Abstract—This work focuses on the design of a supervised-learning recognition system to classify strategic decisions performed throughout the first 15 minutes of DotA2 matches. Main steps involved are: definition of pattern classes, feature extraction and selection, collection of training and testing samples from human gameplay and the learning of an Artificial Neural Network classifier, capable of identifying the correct strategies to be employed at different times throughout matches. Any possible integrations of the developed work with the game’s Bot API are also explored. The main purpose is to ultimately use the model developed to improve strategic decision-making of AI bots in this game. Our solution is evaluated with specific metrics, such as precision, recall, f-measure, among others, which dictate an overall performance of our model. The obtained results show that the built classifier performs considerably better than random classifications. This paper also alerts to other problems that remain unsolved: lack of proper integration of Artificial Intelligence models with the game’s API and the nonexistence of any automatic-labeling methods for DotA2 strategies.

Index Terms—Artificial Intelligence, Real-Time Strategy, Multi-player Online Battle Arena, Machine Learning, Supervised Learning, Bot Development

I. INTRODUCTION

A. Motivation

One of the most noticeable flaws found in games these days, usually has to do with poor Artificial Intelligence (AI) systems (weak performance, inaccurate decision-making and lack of adaptability to unfamiliar scenarios), specifically in Multi-player Online Battle Arenas (MOBAs) where the amount of regular updates performed can be cited as one of the reasons for the low AI research, in these type of games. However, games provide limited environments that can reproduce many characteristics from real world, and where winning or losing have well-defined rules, making it easy to evaluate such criteria. This has driven many AI experts to use games as testbed for development of AI.

The good aspect about DotA2 and other similar MOBAs is the amount of new strategies that are continuously being developed as the game changes, which always keeps the game interesting and challenging to its players. Being an avid player and testing in first hand how faulty and outdated the default AI system is in this game, raised my interest even further in creating agents with better decision-making and overall more adaptive to unfamiliar situations. Game-wise, the team which performs better at an early-stage of the game, is usually the one with higher chances to win, making this stage extremely important. This lead us to focus our development in the early-game phase (in the first 15 minutes of a DotA match), where strategic decision-making is a decisive factor.

In DotA2, there are five different game-roles to be played, with the Support role arguably being the most important in earlier phases of any match, due to being the richest strategy-wise, interchanging between strategies the most, during the first minutes of play.

B. Problem Description

This document focuses on the development of AI Bots for the game DotA2. The decision process of AI agents in this game comes in two forms: tactical, which has to do with specific micro-action behaviour, like moving to a certain location or using an ability; and strategic, which has to do with a higher-level behaviour related with general objectives, like helping an ally or attacking and enemy hero, which represents more complex behaviours, encompassing a whole sequence of micro-actions. The available default AI Bots have different levels of difficulty: in the hardest mode, they show a decent tactical behaviour (mainly because their reaction time is improved) but still have a poor strategic decision-making. As so, the main problem addressed is how to develop a model that has a better control of the strategies applied by agents during early stages (first 15 minutes) of a match.

C. Hypothesis

From the previously described research problem, we can isolate the following testable hypothesis:

- **By developing and training of an Artificial Neural Network classifier, we will be capable of identifying, for the support role, the correct strategies employed during the first 15 minutes of a match, based off the labeling of replays by highly-skilled players.**

D. Objectives

1) Research and Analyze previous and relevant works to our problem, about the application of Artificial Intelligence in games
2) Design and Develop AI agents for DotA2
3) Implement Learning Algorithms that can be used for training the Agents
4) Evaluate the viability of applying AI techniques to improve AI agents’ behaviour in DotA2
II. RELATED WORK

A. Background

With the release of the Real-Time Strategy (RTS) and MOBA genres, AI found a challenging testbed emerging, where not only machine behaviour was important, but also dealing with a high number of complex variables, that would directly affect the chances of winning or losing the game. Features like resource management and coordination, among others, bring a whole new level of difficulty to AI research: agents are now built in uncertainty conditions and complex state spaces, in games where frequently human intuition is a game-deciding factor. DotA was the first game of the MOBA-genre, bringing all these new constraints to the topic of producing AI systems in games.

To better understand specific aspects regarding the proposed solution later on, some prior knowledge of the game can be useful: in short, there are two opposing teams, called Radiant and Dire, whose goal is to destroy each other’s base, in order to win the game. Each team consists of five players, that should work together to take down the Ancient, which is the enemy’s base main structure. There are also three very important concepts to keep in mind: heroes, defensive structures & units and player roles, which can all be found and better understood in DotA2 encyclopedia¹.

B. State of Art

This section will present literature reviews of some works done in this area. Michael Mateas, in one of his works, gave a very good definition of the ‘game AI’ concept [1]:

“The phrase ‘game AI’ covers a diverse collection of programming and design practices(...)What links these practices together under the single term ‘game AI’ is a concern with ‘intelligent’ behavior; that is, behavior that the player can read as being produced by an intelligence with its own desires, behavior that seems to respond to the players actions at a level connected to the meaning of the players actions”

Great examples of AI application started with “Pac-man” [2], which was one of the pioneer games in introducing proper game AI, in which the details of ghosts’ behaviour (enemies) is one of the primary determinants of player experience and critical to understanding the game. Other notable games worth considering, that brought innovations to the scene [3] are Half-Life [4], in which squad AI is used effectively for the first time; Thief [5], which brought an accurate sensory model allowing AI actors to respond realistically to light and sounds; Sims [6], which introduced basic desires driving choices and actions, also modeling emotional interaction between its characters; Total War [7], where thousands of AI-controlled soldiers are featured for the first time, introducing accurate battle simulations; and Facade [8], a first-person interactive drama that makes use of natural language parsers allowing the player to coach the AI’s and guide the story.

In sum, AI has grown a lot in the last few years, providing better player experience, either from improving aesthetic feel to having agents properly competing or aiding the player in some way.

The primary focus of this work is, however, on the development of AI agents for DotA2, which is a game where real-time actions must be taken, having a huge time-constraint for decision making and less time available for planning. Following up, are described works performed in the game and in a similar game-genre, stating their respective contributions to the evolution of AI research in this area.

1) Artificial Intelligence in RTS games: Developing intelligent agents for RTS games has proven to be a very hard task due to constraints of the genre. Early researches identified six main challenges [9]: Resource Management, Decision-making under uncertainty, Spatial and temporal reasoning, Collaboration, Opponent modeling and learning and Adversarial real-time planning.

There are two different types of decision making that need to be taken into consideration in RTS’s: micro-management, responsible for single unit behaviour; and macro-management, responsible for sequencing the overall actions like building units and structures in a certain order, as well as the global strategy and resources management, which involves allocation for different tasks.

A work by Tavares, Azpurua, Chaimowicz [10] explored the important Task Allocation problem in the popular game Starcraft. Starcraft’s application programming interface called BWAPI, intended for algorithm development, was used for implementing the bot in which was applied an algorithm based on swarm intelligence, which is often used for these type of assignment problems like task distribution, and scheduling problems. Genetic algorithms were applied to find a proper agent-task combination that maximizes a global metric of the agent’s performance.

Martinho & Santos [11] also tackled the Task Scheduling problem, essential to the RTS-genre, by implementing an algorithm inspired in wasp-behaviour to schedule unit production in a modified scenario from the game Warcraft III: Frozen Throne. By evaluating their routing-wasp algorithm, results shown that it was most appropriate for situations with continuously high request rates and for large scale scenarios, requiring less computation power than other algorithms to achieve similar levels of performance.

A common problem when creating agents for Strategy games is their lack of adaptability and improvement, which is very common in scripted implementations using classical AI hard-coded approaches e.g., Finite State Machines (FSM). This usually turns out to be a problem: every game match is different and requires different methodologies used by the agent to be successful, which is precisely where these approaches struggle: in building an adaptive behaviour. Cunha [12] tackled this problem, also applying swarm intelligence, using a decentralized approach, to improve an AI in a game called ‘Almansur’.

The works presented above had to do with applying AI to improve agent’s overall Strategy playing the game, which corresponds to high-level decision process, and the highest

¹Online: https://liquipedia.net/dota2/Main_Page
level of abstraction for game comprehension, which is the same concept this document aims to address.

2) Artificial Intelligence in Dota2: In this section are reviewed some recent applications of Machine Learning in Dota2. One important branch of research has to do with detection and classification of heroes’ roles and positions in the game.

A work by Gao, Judd, Wong, Lowder [13] targets the classification of both heroes that players are using and the role they are taking, based on data obtained from available matches replays. The classification method itself used supervised learning techniques, specifically logistic regression and random forest classifiers. Results obtained from this study were very good overall, showing a massive improvement that can be attained by using logistic regression and random forest supervised learning techniques over a random guess, which would be expected to yield less than 1% accuracy.

Eggert, Herrlich, Smeddinck, Malaka [14] continued to explore this subject, providing approaches regarding the construction of complex attributes from extracted low-level data, as well as investigating the applicability and performance of Supervised Machine Learning to classify player behaviour in terms of specific heroes’ roles within a team, by comparing and discussing the effectiveness of a bigger variety of supervised classification algorithms, including support vector machines with sequential minimal optimization(SMO), naive Bayes(NB), and Bayesian networks, in addition to the previously mentioned Logistic Regression(LR) and random forest decision trees(RF), which lead to even better results, with an average of 96% accuracy in correctly determining player roles, using LR.

In MOBA games, correct interaction between players and better planned team fights are ultimately what dictates the winning team. As so, another obvious branch of research in Dota2 is encounter detection and combat results prediction.

Yang, Harrison, Roberts [15] explored this concept by analyzing combat logs from collected matches replays and modeling combat as a sequence of graphs in order to extract patterns that could predict the outcome, not only of team fights but of entire games. Results obtained revealed an accuracy in predicting combat results of 80%, showing that graph theory can certainly be helpful in combat predictions of similar games.

Another interesting work about this subject, by Schubert, Drachen, Mahlmann [16], tackles the Encounter Detection problem by providing a definition of 'encounter' based on spatial positioning, and using an algorithm to extract these, from available match replays. An experimental evaluation based on their collected dataset examined the importance of performance evaluation metrics, and their predictive power and results drawn were that, by evaluating performance metrics, after 10 encounters, there is almost a 90% success in correctly predicting the winning team, proving that the use of algorithms for Encounter Detection is beneficial for this branch of research.

The most popular application of AI in Dota2 is probably the OpenAI Bot, created for the purpose of playing 1vs1 matches. Reinforcement Learning(RL) was applied to build the bot: it started without any prior knowledge of the world, and learned everything from scratch through self-play. This experiment proved to be of great success: after weeks of constant training, the bot managed to beat the top professional human players, showing that self-play can be a learning technique applied to improve the performance of machine learning systems from below human-level to superhuman.

C. Artificial Neural Networks

A work developed by Basu, Bhattacharyya, Kim [17], addresses the impact of using Artificial Neural Network(ANN) systems in pattern recognition problems, identifying research topics and applications which are at the forefront of this challenging field. According to the aforementioned paper, neural networks are mainly characterized by their ability to learn complex nonlinear input-output relationships, using sequential training procedures, and adapting themselves to the data. The most commonly used family of ANNs for pattern classification tasks is the feed-forward type, which includes the Multi-Layer Perceptron(MLP) network.

The MLP consists of an input layer, an output layer, and one or more hidden layers of neurons in between these two, aiming to increase the ability of modeling complex functions. There are typically three steps involved in regard to data processing [18]: training stage, weight determination stage and classification stage. Usually, the training process starts by assigning arbitrary connection weights which are constantly adjusted to minimize the error obtained, until an acceptable model accuracy on the training data is reached. Specifically, in an MLP architecture, the input that a single hidden neuron \( j \) receives from the neurons \( i \) of its preceding input layer, may be expressed as:

\[
net_j = \sum_{i=1}^{t} w_{ij}p_i
\]

(1)

Where \( w_{ij} \) represents the connection weight between input neuron \( i \) and hidden neuron \( j \). \( p_i \) is the data at the input neuron \( i \) and \( t \) is the number of input neurons. After having determined the net sum at a hidden node, an output response is provided at the node using an activation function.

Throughout years, many activation functions have been proposed. One of the most used is the sigmoid function, which is represented as follows:

\[
S(x) = \frac{1}{1 + e^{-x}}
\]

(2)

It can be deducted from the previous formula, that the output of this function will always be in the range (0,1) which is perfect representation of probabilities and classifications, where large negative real numbers will become 0 and large positive numbers will become 1. It has been widely used for this property, which makes a clear distinction between

2More on Dota2. Online: https://blog.openai.com/more-on-dota-2/
predictions. Recently, other alternatives have been used to deal with this issue, like the Rectified Linear Units (ReLU) activation function, computed with the following formula:

\[
f(x) = \max(0, x)
\]  

ReLU is an extremely simple function, which returns zero, when \( x < 0 \) and is linear, when \( x > 0 \). Naturally, its main advantage is that it does not constrain the input space into a small region, avoiding the vanishing gradient problem\(^3\), which is considered a major drawback, inherent to the use of sigmoid functions.

Going back to detailing the usual training process, once the output from the activation function is obtained, the model’s error is calculated using a cost function (e.g., the mean squared error) and its value is back-propagated and used to figure out any necessary updates to the weights in the network, in order to have succeeding outputs to be closer to the target output, consequently minimizing the error for each output neuron and the model as a whole. This method is called Backpropagation [19], one of the most widely applied algorithms in ANN models.

Once the training process is over, the trained network (with the already updated weights) is then tested on new data in order to evaluate its generalization capability and accuracy. The performance of a model can finally be measured by relying on specific evaluation metrics, commonly used in classification tasks.

Overall, the increasing importance and popularity of ANN models to solve pattern recognition problems has been primarily due to their low dependence on domain-specific knowledge and the availability of efficient learning algorithms, inciting the usage of these systems in several different applications [17].

III. SOLUTION

A. Approach

Our approach to improve AI agents’ decision process at a strategic-level in the game DotA2 was built using a Supervised Learning (SL) approach, to tackle a strategy-classification problem. The main idea was to make agents able to classify each game state with a certain strategy (from a list of different strategies) that corresponds to the best one to be employed in that state. The correct strategy to take at a given time would vary, based on each agent’s in-game role and on other relevant features. To accomplish this, high-skilled human gameplay data was captured and annotated by experienced human players, and a classifier was trained and tested, according to gathered info.

Although we had also planned to explore possible integrations of the trained and tested classifier with the DotA2 Bot Scripting API, in order to properly dictate for each Bot which strategy it should be employing at real-time, hopefully resulting in a considerable improvement of their strategy-planning throughout a match, unfortunately this could not be performed due to specific constraints from the DotA2 API, that weren’t initially expected. As so, the whole development process to be explained in this section, will only concern the previous steps. Therefore, our solution was designed to be run in the following steps: Collecting Data, Labeling Replays, Defining relevant features, Training Classifier, Finding Accuracy Results, each described in the following section, regarding implementation.

B. Implementation

1) Collecting and Labeling Replays: Initially, an amount of 100 replay files were collected via the Opendota platform\(^4\). Replays are full recordings of matches that already happened and are saved in a specific file format, which can be queried and run for a full simulation of these past matches. As so, the idea was to analyze the simulations in DotA2 engine, annotating which strategies the support player of the winning team was applying, according to a list of 20 different strategies, which can be consulted at DotA2 Bot API Modes\(^5\). The goal was to create a cause-effect relation between what was happening in-game and which strategy the experienced human support player decided to apply in that situation, so it makes sense to have these strategies being the exact same as those that any default game Bot can already employ.

Strategies had to be hand-labeled, and this process was performed per second of replay, during the 15 minutes of each match, summing up to 900 labeled strategies per replay. Only 10 different replays were used, due to time constraints, effectively resulting in a total amount of 9000 labeled strategies to be used as data.

2) Parsing and Defining Features: To be able to understand when a certain strategy should be employed, a “bridge” between what is currently happening in-game and optimal strategies to be taken needs to be defined. In order to do this, relevant attributes that represent the nuances of different game-states and directly affect strategic decision-making, need to be considered.

Since replays allow for complete simulations of matches, it means they contain all data regarding these matches, although there is no evident way to extract it. To solve this, existing replay parsers like manta\(^6\) and clarity\(^7\) were considered. We ended up choosing clarity due to having clearer code examples\(^8\), showcasing the parser’s capabilities. The community around this project was also very active and the parser itself was still maintained, regularly updated and improved based on the feedback from its users. This parser allowed to break the data contained in replay files into different categories, from which the most important to our case are the entities, which are related to every unit present in-game, like heroes, creeps and structures. Using this tool, we wrote a program that would receive any replay file as input and, for each second

\(^3\)https://medium.com/@anishsingh20/the-vanishing-gradient-problem-48ae7f501257
\(^4\)Open source Dota 2 match data and player statistics. Online: https://www.opendota.com/
\(^5\)Online: https://developer.valvesoftware.com/wiki/Dota_Bot_Scripting
\(^6\)Online: https://github.com/dotabuff/manta
\(^7\)Online: https://github.com/skadistats/clarity
\(^8\)Online: https://github.com/skadistats/clarity-examples
of its match, during the first 15 minutes, would extract 70 planned relevant game features, ranging from basic hero stats, like hitpoints, mana and spatial positions, but also somewhat more complex attributes like distances between heroes and distances to laning and jungling areas, all of which have a great impact on strategic decision-making. Half of our features are the variation of values obtained in the current processed second, from the values they had in the previous second. This is important since it allows us to understand what was happening in the previous moment that lead up to the current second, which has influence in the strategies employed.

By concluding this step, we ended up with a total of 9000 data-points, each with 70 relevant attributes. Although the data wasn’t great in size, it was significant enough to move on to the next step: building, training and testing our classifier.

3) Training and Testing the Classifier: While searching for any Machine Learning(ML) softwares to be used for building, training and testing our classifier, we found Waikato Environment for Knowledge Analysis(WEKA), which contains a collection of algorithms and data pre-processing tools, designed so that users can quickly try out any existing ML methods on new datasets, in very flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing both the input data and the result of learning [20]. This was our software of choice, after reading about some of its benefits, and understanding how straight-forward the interface was.

After having properly analyzed our data with WEKA, it was conclusive that there were class imbalances, with some classes having a very high number of instances while others were not even present in the dataset. WEKA also offered numerous classification and regression algorithms, available to be picked from, and that could be applied to our preprocessed data. A study performed by Patel et. al [21], compared the performance of four different classification algorithms, available in WEKA, using the well-known iris flower dataset\(^9\). The evaluation of the performance of each classifier was made with four commonly used metrics in classification problems, which are also used in our evaluation section. Although the iris dataset is fairly simpler than ours, it presents a similar type of multi-class pattern recognition problem, which lead us to ponder our choice of classifier, based on the conclusions drawn from this study. The results obtained revealed that the MLP classifier had the best performance in all metrics used for evaluation, hence ending up being chosen, to address our own classification problem.

We decided to download an extra package from WEKA’s package manager, which contained a MLP implementation less secretive about its specifics and had more advancements, like dropout, when compared to the default WEKA’s MLP package which was somewhat obsolete.

Regarding its training method, this implementation uses mini-batch gradient descent (with a batch size of 100 training examples and 1000 training iterations), which is a variant from the gradient descent algorithm, treated nowadays as the dominant method used to train deep learning models. The goal of this algorithm is to find model parameters (weights, in NN architectures), updating those to minimize the error of the model in predictions made with the training data. This is performed by making changes to the model that move it along a gradient or slope of errors down towards a minimum error value, hence the name given to the algorithm. The mini-batch gradient descent is a variation of this algorithm, that splits the training dataset into smaller batches, used to calculate model error and update model weights. This specific method has some upsides when compared to other two variations of gradient descent\(^9\):

- It is computationally more efficient than the Stochastic Gradient Descent(SGD) method, which calculates the error and updates the model for every single example in the dataset, causing it to be more computationally expensive than other configurations
- Its update frequency is higher than the other variation (batch gradient descent), allowing for a more robust outcome, avoiding premature convergence of the model to a less optimal set of parameters

When it comes to activation functions used, this implementation uses ReLU as an activation function for the hidden layers and Softmax for the output layer. In fact, the softmax function used in our classifier is basically an extension of the sigmoid function (detailed in the section describing ANNs), but used for multi-class domains, where we have multiple outputs, creating a probability distribution that adds to 1. The aforementioned vanishing gradient problem is also one of the reasons we opted for this NN package, which had the ReLU implemented for hidden units. Overall, this specific activation function brings a good set of benefits when compared to the commonly used sigmoid function:

- It is less computationally expensive, because it involves simpler mathematical operations
- It was found to accelerate the convergence of gradient descent, resulting in faster training [22]
- It does not constrain the input space into a small region, avoiding the vanishing gradient problem

Regarding our NN structure, the final model contained 1 hidden layer with 50 nodes. This was decided by using the rule-of-thumb approach, presented by Panchal & Panchal [23], which is a method for determining the ideal number of neurons per hidden layer, for each case, stating that the number of hidden neurons should be in a range between the size of the input layer (70, in our case), and the size of the output.

\(^9\)Iris flower dataset explained. Online: http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

layer (20 initially, reduced to 8, as will be explained next), and it should be about 2/3 the size of the input layer, plus the size of the output, hence ending up with 50 neurons. Dropout regularization was also applied to our network, with the goal of creating a model that has a better generalization and less likelihood of overfitting the training data. Dropout is a technique where randomly selected neurons are ignored during training. This is a useful concept: by switching some neurons off, it prevents them from becoming too specialized in the data they are training with, having other neurons to step in and handle the representation required to make predictions for the missing neurons, increasing adaptability. Our dropout rate was 50% in hidden layer and 20% in input layer (percentage of units subjected to dropout, in each layer), based on some useful pointers about proper dropout usage, available at [24].

Throughout any of these implementation sub-sections, training and testing was constantly being performed, and analyzing respective obtained results was crucial to understand what needed to be changed. In sum, the “Defining features” and “Training and Testing the Classifier” sections were performed multiple times, depending on the results obtained, altering their specifications iteratively, in order to get a satisfying final outcome.

70% of our data was used to build a training set and the left out 30%, to serve as testing set. 10-fold cross-validation was performed on our training set, in order to flag problems like overfitting, and to give an insight on how the model would generalize to an independent dataset. Once this was done, the model was tested on our previously prepared independent dataset. Our initial results were poor, but after analyzing some metrics displayed in WEKA’s classifier output window, it was clear that class imbalances were affecting the development. Some classes were present in very few data-points of our sets, that the model simply wasn’t able to learn them. In more extreme cases, some class categories only appeared on either the training or the testing datasets, so they were either not learned or never tested, which obviously skews the results. To tackle the issue, we decided to remove some classes and respective data-points, based on the following criteria:

1) classes represented by less than 2% of the total data
2) classes that represent rare in-game situations, hence only present in one of all DotA2 match replays collected as data

After applying this method, 20 output classes were reduced to 8, with these representing the most frequent and common in-game strategies: laneing, attack, roam, retreat, rune, farm, ward and defend_ally. This means that, if AI Agents were ever able to have, in regard to these strategies, a near similar decision-making to the highly-skilled gameplay gathered as data, that would turn out to be a considerable improvement in comparison to the performance of available default Bots.

Despite having hand-picked the “most promising” classes, there were still some imbalances, with half of the 8 classes representing a huge majority of the total data. To solve this issue, there are several different methods that can be applied to imbalanced datasets [25], such as:

1) Undersampling - aims to balance class distribution through the random elimination of the majority class examples/instances. Drawbacks from this method are that it can potentially discard useful data that could be important for the induction process, and also not ideal in our specific case, where would be useful to increase the data size
2) Oversampling - aims to balance class distribution through the random replication of the minority class examples/instances. A major drawback from this method is that it can increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples. In short terms, overfitting means that the predictive power of a classifier will be very good on a specific type of data (the training data), but the model will fail to fit additional data or predict future observations reliably
3) Synthetic Minority Over-sampling - aims to balance class distribution through the generation of synthetic minority examples to over-sample the minority class. This generation is performed by interpolating between several minority class examples that lie together. A great benefit of this method is that, since new instances generated are in fact different and not exact copies, this can help avoid the overfitting problem that occurs in the regular random oversampling.

After researching pros and cons of each, Synthetic Minority Over-sampling Technique (SMOTE) [26] seemed the right choice, since it was also available as one of many pre-processing tools from WEKA. In more detailed terms, when using SMOTE, the minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the k minority class nearest neighbors. The amount of neighbors used for the synthetic generation of new samples will depend on the amount of over-sampling needed. In our case, from the 8 classes that ended up being chosen, half were over-sampled by 300%, which means that, from the 5 nearest neighbors used in WEKA’s SMOTE package implementation, only 3 were picked and new samples are generated in the direction of each. The method applied to generate more samples is quite simple: take the difference between the feature vector (sample) under consideration and its nearest neighbor. Multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration.

With this, we were able to tackle some of the main imbalances between all 8 classes, consequently improving the consistency of the results obtained, which are presented in the following section.

4) Accuracy Results and Evaluation: Our shell script, used to randomly split the data in training and testing datasets, was run 10 times in order to increase the consistency of obtained results.

Due to constraints related with the integration of the ML model with the DotA2 API, that restrained us from testing the classifier in a real-case scenario (using it on a real
DotA2 match), consequently limiting our evaluation process, we relied on the following described metrics, to theoretically evaluate how well would the model actually perform:

- **Classification accuracy & Kappa Statistic**
  Accuracy is often the starting point for analyzing the quality of a predictive model, as it presents the percentage of instances that were associated with the correct class, however it can sometimes be an unreliable metric, since it does not take into account possible dataset imbalances. Other metric is Kappa statistic, a metric that compares our observed accuracy with an expected accuracy (accuracy if classification was random). It can be calculated using the following formula:

\[
Kappa = \frac{observed\text{accuracy} - expected\text{accuracy}}{1 - expected\text{accuracy}} \tag{4}
\]

By making use of the formula, one can quickly realize that having a negative Kappa statistic value means that the observed accuracy is smaller than the excepted accuracy, resulting in a model performing worse than if it was randomly classifying instances.

- **True and False Positive Rates**
  The following metrics are often used in binary classification problems, however they work just as fine in multi-class problems like ours, since they can ultimately be simplified to two-class problems using the one against all approach [27]. By convention, in binary classification, the class label of the minority class is called positive while the class label of the majority class is called negative. Using the mentioned approach, with multiple classes, we compare each one individually against the rest, with the class being compared called positive and all other classes called negative. Rates are calculated using 4 essential concepts: True Positives(TP), which is the number of positive cases correctly classified as such; False Negatives(FN), the number of positive cases incorrectly classified as negatives; False Positives(FP), the number of negative cases that are incorrectly identified as positive cases and True Negatives(TN), the number of negative cases correctly classified as such.

Having the values for each of these 4 concepts, TP and FP Rates can then be calculated using the following formulas:

\[
FPrate = \frac{FP}{FP + TN} \tag{5}
\]

\[
TPrate = \frac{TP}{TP + FN} \tag{6}
\]

- **Precision**
  Commonly used in these type of classification problems, it is a measure of a models exactness. A high precision value is usually an indication of a good classifier. It can be calculated using the following formula:

\[
Precision = \frac{TP}{TP + FP} \tag{7}
\]

Precision and TPRate(also known as Recall) are often used together to fully evaluate the effectiveness of a model: high precision means that most examples classified as positive were indeed positive, while high TPRate means that most positive examples were correctly classified as so.

- **F-Measure**
  Often called F-Score, it measures a test’s accuracy, considering both previous metrics Precision and Recall to compute the score, using the following mathematical expression:

\[
F\text{measure} = \frac{(1 + \beta^2) \times \text{Recall} \times \text{Precision}}{\beta^2 \times \text{Recall} + \text{Precision}} \tag{8}
\]

Where \( \beta \) corresponds to the relative importance of recall versus precision, and is usually set to 1 in problems where both equally matter.

Knowing now the meaning behind some concepts, we can express the previously mentioned accuracy in the following mathematical terms:

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}
\]

Accuracy can be predominantly contributed by a large number of TN examples which, in most circumstances and real-case scenarios, are not nearly as important as FN and FP. Therefore, F-measure is usually a better metric to use, if we are seeking a balance between Precision and Recall and there is an uneven class distribution.

- **Matthews Correlation Coefficient**
  Yet another measure commonly applied to binary classifications, Matthews Correlation Coefficient(MCC) can also be applied to multi-class domains, using the previously mentioned one-vs-all approach, and then calculating the average of each class MCC value. It can be summarized with the following formula:

\[
MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{10}
\]

We also decided to utilize MCC to evaluate our model, since its value is least influenced by unbalanced data, serving as a balanced measure which can be used even if the classes are of very different sizes [28].

- **Area Under the ROC Curve**
  One of the most common metrics used is the Receiver Operating Characteristic(ROC) curve analysis, and the associated use of the Area Under the ROC Curve(AUC) to assess overall classification performance. The ROC curve is represented by a graph showing the performance of a model at all classification thresholds. This curve plots
two already mentioned parameters: FP rate on the x-axis and TP rate on the y-axis. Again, we use the one vs. all approach to draw ROC curves for each class, and look at the weighted average.

After having individually analyzed all of the previously detailed evaluation metrics, it can be stated that, although it presents far from perfect results, our classifier is still considerably better than any random classifications, and we can only speculate that it would improve DotA2 default agents’ decision-making. All results obtained can be summarized in Table I, which hopefully serves as a viable measurement of our classifier’s performance, and from which we can conclude the following:

- **Classification Accuracy** is, on average, **54.519%**, which is considerably better than any similar 8-class model making random classifications (which would yield, on average, a 12.5% accuracy, assuming equally balanced classes)
- **Kappa Statistic** has, on average, a value of **0.37**, which, according to the scale created by Landis and Koch [29], is considered “fairly”, better than a random model
- **TP Rate and FP Rate** are, on average, **0.545** and **0.172**, respectively. It is quickly deductible from these values that, more than half of actual positive examples were correctly identified as such and only a small percentage of negative events were wrongly categorized as positive (less than 20%)
- **Precision** is, on average, **0.558**, meaning that, more than half the examples classified as a specific class, were correctly identified
- **F-Measure** is, on average, **0.535**. Since it is a combined metric, it is hard to think of an intuitive meaning for the result
- **MCC** is, on average, **0.385**. This metric returns a value between -1 and +1, with +1 representing a perfect prediction, 0 as no better than random prediction and -1 the worst possible prediction. Our result is considerably higher than 0, meaning it is better than random
- **AUC** is, on average, **0.785**. This metric varies between 0 and 1, with an area of 1 representing a perfect model and an area of 0.5 representing a worthless model [30]

### IV. Conclusion

#### A. Main Contributions

The goal of this work was to develop a supervised learning model that would have better control over which are the ideal strategies to be employed by AI Bots, during a DotA2 match. In this document we covered the implementation process of our solution, and the accuracy results obtained from testing our SL classifier. By analyzing these results, available on the previous section, we can verify that our hypothesis, which declares that by building an ANN classifier, we are able to identify the correct strategies employed during early stages of a match, was successfully proven.

Overall, three important contributions of this thesis can be highlighted:

1. A java code that receives any DotA2 replay as input, which extracts relevant features, useful for strategic decision-making, with these being processed for each second of the match. Hopefully this program will be useful for any future works regarding the topic of ML in DotA2.
2. A trained and tested NN model, which had decent evaluation results, and would, in theory, significantly improve the strategic decision-making of the available default DotA2 Bots, had it been successfully integrated with the Scripting API.
3. This document also pointed out existing issues that have a negative impact in the development of ML systems for DotA2 and that are hopefully tackled and dealt with in future works, like not having any available methods (to our knowledge) of automatic-labeling for DotA2 replays.

#### B. Future Work

Although we had started with the idea of building 5 classifiers, one for each in-game role, the support role ended up being chosen, for the aforementioned reasons. As so, and due to time constraints, many experiments and tests regarding the other 4 roles have been left for the future:

1. Properly building other 4 classifiers, which would require the extraction of different features, regarding each role specifications.
2. In contrast to the support role, some others that were left out, present richer decision-making patterns in later stages of the game, which indicates that, more interesting results would be obtained if replays were to be labeled for the entirety of each match duration. This would allow each role to an equal chance of presenting complex behaviors, also improving our solution, which would ultimately be able to dictate decisions for Bots to employ, during the entirety of any match.
3. Regarding our dataset, its augmentation by labeling more matches would probably result in more accurate and consistent results, since it would increase the diversity of in-game situations. This can be easily achievable in the future to improve the work, since the task itself is not hard, just time-consuming. Ideally, a future work that would greatly help, would be coming up with a method to allow automatic-labeling of DotA2 replay matches, although this seems very hard to accomplish, due to the complexity and amount of variables present in MOBA environments.

#### REFERENCES

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Accuracy(%)</th>
<th>Kappa Statistic</th>
<th>TP Rate</th>
<th>TP Rate</th>
<th>Precision</th>
<th>f-Measure</th>
<th>MCC</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59.0725</td>
<td>0.4266</td>
<td>0.591</td>
<td>0.164</td>
<td>0.592</td>
<td>0.581</td>
<td>0.441</td>
<td>0.812</td>
</tr>
<tr>
<td>2</td>
<td>54.6485</td>
<td>0.3808</td>
<td>0.546</td>
<td>0.165</td>
<td>0.577</td>
<td>0.543</td>
<td>0.403</td>
<td>0.787</td>
</tr>
<tr>
<td>3</td>
<td>56.2084</td>
<td>0.3977</td>
<td>0.562</td>
<td>0.161</td>
<td>0.552</td>
<td>0.544</td>
<td>0.412</td>
<td>0.822</td>
</tr>
<tr>
<td>4</td>
<td>54.3145</td>
<td>0.3367</td>
<td>0.543</td>
<td>0.209</td>
<td>0.521</td>
<td>0.519</td>
<td>0.344</td>
<td>0.741</td>
</tr>
<tr>
<td>5</td>
<td>49.9628</td>
<td>0.3295</td>
<td>0.500</td>
<td>0.143</td>
<td>0.606</td>
<td>0.505</td>
<td>0.368</td>
<td>0.807</td>
</tr>
<tr>
<td>6</td>
<td>51.8903</td>
<td>0.3717</td>
<td>0.519</td>
<td>0.132</td>
<td>0.582</td>
<td>0.533</td>
<td>0.391</td>
<td>0.769</td>
</tr>
<tr>
<td>7</td>
<td>58.8934</td>
<td>0.4143</td>
<td>0.589</td>
<td>0.169</td>
<td>0.616</td>
<td>0.588</td>
<td>0.443</td>
<td>0.807</td>
</tr>
<tr>
<td>8</td>
<td>51.5584</td>
<td>0.3086</td>
<td>0.516</td>
<td>0.212</td>
<td>0.477</td>
<td>0.487</td>
<td>0.304</td>
<td>0.736</td>
</tr>
<tr>
<td>9</td>
<td>51.4830</td>
<td>0.3433</td>
<td>0.515</td>
<td>0.174</td>
<td>0.503</td>
<td>0.500</td>
<td>0.349</td>
<td>0.763</td>
</tr>
<tr>
<td>10</td>
<td>57.1586</td>
<td>0.3911</td>
<td>0.572</td>
<td>0.188</td>
<td>0.556</td>
<td>0.547</td>
<td>0.390</td>
<td>0.807</td>
</tr>
</tbody>
</table>