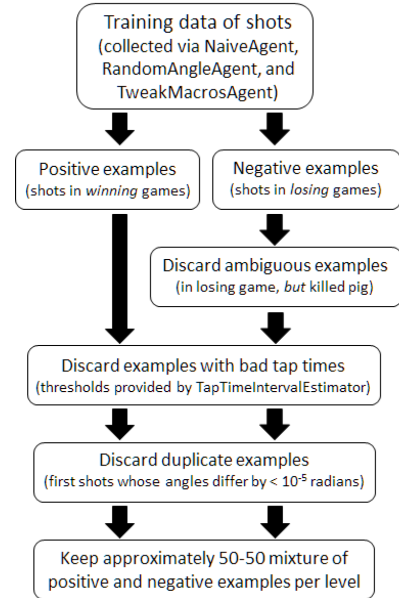

Figure 1: Filtering examples.

In order to clarify our definition of negative examples, we filter out ambiguous negative examples, which we define as those shots in losing games that killed pigs. Since killing a pig is a desired secondary outcome, shots that achieved this objective despite being in a losing game are not categorized as negative examples and therefore discarded.

Furthermore, our RandomAngleAgent attempts a large number of shots whose angles are valid but had poor tap times, either tapping too early (causing birds to fly offscreen) or too late (where taps would not register because the bird had already reached its impact point). Since our models for simplicity learn about shot angles and ignore tap-time information, we filter examples featuring bad tap times in order to discard noisy data. Data is thus filtered using the thresholds of tap time generated by our tap-time estimator, which we explain in Section 3.4.

Because each level's game state before any shots are taken is (nearly) constant, we also filter duplicate shots from our datasets. Every first shot taken at each level occurs in a game state essentially identical to all other first shots in that level. We thus identify *duplicate* shots as first shots of a given level whose shot angles differ by no more than 10^{-5} radians. We filter duplicate shots from our positive and negative datasets separately. We do not attempt to detect duplications on shots other than the first shot in a level.

Finally, because our experiments are considered active tasks, there is no natural positive-to-negative ratio of examples. For this reason, we provide to our learning algorithms an approximately even mix of positive and negative examples per level. This is achieved by randomly discarding excess examples from some levels. From 724,993 games involving 3,986,260 shots, we end up with 224,916 positive and 168,549 negative examples.

3.3 Representing Features about a Game State

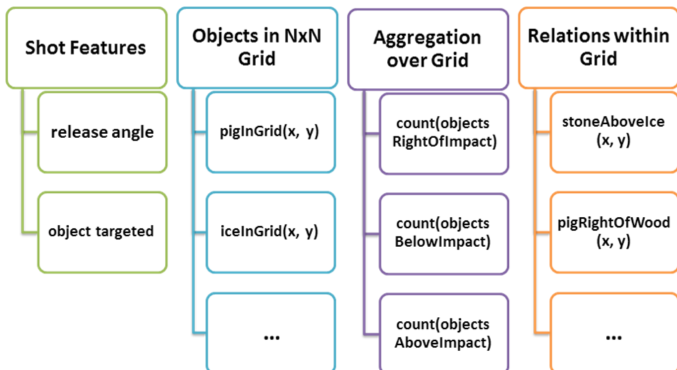

Figure 2: Features represented in AngryBirdsGridExamples. “Counting an object” counts the number of cells containing that object. Relations like *stoneAboveIce(x,y)* are true if cell *x-y* contains ice and some cell at or above *y* in column *x* contains a stone.

We construct an example by collecting values for various features (see Fig. 2) within the game state before a shot is made. An AngryBirdsGridExample contains the shot release angle and object targeted (pig, ice, etc.). It is important to note that these two features are considered as input features in our example model and not as outputs. Furthermore, these examples represent a localized portion of the game state as a 7x7 grid of cells located around the shot impact point (see Fig. 3). We encode the contents of each cell in a series of 2D arrays of Boolean values, based on whether or not they contain a game object: pig, ice, wood, stone, or TNT. To further enhance our feature representation, we created a terrain detector to identify immovable wall areas in game states and include them as features in our grid examples. Note that one cell in the grid can have multiple features being true for that that cell.

Our grid examples also compute features that encode relative positions and counts of objects within the grid. As such, our grid examples calculate whether a given cell contains pigs above, below, and to the right of structures, as well as the counts of objects above, below, and to the right of the shot’s predicted impact point. We do so to learn whether the relative positions of structures and pigs within a game state affect the outcome of a given shot.

Additionally, we do not consider features that are the same across all candidates of a given shot, such as the total number of pigs in the current game state. Because our chosen models are linear and weight each feature separately, features constant across all candidate shots do not result in varying scores across these shots.

3.4 Choosing the Tap Time

For our experiments, we consider tap time as a feature separate from our grid-example representation. Tap time is thus not learned by our standard algorithms. We instead built a separate *TapTimeIntervalEstimator*, a simple estimator which keeps track of tap times in training data to determine the best tap intervals for each bird type (other than red).

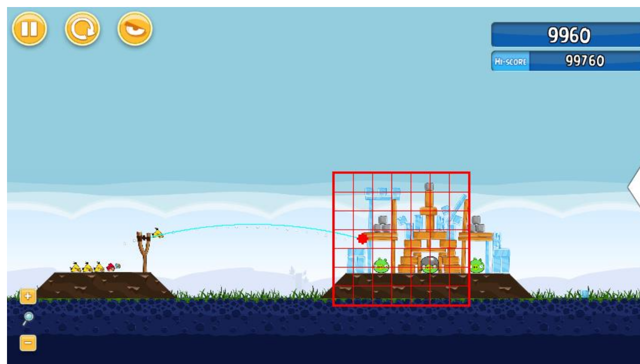

Figure 3: Visualization of the grid we use to construct our features for an example. The grid is placed at the estimated impact point (marked with a red asterisk) for that shot. We do not center the grid around the impact point because, due to physics, the consequences of a shot are largely to the right of the impact.

Since tap times vary across levels due to distance between slingshot and impact, we normalize tap times instead as a *fraction of the impact time*. Impact time is calculated as the estimated time it takes for a bird to reach its impact point without utilizing its tap. The provided code performs this calculation for us.

For each bird requiring a tap, the TapTimeIntervalEstimator counts tap-time fractions (separately for high and straight shots) that resulted in either a win or a loss. Taking the ratio of wins to losses results in visible peak intervals in which best taps were made for certain bird and angle combinations. Finally, during experiments, we select from these intervals using triangle distributions to compute desired tap times for a given shot whose angle has been selected by our previous described algorithms. (We performed some informal experiments and our tap-time method leads to slightly better performance than the method in the provided NaiveAgent, but we do not report on those experiments here.)

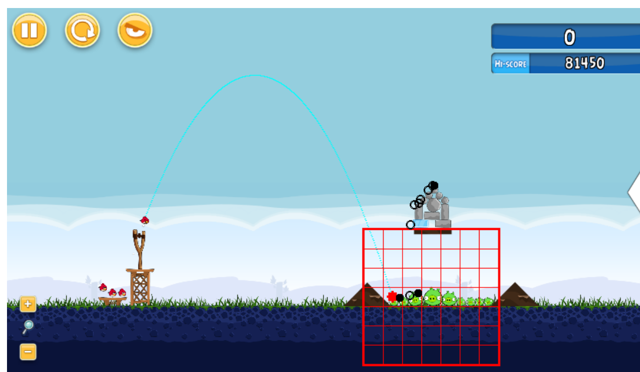

Figure 4: Consideration of candidate shots during Angry Birds gameplay. Each black point indicates the impact location of a candidate shot. The scores of the candidate shots are normalized between [0, 1]; solid black circles indicate candidates with normalized score ≥ 0.5, while open circles indicate candidates with scores < 0.5.

3.5 Learning Models and Playing Angry Birds

After categorizing and filtering our datasets, we divide them into training and tuning sets used by our learning algorithms to learn models of good shots. To address overfitting, our implementation of the Weighted Majority Algorithm uses the tuning data to determine and save the iteration of trained weights that performs the best on the tuning dataset (more precisely, we choose the iteration with the lowest average rank of the top 25% of the positive examples in the sorted list of predicted scores on the tune set). Our implementation of the Naïve Bayesian Network does not incorporate any correction factors, and instead combines the training and tuning datasets.

After completing training, we use our learned models to play Angry Birds. For every game state, we consider several dozen candidate shots (depending on how many pigs and other objects are in the current scene), considering both high- and straight-angled shots for each potential target. Each candidate shot is presented as a grid example to be evaluated by the learning algorithm being tested: the Weighted Majority Algorithm returns a net weighted sum score, while the Naïve Bayesian Network returns a calculation of the odds of winning given the features present in the grid example. We then consider the five highest-scoring candidate shots, choosing among them proportionally to each of their scores, and the selected candidate shot is executed in the current real-time Angry Birds game (see Fig. 4).

4 Experimental Methodology

For our experiments, we train our two algorithms 21 times each on data from Levels 1 to 21, creating separate models for use on each level. Each model was trained on training examples from all levels *except* the level it would be used to play (e.g., a model trained on all levels except the first level is only used to play the first level). We then test our sets of learned models on Levels 1 to 21.

In running the experiments, we start Chrome instances with all Angry Birds levels (1-21) unlocked. Our process for deciding which level to play is as follows.

- First, from 1 to 21, play each level *once.*
- Next, again from 1 to 21, play once those levels *not yet solved*. Repeat this until all levels solved.
- When all levels solved, play the level with the best ratio of *number of times a new high score was set* over *number of times level was played*.

5 Results and Discussion

Fig. 5 shows the performance of our learning algorithms and of our experimental control as a function of the *number of shots taken* versus the *sum of the highest scores* for each of the 21 levels (only scores in winning games are counted). We play 300 shots, which takes about 75 minutes. The results are averaged over ten repeated runs. Both our learning methods perform better than our experimental control, the provided NaiveAgent.

We also show in Fig. 6 the performance of our algorithms when, instead of using 21 differently trained models, we use

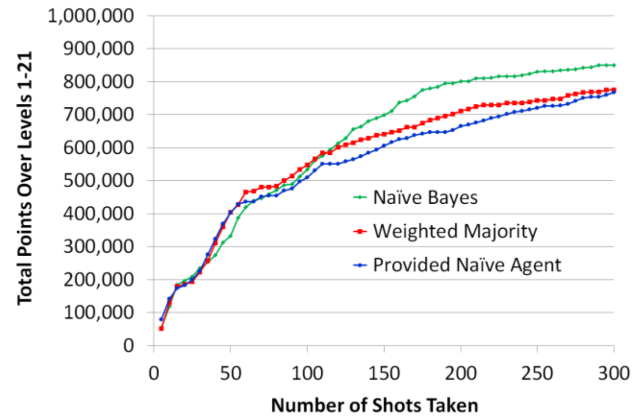

Figure 5: Performance of our learning algorithms trained on all levels except the current level.

a *single* model trained on all 21 levels to play all the levels. This figure shows the difference in performance when the learners have the advantage of playing each level many times before being evaluated. When comparing results reported in various papers it is important to note whether or not learners have training examples from the levels being played (Fig. 6) or the levels are novel (Fig. 5). It is also interesting to see that, in our experiments, our Naïve Bayes model trained on all but test levels eventually scores similarly to our NB model trained on all levels. Apparently Naïve Bayes is able to learn the general skill sufficiently well to largely compensate for not having any training examples for the level currently being played.

The mean areas under the five curves in Fig. 6 are as follows: 167 Million (Naïve Agent), 175M (WMA trained on all but current level), 187M (NB trained on all but current level), 205M (WMA trained on all levels), and 205M (NB trained on all levels). Using an unpaired two-tailed t-test, the differences between the NaiveAgent are statistically significant for all but WMA (p-values of less than 0.0001 for the two versions trained on all levels and p= 0.01 for NB).

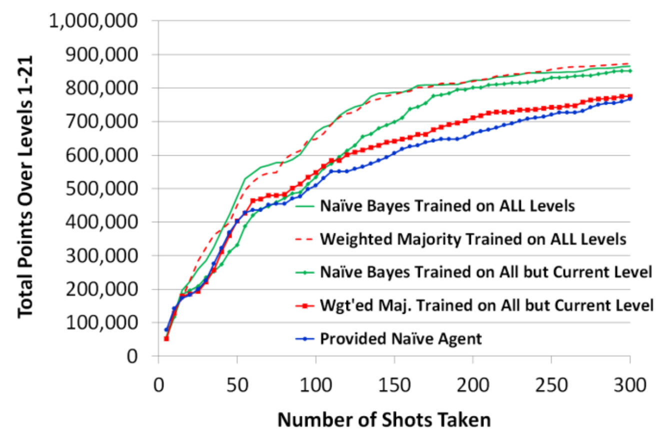

Figure 6: Performance of our learning algorithms trained on both all levels and all levels except current level.

Our experiments reported here only involved Levels 1-21, for both training and testing, but we also let our agents run on Levels 22-42 a small number of times. Tables 1 and 2 show the highest scores we ever found in playing about four million games (note that these games were all played after we fixed the bug in the score-reading code). We present them as a reference for use by other researchers. The total of the best scores in Levels 1-21 is 1,267,530 and for Levels 22-42 is 1,532,710. Comparing these totals to the results in Fig. 5 and 6, combined with the fact that even in many-hour runs, our algorithms would reach only slightly more than one million points, one can see there is sizable room for improvement. It is interesting to see how many levels can be solved with a single shot! Unfortunately, if the best sequences are replayed verbatim they do not come close to replicating the scores in Tables 1 and 2 without performing many replications, showing that results in Angry Birds are quite sensitive to the initial conditions (a given level's initial states are nearly but not totally identical across runs) as well as any non-determinacy in the game's physics engine.

Table 1: Highest scores found for Levels 1-21, formatted as: level (shots taken) score.

1	(1)	35,900	8	(1)	59,830	15	(1)	57,310
2	(1)	62,890	9	(1)	52,600	16	(2)	71,850
3	(1)	43,990	10	(1)	76,280	17	(1)	57,630
4	(1)	38,970	11	(1)	63,330	18	(2)	66,260
5	(1)	71,680	12	(1)	63,310	19	(2)	42,870
6	(1)	44,730	13	(1)	56,290	20	(2)	65,760
7	(1)	50,760	14	(1)	85,500	21	(3)	99,790

Table 2: Highest scores found for Levels 22-42.

22	(2)	69,340	29	(2)	60,750	36	(2)	84,480
23	(2)	67,070	30	(1)	51,130	37	(2)	76,350
24	(2)	116,630	31	(1)	54,070	38	(2)	39,860
25	(2)	60,360	32	(3)	108,860	39	(1)	76,490
26	(2)	102,880	33	(4)	64,340	40	(2)	63,030
27	(2)	72,220	34	(2)	91,630	41	(1)	64,370
28	(1)	64,750	35	(2)	56,110	42	(5)	87,990

6 Future Work

Our experiments open up many diverse possibilities for future work. Our definitions of good and bad shots, which play key factors in supervised learning, are probably insufficient. Another interesting possibility is to learn separate models for each type of bird, as well as for high shots, low shots, first shots, last shots, and shots when only one pig remains. Currently we only consider the two trajectories the provided code calculates and it might be worthwhile to consider more candidate angles for each object targeted.

We only considered two learning methods in this initial work and more should be considered, including reinforcement learning approaches which have worked well for tasks such as RoboCup [Stone & Sutton, 2001], plus additional supervised learning methods such as support vector machines and neural networks.

We would also like to go beyond Naïve Bayesian Networks by designing Bayesian networks with dependencies among features. Hand-coded engineered dependencies could serve to improve Bayesian Network performance on the Angry Birds testbed. In addition, we can use algorithmic search to find good Bayesian Network structures.

Another interesting task to consider is to use teacher demonstrations or teacher-provided solutions in supervised learning algorithms. In Angry Birds, human intuition proves helpful in solving complex levels that require rolling birds along angled terrain and knocking over secondary objects. Given a graphical interface that shows potential trajectories and calculates impact points, human teachers could suggest good shots and even provide complete solutions that score well. We could then use these solutions to train machine learning algorithms to play more intelligently and attempt to emulate human intuition. Potential challenges include gathering enough data for training (as teacher demonstrations are time consuming) and generalizing the learned playing techniques across all levels, as some solutions may only be valid for a particular complex level.

7 Conclusion

The Angry Birds testbed serves as a challenging problem for machine learning. We present and empirically evaluate a design for creating a computerized player based on machine learning. While our learning algorithms are able to statistically significantly outperform the provided NaiveAgent, there is much room for improvement when dealing with the Angry Birds task and many opportunities for future work. Our empirical results provide a baseline for the performance of future machine learning (and other AI) methods.

A key aspect of our approach is that we represent training examples in a manner that we strove to make independent of any specific game level, which we did by creating features whose semantics were relative to a candidate shot's impact point (*shotAngle* and *objectTargeted* were the only exceptions). We argue that the primary goal should be to learn the general task of playing Angry Birds, rather than aiming to learn how to play specific levels. This can be addressed by making sure models for choosing shots for Level i are *not* trained with any examples from Level i.

Acknowledgments

This work was supported by the University of Wisconsin-Madison, USA. Authors ANC and LX are undergraduate students doing independent study under the direction of author JS. Jochen Renz provided valuable assistance that we greatly appreciate.

References

[Mitchell, 1997] T. Mitchell. *Machine Learning.* The McGraw Hill Companies, Inc., Boston, MA, 1997.

[Russell and Norvig, 2010] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* (3rd ed). Pearson Education, Inc., Upper Saddle River, NJ, 2010.

[Littlestone, 1988] N. Littlestone. Learning Quickly When Irrelevant Attributes Abound: A New Linear-threshold Algorithm. In *Machine Learning*, pages 285-318, 1988.

[Stone and Sutton, 2001] P. Stone and R.S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the* 18th *International Conference on Machine Learning,* pages 537-544, Williams, MA, 2001.