[[1]](#footnote-1)

Assignment 2 - Report

Jose Zavala, *Student, University of Essex*

*Abstract*— This report defines the initial proposal of the creation of a Decision Tree Classifier model that can learn to *play* Tic-Tac-Toe by initially using Supervised Learning with a dataset generated by a Monte Carlo Tree Search with UCT that can successfully play the game, to the use a type of Reinforcement Learning to further tune the model to become a strong adversary in the game. The aim is to create a Decision Tree Classifier capable of recognizing the best possible move for a given game-state, without considering any other factors and without assuming any action on his adversary. This greedy approach has been successful in previous Tic-Tac-Toe playing agents, as well as the reinforcement learning techniques used have been used in the past with more complex models to successfully play more difficult games such as Go. It is expected that the proposed model will be able to recognize the strongest moves in the initial dataset since they should statistically be played more, as well as the reinforcement learning approach taken will accentuate the successful moves over the not-so-successful ones. A timetable of how the project will be taken into action is presented.

# INTRODUCTION

T

HIS project aims to implement a small-scale version of the Expert Iteration (ExIt) algorithm originally by [1] to create an agent that can successfully play a Full Information game such as Tic-Tac-Toe. We explore the definition and implementation of the Expert Iteration algorithm in further sections.

ExIt is based on Reinforcement Learning algorithm methodology [2] [1] [3].

The project will explore the use of a Decision Tree Classifier to imitate the behavior of an already existing Monte Carlo Tree Search solution, to then use simple reinforcement learning techniques to strengthen the model’s performance [2].

DTs are to be used since they’ve been demonstrated to imitate MCTSs in the past, as well as they offer a more memory and time efficient method to MCTS in practice [4].

If performance results are satisfactory after experiments are conducted, this will show that, at least for Full-Information games with small state, simpler models can learn to predict moves given enough data about the current state of the game.

The report first shows previous research on MCTS and Decision Trees on the topic. Then, it further explains the model that we are used for training, as well as the MCTS that we will use to generate the initial training data. After this, the dataset recollection methodology is explained. Ultimately, the experiments to evaluate and further improve the performance of the DT are described.

# Background

The famous Tic-Tac-Toe, or OXO, is a “Full Information” game, which refers to the fact that all players have complete knowledge of the state of the match at any given point in time, unlike games like “Poker”, in which certain information (i.e. the hand of a given player) is unavailable [5] [6]. Other examples of this type of games are Chess, Go, and Checkers.

## Tic-Tac-Toe Knowledge

The size of the board for a game of Tic-Tac-Toe is 3x3 and each slot can have any of three symbols (‘X’, ‘O’, *empty*) at any given time, so intuition indicates that there are or 19,683 possible game states [6]. However, these are limited by both conditions that restrict the feasibility of a game state of occurring (Figure 1) and the fact that by rotating the board some game states are equivalent (Figure 2) [6].

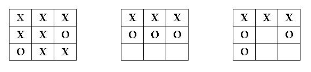


Figure 1 - Infeasible game-states [6]

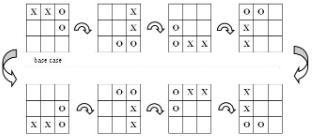


Figure 2 - Equivalent game-states [6]

These conditions reduce the intuitive number to 765 unique game-states [6].

## Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a commonly used technique for the creation of agents capable of playing these types of games [2] [7].

MCTS uses a variation of the Upper Confidence Bounds (UBC) algorithm, called UCB applied to trees (UTC) [8]. This multi-armed bandit approach takes the action that takes to a node which maximizes Equation 1 for the current state [8]. Where is the number of wins after visiting the ith node, is the number of simulations after the ith node and is the total number of simulations ever considered [8].

Equation 1 - UCT Node Selection [8]

This allows the algorithm to explore the possible scenarios of different games just by correctly encoding its mechanics into the algorithm, which demonstrates the effectiveness of MCTS for tracking future game states and predicting the optimal move based on that information [2] [7]. However, its implementation is inherently time and resource expensive since it needs to recreate a full tree in each turn since the game state changes with every move. Also, the random nature of the algorithm allows to overlook specific game paths that may result on a loss, since these may be the consequence of “weird” plays.

MCTS has been used for agents capable of playing turn-based board games, such as Hex [9] and Go [2] [1], due to its capability of lookout for multiple future scenarios using the default random policy or enhanced policies based on successful strategies [9]. MCTS is also frequently used since it is capable of playing its target game successfully immediately after deployment without the need of any training based on expert play given that it was implemented with knowledge of the game’s mechanics [2] [1] [7] [9].

The success of MCTSs in more complex games such as Go can be observed in a recent model, referred to as *Alpha Go Lee*, that uses two networks: A Policy Network that assigns probabilities to moves trained first by supervised expert play and subsequently refined by policy-gradient reinforcement learning, and a Value Network that outputs a position evaluation trained by reinforcement learning to predict the winner of games of the Policy Network against itself [2]. These two networks were combined with a MCTS to provide a lookahead search, using the Policy Network to narrow down the search of high-probability moves and the Value Network to evaluate positions on the tree [2]. This model went as far as defeating the world champion of Go, Lee Sedol, back in 2016.

## Expert Iteration

Expert Iteration (ExIt) is an algorithm first introduced in [1]. It is a Reinforcement Learning algorithm based on the dual-process theory that explains that humans have two systems of thought: System 1 focus on automatic and unconscious thought, also known as intuition; while System 2 is a slow, conscious, explicit and rule-based mode of *reasoning* [1]. ExIt makes use of MCTS as an analogue of System 2, assisting the training of a neural network that works more similarly to System 1 [1].

To understand ExIt, we first need to introduce Markov Decision Processes and Imitation Learning:

**Markov Decision Processes** (MDPs) aim to generate *policies* for optimal play in a two-player, Full Information, zero-sum games that can maximize the rewards for the respective player [1]. A *policy*  is formed by taking an action in a given state . It uses a Value Function , that denotes the reward obtained after following policy in a given state . In a Reinforcement Learning problem, in which we do not know all rewards, the function is introduced, that defines the value of taking action in a state and then following the optimal policy afterwards [1].

**Imitation Learning** attempts to solve MDP by mimicking an existing *expert* policy, which is provided by an expert [1]. The policy learnt through imitation is called the *apprentice* policy, and it needs a dataset of states of expert play with the targets provided by the expert in order to attempt to predict it [1]. The two solutions are: Direct action prediction by imitating optimal move provided by the expert , imitating the expert; Estimate the action-value function and act in a greedy selection algorithm for the optimal move by a trade-off of prediction errors and their cost [1].

ExIt is built upon Imitation Learning by implementing an expert improvement step. The following is a description of the algorithm.

Figure - Expert Iteration Algorithm [1]

At each iteration , the algorithm proceeds as follows: we create a set of game states by self-play of the apprentice . In each of these states, we use our expert to calculate an Imitation Learning target at (e.g. the expert’s action ); the state-target pairs (e.g. ) form our dataset . We train a new apprentice on (Imitation Learning). Then, we use our new apprentice to update our expert (expert improvement). [1]

This algorithm was implemented by [1] using a Neural Network as the apprentice to play the game of Hex with successful results. A similar implementation of this algorithm was used by the *AlphaGo Zero* team in [2], which used a Deep Neural Network for the apprentice to learn to play the game of Go. This version of the *AlphaGO* agent surpassed the previously mentioned *AlphaGo Lee*, winning 100-0 matches against it.

## Decision Trees

In Machine Learning, a Decision Tree (DT) is a hierarchical tree model for supervised learning, in which recursive splits determined by the value of a certain *attribute* of the provided example conduct to a leaf node that contains a classification [4].

Different implementations of the Decision Tree algorithm exist in the literature, namely Classification and Regression Tree (CART), Iterative Dichotomiser 3 (ID3), C4.5, and C5.0 [4] [10]. All of these algorithms have the same basis for construction, which is to recursively split a node based on the attributes of a feature and their relation to the classified class in the training examples [4]. When no further splits can be performed, the most represented class in the remaining examples is set as the leaf node [4].

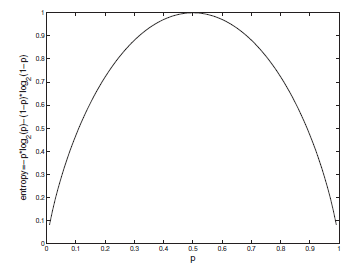


Figure 4 - Entropy Function for a two-class problem [4]

Equation 2 - Entropy Function [4]

Equation 3 - Entropy Function for Feature [4]

Equation 4 - Information Gain [4]

To determine the splits of the tree, the entropy function will be used to determine Information Gain [4]. Other metrics for node splitting exist, such as Gini Index, but Information Gain was selected for its simplicity of use and because it favors unique values in a split [4]. The Entropy Function (Equation 2) allows us to know the homogeneity of the dataset for a given class, where is the probability of classifying for class i. For a binary classification problem, this equation is graphed in Figure 4, where while probability of one given class increases, the probability of the second decreases; this causes a peak of entropy when both classes are equally represented. The more homogeneous it is, the more relevant it becomes to know which action to take [4]. To calculate the entropy of the dataset when a given feature (a cell in this case) is selected, Equation 3 is used, where n represents the number of attributes for a given features and K the number of classes. This equation let us get a weighted mean for the entropy of the dataset if it was discriminated by a given feature. The information gain takes the Entropy of the dataset (Equation 2) and the Entropy of the dataset if segmented by a given feature (i.e. cell in the board) (Equation 3) was to be selected. The feature that gives the most Information Gain is selected to split the node at the given game state.

This algorithm permits DTs to create a set of recursive rules that distinguish which features are more relevant in higher stages of the decision process. Also, they provide an efficient model for predicting decisions based on the whole provided context. The build methodology also synergizes well with the MCTS algorithm, since the leaf nodes predict for the actions that are most popularly represented by the available examples [4].

# Methodology

The goal of this project is to implement the Expert Iteration algorithm using a Decision Tree classifier as the *apprentice* policy and a Monte Carlo Tree Search as the improved *expert*, to then evaluate the performance for learning to play the game of Tic-Tac-Toe.

## Dataset Collection

To gather data to train the *apprentice policy*, the MCTS implementation provided by the Monte Carlo Tree Search Research Hub (<http://mcts.ai/code/python.html>) was used as the target *expert* [1]. The original code provides an implementation of the UCT algorithm for three games: Nim’s Game, OXO (Tic-Tac-Toe), and Othello. The Code was modified to save the game states during a match from the perspective of the player whose current turn is being played.

Previous research mentioned in II.A indicates that there are 765 unique game states in the board at any possible time [6]. However, since neither MCTS nor DT can determine when a state is a rotation of another game state, we undo this knowledge to have a number of 6,120 game states. To be sure to capture all this, an arbitrary number of 10,000 games will be stored as to also account for the validation set. No repeated game states will be removed since they reflect the frequency the MCTS makes those choices.

To allow our *apprentice policy* to play either as “X” or “O”, we encoded the board of the game depending on the current player, and not as “X”’s and “O”’s, , being “X” if it indicates a piece placed by the current player, “Y” if it was placed by the rival, and “\_” represents an empty slot (Figure 7).

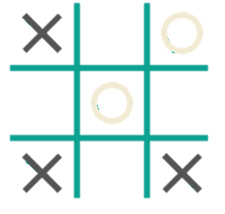


Figure 5 - Game State

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 2 | 0 | 2 | 0 | 1 | 0 | 1 |

Figure 6 - Original Encoding by MCTS Research Hub

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Y | \_ | X | \_ | X | \_ | Y | \_ | Y |

Figure 7 – Our Encoding for Second Player's (O) Perspective

Upon inspection of data, it was decided that every move should be stored, even from the perspective of a losing agent. This was determined since MCTS, our *expert*, will always perform the best possible move available for a given scenario [8]. In a naïve and optimistic approach, even if the agent loses, it means that no better move was found by the *expert* for that scenario [8]. Also, it is assumed that in the long run, the MCTS algorithm will chose the best moves more frequently over the non-optimal ones [2] [8], which the DT implementation should capture. A representation of a possible resultant DT trained on this *expert* is given in Figure 8.

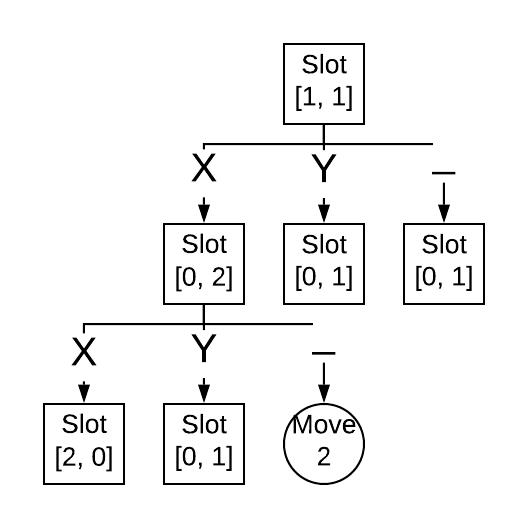


Figure 8 - Decision Tree Representation for State in Figure 5 from Second Player’s (O) Perspective with the new encoding

## Apprentice Policy Model

To achieve the goal, a CART with Information Gain will be used as the *apprentice policy* [1][4] [10]. We will use the implementation provided by Scikit-Learn (<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>) [11]. This implementation was selected due to its ease of use and its integration of fast data-structure libraries in Python such as NumPy.

Originally, it was set that the model would receive the state of the board as an input of 9 features (Figure 7), each one symbolizing a position, and target those results to the cell that was selected by the MCTS algorithm. However, since Scikit-Learn’s DecisionTreeClassifier class does not support categorical (i.e. our initially proposed encoding) and only numerical, we used One-Hot Encoding in runtime to feed the DT an understandable feature vector, transforming it from a 9 feature vector to 27; a significant but necessary feature space dimensionality increase. While the collected dataset will remain the same, the encoding in runtime will look as in Figure 9 for each feature (i.e. slot in the board):

Figure - One-Hot Encoding of the Proposed Feature Vector

This issue also occurs with the target value (i.e. player’s move). Even though it is a numerical value and it’s accepted by Scikit-Learn’s DT implementation, the distribution of values with numerical ranges may cause problems while training as it could assign inexistent priority of some slots over others. To solve this, One-Hot Encoding was also used for the target column in runtime.

The DT was selected for our *apprentice policy* for two major reasons: First, it will be able to identify the most commonly selected plays by the *expert* for a given game state. Since the game has a very limited number of states [6], the DT should be able to determine which choices were most played overall, removing the *random policy* factor of the MCTS algorithm [8]. The second reason is that, since the DT is going to be trained to emulate the *expert* with Imitation Learning [1], and not merely to learn “how to play”, the resulting tree will provide more insight about the relevance of certain cells over others at given game states.

This allows the DT to understand which cells are more important for the MCTS algorithm in a given situation having enough training examples to choose from. To “understand” the importance of a cell is to learn which cell’s states possess less entropy as to correctly discriminate the next action taken. At a leaf node, the action taken is the one with the most representation available, which statistically should be the most effective move performed by the target MCTS [8]. This is useful if we further need to manually analyze the *apprentice policy* to understand its decisions.

As a last note, it is important to note that Decision Tree algorithms **work poorly on unseen examples** [4]. Since our *apprentice* is trained using only the *expert* play, moves generated by the *Expert* Tree algorithm may not always be optimal, which may cause that the model predicts invalid moves. This is solved by observing the prediction during play and selecting a random available move if an invalid move was predicted.

## Apprentice Training

To ensure that the Decision Tree implemented is a successful *apprentice policy* for our model, we need to train and evaluate its performance on imitating the *expert*. To ensure we are training the best model for our *apprentice*, we are going to evaluate for different metrics using 10-Fold Cross Validation, with the GridSearchCV method provided by

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0:0] | [0:1] | [0:2] | [1:0] | [1:1] | [1:2] | [2:0] | [2:1] | [2:2] | Move |
| \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | \_ | 4 |
| \_ | \_ | \_ | \_ | Y | \_ | \_ | \_ | \_ | 0 |
| Y | \_ | \_ | \_ | X | \_ | \_ | \_ | \_ | 2 |
| X | \_ | Y | \_ | Y | \_ | \_ | \_ | \_ | 6 |
| Y | \_ | X | \_ | X | \_ | Y | \_ | \_ | 3 |
| X | \_ | Y | Y | Y | \_ | X | \_ | \_ | 5 |
| Y | \_ | X | X | X | Y | Y | \_ | \_ | 1 |
| X | Y | Y | Y | Y | X | X | \_ | \_ | 7 |
| Y | X | X | X | X | Y | Y | Y | \_ | 8 |

Table 1 - Capture of a single MCTS game using the proposed encoding

Scikit-Learn (<https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html>) [4] [11].

The metrics to be evaluated are Split Criterion and Maximum Tree Depth. Split Criterion will compare between Gini and Information Gain, in order to solidify our selection of Information Gain to split the nodes in the DT; Maximum Tree Depth will ensure that overfitting is not an issue for our classifier to imitate the *expert policy*, this is done by pre-pruning the tree when a maximum depth is reached [4].

To determine the best parameters we will only evaluate our Cross Validation process using the accuracy score, since the only purpose of the *apprentice policy* is to faithfully imitate the original MCTS, a.k.a. our *expert* [1].

## Expert Model

As stated in [1], the role of the *expert* is to perform exploration and accurately determine strong move sequences from a single position. This makes tree search algorithms the canonical choice for the *expert*. We decided to use MCTS as our *expert* for this project based on the work done by [1] and [2], which had successful results in the past.

Since MCTS uses repeated game simulations to estimate the value of a states, expanding the tree further in more promising lines and then selected the **most repeated** play, we are confident that it will synergize well with our selected *apprentice* policy, described in the previous section.

The implementation of the MCTS algorithm is heavily based on the code provided by the Monte Carlo Tree Search Research Hub (<http://mcts.ai/code/python.html>), with minor modifications to add our *apprentice* policy.

Reinforcement Learn will be performed following the original ExIt algorithm, with a small variation. The first iteration of the *apprentice* will be trained using only a small sample of the original dataset generated by *expert* play, rather than the whole as the literature suggests [1] [2]. Further iterations will be trained by creating a move set of *N* number of games between our *Expert* with the selected *apprentice policy* (DT) against the original MCTSwith the default policy (random play). This was decided in order to prevent the new dataset of only focusing on previous optimal moves, opening the possibility of new unseen strategies to emerge.

Literature on the ExIt suggest that an online version of the algorithm aggregates all datasets generated so far at each iteration on the Imitation Learning step [1]. We chose against this since our selected *apprentice policy* will only classify the most prominent move in the dataset. Small aggregations to the *expert* play dataset on each iteration, as suggested by [1], may cause that some optimal plays are ignored since their incidence is small compared to the previous dataset. By creating a new dataset on each iteration , we make sure that each *apprentice policy* is trained with maximum representations of the moves selected by the version.

To ensure that our *apprentice policy* is learning and becoming better than its previous iterations, each 10 versions of the *Expert* we will make the latest version compete with an iteration 10 steps *younger*. We will store this results for evaluation.

All models and move datasets are stored for later evaluation and to further continue training if needed.

# Experiments

All the processes described in Section III explain how the initial training and classification performance of the model will be realized. However, we now need to test the ability of the model as a candidate to compete against other agents in matches of Tic-Tac-Toe.

An approach similar to the one described to train the previous version of *Alpha Go Zero* in [2] will be used, in which we first train the model with “expert play” (i.e. the MCTS agent), to then implement self-play to further refine winning moves.

## Initial Performance Testing

To test the abilities of the DT, we will compare its gameplay against the original MCTS agent used to train it. We will observe its win\_rate score, which is only the percentage of the games that resulted in draw or a win, in order to aim for a No-Loss strategy performance for the DT.

100 games will be played against the original MCTS to calculate an initial win\_rate score. The games will be stored as explained in section III.B, but instead of storing all games, the model will be fine-tuned with the movements of the winning agent. This was not done before since we assumed an optimal strategy from the MCTS in all scenarios, but with enough examples for basic play, we are now focusing on learning the best strategies. This method of further learning was also performed by [2], in which they first explored Go games between expert human players and then let their own models to further play against previous iterations of themselves for faster learning with successful results.

## Further Training

After the initial plays with the MCTS agent, a second version will be developed with the new training examples. 10 games will be played against the first version, and the new win\_rate score will be saved. Again, new training examples will be saved from the winning agent’s moves. These moves will be used to train a new agent each time [2]. To avoid overfitting on the same decisions each time, a random factor will be added, which will consist on selecting a random available move with a probability of 10% each turn.

This process will allow us to understand if the DT is able to learn from successful strategies if enough training examples are provided. The success of the model will be determined with the change of the win\_rate score over time.

# Discussion

As mentioned in Section IV, the performance of the DT as an agent capable of playing Tic-Tac-Toe will be measured with the win\_rate score over time.

Even though we are implementing the basic process used to train *Alpha Go* for the far more complex game of Go [2], the team behind that agent was using a Reinforcement-Learning technique, which allowed the final model to find optimal strategies over time in a more dynamic environment.

There are several limitations to our approach, the first and most notorious one is that the model we are training is a supervised learning classifier and not a reinforcement learning agent [4]. This means that it will only go as far as imitating the data it was trained with, and not to actually learn successful strategies (Figure 8). Therefore, we expect small performance improvement after competing against past versions of itself, since the only change is that it will add more weight to successful movements in a given game-state without knowledge on the opponent’s strategies nor previous time steps.

Another limitation is the way data will be stored in the Experiments section. Since only the actions already recorded by the classifier are going to be played, this may cause the model to overfit on a set of plays, which then may lose against the original MCTS model.

# Conclusion

This project will aim to implement a DT that at first will imitate the behavior of a MTCS playing a game of Tic-Tac-Toe, to then use reinforcement learning techniques [2] to try to obtain more information about game-winning plays.

It is expected that the DT will learn from the MCTS with a high performance, but with a considerable classification error, since MCTS has a random factor which will cause it to perform different moves at the same state for different games, which will impact the classifier performance on initial testing.

For the reinforcement learning part, it is expected that the DT will improve its win\_rate score after some iterations against past versions of itself, but since it will only consist of adding new training examples of more efficient plays this increase will be limited.

# References

|  |  |
| --- | --- |
| [1] | T. Anthony, Z. Tian and D. Barber, "Thinking Fast and Slow with Deep Learning and Tree Search," in *31st Conference on Neural Information Processing Systems*, Long Beach, 2017. |
| [2] | D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature,* vol. 550, pp. 354-359, 2018. |
| [3] | D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv,* p. 19, 2017. |
| [4] | E. Alpaydin, Introduction to Machine Learning, Massachusetts: Massachusetts Institute of Technology, 2014. |
| [5] | S. Sriram, R. Vijayarangan, S. Raghuraman and X. Yuan, "Implementing a no-loss state in the game of Tic-Tac-Toe using a customized Decision Tree Algorithm," in *International Conference on Information and Automation Information and Automation*, Zhuhai/Macau, 2009. |
| [6] | A. Bhatt, P. Varshney and K. Deb, "In Search of No-loss Strategies for the Game of Tic-Tac-Toe using a Customized Genetic Algorithm," Indian Institute of Technology, Kanpur, 2008. |
| [7] | S. S. Sista, *Adversarial Game Playing Using Monte Carlo Tree Search,* Cincinnati: University of Cincinnati, 2016. |
| [8] | L. Kocsis and C. Szepesv ́ari, *Bandit based Monte-Carlo Planning,* Budapest: Computer and Automation Research Institute of theHungarian Academy of Sciences, 1999. |
| [9] | B. Arneson, R. B. Hayward and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 2, no. 4, pp. 251-258, 2010. |
| [10] | L. Breiman, J. Friedman, C. J. Stone and R. A. Olshen, Classification and Regression Trees, Wadsworth : Chapman & Hall, 1984. |
| [11] | F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and É. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011. |

# Appendices

## 10-Fold Cross Validation Grid Search Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Criterion | Max Depth | Mean Test Score | Standard Deviation Test Score | Test Score Rank | Mean Train Score | Standard Deviation Train Score |
| gini | **11** | 0.758204512 | 0.002848817 | 20 | 0.76169701 | 0.000264978 |
| gini | **13** | 0.758845757 | 0.002546861 | 9 | 0.762036718 | 0.000238391 |
| gini | **15** | 0.758822856 | 0.002553542 | 18 | 0.76203799 | 0.000237207 |
| gini | **17** | 0.758845757 | 0.002530212 | 9 | 0.76203799 | 0.000237207 |
| gini | **19** | 0.758834307 | 0.002532222 | 14 | 0.76203799 | 0.000237207 |
| gini | **21** | 0.758845757 | 0.002512764 | 9 | 0.76203799 | 0.000237207 |
| gini | **23** | 0.758845757 | 0.002550188 | 9 | 0.76203799 | 0.000237207 |
| gini | **25** | 0.758834307 | 0.002540227 | 14 | 0.76203799 | 0.000237207 |
| gini | **27** | 0.758868659 | 0.002529318 | 4 | 0.76203799 | 0.000237207 |
| gini | **29** | 0.758834307 | 0.00253756 | 14 | 0.76203799 | 0.000237207 |
| entropy | **11** | 0.758215962 | 0.002828559 | 19 | 0.761761899 | 0.00023678 |
| entropy | **13** | 0.758903012 | 0.002526869 | 1 | 0.762035446 | 0.000232081 |
| entropy | **15** | 0.75888011 | 0.002538356 | 3 | 0.76203799 | 0.000237207 |
| entropy | **17** | 0.758891561 | 0.002576225 | 2 | 0.76203799 | 0.000237207 |
| entropy | **19** | 0.758868659 | 0.002557987 | 4 | 0.76203799 | 0.000237207 |
| entropy | **21** | 0.758868659 | 0.002583134 | 4 | 0.76203799 | 0.000237207 |
| entropy | **23** | 0.758845757 | 0.002568001 | 9 | 0.76203799 | 0.000237207 |
| entropy | **25** | 0.758834307 | 0.002550511 | 14 | 0.76203799 | 0.000237207 |
| entropy | **27** | 0.758857208 | 0.002572332 | 8 | 0.76203799 | 0.000237207 |
| entropy | **29** | 0.758868659 | 0.002555207 | 4 | 0.76203799 | 0.000237207 |

1. [↑](#footnote-ref-1)