CMP304 Artificial Intelligence Unit 2 Report

**Callum Myers**

**2102549**

# **Introduction and Background**

The application I have submitted is an AI driven player classifier, using a supervised learning decision tree classifier model. It reads in data about players from the Rocket League 2021-2022 professional e-sports tournament, RLCS, and classes them into different categories based on their performance. It was a suitable application choice because it has a large enough dataset and has many statistics which can be used to place players in different classes. My solution assumed that users would know a suitable amount about the game and tournament, such as which areas players come from, and what different statistics mean. I will however explain this information later in this document. The libraries required for this program to run are:

* Pandas
* Matplotlib
* Sklearn

# **Data Specification**

The dataset used in the program (Kaggle, no date) is a large dataset with 106,795 entries in it. Each entry relates to a specific player’s performance in a match. There are 103 columns however in the program only a small selection of these are used.

The columns used are player\_id, a string used to uniquely identify each player, team\_region, a string showing which region each player was playing in, core\_goals, an integer showing the number of goals a player scored during a match, core\_assists, an integer showing the number of assists a player had during a match, core\_saves, an integer showing the number of saves a player had during a match, demo\_inflicted, an integer showing the number of demos a player achieved during a match and movement\_percent\_supersonic\_speed, a float showing the percentage of time in per match during which the player was at maximum speed.

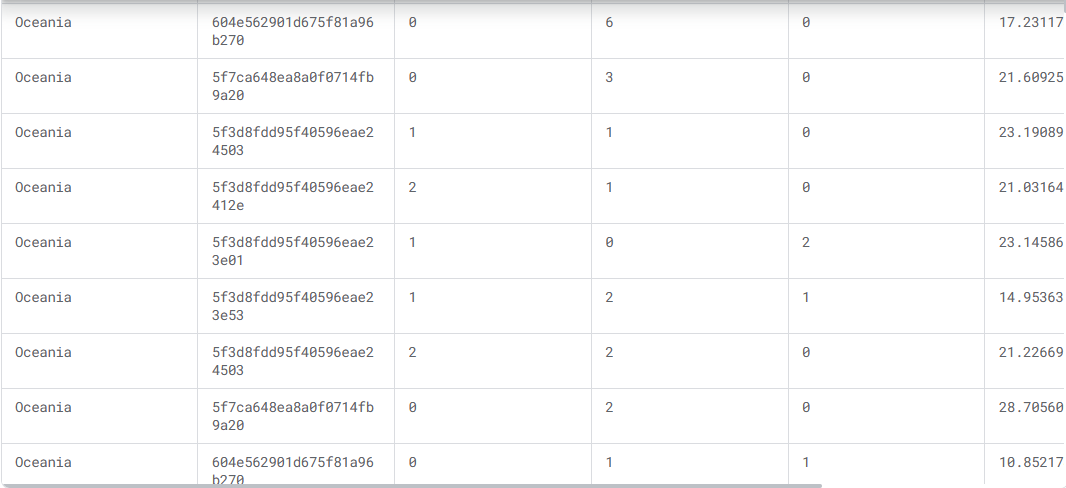


Photo from dataset page (Kaggle, no date) filtered to show the columns used in the program.

From left to right: team\_region, player\_id, core\_goals, core\_assists, core\_saves, movement\_percent\_supersonic\_speed

# **Methodology**

The program is made up of four python files: DataPreperation.py, DataAnalysis.py, UserTesting.py and main.py. I will go through what each file does in turn.

Firstly, DataPreperation.py is a file which creates functions to read in and prepare the data for use by the other files. The readData() function uses pandas.read\_csv() to read in the data from the matches\_by\_players.csv file from the dataset folder (Kaggle, no date) and stores this into a variable called players\_data. Next, it creates another variable called filled\_data which uses .fillna() to fill any empty slots with a given value, in this case 0 was used. This is done to ensure there are no NaN (not a number) errors given when using this file in other places. The next function is selectRegion(), which brings in the region variable as a parameter. This function then takes whatever value is in the region parameter and filters out the players\_data from before to only contain entries where the value in the region\_id slot matches the value stored in region. This new filtered dataset is then returned. Finally, there is the plotElbow() function. This firstly created a variable called elbow\_check which is given a set value, and this is the maximum number of clusters to run the next part on. The function then loops for this elbow\_check variable’s value and runs the kMeans function on each iteration using the filled\_data dataset, then stores an inertia value into the list wss. This list then plots a point on a graph for each of the values contained within it, and this is displayed to the user when the function ends. Using this function, the user can see the point at which the optimal number of clusters have been reached to classify players.

Next, DataAnalysis.py is a file which controls analysing the data provided and classifying each player from the dataset into a class. These classes are as follows:

* Maniac – players with above average stats in the demo\_inflicted and movement\_percent\_supersonic\_speed columns.
* Goalscorer – players with above average stats in the core\_goals column and below average stats in the core\_saves and core\_assists columns.
* Playmaker – players with above average stats in the core\_goals and core\_assists columns, and below average stats in the core\_saves column.
* Goalkeeper – players with above average stats in the core\_saves column and below average stats in the core\_goals and core\_assists columns.
* Defender – players with above average stats in the demo\_inflicted and core\_saves columns, and below average stats in the core\_goals column.
* All-rounder – players with roughly average stats in the core\_goals, core\_saves and core\_assists columns.
* MVP – players with above average stats in the core\_goals, core\_saves and core\_assists columns.
* Weak link - players with below average stats in the core\_goals, core\_saves and core\_assists columns.
* Unclassified – default catch class for any players who do not fit into any of these classes.

The first function of the file is the classifyPlayers() function. This function takes in two parameters: a dataset stored as players and a list of matches played per player. It then begins by getting the total goals each player scored across all matches they participated in by using the .sum() function on each of the stat columns required, then adds all of these total values into a list created for each column, and checks to see what the maximum for each list is. Once these have been calculated, the next section then iterates over the dataset again and uses the totals calculated, divides this by the number of matches each player has played to get their average goals per game, and divides this by the maximum of the list to normalize the data between zero and one. By using the average goals per game this balances out the data, since some players played more games than others their stats would be expected to be higher, and this can invalidate the results since it would not reflect how effective they actually are, since these stats could actually be low for how many matches they played, but would still be well above the values of players who only played for less of the tournament. These normalized stat values are then added to their own lists to be used again later. Next, the overall average stats per game are calculated by running the mean() value on each of the normalized lists to get an average, and stores these averages in variables. It then again iterates through the players dataset and calculates each player’s normalized data, then using a range of if else statements it compares each set of normalized stats to the averages to select which class to put each player in. These class names are then stored in a dictionary called playerClassifications which stores a class name next to each player’s ID. Finally, it sends this new dictionary into the drawPieCharts() function from UserTesting.py (which will be explained later in this document) to output the results of the calculations. The next function of this file is called classifyPlayersAI(), which again brings in a dataset labelled as players, and a list of matches played per player. This function works similarly to classifyPlayers() however is done using the AI. First, it created 2 lists: X and Y. X is used to store the relevant statistics of each player, and Y is used to store the labels attached to each player. It then iterates over the dataset, and for each row it adds each player’s average stats per game to the X lists by first putting them into a row and then adding this row to X. It then runs through the labels applied to each player by classifyPlayers() which are currently stored in playerClassifications, and adds each one to the Y list by giving each label an integer between one and nine, for example if a player has the maniac class stored in playerClassifications it will add a number one to the Y list. After leaving this loop, it then uses the train\_test\_split() function to split all of the data into two, with one section used for learning (x\_train and y\_train) and one section used to test the AI’s learning (x\_test and y\_test). Next, a decision tree classifier is made and stored as clf, then using the .fit() function it fits in the x\_train and y\_train datasets to teach the AI the rules for classifying the players based on the stats attached to each label. Using the x\_test dataset, which is the unseen portion of the data, the function runs the clf.predict() function and stores these predictions in the y\_predict list. By iterating through this list it stores each prediction in the dictionary called playerClassificationsAI, storing each prediction on a new row, and sends this dictionary to the drawPieChartsAI() function from UserTesting.py (explained later) to display the predictions the AI made as a chart. Finally, the function creates a classification report and prints this to the user which will show the precision of each class, recall rates, F1-scores and overall accuracy.

The next file is UserTesting.py which is simply used to output graphs to plot results. There are two functions however there is only one very minor change between the two. First, they both create variables to count the number of occurrences in each classification of players. Then, it iterates through the given dictionary stored as classes, and checks what is stored in each row. In drawPieCharts(), this will check for the string of each class, whereas in drawPieChartsAI(), it will check for the integer instead as this is how the AI is set to store predictions. When it finds the correct classifier, it adds one to that classifier’s variable counter, and if the integer cannot be found within the if elif statements then it will add one to undefined. It then defines labels for each section of the charts, which are just names of the classes, then sets the sizes to be the variable counters for the appropriate class. Finally, it plots these using the .pie() function, rounds each section to a one decimal point percentage, and outputs this to the user.

The final file is main.py, and this is simply used to run appropriate functions from other files. It first runs the readData() function from DataPreperation to read in the file. Then, it gives the user an option to view the optimal number of clusters using the showElbow() function, if the user inputs 1 it will draw the graph, if they input anything else it will skip this function. Next, it creates a list called valid\_regions, which contains all regions available in the dataset under the team\_region column. It then runs a while loop which will continuously run until the player types quit to exit the program. Within this loop, the user is asked to select a region to get player classes from. This then runs the selectRegion() function using this input, after validating the data input against the valid\_regions list, and then narrows the data down to this region before grouping it by player\_id. Next, it uses the size of the dataset to calculate the number of matches each player played. It then uses this dataset and matches\_played list to run the classifyPlayers() and classifyPlayersAI() functions, until the user types quit to exit the program.

# **Results and Conclusions**

My application runs as expected; however, I do not feel the AI seems to be overly effective at learning. I found the overall accuracy of the AI’s classification to be from 38%-50% when compared to the manual classification.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

The chart on the left displays the classification of the Europe region of the dataset, and the right is under the same test case with the AI. The proportions of each class seem to be mostly accurate, however the percentages are not. This was done with a 0.2 test size value.

A screenshot of a graph

Description automatically generated

This is my classification report (Statology, 2022) from this test. As the data shows, the most accurate class for the AI was the maniac class, with a precision score of 53% and a recall score of 55%. My worst scoring class was defender, with a 19% precision rate and a 27% recall rate. The AI seems to have trouble identifying defenders and weak links, which may be because defender has relatively specific requirements compared to the others, and weak link would be the last class which the AI would check therefore it may have already classified players before getting there.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

These graphs above are taken from running the program with the region set to select North America and the test size value of 0.1. It shows a similar trend to the Europe results with defenders again being a small class and difficult for the AI to identify the players in it correctly, but maniac did the best again.

These results overall show that the AI is having difficulty accurately putting players into uncommon classes with specific rules. This is most likely to be because these classes are not used often therefore the amount of exposure that the AI has to these classes in the training set will be limited, and it may not fully understand what the rules of these specific classes are. This could be remedied with a dataset which is specfically engineered to have more entries into these classes, which would show the AI when to use it more effectively, or having a larger dataset may also help since it is more likely to be naturally exposed to the rules more often. There is also 9 classes for the AI to choose from, each with very different rulesets, and therefore means the AI may fall down the decision tree but classify the player before checking the node which contains the correct class.

If I were to expand on this model in the future, I would look further into the sklearn library to see which other models I could look into using. I would also look into having it use two separate large datasets, allowing it to train and test from a separate dataset before the user can use it to classify their own. I feel it would also help if I were to experiment with class weighting since there is a rather large imbalance between the amount of ocurrences within each class, most prominent when looking at the Europe results above, where weak link has only 3 occurences while maniac has 124.

# **References**

Data Camp (2022) *Classification in Machine Learning: An Introduction.* Available at: <https://www.datacamp.com/blog/classification-machine-learning> (Accessed: 5 May 2024)

Datagy.io (2021) *Pandas Mean: Calculate Pandas Average for One or Multiple Columns.* Available at: <https://datagy.io/pandas-mean-average/> (Accessed: 4 May 2024)

Digital Ocean (2019) *How To Build a Machine Learning Classifier in Python with Scikit-learn.* Available at: <https://www.digitalocean.com/community/tutorials/how-to-build-a-machine-learning-classifier-in-python-with-scikit-learn> (Accessed: 6 May 2024)

Finxter (2024) *5 Best Ways to Fill Missing Values in a DataFrame with Python.* Available at: <https://blog.finxter.com/5-best-ways-to-fill-missing-values-in-a-dataframe-with-python> (Accessed: 4 May 2024)

Kaggle (no date) *Rocket League Championship Series 2021-2022.* Available at: <https://www.kaggle.com/datasets/dylanmonfret/rlcs-202122?resource=download> (Accessed: 30April 2024)

Stack Overflow (2012) *Renaming Column Names In Pandas.* Available at: <https://stackoverflow.com/questions/11346283/renaming-column-names-in-pandas> (Accessed: 4 May 2024)

Stack Overflow (2012) *Finding The Average Of A List.* Available at: <https://stackoverflow.com/questions/9039961/finding-the-average-of-a-list> (Accessed: 5 May 2024)

Stack Overflow (2013) *How can I iterate over rows in a Pandas DataFrame?* Available at: <https://stackoverflow.com/questions/16476924/how-can-i-iterate-over-rows-in-a-pandas-dataframe> (Accessed: 6 May 2024)

Statology (2022) *How to Interpret the Classification Report in sklearn (With Example).* Available at: <https://www.statology.org/sklearn-classification-report/> (Accessed: 6 May 2024)