Monte Carlo Tree Search for Character Selection in the Game Dota 2

Monte Carlo Tree Search für die Charakterauswahl in Dota 2
Bachelor-Thesis von Leo Valentin Kettner aus Frankfurt am Main
Tag der Einreichung:

1. Gutachten: Prof. Dr.-techn. Johannes Fürnkranz
2. Gutachten: Tobias Joppen

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Monte Carlo Tree Search for Character Selection in the Game Dota 2
Monte Carlo Tree Search für die Charakterauswahl in Dota 2

Vorgelegte Bachelor-Thesis von Leo Valentin Kettner aus Frankfurt am Main

1. Gutachten: Prof. Dr.-techn. Johannes Fürnkranz
2. Gutachten: Tobias Joppen

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den March 21, 2017

(Leo Valentin Kettner)

Abstract

Dota 2 is a complex video game in which players choose from a large pool of different characters before the game starts. The created team composition significantly determines how the game will play out and who has better chances of winning but it is not obvious how to choose well.

In this thesis we explore how the strength of team compositions can be evaluated with machine learning and how this knowledge can be used to build an application using Monte Carlo Tree Search that can play the character selection sub game well in order to provide good character choice recommendations for real players.

Contents

List of Figures

1 Introduction

Dota 2 is a complex multiplayer video game in which the initial selection of characters for the two teams plays a big part in the following play out of the game. Each of the five players on the two teams needs to select one of the 112 available characters in turns. The players need to take into account how well the heroes work together and how well they work against the heroes on other team in order to have the best chances of winning the match.

The goal of this thesis is to analyze how a computer can learn to play this part of Dota 2 and in turn provide good character selection recommendations to real players.

To achieve this goal methods from the field of machine learning were used in order to learn to predict who will likely be the winner of a match given the teams' character selections. This needed large amounts of training data which was gathered over an API provided by the developers of Dota 2.

Machine learning is the science of making computers able to learn about, model and predict different forms of data. It is widely used for example to recognize faces in photos, find content similar to a user's interest or understand voice commands.

We evaluated multiple candidate machine learning methods on how well they perform on this problem.

Since we were interested in suggesting good character choices dynamically during the character selection phase in which the teams are not yet finalized we used the predictions for finalized teams with the Monte Carlo Tree Search method for finding character choices with a higher likelihood of winning.

Monte Carlo Tree Search is a general game playing algorithm which has shown good results in other games like Go which are otherwise hard for a computer to play well. We implemented it for the Dota 2 character selection and analyzed some improvements specific to this domain.

In the *Dota 2* section the foundations of the game are explained to help understand the depth of the game and the choices made throughout this document.

The *Predicting Match Outcome Based on Team Composition with Machine Learning* section describes what kind of data we gathered and how well different algorithms perform when using the data to learn to predict strength of team compositions. A best performing machine learning model was selected to be used with Monte Carlo Tree Search which is explained in its section.

In the *Recommending Hero Picks with MCTS* section we adapt and implement the general version of Monte Carlo Tree Search and evaluate different improvements that can be applied to it. This comes together to create an application which can play against itself, or with inputs provided by the user can recommend hero choices in real games.

The *Conclusion and Future Work* section summarizes our work and gives ideas for future improvements and research

2 Dota 2

2.1 Overview

Dota 2 is a MOBA (multiplayer online battle arena) or ARTS (action real-time strategy) video game developed by *Valve Corporation*. It has a large player base averaging more than 600.000 simultaneous players in 2016 [6] and a strong competitive scene featuring the tournament with the largest prize pool of any eSports title with over 20 million dollars for *The International 2016* [1, 9].

2.2 Introduction to Gameplay

Dota 2 is played by 10 players divided into 2 teams of 5 players each: the *Radiant* and the *Dire*. To win the game a team has to destroy the other team's base while preventing the other team from doing the same. This usually takes between 25 and 60 minutes but this is not restricted by any rule in the game and a match's duration is open ended.

Players control their units like in other real-time strategy games like *StarCraft* but they usually manage only one unit: their hero. A hero is one of 112 playable characters that is chosen by each player before the game starts. This is called the (hero) picking phase. Every hero is unique and their playstyle is defined by their abilities which can either be active spells usable with mana points (or other resources) or passive abilities. Spells range from simply dealing damage to complex effects like teleporting units, disabling enemies from using abilities, summoning creatures and much more.

In addition to their spells every hero also has a variable amount of each of the attributes: agility, intelligence, strength, attack damage, attack speed, attack range, physical resistance, magical resistance, hit points, mana points and movement speed. Some of these attributes influence each other: for example every point of strength adds health points and every point of intelligence adds mana points.

Over the course of the game heroes gain experience and levels increasing their attributes and giving them access to more powerful versions of their abilities and gold which is the currency used to buy items which in turn grant more attributes or abilities.

In order to accomplish their objective of destroying the other team's base, players can directly attack enemy fortifications, gain gold and levels without fighting the other team in order to be stronger at a later point during the game, or engage enemy heroes to kill them which in addition to gold and experience also takes the killed hero out of the game for up to 100 seconds until they are revived.

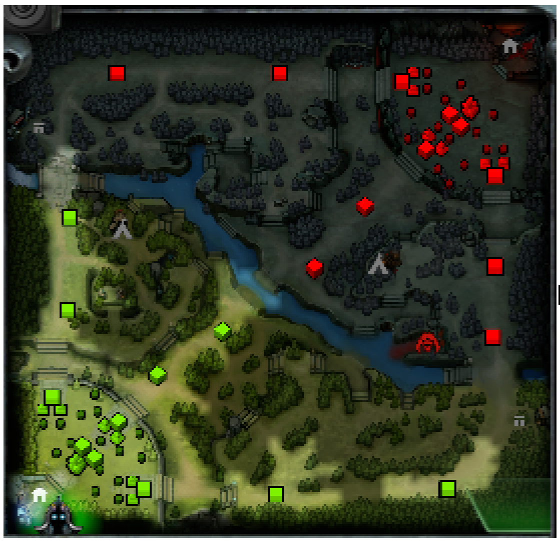

Figure 1: The Dota 2 minimap showing the Radiant base in the bottom left and Dire base in the top right, the three lanes and the towers on them connecting the bases and the river that divides the two sides.

As seen in figure 1 fortifications are placed along the 3 main paths called "lanes" that connect the two teams' bases and need to be destroyed before being able to take on the main base. These fortifications are computer controlled stationary units called towers that deal damage to nearby enemies and allow allied heroes to quickly teleport onto them in order to help their team. Every 30 seconds computer controlled units called "creeps" start walking from their base towards the other base on each lane. Individual creeps are weak but are needed in numbers to overcome the enemy towers. Killing creeps is the main way of earning gold and experience early in the game.

2.3 Hero Differences and Team Composition

The amount and uniqueness of the available heroes plays a big part in the complexity of Dota 2. For a player, it is important to know what every hero is capable of and good at doing and how they relate to the other heroes in a given match. Some heroes might be better in matches with players new to the game, but perform worse or require different strategies when playing against veterans.

Heroes can excel at many different things like:

- dealing damage from far away,
- quickly closing the gap to an enemy,
- dealing damage in an area versus many smaller units,
- creating many small units to destroy towers quickly,
- preventing an enemy from using their spells,
- preventing an enemy from running away,
- preventing an enemy from using their physical attack,
- healing allies,
- amplifying damage dealt from allies,
- providing information about the location of the enemy heroes,
- being mobile in order to evade the enemy heroes
- fighting one enemy hero alone
- ...

Some of these skills are especially useful against other specific skills (they *counter* them): For instance, if the other team has a hero that can turn invisible then it is useful to in turn have a hero that can see invisible units, or if an allied hero is able to deal a lot of damage but has low hit points then it is useful to have abilities that can decrease the damage they take.

Heroes are often coarsely categorized into different roles. Heroes called *carries* are weak in the beginning of the game but excel at gaining gold and experience quickly which makes them stronger as the game goes on. These heroes usually gain more attributes per level, deal their damage with their physical attacks and have spells that become more powerful in some form as the game goes on instead of dealing a fixed amount of damage.

The opposite of a carry is a *support*. A support is strong early in the game and tries to make sure that the game goes well for the carry on their team. Supports usually have healing spells, disabling spells that prevent enemy heroes from killing their carry or prevent them from fleeing when the carry tries to hit them.

A more specialized role is the *split pusher*. Pushing means pushing allied creeps towards the enemy base by destroying enemy creeps and towers on the way. The split pusher is a contrast to the more intuitive form of classical pushing where all heroes of a team decide to try to push on the same lane which usually leads to a fight against the whole enemy team as they try to defend. Instead, they try to push unpredictably and alone in order to force multiple enemies to defend the pushed lane. When the enemies arrive, the split pusher leaves the lane but now knows that their team has an advantage on other parts of the maps as some enemy heroes are now busy defending.

The skills heroes excel at and the roles above are not exhaustive, this section merely tries to relay a feeling for the complexity of the game.

The **team composition** is the selection of heroes for each team viewed as a whole. Depending on which heroes were chosen a team can be good or bad at different things. A good team composition requires heroes that work well together and work well against the other team's heroes.

The team composition determines which strategy of the team as a whole gives the best chances at winning the game. Picking many pushers and heroes who are good early in the game will lead to a team that should try to win the game early as this is where they have the most advantage. Picking too many carries on the other hand might lead to an early loss as this is the phase when the team is the weakest.

Finding good team compositions and strategies becomes more important at higher levels of play where the game becomes sharper and there is less room for error. A team composition is also hard to evaluate in a vacuum by just looking at the heroes. Often more is learned by actually playing out a composition and seeing what works well in the match.

2.4 Metagame and Game Balance Patches

The metagame describes which set of strategies, heroes and team compositions is at a given point in time considered good or bad. Most teams follow the metagame but sometimes a team will invent a new strategy or make an old one popular which if successful evolves the metagame and forces other teams to adopt this strategy or in turn invent an even newer strategy that is good against the former.

The evolution of the metagame is aided by changes or patches to the game's balance. Balance describes the entirety of the game rules and how they work together. Good balance usually means that there are a lot of viable strategies and that no particular hero is seldom chosen because the respective hero is considered too weak. On the other hand a metagame where there is only one viable strategy would be considered to be badly balanced.

To improve balance and shake up the metagame in order to make it less stale for the players, Dota 2 receives balance patches regularly. These patches usually change some heroes or mechanics in order to allow for a new set of strategies. If games tend to end too early the towers might be made stronger, or if a hero is picked in the majority of matches they will be made weaker.

This thesis was written on the version 6.88 of Dota 2.

2.5 Picking Phase

The picking phase takes place before the main game. During it every player chooses a unique hero (no two players can play the same hero) while communicating with the rest of the team about their preferences. There are different *game modes* that decide how the picking phase progresses and influence the resulting team compositions.

Some of the game modes are:

All Pick Players pick simultaneously during 75 seconds. Simultaneously means that there is no temporal ordering in the picks, a player can pick at any time. This is the most commonly played game mode in unranked games (games which do not influence the players visible skill rating).

All Random Players get random heroes assigned. Every player has the option to re-random their hero once which will never give them the same hero again.

Ranked All Pick Instead of picking at the same time teams take turns picking one hero each. Turns lasts 35 seconds and any player of the team can pick, first come first served. Before the picking phase there is also a *banning phase* which limits the available hero pool for the following picking phase. Each player simultaneously has the option to nominate one hero to be banned. Half of the nominations are randomly selected and banned.

Random Draft The hero pool is limited to 50 random heroes. Teams take turns picking like in Ranked All Pick.

Captains Mode Bans and picks are interleaved in a specific order. This mode is meant to be played in tournaments where one player is the team's captain and selects the bans and picks.

Captains Draft A mixture of Random Draft and Captains Mode. There is a specific ban and pick ordering but the hero pool is limited to 27 random heroes.

Some game modes are better suited for competitive matches than others. All Random will often lead to bad team compositions and it is frustrating to play with random heroes at a high level of play.

All Pick is preferred by most players as they often have specific heroes in mind which they already decided to play before the game begins. All Pick evolved to Ranked All Pick so players would not wait until the very last second to pick their heroes so that the other team could not react to or counter their pick.

The mode which places the most emphasis on planning team compositions is Captains Mode. Interleaving bans and picks allows teams to gradually establish a strategy and react to the other team's picks. Both preparation and the metagame influence what the captains have in mind when picking in this mode.

In order to get a break from the current metagame, some players like modes with a limited random hero pool like Random Draft and Captains Draft. These modes can still be played competitively but force players to make new strategic choices for their team compositions by not allowing every hero to be picked.

3 Predicting Match Outcome Based on Team Composition with Machine Learning

3.1 Machine Learning

In order to recommend heroes that will improve our winning chances we needed an algorithmic way of determining how good a team composition is. Instead of trying to come up with such an algorithm by hand and programming our own ideas into it we used techniques of the field machine learning to model how different team compositions lead to different outcomes of the game.

In our case this is done with *supervised learning*: We present the computer a large number of matches in form of their team compositions and outcomes of the matches and the computer learns to predict the outcomes of new matches that it has not seen yet.

Each single match in the training data is called a *sample*. A sample has its *features* which together are some way to represent the team composition, and its output or label which is the winner of the match.

The outcome of a match can either be the Radiant or the Dire winning which is called a *classification problem* as every match is part of one of the two classes.

The class is what we try to predict with machine learning: Given a team composition which side is most likely to win.

The process of learning can be accomplished in different ways with varying success by various machine learning algorithms which are explained in the *The Learning* section.

3.2 The Data

Our data came from an API provided by the developers of the game which allowed us to query recently played matches. Over the course of five days we gathered 180.000 matches of the *very high* skill bracket.

The API divides matches based on the players in them into three skill brackets: normal, high, very high. Very high was chosen to get matches played in the most competitive setting since we wanted to learn something about the game on its highest level.

The prediction is dependent on what data is used as input for the machine learning algorithms. The predictions will be more accurate for matches that fit our data selection. For example matches from the normal skill bracket are not predicted as well as matches from the very high skill bracket because different team compositions are stronger at different levels of play.

The game mode was not filtered as the mode only influences how heroes are picked but not how the game plays out after the heroes have been picked. So any game mode provides usable data.

Matches in which a player left the game before it was over were discarded as the outcome is influenced stronger by the fact that a team has to play with fewer players than by the actual team composition.

Each sample in the data set is part of one of two classes: The winner of the match is the Radiant or the Dire team.

In the current gameplay patch of Dota 2 the win rates of the two teams are not perfectly balanced. The Radiant team wins about 55% of the matches and the Dire only 45%. A naive machine learning algorithm might use this fact and always predict that the Radiant will win which is a bad prediction but would still make it correct 55% of the time. To combat this bias we undersampled our data set which means we intentionally left some data out to have the same number of Radiant wins as Dire wins in the final set.

The resulting data contains 140.000 samples which are split into a training, a validation and a testing set with an 80-10-10 split. The training set is used as the input data to train the selected machine learning algorithms, the validation set is used to evaluate the algorithms and tweak their parameters, and the testing set is used to evaluate the final score of what we believed to be the best algorithm in the previous step.

3.3 The Learning

As the features of each sample we chose a 224 dimensional vector in which each element represents whether a specific hero was chosen in that match. As there were at the time of writing 112 available heroes the first 112 elements represent the heroes of the Radiant team, and the next and last 112 elements the heroes of the dire team. Out of the elements of each team there always have to be exactly 5 heroes selected. A picked hero is represented with a 1 and an unpicked hero with a 0. The specific mapping of which hero corresponds to which vector element is arbitrary and unimportant as long as it is consistent.

Each sample also contains the winner. The Radiant winning is represented as a 1 and the Dire winning as a 0.

The machine learning framework used is *scikit-learn*[1]. It implements many tools and algorithms used for machine learning in Python[2] and is open source and free.

What follows are the results of trying out various classification algorithms in scikit-learn on our data. We explain the concept of each algorithm and then how they compare against each other in the *Evaluation* section.

3.3.1 Logistic Regression

Logistic regression [7, p. 317-320][11] estimates the probability of a sample belonging to a class with the logistic function:

$$\frac{1}{1 + \exp -(c + wX)}$$

where X is a vector representing the features of the samples, w is a vector of weights or coefficients of the same size as the feature vector and c is the intercept.

Each coefficient corresponds to the feature it is multiplied with and together a linear combination of the features is formed to which the intercept is added.

After training we can analyze which coefficients were assigned to which features. For example the 55th feature corresponds to hero number 55 (*Omniknight*) being on the Radiant team. This feature had a coefficient of -0.675 and its counterpart for hero number 55 being on the Dire team (feature number 55 + 112 = 167) a coefficient of 0.612. These values represent how strongly this hero influences the probability of this match being part of the class *team Dire wins* because the algorithm predicts membership of the class with the lower label and the Dire winning has label 0 while the Radiant winning has label 1.

Looking at the formula which computes this probability we can see that a negative coefficient makes the probability estimate for the Dire winning fall and a positive one rise. Thus hero number 55 makes it more likely for whatever team they are on to win since their Radiant coefficient is negative and their Dire coefficient is positive. In fact hero number 55 had the best coefficients for both teams.

On the other hand hero number 59 (*Broodmother*) had a Radiant coefficient of 0.473 and a Dire coefficient of -0.538. This hero is detrimental to whatever team they are on. Or hero number 57 (*Huskar*) with coefficients 0.104 and 0.107 who always makes the Dire win more likely no matter what team they are on.

In logistic regression each feature contributes to the prediction independently of other features. The model can only tell that some hero generally increases the winning chances of a team but it is not able to see that two heroes together might have good synergy which would increase their winning chances when they are picked together.

[1] `http://scikit-learn.org/`
[2] `https://www.python.org/`

Decision trees learn simple true or false decision rules that form a tree. New samples are then classified by following the tree to a leaf at which point the decision is done [3, ch. 2][12].

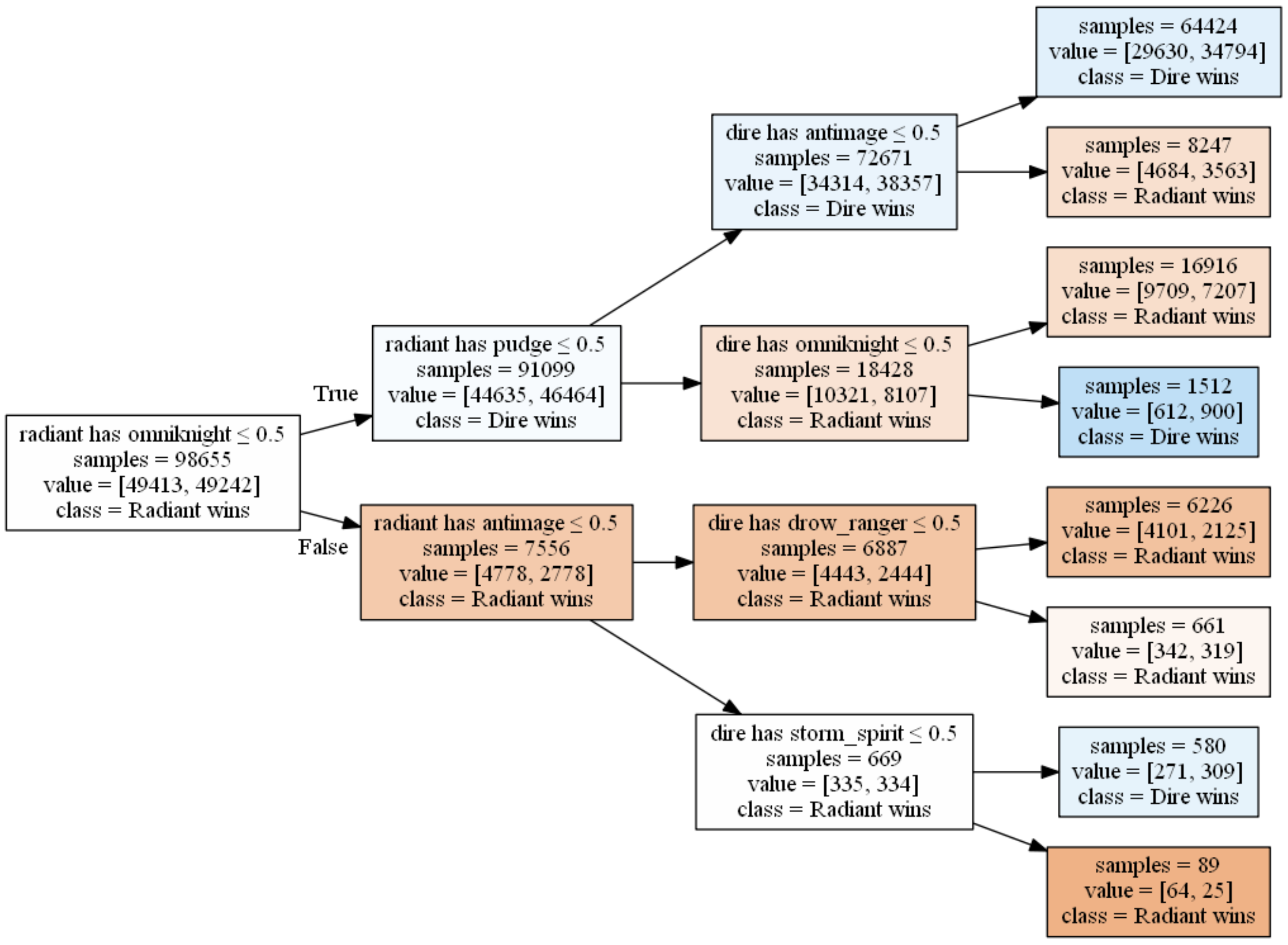

Figure 2: An example decision tree on our data.

Figure 2 shows an intentionally small decision tree generated on our training data. The tree starts on the left side at its root node.

The root node makes a decision based on the feature corresponding to Omniknight being on the Radiant team is smaller than or equal to 0.5. Since our features are all 0 or 1 this simply asks whether Omniknight is not on the Radiant team. If this is true (Omniknight is not on the Radiant team) we follow the upper arrow to the next node. If false we follow the bottom arrow. This continues until we hit a leaf node (the end of the tree) at which point we classify the sample based on what the node in the figure displays as its class.

Each node decides which class belongs to it based on the training data. This is represented in the "value" category in the figure which shows how many samples of the training data are part of the two classes. The more one class dominates the node the more strongly colored is the respective node.

Analyzing the tree with our knowledge of the game, we can see that the first decision considers the strongest individual hero (Omniknight). If Omniknight is indeed on the Radiant team then the Radiant is already much more likely to win than the Dire. Following the lower edges the tree checks if Antimage is on the Radiant team and Storm on Dire. We know that Antimage usually performs well against Storm

Spirit (Storm Spirit is countered by Antimage) so this line of decisions makes sense and the Radiant should win more often if they have Antimage and the Dire have Storm.

Here we also see a limitation of decision trees. Since the tree is organized in a hierarchy, it is hard to represent the concepts of counters: If we have multiple hero pairs like Antimage and Storm Spirit where one hero is very good against the other then there will be no good place to put these counters in the tree as they do not depend on other decisions but would always influence following decisions or be influenced by previous decisions because of the hierarchical nature of the tree. Furthermore, each counter pair would have to be in the tree twice since Antimage still counters Storm Spirit when the teams are reversed but the decision tree cannot ask *Is Antimage on one team and Storm Spirit on the other?* in one step.

Another similar limitation is that we know that it is bad to have too many carry type heroes on one team but there is no good way to count types of heroes in a decision tree.

To get a better model out of the training data we can in comparison to the tree in figure 2 increase the depth of the tree. At the same time we do not want the depth to become too high or else the tree will be overfitted which means that it will resemble the training data too closely which lessens its ability to predict new samples.

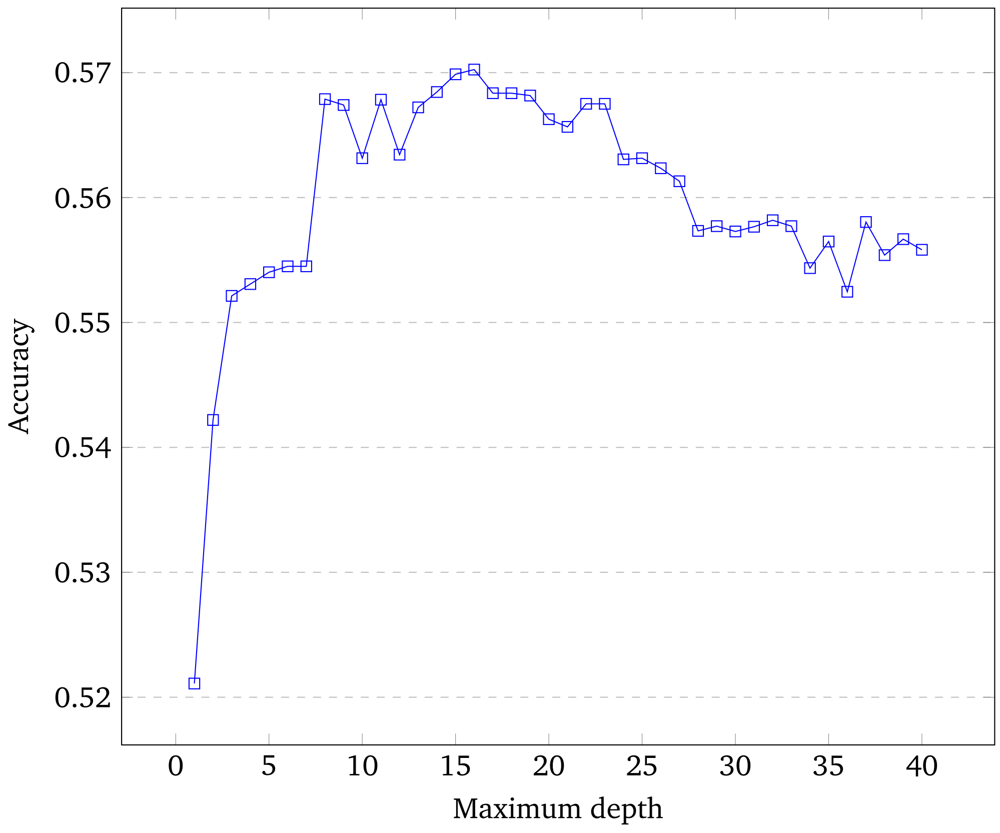

Figure 3: A graph showing maximum depth of the decision tree versus achieved accuracy.

In figure 3 we see a graph showing the maximum depth of a decision tree trained on the training data and accuracy (ratio of correctly classified samples) on the validation data.

At a depth of 16 we achieve the highest accuracy of around 0.57.

3.3.3 Random Forests

Random forests [4, 13] build on decision trees by using a forest of many trees instead of only a single tree.

Each tree is trained on a random subset of the training data, and to predict a new sample the predictions of the individual trees are averaged together. This decreases variance compared to just using a single decision tree which means that the algorithm is less volatile when the training data changes. A single decision tree might make very different predictions after a small change to the training data whereas a random forest is more robust.

Additionally to training on a random subset of the data the trees in the random forest also only consider a random subset of the available features when learning their decision rules as opposed to all features. This yields less accuracy in a specific tree but decreases variance and overfitting in the whole forest.

Having many specialized decision trees alleviates the limitations mentioned earlier. We would not need to fit all the counters in one tree anymore. Instead, they could be spread out over many so that the final forest can use them all. Still, even in random forests such a thing like the counting of carry heroes can not easily be done.

The more trees a random forest contains the better the accuracy, but from some point on returns diminish. This is what we see in figure 4. The graph suggests that a further tree count increase could still provide slightly more accuracy but at this point memory and cpu usage start becoming too high for that to be feasible. We decided on 200 trees as a good value.

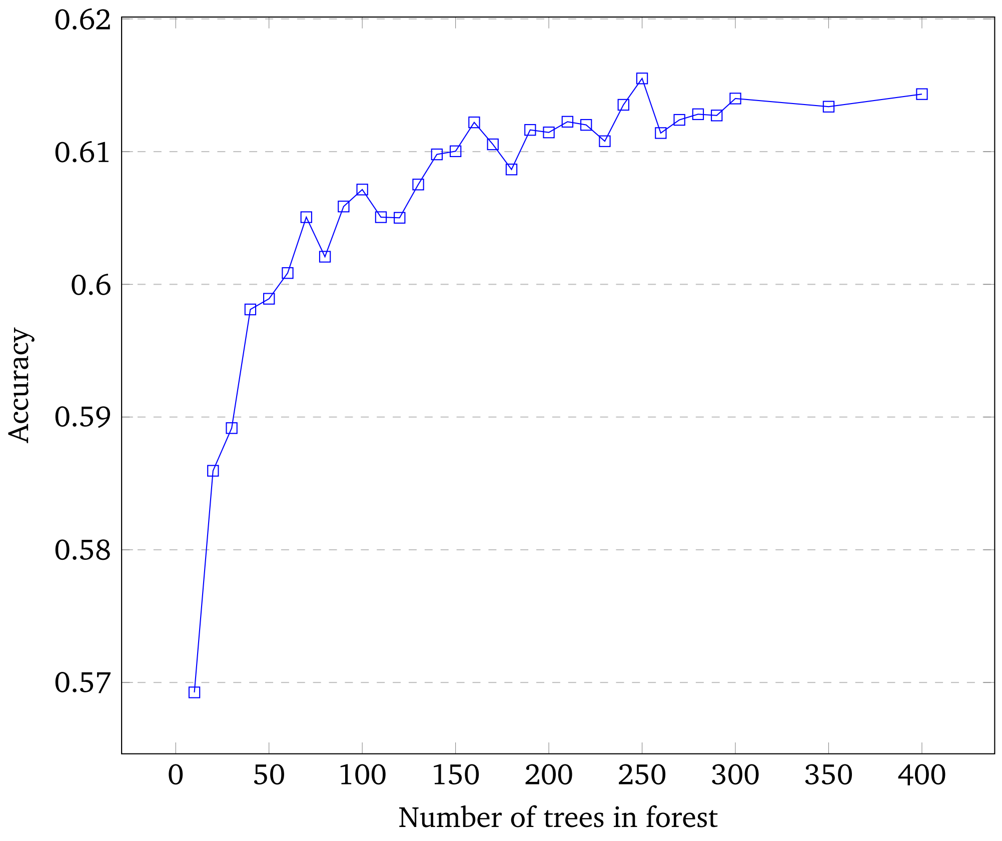

Figure 4: A graph showing number of decision trees in the random forest versus achieved accuracy.

3.3.4 Gradient Tree Boosting

Like in the random forest method, gradient tree boosting [10, 14] also uses many individually weak decision trees to obtain a strong final model. However, where in random forests the trees are trained separately without influencing each other, gradient tree boosting starts with some initial model and then repeatedly adds a single decision tree that would maximally improve the combined model. This new model consisting of the initial model and the first tree is then used as the basis for adding the next tree and so on. In random forests all trees weigh into the final prediction equally whereas in gradient tree boosting each tree has a factor assigned to it which can make it matter more or less in the final result.

Like in random forests having more trees is better but diminishing returns set in as seen in figure 5. The accuracy does not rise after 75 trees.

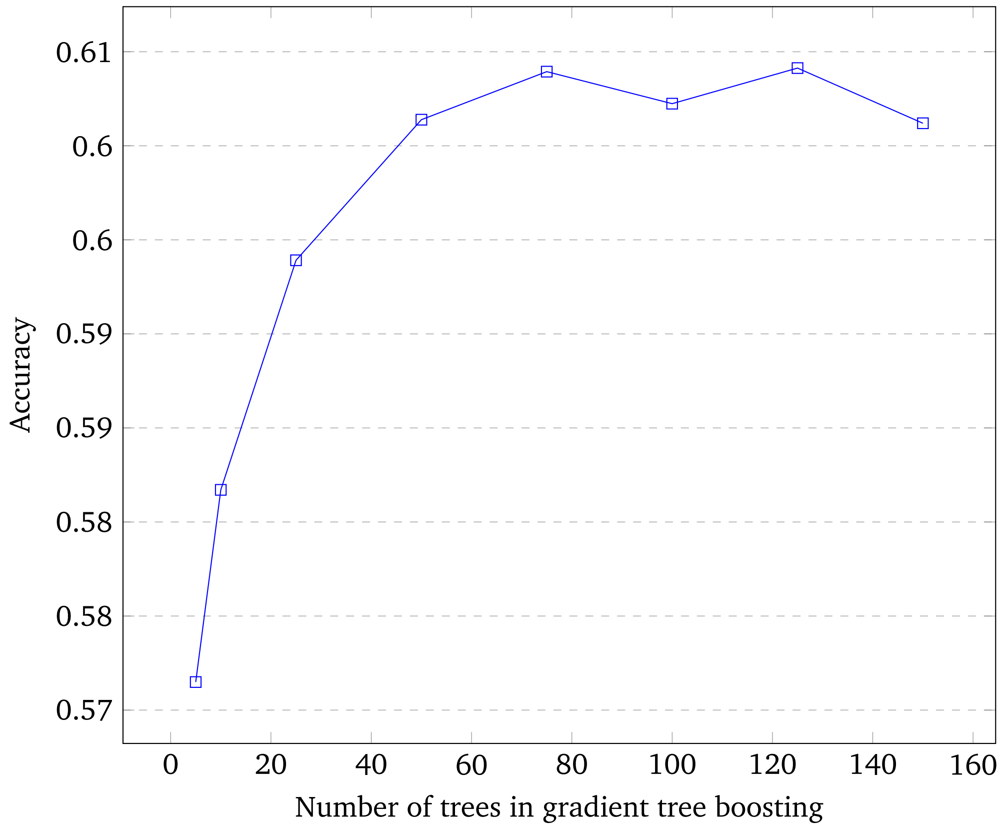

Figure 5: A graph showing number of decision trees used in gradient tree boosting versus achieved accuracy.

3.3.5 Nearest Neighbors

In nearest neighbors [15] new samples are predicted by comparing them to already known samples that are near (resemble closely) to the new sample. This contrasts with how information is extracted out of the training data in methods like logistic regression or decisions trees: In nearest neighbors, all training data is directly remembered and then compared to a new sample.

In order to find data similar to a new sample we need to choose a distance metric which will compare the closeness of two samples. This could be simply the euclidean distance but since our features are all boolean we use the Manhattan distance which in our case is just the number of unequal dimensions. For example, a match of heroes 1,2,3,4,5 versus heroes 11,12,13,14,15 in which we swap hero 1 with hero 6 will have a Manhattan distance of 2 to the original since two dimensions changed in the feature vector. When we compare a match to another in which every hero has changed we get the maximum distance of 20.

Nearest neighbors can use a fixed number k of neighbors (the nearest k samples) to predict a new sample or use all the neighbors within a given distance. The resulting prediction is the majority vote of the neighbors where a neighbor's vote is weighed by the inverse of its distance so that a nearer neighbor influences the result more than a neighbor farther away. We chose the fixed number of neighbors version as it was more than twice as fast to predict new samples without having worse results.

In Figure 5 we see the effect of different numbers of neighbors. The graph falls off after 180 which seems to be the best value.

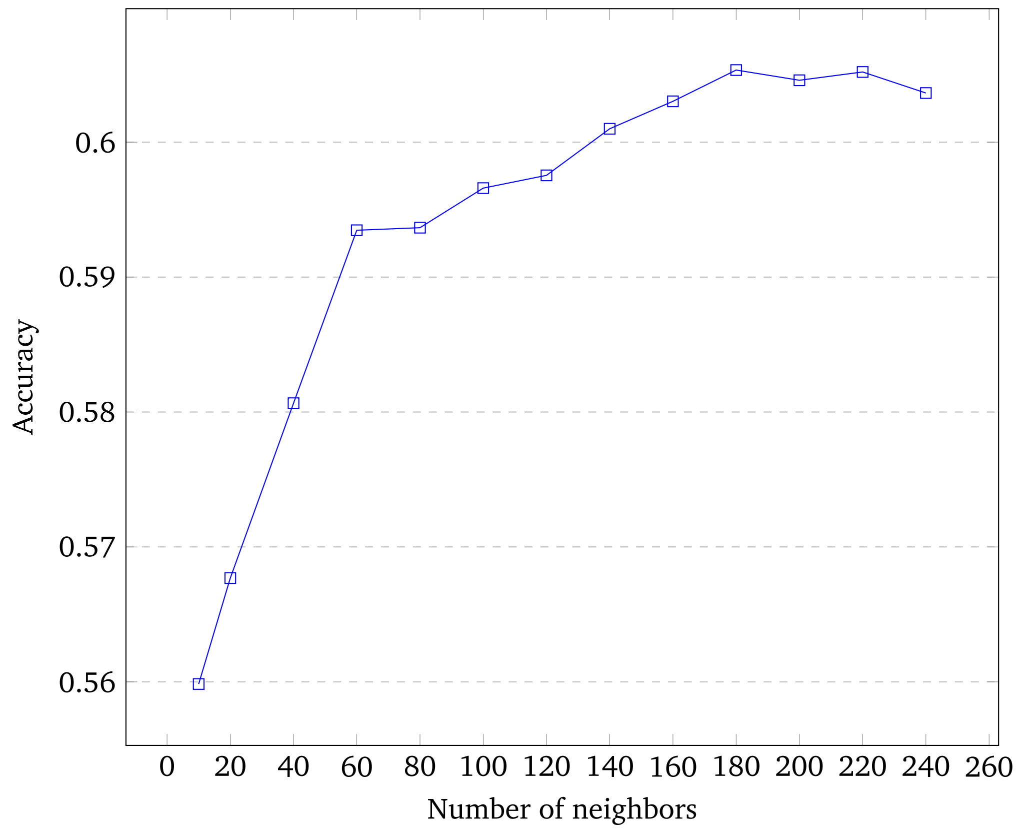

Figure 6: A graph showing number of neighbors in the nearest neighbor algorithm versus achieved accuracy.

3.4 Evaluation

Figure 7 shows how the previously chosen algorithms performed on the test data set. Speed is included as it can be important for the Monte Carlo Tree Search algorithm as explained later.

Algorithm	Accuracy	Speed
Logistic regression	0.616	0.056
Decision tree	0.565	0.039
Random forest	0.62	11.5
Gradient tree boosting	0.608	0.075
Nearest neighbors	0.599	29.605

Figure 7: A table showing accuracy (ratio of correctly classified samples) and speed (time in seconds it takes to predict 1000 samples on our computer) of machine learning models on the test data set.

All methods except decision tree perform with similar accuracy but there are large differences in speed. The accuracy of roughly 0.6 with different approaches suggests that it is not shortcomings in the machine learning methods that prevent a higher accuracy but that the data is inherently noisy and can not be perfectly predicted. This is consistent with our assumptions about Dota2: We know that team compositions only influence which team will win a match but not decide it and that the same team compositions playing against each other multiple times will have different outcomes depending on how the players perform. Our data shows that compositions are enough to correctly predict at least 60% of the matches. This accuracy is likely influenced by different balance patches and by how well the players pick. In a perfectly balanced games with all players making perfect hero choices all matches would have the same probability of the Radiant or the Dire winning. Our accuracy is a mixture of game balance, player skill and how good our prediction is; it cannot be categorized as good or bad in a vacuum, we can only compare different methods on the same data.

Logistic regression performs as well in terms of accuracy as other more powerful methods. Despite our knowledge of how important hero combinations are, on our data, logistic regression which performs predictions based on single heroes, is not beaten significantly by more complicated models. A plausible reason for this is that our data is not uniformly sampled: in all matches in our data the players already tried to make good hero choices that are likely to produce good compositions. To humans, such obviously bad compositions like a team of 5 carries would be standing out which is something logistic regression could not detect. But obviously bad compositions do not occur often enough when good players pick to matter much in terms of accuracy.

If we had used totally random team compositions as our data, logistic regression might have scored worse than other methods. However, the All Random game mode is not played often enough for us to have enough samples of it. Additionally, in All Random games the players can still make the choice of getting one new random hero to improve their team composition.

3.5 Different Feature Selections

Instead of a 224 dimensional vector representing heroes chosen with 0 and 1 there are other ways to represent a match as a vector of features. Two other ideas were tried but did not achieve better results.

The raw information of which hero belongs to which team could be represented in fewer dimensions by directly using the hero identifier in a 10 dimensional vector where the first 5 components represent the heroes on the Radiant and the last 5 the Dire. This representation might appear more intuitive to a human being, thinking about the data but it has problems in practice.

Firstly it introduces ambiguities in the way a match with the same heroes can be represented by reordering the hero identifiers, this is not possible with the 224 dimensional representation. The problem can be circumvented by ordering the hero identifiers in each team, however, small changes in hero selection could then cause large changes in the resulting feature vector which is undesirable for most machine learning algorithms in practice.

Secondly it makes it harder for the machine learning algorithms to interpret each feature as it has to learn this complex encoding in which each individual feature is only nominal but not ordinal: it does not make sense to compare the numerical values of the features to each other as they are identifiers for heroes and not some quantity like height. We can see that this leads to, for example, logistic regression working badly because coefficients for this feature vector are meaningless since all they do is to scale their input. Decision trees also fail since the simple decision of whether a specific hero is on one team will need a large amount of depth in the tree. This is caused by all of the trees decisions having to be of the form *is feature number X smaller than number Y*.

The other idea is to use features that have intrinsic meaning in the game. Instead of hero identifiers we can represent a hero by its attributes. The attributes form a description of a hero and can be compared to each other. This can be done for each hero individually so that if the attributes *damage* and *armor* are chosen as features, 2 components of the feature vector for each hero will be used. Or it can be done additively where all attribute values are added up respectively to get a total damage and armor value for each team.

This approach worked better than the previous one but still performed much worse than the 224 dimensional vector. This outcome is expected given our knowledge of the game, since heroes are much more defined by their abilities than their attributes. Thus there is not much to learn about a hero by looking only at its attributes. On the other hand, abilities cannot be used directly as they are complex descriptions of their effects in the game written in our language and are hard to convert into data which a computer understands.

4 Monte Carlo Tree Search

4.1 Game Theory and Combinatorial Games

Game theory [5, ch. 2.2] formalizes the concept of one or more players interacting in games defined by some set of rules resulting in some reward or payoff for the players.

A game consists of an initial state which can be transformed into other states by the players selecting actions to play. Which actions are legal in which state is determined by the rules of the game. Some states are terminal states. These represent a finished game in which no further actions can be taken. Terminal states assign rewards for each player individually and players try to maximize their reward.

Combinatorial games [5, ch. 2.21] are specific types of games for two players. They are described by the properties:

Zero-sum The rewards over all players sum up to zero. One player's gain is always the other player's loss.

Perfect Information The game state is fully known to all players. There is no hidden information.

Deterministic The game contains no random elements. The result of the same action in the same state is always the same, and likewise for the rewards.

Sequential Players take turns playing their actions. Actions are not played at the same time.

Examples of such games are Chess and Tic Tac Toe. The game state in Tic Tac Toe describes the the markings in each field. In the initial state every field is empty. Actions are selecting an empty field to mark and the game is over when all fields are marked or one of the player has 3 markings in a row. To make it zero-sum we could assign a reward of 1 to the player with three markings in a row and 0 to the other. If the game is a draw we could reward both players with 0.5.

For simple games like Tic Tac Toe the best way of playing them can be found but this quickly becomes infeasible for more complex games like Chess or Tic Tac Toe in 3 dimensions on a 5x5x5 board [2, p. xii].

One solution to allow computers to play such complicated games nonetheless is Monte Carlo Tree Search.

4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [5] is an algorithm used to let a computer find good moves in game theoretic games without necessarily solving them. Tree search refers to the game tree of states that the game can be in as nodes which are connected by decisions or moves by the players as edges. Monte Carlo refers to Monte Carlo methods which involve repeated random sampling for solving some problem.

For our purposes we are only interested in a specific type of games which are played by two players taking turns making discrete moves with deterministic effects, who have perfect information about the game state and who are in strict competition with each other. Chess, Go and Tic Tac Toe are examples of these games.

MCTS starts with the root node (the initial state) and expands its game tree while running. For every expanded node it keeps statistics, for example the number of losses and wins that were randomly sampled for the node. This is done by repeatedly performing iterations of the following steps [5, ch. 3.1]:

Selection Recursively starting at the root node descend into children that are not leaf nodes (do not represent terminal states in the game) and have unvisited children.

Expansion The selected node is expanded with one of its children. The criteria by which to select and expand nodes with which child is called *Tree Policy*.

Simulation Run a simulation of the game starting with the child selected in the previous step until a terminal state is reached by choosing from the available actions repeatedly. This yields an outcome of the game (for example whether it is a loss or win for the player). The criteria by which to select actions and compute the outcome is called *Default Policy*.

Backpropagation Backpropagate the outcome up the tree.

In figure 8 we see a visual representation of MCTS. Each node remembers its number of play outs and number of wins from the perspective of the currently moving player. In the selection phase we follow the arrows to the selected node which is then expanded to one of its child nodes. This child node starts with 0 wins and 0 play outs and is then simulated once which turns out to be a loss. This result is propagated back up the tree and all nodes on the path are updated. These steps are repeated until some constraint

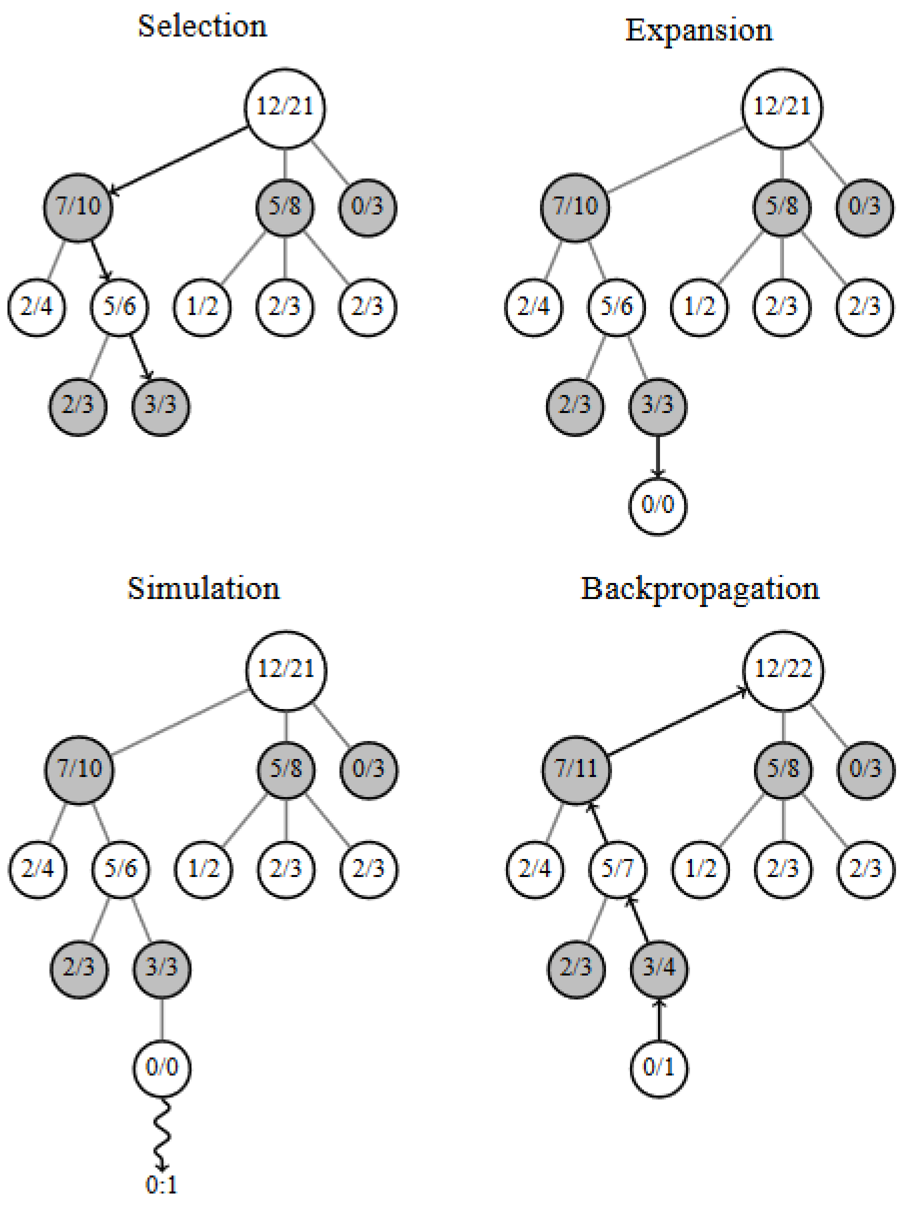

Figure 8: A visual example of the MCTS algorithm. [16]

in form of time, processing power or number of iterations is reached. At that point we select the most promising action to play according to some critera. For example the action leading to the node with the highest ratio of wins to total simulations could be selected.

We use a variant of MCTS called *Upper Confidence Bounds for Trees* (UCT) [5, ch. 3.3]. UCT interprets the Tree Policy as a multi-armed bandit problem. A multi-armed bandit problem describes a problem where one of multiple actions needs to be chosen repeatedly in order to maximize a reward over all plays, for instance, when playing a bandit slot machine. In this situation there is a trade off between exploiting the reward of known good actions by simulating them further and exploring other actions that have not yet been visited or appear to have a low reward.

UCT tries to minimizes the growth of the regret which is the loss in reward due to not playing the best action by always selecting the child node j which maximizes

$$X_j + 2C_p\sqrt{\frac{2ln(n)}{n_j}}$$

where X_j is the already estimated reward of child j, n is the visit count of the parent node, n_j is the visit count of child j and $C_p > 0$ a constant. The estimated reward is the total cumulative reward of the random playouts divided by the total number of playouts. It represents the exploitation of known rewards and is weighed against the second part of the formula which represents exploration as it grows when child j has been visited few times in relation to its parent. Adjusting C_p results in more or less exploration and good values are domain dependent.

5 Recommending Hero Picks with MCTS

5.1 Introduction

To fulfill our goal of providing good hero recommendations we implemented MCTS with UCT for the Dota 2 hero picking phase. Good recommendations in our case means recommending heroes with which the player's team is most likely to win based on the previously picked heroes. Not taken into account are player specific preferences of liking or disliking to play some heroes as this is something completely subjective to each player.

With the final application a player can evaluate partial or full team compositions, get good candidates for the next hero to pick or ban or let the application simulate the picking phase by picking against itself.

In our model of the picking phase as a combinatorial game, the game state is given by each team's heroes, the banned heroes and whose turn it is to pick. Actions can be either picking or banning some number of heroes. Whether the next action is a pick or ban and the number of heroes to choose is given by the specific game mode.

The implementation was held generic to allow using any of the game modes Captains Mode, Captains Draft, Random Draft, Ranked All Pick.

When both teams have 5 heroes the game is over (but for the human players the main game begins) and the team compositions are evaluated into rewards to be used in MCTS. The reward for a terminal state is the probability for each player (as in teams of human players) to win the game and is predicted by one of the machine learning models described previously. This highlights the useful property of MCTS that it only needs to be able to compute rewards for terminal states but does not require any other domain knowledge. By having models that can predict the chances of finalized teams winning we can recommend good choices even for partial teams.

This is only an approximation of the real picking phase. In particular we assume each team picks in unison whereas in reality players perform their own picks and might not communicate with the rest of the team. We also assume that each team's goal (reward) is only to win. This is often true but some players might instead play for fun or have some specific heroes in mind which they want to play regardless whether this gives them the highest chances of winning.

As described earlier, the most promising machine learning models are logistic regression and gradient tree boosting due to their high accuracy and fast speed. Speed is important as MCTS generally provides better results by running more iterations and our use case is running it in a soft real time scenario where the user has to decide on a hero within for example 35 seconds when playing Ranked All Pick.

However our logistic regression model only weights heroes individually and statically by learning their coefficients. If this was really what decides games then the best action would be to always pick the available hero with the best coefficient without taking other heroes into account, and using MCTS would be unnecessary. We know that is not really a good way to pick and thus chose gradient tree boosting as the best candidate.

5.2 Implementation

Our implementation as seen in figure 9 roughly follows the description in [5][ch. 3.3]. This snippet is simplified for readability: some part are left out and banning is not implemented.

```
class State:
  def is_terminal(self):
    ...
  def get_actions(self):
    ...
  def get_next_state(self, action):
    ...

class Node:
```

```python
  def __init__(self, state=State(), incoming_action=None, parent=None,
   total_simulated_reward=0, visit_count=0):
    self.state = state # s(v)
    self.incoming_action = incoming_action # a(v)
    self.parent = parent
    self.total_simulated_reward  = total_simulated_reward # Q(v)
    self.visit_count = visit_count # N(v)
    self.children = list()
    self.tried_actions = set()
    self.remaining_actions = self.state.get_actions() # All actions initially
  def expand(self):
    action = random.choice(self.remaining_actions)
    self.remaining_actions.remove(action)
    child = Node(self.state.get_next_state(action), action, self)
    self.children.append(child)
    return child

def uct_search(Cp, time_limit):
  start_time = time.time()
  while (time.time() - start_time) < time_limit:
    node = tree_policy(root_node)
    reward = default_policy(node.state, model)
    backup(node, reward)
  best = best_child(root_node, 0)
  return best

def tree_policy(node, Cp):
  while not node.state.is_terminal():
    if len(node.remaining_actions) != 0:
      return node.expand()
    else:
      node = best_child(node, Cp)
  return node

def best_child(node, Cp):
  constant = math.log(node.visit_count)
  return max(node.children, key=lambda n:
    n.total_simulated_reward / n.visit_count + Cp * math.sqrt( constant / n.visit_count))

def default_policy(state):
  for_radiant = state.radiant_moved()
  while not state.is_terminal():
    action = random.choice(state.get_actions())
    state = state.get_next_state(action)
  return compute_reward(state, for_radiant, model)

def compute_reward(state, for_radiant):
  radiant_win = predict_radiant_win_probability(state)
  if for_radiant: return radiant_win
  else: return 1 - radiant_win

def backup(node, reward):
  while node != None:
    node.visit_count += 1
    node.total_simulated_reward += reward
    reward = 1 - reward
    node = node.parent
```

Figure 9: A code snippet of our MCTS implementation in Python.

The State class represents all possible game states of the picking phase. It keeps track of the heroes in each team and the banned heroes. The Node class is a node in the MCTS game tree. The uct_search function repeatedly performs the steps of MCTS until a time limit is met and then returns the child node with the highest average reward by calling best_child with C_p set to 0.

Using a time limit instead of a maximum number of iterations makes sense for our use case of a real player using the application in a real game but it means that some of the results can depend on the speed of the computer they are run on. For example a faster computer will be able to run more iterations in the same time which could result in better hero suggestions and vice versa. For comparison our computer which was used for everything is able to run roughly 3000 to 5000 iterations per second.

The exploration term in the UCT formula in the code is changed slightly to remove unnecessary multiplications which are now reflected directly in the C_p parameter and $ln(n)$ is only computed once per best_child call.

Tree Policy and Default Policy choose actions at random to expand or simulate nodes.

Not seen in figure 9 is the code that performs the simulation of the picking phase consisting of receiving hero picks from the players (using MCTS or asking the user) and managing the game state. There an optimization is performed which after the other team's action reuses the relevant part of the game tree which the previous invocation of MCTS might have already built.

When games are done the machine learning model evaluates the final team compositions to find the winner. Here we have more information than just which team won as scikit not only predicts the class of a sample but can also give the probability of class membership. Instead of noting just the winner we see by how large a margin the winner has won. The average of this winning probability across all play outs is used as the average reward whenever we let different versions of the MCTS implementation play each other in following experiments.

5.3 Improvements

5.3.1 Speed

Because of the real-time requirement the speed of the implementation is important in order to achieve as many iterations of MCTS as possible.

To analyze our implementation we used the project *line_profiler*[3].

Our implementation uct_search spent 48% of its running time in tree_policy and default_policy respectively. In tree_policy 83% was spent with the call to best_child which is already a simple function that cannot be optimized further easily. In default_policy 71% was spent with the machine learning prediction of the simulated game which shows that the speed of our models matters for the implementation in comparison to time spent in other places.

5.3.2 Exploration Parameter

To find good values for the exploration parameter C_p we let the application play itself in the Ranked All Pick mode without the initial banning. One player always uses a C_p of 1 while the other varies C_p. Both players use the same time constraint, and which player plays as the Radiant or Dire and who has the first pick is randomized in each game since there might be an inherent advantage for one side or the team with the first pick.

The results are shown in figure 10.

[3] `https://github.com/rkern/line_profiler`

time \ C_p	0	2^-6	2^-5	2^-4	2^-3	2^-2	2^-1	1	2^1	2^2	2^3	2^4
0.1	0.55	0.5	0.58	0.54	0.53	0.55	0.5	0.51	0.46	0.52	0.48	0.49
1.0	0.54	0.56	0.56	0.58	0.58	0.51	0.52	0.5	0.55	0.47	0.51	0.47
10.0	0.56	0.59	0.58	0.57	0.59	0.59	0.54	0.51	0.51	0.49	0.51	0.48

Figure 10: A table showing the average reward of a MCTS player with varying values of the exploration parameter against itself with the exploration parameter set to 1.

Powers of 2 were used to get a varied ballpark of candidates and different time constraints to see the MCTS player in different usage scenarios. Each time - C_p combination was played roughly 50 times. Curiously the expected reward with both players using a C_p of 1 is not exactly 0.5; but this is probably caused by statistical variance.

We see rewards generally go higher than 0.5 with C_p values getting smaller than 1 and vice versa. The time constraint of 10 seconds which was the most important one to us as it is closest to using the application for real games peaks at a C_p of 2^-6, 2^-3 and 2^-2. Out of those 2^-3 (0.125) has the highest average reward over the 3 time constraints combined so we decided on it as the best C_p value.

We also investigated the C_p values in the Captains Mode game mode. Because this mode requires more decisions and includes banning, a different C_p value might be optimal. This is shown in figure 11. We only used the time constraints 1 and 5 seconds to reduce the time it takes to run the experiment and each time - C_p combination was played roughly 30 times.

time \ C_p	0	2^-6	2^-5	2^-4	2^-3	2^-2	2^-1	1	2^1	2^2	2^3	2^4
1.0	0.51	0.47	0.56	0.57	0.56	0.5	0.54	0.47	0.5	0.51	0.46	0.51
5.0	0.53	0.56	0.62	0.59	0.56	0.54	0.45	0.51	0.45	0.49	0.56	0.46

Figure 11: A table showing the average reward of a MCTS player with varying values of the exploration parameter against itself with the exploration parameter set to 1 in the Captains Mode game mode.

We see a similar trend as in figure 10 and values that do not differ too strongly. This time the peak of the longer time control is at 2^-5 which tells us that the exploration-exploitation trade off should be more on the exploitation side in Captains Mode than it is in Ranked All Pick.

5.3.3 Tranpositions

In some games including the Dota 2 picking phase the same game state can be reached via different paths through the game tree: One team first picking hero A and then B results in the same state as picking B first and then A, assuming the other team has the same heroes in both cases. This is called a transposition [5, ch. 5.2.4].

In these games the performance of MCTS can be improved by making use of the information that the game tree is a directed acyclic graph instead of the normal tree. Where previously nodes with the same state were treated as totally unrelated nodes they can now share information.

We implemented transpositions according to the UCT2 version described in [8]. In this version the previously mentioned UCT equation used in best_child uses the average reward of all transpositions of the child instead of the average reward of the specific child node which can have many transpositions in other nodes in the game tree built by MCTS.

We did not implement UCT3 as it is more complex, slower and only recommended when the time to simulate games dominates the time it takes to update the additional statistics in UCT3 [8, ch. V] which is not true in our implementation.

uct_search Creates or reuses an existing transposition table (in the case of being used for all the picks in the same game). The transposition table stores information about transpositions indexed by their game states.

Node Nodes no longer store their accumulated reward as this information exists in the transposition table.

backup In addition to updating the nodes of the current path through the game tree the transpositions of the path are updated as well.

best_child Uses the average reward of the transposition of the current node instead of the average reward of just the current node.

Since usage of transpositions might have an effect on the optimal C_p values we recreated the experiment of figure 10. To speed up the process only the most important time control of 10 seconds was used. The results are shown in figure 12.

C_p / time	0	$2^{-}6$	$2^{-}5$	$2^{-}4$	$2^{-}3$	$2^{-}2$	$2^{-}1$	1	2^1	2^2	2^3	2^4
10.0	0.54	0.58	0.62	0.57	0.56	0.58	0.48	0.53	0.5	0.47	0.47	0.5

Figure 12: A table showing the average reward of a MCTS with transpositions player with varying values of the exploration parameter against itself with the parameter set to 1.

Each time time constraint was played roughly 30 times. $2^{-}5$ (0.03125) has the best win rate thus we used it for C_p in the transposition implementation.

We repeated the experiment for Captains Mode as seen in figure 13 with roughly 30 play outs per C_p value.

C_p / time	0	$2^{-}6$	$2^{-}5$	$2^{-}4$	$2^{-}3$	$2^{-}2$	$2^{-}1$	1	2^1	2^2	2^3	2^4
5.0	0.55	0.57	0.6	0.57	0.6	0.52	0.54	0.51	0.53	0.47	0.46	0.51

Figure 13: A table showing the average reward of a MCTS with transpositions player with varying values of the exploration parameter against itself with the parameter set to 1 in the Captains Mode game mode.

The peaks are at $2^{-}5$ and $2^{-}3$. $2^{-}5$ was also the best value in the Ranked All Pick experiment so for the transposition implementation we can use it as the optimal C_p value in both game modes.

5.3.4 Domain Knowledge Enhancements

MCTS can be improved by incorporating domain knowledge [5, p. 16] earlier into the algorithm instead of using only the results of random play outs. With our data we measured the win rate of individual heroes and used this information in the MCTS implementation.

We changed default_policy to not choose totally random actions to simulate a game as this is not how real games usually play out. Instead of selecting from the remaining heroes uniformly, default_policy is now biased to choose heroes with higher win rates more often. This ensures that it is still possible to randomly choose any hero while making it more likely to choose individually good heroes.

No such bias is used when the next action is a ban instead of a pick as it not clear whether banning heroes with strong win rates is beneficial as that hero is banned for one's own team as well.

Heroes are weighed linearly with their win rate: a hero with twice the win rate of another is twice as likely to be selected.

The same change was also applied to the expand method and the results are seen in figure 14.

enhancement	default_policy	expand	both
outcome	0.48	0.49	0.48

Figure 14: A table showing the average reward of MCTS with domain knowledge used to try to improve the default_policy and expand methods playing against itself without the changes.

Each matchup was played roughly 100 times with a time constraint of 1 second. We see all of our changes worsen the win rate. It is not clear why this happens but one candidate explanation was that the increased computation cost leads to fewer number of iterations and this is not compensated by what was gained with the changed.

To test this we repeated the experiment but this time with an iteration limit of 2500 instead of a 1 second time constraint as seen in figure 15.

enhancement	default_policy	expand	both
outcome	0.51	0.52	52

Figure 15: A table showing the average reward of MCTS with domain knowledge used to try to improve the default_policy and expand methods playing against itself without the changes. Instead of a time limit an iteration is used.

This time a slight improvement was measured. Thus a more efficient implementation might make this change worthwhile.

5.4 Analyzing Team and Game Mode Balance

With an asymmetrical playing field the Radiant and the Dire sides in Dota 2 are not equal in win rate: the Radiant wins about 55% of the matches when humans play. Our machine learning models were trained with undersampling so from their perspective this imbalance does not exist.

The game modes are also not perfectly balanced. In Ranked All Pick one team has the first pick and the other the last, and in the more complicated Captains Mode the first pick is offset by giving the other team two picks in a row and the last pick. Intuitively having the first pick gives one the initiative and the ability to get one of the overall best heroes, on the other other hand the last pick can react to all the earlier picks and cannot be countered.

To see how strongly this affects the outcome in our MCTS implementation we let it play against itself in Ranked All Pick and Captains Mode and recorded the results as shown in figure 16. Each game mode was played roughly 45 times.

Game mode	Radiant	First pick
Ranked All Pick	0.46	0.53
Captains Mode	0.52	0.6

Figure 16: A table showing the average reward of MCTS in relation team and first pick.

We see that in Ranked All Pick MCTS actually performs better on the Dire than on the Radiant and slightly better on Radiant in Captains Mode. This was likely caused by some interaction between the machine learning model and MCTS which favors one side in some situations.

The side with the first pick has a clear advantage in both game modes but this is more pronounced in Captains Mode where the first pick has a 60% win rate. This shows that at least when MCTS plays with our machine learning model both game modes are not perfectly balanced. We could not find statistics on real games on this matter as our data does not include which team had the first pick. However in tournaments both side and first pick advantage are offset by both teams participating in a coin flip where the winner decides if they want to choose their preferred side or whether they want the first pick.

We also looked at how MCTS builds its expected rewards in Captains Mode and noticed that it does not value the initial bans in Captains Mode much. In our simulations the top visited choices' (which have accurate scores due to a large number of play outs) average rewards were usually within 0.02 of each other. This means that no matter what is banned it will not influence the expected reward much and that there are no decisively good bans. This can be intuitively explained: because these bans happen before any hero is picked and all bans apply to both teams. The initial banning is a personal preference of human players who dislike playing against some particular heroes, have a strategy they want to try already in mind or know that this particular other team is very good with some heroes. Since MCTS has no previous preferences or background knowledge of the players these bans do not matter much to it.

Bans in the later stages of the picking phase still had less impact on the expected reward than picks but they had more impact than earlier bans.

5.5 Evaluation

For the final evaluation of our MCTS implementations we implemented a random player who always chooses totally random actions and a win rate player who always picks or bans the hero with the highest individual win rate in our data set (or cumulative win rate if multiple heroes need to be chosen). We let these new types of players, the initial MCTS implementation and the transposition implementation (both with their optimal C_p values) play matches against each other in different game modes with a 10 second time constraint. Each matchup was played roughly 40 times. The results area shown in figure 17.

Gamemode	Player	Random	Winrate	Initial	Transpositions
Ranked All Pick	Random				
	Winrate	0.8			
	Initial	0.82	0.56		
	Transpositions	0.84	0.64	0.53	
Captains Mode	Random				
	Winrate	0.72			
	Initial	0.81	0.66		
	Transpositions	0.82	0.6	0.52	
Random Draft	Random				
	Winrate	0.75			
	Initial	0.78	0.64		
	Transpositions	0.79	0.66	0.56	

Figure 17: A table showing the average reward of different MCTS implementations against each other in different game modes. Each probability is the win rate of the player of the left column versus the player in the top row.

We see that against the random player all other players win decidedly with expected rewards between 0.75 and 0.85. The Transposition player performs slightly better than the initial player which performs slightly better than the win rate player. In Captains Mode the win rate player performs worse than the other two likely because it takes high win rate heroes away from itself by banning them.

Against the win rate player both the initial and transposition player perform well with rewards ranging between 0.56 and 0.66 but the initial player is worse than the transposition player by roughly 0.1 expected reward in ranked all pick than in the other modes.

Finally the improvement of the transposition player over the initial player is noticeable in all game modes but strongest in Random Draft.

We note that it is easy to beat a random player decisively with the simple heuristic of the win rate player while MCTS cannot improve the results over the win rate player as decisively. Transpositions provide a slight further improvement but this could be made larger by implementing them more efficiently in order to raise the iterations per second.

5.6 Making a Useful Application

We wanted to turn our results into a useful application which could be used when playing real games of Dota 2.

To make it easier to read the recommended action output we convert the internal hero representation which is just the hero's identifier back into the corresponding name the hero is known under in Dota 2. When users enter a hero that has been picked into the application they can write the hero's name. Since user's spelling might contain typos we search for the closest matching name.

When recommending heroes the application displays a number of best choices and their expected rewards. This way users can make the final decision themselves in light of the information provided by the application.

As Dota 2 regularly receives balance patches that drastically change which heroes are good in which situations it is important that our application is easily updateable. This does not require much manual work because every step is implemented programmatically. We can gather new match data and train models on this data in a few lines of code. This is all that is needed to adapt to a new balance patch. The MCTS implementation only relies on the machine learning model and does not require any changes.

6 Conclusion and Future Work

6.1 Conclusion

We have shown how machine learning methods can be used to predict outcomes of Dota 2 matches given the match's team composition. The methods of logistic regression, decision tree, random forest, gradient tree boosting and nearest neighbors were evaluated on the data that we collected. All methods except decision trees achieved comparable accuracies between 0.6 and 0.62 but logistic regression and gradient tree boosting were multiple orders of magnitude faster in their predictions than the others. We have described why despite comparable accuracy and speed the gradient tree boosting method is nonetheless better suited for giving character recommendations than the method of logistic regression.

Monte Carlo Tree Search with Upper Confidence Bounds for Trees was implemented for the Dota 2 character picking phase. MCTS provided significantly better character recommendations than simpler methods. Good values for MCTS' exploration parameter were experimentally determined which improved playing strength significantly.

The transposition enhancement was implemented and compared to the basic implementation. It increased playing strength significantly. An enhancement using individual win rates of heroes in the expand and the simulation step was implemented but only performed slightly better with an iteration constraint and worse with a time constraint than the basic implementation.

We analyzed the balance of the Radiant and Dire and of the game modes Ranked All Pick and Captains Mode and found that the side with the first pick is significantly more likely to win the game than the other team.

The final application is capable of playing against itself and can provide users recommendations for matches they are actually playing in.

6.2 Future Work

Our implementation of MCTS is good but not yet perfect and could be improved in a number of ways:

- It could be reimplemented in a more optimized programming language. Python allows for quick programming and prototyping but a lower level language like c++ would probably be faster and use less memory.
- A greater variety of domain knowledge enhancements could be investigated and the performance of the hero win rate enhancement improved. Finding and adding more heuristics in the expand and simulation steps would likely improve performance per iteration as we have already seen. Here more of our personal ideas and experience as a Dota 2 player of what constitutes better and worse team compositions could be tried out, for instance not picking too many heroes which fill the same role.
- Improving the machine learning models would make MCTS perform even better.
- The exploration parameter could be analyzed in more depth. We only looked at powers of 2 to get an overview over promising values. Checking smaller intervals might lead to further improvements.

The machine learning process could be improved by gathering more or better suited data. We noted that the game mode from which our data was gathered influences the team compositions. Because experienced players already try to pick well we do not get uniformly random samples over all possible team compositions. Sampling data from totally randomized matches might improve model strength. Besides this we did not see any obvious improvements but machine learning is a large field with many approaches. Trying more machine learning algorithms would perhaps lead to improvements in predictive accuracy, considering that we might have missed a good candidate in the algorithms that we evaluated. Especially deep neural networks have shown good results in other domains and might work well here too.

References

[1] The International 2016. http://www.dota2.com/international2016/overview/. Accessed on 2017-03-03.

[2] József Beck. *Combinatorial games : Tic-Tac-Toe theory*. Cambridge University Press, 2008.

[3] Leo Breiman. *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[4] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[5] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*, 4, 2012.

[6] Steam Charts. http://steamcharts.com/app/570. Accessed on 2017-03-03.

[7] Samprit Chatterjee and Ali S. Hadi. *Regression analysis by example*. Wiley-Interscience, 2006.

[8] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and move groups in monte carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395, Dec 2008.

[9] Esports Earnings. http://www.esportsearnings.com/tournaments. Accessed on 2017-03-03.

[10] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38:367–378, 2002.

[11] Scikit Learn. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression. Accessed on 2017-03-03.

[12] Scikit Learn. http://scikit-learn.org/stable/modules/tree.html. Accessed on 2017-03-03.

[13] Scikit Learn. http://scikit-learn.org/stable/modules/ensemble.html#forests-of-randomized-trees. Accessed on 2017-03-03.

[14] Scikit Learn. http://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting. Accessed on 2017-03-03.

[15] Scikit Learn. scikit-learn.org/stable/modules/neighbors.html#classification. Accessed on 2017-03-03.

[16] Mciura. https://commons.wikimedia.org/wiki/File:MCTS_(English).svg. Accessed on 2017-03-03. Mciura (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons.